ugobe®

LIFE FORMS

Pleo Programming Guide

Confidential – Not for external distribution/publication!

Table of Contents

Introduction

Pleo is designed to be modified and extended in multiple ways, from the simplest, such as replacing a built-in sound with one of your own – to the most complex, such as writing a new application for Pleo using the built-in scripting language.

This guide will show you how to use these various techniques to modify and extend your Pleo.

There are many tools associated with developing Pleo's applications and extensions. This document attempts to describe each of these tools in order to allow a wide range of users to modify and extend Pleo's functionality.

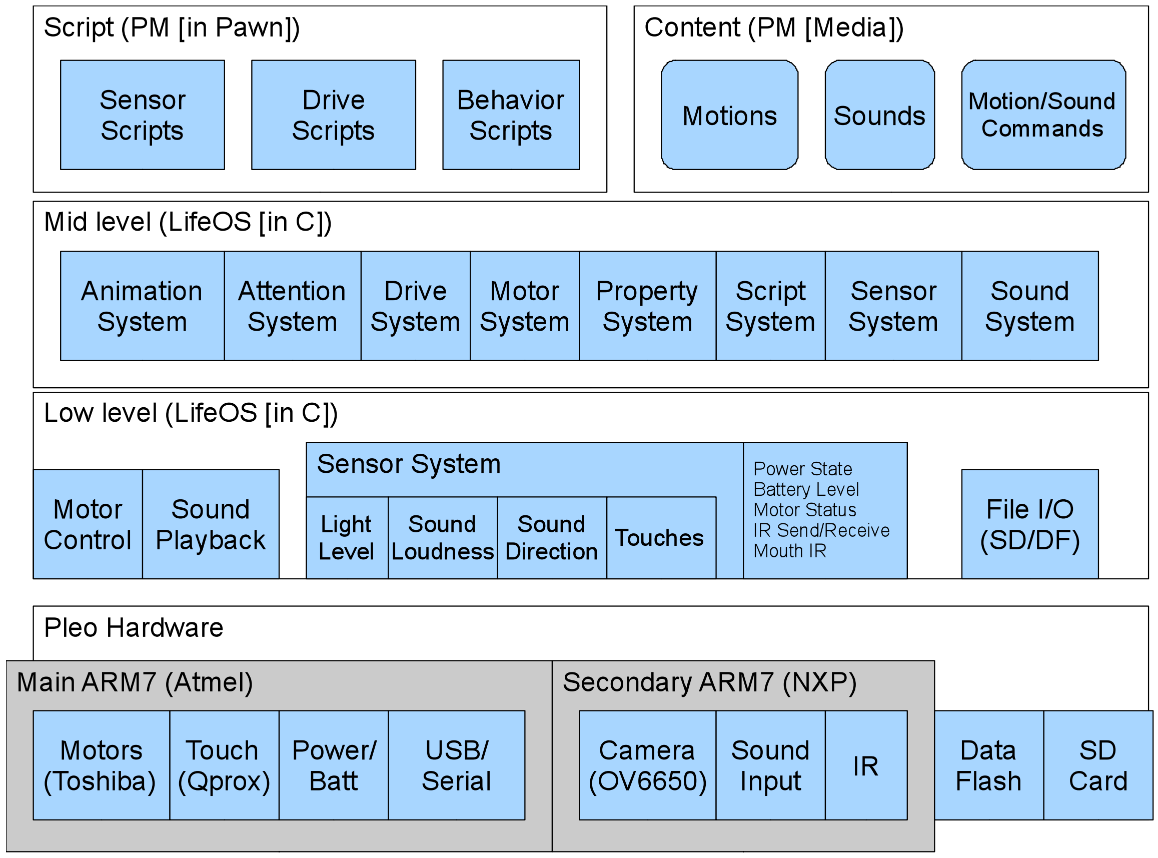
The software built into Pleo is referred to as the *LifeOS*. The LifeOS architecture closely resembles the standard operating system model, using a layered system of modules. The LifeOS is broken down into three major layers:

- **low-level:** This layer is an interface to the real hardware components. This includes the sensor drivers that read sensor information, passing it up to the mid-level layer, and output systems like the motor controllers, which move joints.
- **mid-level:** This layer provides the application services to the high-level. It performs much of the processing of the sensor input, and provides the native function interfaces to the high-level layer.
- **high-level:** This layer is where the majority of applications will reside, implemented using the Pleo scripting language.

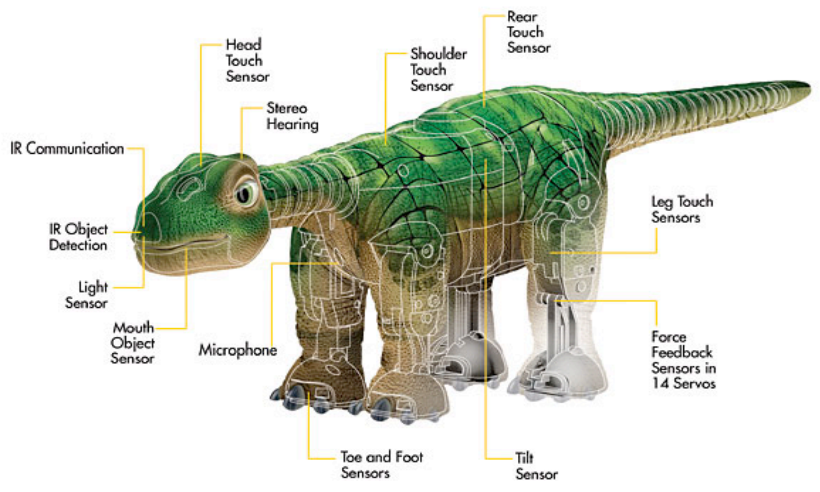
The *Architecture* chapter will summarize the available hardware resources and the main software components in the mid-level. Later chapters will explain how to create new applications for Pleo.

Architecture

Pleo consists of hardware and software components:



Hardware



Pleo hardware consists of a body with:

- 2 ARM7 CPUs: One in Pleo's head to handle camera, sound input, IR send/receive, mouth IR interrupter, and head and chin sensor input. Another ARM7 in the body handles motor control, touch sensors, sound output, and the high-level layers of the Pleo software.
- 14 motors: 2 per leg (shoulder/elbow and hip/knee), 1 for the torso, 2 for the tail (horizontal/vertical), 2 for the neck (horizontal/vertical) and one for the eyes and mouth.
- 8 touch sensors: 1 per leg, 1 on Pleo's rear, 1 on the back or neck, 1 on the chin, and 1 on the head.
- 4 foot switches: 1 per foot.
- 1 IR transceiver: For communication with other Pleos and future Ugobe life forms. May be compatible with some standard IR remotes.
- 1 IR interrupter in the mouth: Detects the presence of an object in Pleo's mouth.
- 1 orientation (or 'tilt') - sensor: Recognizes Pleo's current orientation, from six possible states.
- 1 shake sensor: Detects whether Pleo is being shaken to wake him from sleep or low-power mode.
- 2 microphones: Detect sound volume changes and direction.
- 1 camera: Detects light levels, identifies and tracks objects (color blobs).
- 1 USB port: For control and programming.
- 1 battery: Reads current charge and temperature.

(See the *Pleo Hardware Specification* for more detail)

Software

The software that runs in Pleo is divided into three layers:

- Low-level layer (or Driver)
  The low-level software deals directly with hardware, including the motor control, sensors, SD Card, battery, USB, camera, sound input, and sound output. It is implemented as a set of drivers to each hardware component. Sensor information is provided to the mid-level layer through a Blackboard system.
- Mid-level layer
  The mid-level layer provides the application functional support to the high-level scripting layer. This layer contains all the various systems of Pleo. The high-level C system components include:

  - **Sequence system**
  - **Sound system:** the sound system is responsible for the playback of sound resources. Sound resources can be included as part of an application, built into Pleo, or captured from Pleo's microphone input.
  - **Motion System:** the motion system is responsible for controlling all of Pleo's motors, or joints. It provides an API to control individual joints, playback -preconstructed motion resources, and provide detailed feedback about the current status of all the joints.
  - **Property System:** the property system allows the interchange of data between the mid-level and the script layers, or between different Vms.
  - **Script System:** the script system implements the Pleo virtual machine and allows applications to load, unload, and execute other script resources.

- High-level layer
  The script layer implements the highest-level functionality of Pleo. This is essentially Pleo's personality, determining how and when he responds to sensor input and internal goals.

Pleo Development Kit (PDK)

The Pleo Development Kit is a package of tools, documentation, and samples to assist in writing new applications for Pleo. This chapter details the use of the tools included in the PDK and gives a brief overview of how to get started writing Pleo applications.

Prerequisites

The development environment for Pleo applications is based primarily on open source tools. This includes Python, Pawn, GTK, and wxWidgets, among others.

Note: Pleo applications at Ugobe have been developed using both Windows and Linux hosts. Other operating systems may function properly, but are not tested or supported by Ugobe.

Python

There are currently two possible distributions of the PDK – binary and source. Almost the whole package is the same, except for the bin directory.

The binary version is packaged using the py2exe tool, thus obviating the need for any installed tools before using the build tools. The main build tool is named ugobe_build_tool.exe, located in the pdk/bin directory. The main issue here is that it is Windows-only.

The source version of the PDK requires only a Python installation and - depending on the version of Python used – an extra Python module or two. The Python tool set uses the ElementTree module, which is built into Python 2.5 and later, but needs to be installed in Python versions 2.4 or earlier.

Pawn

Since the Pleo firmware is built with a specific version of the Pawn run-time code, we include the matching Pawn compiler as part of the PDK. Note though that you can still use the released distributions of the Pawn system directly from CompuPhase (http://www.compuphase.com/pawn/pawn.htm), if it matches or is compatible with the version of Pawn built into Pleo. As a reference:

- Pleo 1.0.x firmware uses Pawn version 3.2.3664
- Pleo 1.1.x firmware uses Pawn version 3.3.3930

Other Notes

Some of our Python code works directly with a tethered Pleo. In this case, we need a Python module to communicate over serial ports. We use the pyserial module located here: http://pyserial.sourceforge.net/

We have developed a number of GUI tools in-house, also using Python. The vast majority of these have been developed using PyGTK and the Glade GUI building tools.

Much of the initial development of Pleo took place on Linux, using Make and Makefiles. In order to ease the work on the part of the Windows folk, we found the cygwin package invaluable during development. It should be noted the first versions of the build tools were also developed in the cygwin environment, so there may still be some 'unix-isms' in place. But we have made some effort to ensure that the tools work well in a standard Windows command prompt as well.

Directory structure

The Pleo Development Kit (PDK) is the collection of tools needed to develop projects (or applications) for Pleo. The tools consist of applications written by Ugobe to process raw media assets to the compiled, optimized format for Pleo, utilities to aide in the creation of media resources, and include files that define the interface available to the scripting language. There are also third-party tools required to build projects for Pleo. These are described next.

The file structure for the PDK is currently laid out as:

```
/pdk
    /bin
      ...tools needed to build and convert resources...
    /include
        Animation.inc
        File.inc
        Motion.inc
        ...other global includes...
        /common
            ...constants and types used across Ugobe
products...
        /pleo
            properties.inc
            sensors.inc
            joints.inc
            ...other Pleo-specific includes...
    /examples
        /template
            template.upf
            sensors.p
    /media
        /motions
            ...motion csv files from animation software...
        /sounds
            ...sound wav files for sounds that Pleo can
play...
        /commands
            ...command csv file examples...
    /tools
        /drivers
            ...Windows Pleo drivers...
```

Projects

The build system is based on the project concept. Each project is – typically – created in its own directory, with one Ugobe project file (UPF) that contains all information necessary to build a complete file set, ready for an SD Card.

A project can consist of the following file types:

- **Sounds:** Currently, we support 11k, 8-bit mono sound playback. Any sounds referenced in the UPF file will be converted using the built-in Python audioop module. Optimally, you should use the highest quality source files as possible, and the build tools will convert them to the required output format. After processing, they become USF (Ugobe Sound Files) files.
- **Motions:** Motions are the animation data we use to animate Pleo at run-time. The animations are exported from 3dsMax as CSV files and during build time become UMF (Ugobe Motion Files) files.
- **Scripts:** Scripts are Pawn source code (.p files), which contain the controlling script code. After processing they become AMX files.
- **Properties:**

Building projects

Once the project file contains all references to the resources that will be used, and how they should be built, it is processed with the ugobe_project_tool application. This application processes the UPF file, building each resource as appropriate, and places the result in the build folder.

Loading Files to the SDCARD

Once the project has been built, the files in the build folder can be copied to the SD Card. After the resource files (AMX, USF, and UMF) are placed on the card, the SD Card is inserted into Pleo. If Pleo is currently powered on, the firmware will attempt to start execution of the scripts on the SD Card.

Update: Recent versions of the PDK now combine all sound, motion and script resources into one file – called a Ugobe Resource File. This file is named based on the top-level elements name attribute. So, only this one file is now needed.

Note: see the section on *Shadowing Resources* for a more detailed discussion of how resources are located and loaded.

Hello Pleo

This section gives a quick walkthrough of a very simple Pleo application.

The Hello Pleo application waits for any touch sensor to be triggered, and plays a sound in response.

Ensure that the PDK has been installed. This includes all the tools necessary to compile motions, sounds and scripts into the formats playable on Pleo. In the following sections, we use {pdk} to signify where the Pleo Development Kit has been installed.

To start a new project, it is easiest to simply copy the {pdk}/examples/template to a new folder within the examples folder. For this sample, we will simply name it “sensor_test”. You will also want to rename the default template.upf to sensors.upf. You should have a directory structure that looks like the following:

```
/sensor_test
      /sounds
            beep.wav
      sensors.upf
      sensors.p
```

Open sensors.p. In the on_sensor function, add this code:

```
case SENSOR_HEAD:
sound_play(snd_beep);
```

This will cause the beep sound to be played whenever the head sensor is triggered (pressed or released).

To build this application, open a command window (a terminal window, bash shell or cygwin window). Change directories – using ‘cd’ – to {pdk}/examples/sensor_test. At the prompt, type the following:

```
../../bin/ugobe_project_tool.exe sensor_test.upf rebuild
```

This will execute the main build tool – ugobe_project_tool – which will parse the given UPF file and compile each listed resource – sound, motion, etc. – into a newly created ‘build’ folder.

If all resources are built correctly, then a pleo.urf (Ugobe Resource File) will also be created in the build folder. This is a combined file of all resources, such as sounds, motions, and scripts.

Copy this pleo.urf file to a blank SD Card. Insert the SD card into a Pleo that has been turned off, and then turn Pleo on. When you touch his head, you should hear the beep sound.

Media Creation

Media is clearly an important element when creating Pleo applications. There are two main resources that are needed by Pleo – not including scripts: motions (synonymous with animations) and sounds.

This section will give an overview of how these resources are created.

Making Motions

Currently, the only way to generate motions for Pleo is through a special exporter created especially for 3dsmax (versions 7+). There is a Pleo model – or rig – that is animated by the animator and then exported to a format usable by the project-building tool for Pleo.

The 3dsmax exporter will export motion data as comma-separated value (CSV) files. These files are fed to the Pleo project tool, which will write out Ugobe Motion Files (UMF).

Making Sounds

Pleo can currently play back only one sound at a time. The format of the audio should be 11k, 8-bit, mono. The data should be in a .WAV file for use by the Pleo project building tools.

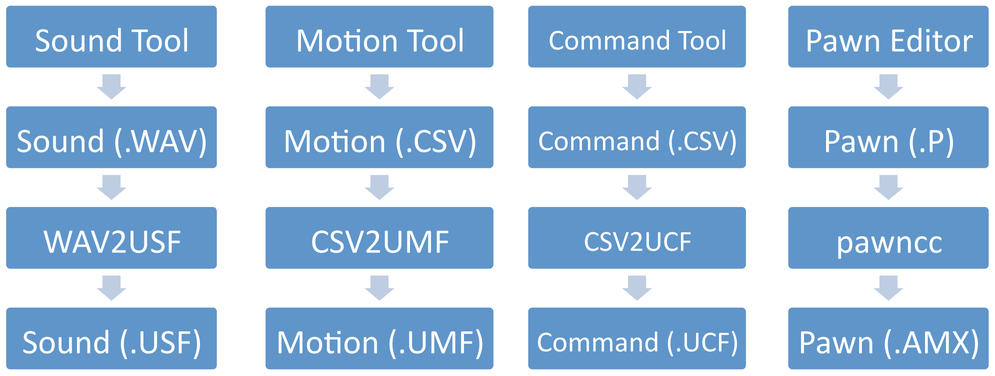
The inputs to the Pleo project tools are .WAV files. These files will be converted to Ugobe Sound Files (USF).

Making Commands

Making Scripts

Build Tools

The PDK includes the build tools necessary to convert resources from a common source format to a format that has been optimized to play back or execute within Pleo. The following graphic illustrates this:



As seen above, the raw resource assets get converted from a source format into an optimized binary format that is then playable on Pleo.

The first row above shows that there is a tool or editor for each raw resource type. For Sound, this may be SoundForge. For Pawn, this may simply be Notepad, vi or Quincy (the IDE that accompanies the Pawn distribution from CompuPhase).

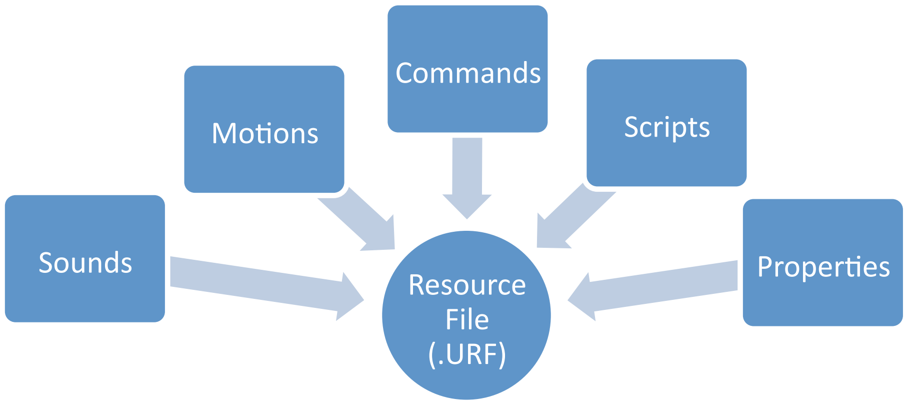
The second row shows what data format our tools expect as input. For Sound, that is WAV. For Motions and Commands, comma-separated-value text files. And for Pawn, Pawn source files.

The third row shows the individual build tool that handles that resource type. For Sounds, that is wav2usf[.py]. For Motions, csv2umf[.py]. For Commands, csv2ucf[.py]. And for Pawn, the (included) Pawn compiler. See the Source vs binary section for more details on what the .py signifies.

The last row shows the final output of the individual build tools. Each of these is an optimized binary format that can execute directly within Pleo.

Ugobe Project File

The Ugobe Project File (.UPF) is an XML-formatted file which lists all of the resources required for any project. The main build tool – ugobe_project_tool – parses this file, and calls out to each of the build tools for each type of resource. It then binds all of these compiled resources together into the final Ugobe Resource File (.URF).



Here is an example UPF file from the PDK:

```
<ugobe_project name="template">

 <options>
   <set name="top" value="../.." />
   <include value="./include:${top}/include" />
   <tools>
      <pawn value="pawncc %i -V2048 -O2 -S64 -v2 -C- %I TARGET=100 -o%o" />
    </tools>
    <directories>
      <build value="build" />
      <include value="include" />
    </directories>
    <umf value="3" />
    <folders />
  </options>

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

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

  <resources>

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

    <!-- Motions -->
    <motion path="${MOTIONS}/bow.csv" />

    <!-- Commands -->
<!--    <command path="${COMMANDS}/name.csv" /> -->

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

```
37     <script path="main.p" />
38
39     <!-- User Properties -->
40 <!--    <user_property path="my_prop" /> -->
41
42   </resources>
43 </ugobe_project>
```

At Line 1, we see the root ugobe_project element. The one attribute is name, which is used as the name of the output URF file.

Lines 3-15 give options to the build tools. These are described in more detail under each tool description.

Lines 17-22 define some path macros. This makes it easier to refer to where the resources are located and allows moving those resources and minimizing changes needed to the UPF file.

Lines 24-42 list all of the resources that make up this project. Each element defines the type of resource, where that resource lives, and optionally contains additional build instructions. More detail is given in each build tool description.

Built Tools: Source vs. Binary

When we originally build the tools that process each resource, we did so with the open source Python scripting language. We still use the source versions of these tools internally at Ugobe.

When we were putting together the PDK, we decided it would be more efficient to package these Python source tools as binaries for each platform. The easiest way to accomplish this is with other freely available tools. For Windows, we use the py2exe module. For OS X, we use the py2app module. And for Linux we use the cx_Freeze module.

Project layout

A typical project folder will look something like this:

```
/project
      /sounds
            sound1.wav
      /motions
            motion1.csv
            motion2.csv
      /commands
            command1.csv
      project.upf
      sensors.p
      main.p
```

Note that in the PDK, to save space, we refer to the global media folder, so many of the examples do not have a local sounds, motions or commands folder. But the build concepts remain the same.

When we build this project, we will typically use a command line like so:

```
python ..\..\bin\ugobe_project_tool project.upf rebuild
```

The first part is the 'master' build tool – in this case we are using the Python source version. This loads the given UPF file, and will call out to the individual build tools for each resource type.

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

*** Prepocessing ***
  Updating enumeration XML 'sounds.xml'
  Creating enumeration 'include/sounds.inc'
  Updating enumeration XML 'motions.xml'
  Creating enumeration 'include/motions.inc'
  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 ../../media/sounds/growl.wav to
temp/sounds/growl.usf adpcm:0 pitch
:1 freq:11025
  Converting ../../media/motions/bow.csv to
temp/motions/bow.umf, UMF3 format
done writing umf3 file; average vector length=15, count=98;
frames=102
  Building script 'sensors.p'
  execute 'sensors.p'
  @ pawncc sensors.p -V2048 -O2 -S64 -v2 -C-  -iinclude -
i..\..\include TARGET=1
00 -otemp/scripts/sensors.amx
Pawn compiler 3.3.3951M                Copyright (c) 1997-
2008, ITB CompuPhase

Header size:            192 bytes
Code size:              244 bytes
Max. overlay size:     2048 bytes; largest overlay=0 bytes
Data size:              128 bytes
```

```
Stack/heap size:        256 bytes; estimated max. use=43
cells (172 bytes)
Total requirements:    2624 bytes
  Building script 'main.p'
  execute 'main.p'
  @ pawncc main.p -V2048 -O2 -S64 -v2 -C-  -iinclude -
i..\..\include TARGET=100
-otemp/scripts/main.amx
Pawn compiler 3.3.3951M                 Copyright (c) 1997-
2008, ITB CompuPhase

main.p(30) : warning 225: unreachable code

Header size:            148 bytes
Code size:              204 bytes
Max. overlay size:     2048 bytes; largest overlay=96 bytes
Data size:              120 bytes
Stack/heap size:        256 bytes; estimated max. use:
unknown, due to "sleep" i
nstruction
Total requirements:    2572 bytes

1 Warning.

*** Writing build/template.urf ***
Version is 0
Build Time is 1216936757 (Thu Jul 24 17:59:17 2008)
writing temp/sounds/growl.usf (0x1000) at 0x200L
writing temp/motions/bow.umf (0x2000) at 0x4200L
writing temp/scripts/sensors.amx (0x4000) at 0x4800L
writing temp/scripts/main.amx (0x4001) at 0x4a34L
writing UGSF toc at 0x4e00L
writing UGMF toc at 0x4e30L
writing UGCF toc at 0x4e60L
writing  AMX toc at 0x4e68L
writing PROP toc at 0x4ec0L
  URF file fits: 20176 of 3649536. 3629376 free

Adler32 crc is F227783F
build time: 0.768000 sec
```

From this output, we see the following sections:

First, since we are issuing the ‘rebuild’ command, we clean up everything from previous builds. This includes the temporary folder with all built resources, the final build directory where the URF is put, and the include folder with the intermediate includes files for each resource type.

Next, we are generating the include files to use within the Pawn script. This consists of creating a list of each resource, grouped by type, and then assigning each an ID. This information is then written to an inc file. By default, these include files are placed in a local ‘include’ folder. This can be changed via the <options/directories/include> element in the UPF file. For example, this is the sounds.inc file generated from the PDK template project:

```
/************************************************\
 *                  WARNING                     *
 *     This file was automatically generated by    *
 * enumgen.py.  Do not edit directly or put under  *
 *               source control.                 *
\************************************************/

enum sound_name {
  snd_none  =     0,
  snd_min   =  4096,
  snd_growl =  4096,
  snd_max
};
```

Note that the names of the sounds used within Pawn are simply the original resource names, with a "snd_" pre-pended. For example, what started as growl.wav is named snd_growl, with an ID of 4096.

Note that each resource type gets its own range of IDs, so that they do not overlap. Currently, these ranges are:

Resource Type	Range in Hex	Range in Decimal
Sounds	*0x1000 - 0x1FFF*	*4096 - 8191*
Motions	*0x2000 - 0x2FFF*	*8192 - 12287*
Commands	*0x3000 - 0x3FFF*	*12288 - 16383*
Scripts	*0x4000 - 0x4FFF*	*16384 - 20479*
Properties	*0x5000 - 0x5FFF*	*20480 - 24575*

We use these hex ranges in many areas since it is easier to read, and gives us a wider range of values for each type.

Back to the build output: we now see each resource listed in the UPF being built by the individual tools. In this case, we are building the sound, the motion file, then the two script files. Additional options for these builders are specified in the <options> element of the UPF file.

Last, after all the individual resources have been built successfully, the build tool will combine all of these resources into one Ugobe Resource File (URF). By default, this will be put into a local 'build' directory. This can be changed via the <options/directories/build> element.

Sound processing

The tool used to convert input wave files into the Ugobe Sound File (USF) is wav2usf.py. In the binary tools release, the compiled version of this tool is built into the binary folder for a specific platform.

We use the built-in ‘audioop’ Python module to perform the actual data conversion from the source wave file into the Pleo-supported PCM or ADPCM format, at the specified bit-rate.

It is recommended that the input wave files be uncompressed, mono, 44kHz files. The output can be PCM or ADPCM, at various bit-rates – but we suggest keeping it to 11k or 22k. To specify the compression and data rate for all sound files in a project, use an <options/sound> element. For example:

```
<options>
    …
    <sound adpcm=”true” rate=”11025” />
</options>
```

You can also set these properties on individual files by adding these same attributes to a specific sound element in the resources element. For example:

```
<resources>
    …
    <sound path=”${SOUNDS}/growl.wav” adpcm=”false”
rate=”22050” />
    …
</resources>
```

This overrides any settings made in the <options/sound section>.

Motion processing

Motions are created at Ugobe using Autodesk 3ds Max Studio, and some custom export scripts to generate CSV file representations of Pleo motions (sometimes also referred to as animations). The freely available MySkit program can also be used to generate motions for Pleo.

The build tool used to process motion CSV files into Ugobe Motion Files (UMF) is called csv2umf.py. In the binary tools release, the compiled version of this tool is built into the binary folder for a specific platform.

The csv2umf tool takes the frame-based description of the motion, creates vectors for each joint that match this, and write it to a binary format (a .umf file). A motion is called out in the UPF like so:

```
<resources>
    …
    <motion path="${MOTIONS}/bow.csv" />
    …
</resources>
```

This example would result in a motion resource with the name of mot_bow and an ID in the range of 0x2000 – 0x2FFFF.

Additional processing options for motion files can be placed in the <options/motion> element. For example:

```
<options>
    …
    <motion version="3" />
    …
</options>
```

This example sets the UMF output version for all motions to version 3. This is not needed for Pleo 1.1, since this is the default – and only – version supported in Pleo.

Command processing

Commands are groups of sounds or motions, with associated property values or ranges, which are processed in the Pleo firmware. The firmware compares the properties in each command entry with the current property values in Pleo, and will choose a sound or motion out of the resulting set that matches the current state of Pleo.

Commands are typically built by hand, either in a spreadsheet program like Excel, or directly in a text editor like Notepad. They are in a comma separated value (CSV) format.

The build tool that processes Command CSV files is called csv2ucf.py. It takes as input the Command text file in CSV format and writes it to a Ugobe Command File (UCF). This UCF is then included in the final resource file (URF).

Commands are called out in a UPF with a line like this:

```
<resources>
    …
    <command path="${COMMANDS}/hunt.csv" />
    …
</resources>
```

Pawn script processing

Pawn scripts provide the logic behind any Pleo application. They can be authored in a program as simple as Notepad, or a complete environment like Quincy – the IDE that is included with the CompuPhase Pawn distribution.

The build tool that processes Pawn script is pawncc – the Pawn compiler. The version distributed by CompuPhase should work for build Pleo applications, if it is of a certain minimum version (3.3 at the time of this writing).

But in order to ensure good builds, we have included the Pawn compiler that we have built and tested at Ugobe, and that we know matches the Pawn run-time built into Pleo.

To specify a script in a UPF, use something like:

```
<resources>
    …
    <script path="${SCRIPTS}/main.p" />
    …
</resources>
```

Each script specified in the UPF will be built using the command line as specified in the <options/tools/pawn> element. For example:

```
<options>
  <tools>
    <pawn value="pawncc %i -V2048 -O2 -S64 -v2 -C- %I
TARGET=100 -o%o" />
  </tools>
</options>
```

Notice there is no path specified to the pawncc executable. This is why the Pawn compiler either needs to be installed somewhere on the PATH, or the use of the provided build.bat and/or build.sh scripts need to be used. In this way the UPF is completely cross-platform, since it used the underlying Python functions for path handling, which works across platforms.

For the example command line, we have the following options:

- %i: this is a place holder for the input .p filename
- -V2048: this turns on overlay generation, setting a maximum overlay size of 2048 bytes.
- -O2: this turns on full optimization, which includes opcode packing, reducing the code size by about 35% on average
- -S64: this sets the stack size, in cells
- -v2: this sets the verbosity level of the compiler output. Useful to see more data about the scripts being built
- -C-: this turns off compression of the code. Compression is NOT compatible with Pleo, since we cannot guarantee enough space for the code when it is compressed (it gets decompressed on load)
- %I: this is a placeholder for the include path set in the <options/include> element

- TARGET=100: this sets the target life form that this script is written for. Currently, we only have one life form. See pdk/include/default.inc for possible values
- -o%o: this sets the output, with the %o representing the output location of the resultant AMX file. By default this will be in ./temp/scripts/

Pleo Scripting

The mid-level layer contains four different virtual machines, or VMs (referred to as an Abstract Machine in the Pawn documentation). It is important to understand how each one is designed and should be used.

All VMs may contain the special functions *init* and *close*. The *init* function will be called when a script is first loaded into a VM, and the *close* function will be called just before a script is unloaded from a VM.

Sensor VM

The Sensor VM is used to handle all sensor activity. A special *on_sensor* function will be called when a sensor is 'triggered'. On startup, the Sensor VM will look for a script names *sensors.am*x.

A sensor is triggered when it changes its state or value to enough that is considered interesting. For example, a touch sensor changes state when touched, and also when released. In this case, there will be a trigger at the time the sensor is touched, and again when the sensor is released.

Another example might be the IR sensor, which will trigger when the IR module receives data from another Pleo or another device.

The *on_sensor* function has the following signature:

```
bool: on_sensor(time, sensor_name: sensor, value);
```

where:

- *time* is the time at which the sensor was triggered, given in milliseconds since Pleo was powered on
- *sensor* is the ID of the sensor. See sensor_name.
- *value* is the new value of this sensor after the trigger

If the *on_sensor* function returns *true*, then the sensor trigger will be reset. If the *on_sensor* function returns *false*, then the sensor trigger will not be reset, which means the *on_sensor* will be called again for this sensor trigger (though there may be other sensor triggers that occur first).

See the *Sensor* section for details on each sensor.

Main VM

The *main.amx* script is designed to be the controlling script for a Pleo application. It should contain a *main* function, which will be called after any *init* function. This function is designed to run forever. If the *main* function returns, the native code will simply call the *main* function again.

Behavior VM

The Behavior – or Aux - VM is designed to load and run script dynamically. The native functions *vm_exec* and *vm_exec_id* will load and execute scripts in this VM by default. These scripts should contain a *main* function that will be called after any *init* function has finished executing.

The *main* function in this script differs from the main script in that when the *main* function in the aux script finishes, it will not be called again. But it will set a property to indicate that the script has finished executing, allowing the main controlling script to call out another script if desired.

After the *main* function returns, the *close* function will be called.

If a script is loaded into the Aux VM, and a script is already running, the existing scripts *close* function will be called. When it completes, the new script will be loaded, its *init* function being called, if any. Then its *main* function will be called.

User VM

The User VM is used to execute the special *init.amx* script. This script is the first one looked for when Pleo starts or when an SD Card is inserted into Pleo.

The init.amx can be loaded while a Pleo application is already running, allowing this script to perform operations independent of the running Pleo Application.

> Note: Pleo 1.1 changed the name of the startup script from ‘init.amx’ to ‘startup.amx’.This allows the creation of one SD Card that can support the compiled Pawn format for both Pleo 1.0 (Pawn 3.2) and Pleo 1.1 (Pawn 3.3).

This script can also control the currently executing Pleo application; that is, it may unload the currently executing application or load a new application.

Since unloading applications can adversely effect the operation of Pleo, only scripts signed by Ugobe can perform this operation.

Scripting Limits

The Pleo firmware has a very limited amount of memory, so it is important that scripts use only the minimum amount possible. This sections details how memory is allocated and used for Pawn scripting and some techniques that allow scripters to identify issues before they even run their applications.

There are two memory components when dealing with Pawn scripts: code and data. Code is the generated Pawn opcodes from the Pawn .p source files. Data consists of the variables of a program, and also the stack for that script.

The Pawn compiler compiles a Pawn .p source file and it generates an .AMX file, which can then be loaded into the Pleo firmware in one of the four available VMs. The layout of an AMX file is like so:

- Header:
- Code:
- Data:

The Pleo firmware reserves a section of memory called a code pool. It is currently 8K in size. This pool is shared among all of the Pawn VMs in the firmware.

Each VM has reserved a block of memory for any scripts data and stack. The current limits are:

- Sensor:
- Main:
- Behavior:
- User:

If any attempt to load a script that would not fit in these limits, the load will fail. See the log output with the log type ‘vm’ enabled.

In order to catch these kinds of errors early, there are mechanisms in the Pawn compiler to check for these limits. This includes either pre-processor directives or command-line options. We will detail both.

Pleo API

Every script loaded into any of the VMs has access to a set of functions exposed by the Pleo firmware. All of these functions are defined in the include files that are part of the PDK.

In the include folder, you will find (at least) the following includes:

Animation.inc: interface to control Animation System, or Motion Commands

Application.inc: interface to control application loading and unloading

Joint.inc: interface to control individual joints

Log.inc: interface to logging functions

Motion.inc: interface to control motion playback

Property.inc: interface to Property System, or Blackboard

Resource.inc: interface to the Resource Manager

Script.inc: interface to script VMs

Sensor.inc: interface to sensor system

Sound.inc: interface to control sound playback

String.inc: interface to string functions

Time.inc: interface to time functions

Util.inc: interface to misc. utility functions like rand.

Refer to the Pleo Reference Guide for more include file specifics.

In the following sections, the ${sdk} mnemonic will indicate the location where the PDK has been installed.

Joint System

The joint API allows script to move joints, get joint positions, status, etc. There are multiple coordinate systems in the Pleo motion control system. At the highest level, we use angles in degrees, and at the lowest level we use voltage (VR) values. The script can use either system, but in most cases will use the default degree system.

Here is the list of joint IDs, from ${sdk}/include/pleo/joints.h:

```
enum joint_name {
  JOINT_RIGHT_SHOULDER
  JOINT_RIGHT_ELBOW
  JOINT_LEFT_SHOULDER
  JOINT_LEFT_ELBOW
  JOINT_LEFT_HIP
  JOINT_LEFT_KNEE
  JOINT_RIGHT_HIP
  JOINT_RIGHT_KNEE
  JOINT_TORSO
  JOINT_TAIL_HORIZONTAL
  JOINT_TAIL_VERTICAL
  JOINT_NECK_HORIZONTAL
  JOINT_NECK_VERTICAL
  JOINT_HEAD
};
```

Each joint has a range of possible angle values, which can be retrieved using the following functions:

```
joint_get_min(joint_name: joint, angle_type: type);
joint_get_neutral(joint_name: joint, angle_type: type);
joint_get_max(joint_name: joint, angle_type: type);
```

The type parameter of type enumeration angle_type should be set to angle_degrees.

The main joint movement function is:

```
joint_move_to(joint_name: joint, value, speed, angle_type:
type);
```

The speed parameter for joint_move_to is in angle_type units per second.

Joint information functions include:

```
joint_get_attribute(joint_name: joint, joint_attribute:
type);
joint_get_position(joint_name: joint, angle_type: type);
bool: joint_is_moving(joint_name: joint);
```

The attributes of a given joint that can be retrieved include:

```
enum joint_attribute {
    ja_none,        // N/A
    ja_status,      // RO
    ja_position,    // RO
    ja_pwm,         // RO
    ja_load,        // RO
    ja_speed,       // RO
    ja_time,        // RO
    ja_setpoint,    // RO
    ja_deadband,    // RW
    ja_threshold,   // RW
};
```

Finally, control over whether script or motion files control a given joint can be specified using this:

```
joint_control(joint_name: name, who);
```

where who should be set to 0 so motion files control the joint, or 1 so that scripts can control the joint (for example, by using the joint_move_to() function).

Motion System

Rather than only control individual joint movements through script, which can be powerful but tedious to implement for complex animations, LifeOS also provides a motion file playback system.

Motion files can be generated with a variety of tools (TBD). A motion file is a binary file conforming to the Ugobe Motion File format (UMF), which consists of a header followed by one or more vectors. Each vector consists of a joint name, a desired destination angle, and a desired time at which that angle should be reached.

Motion files can be started playing using:

```
Motion: motion_play(motion_name: name);
Motion: motion_play_file(const file_name[]);
```

where name is the name of a motion file specified in your project's Ugobe Project File (UPF). The project build system generates an enumerated value for each listed motion file with "mot_" prefixed to it, and stores them in motions.inc. For example:

```
/***************************************************\
 *                    WARNING                      *
 *     This file was automatically generated by    *
 * enumgen.py.  Do not edit directly or put under  *
 *                source control.                  *
\***************************************************/
enum motion_name {
      mot_none       = 0,
      mot_min        = 8192,
      mot_my_motion  = 8192,
      mot_max
};
```

Alternatively, the motion_play_file() function can be passed the name of a file on the SD card, and would thus not need to be listed in your UPF. Note that you need to specify the base of the filename, excluding the '.umf' extension.

Both of these functions return a handle to the motion file, which can be used in some of the functions described below.

There is also a special handle value:

```
Motion_Any = 0xAA
```

This is useful when calling the motion_stop() function, to stop all motions.

You can take control over a currently playing motion file using these functions:

```
motion_pause({Motion, motion_name}: motion, bool: pause);
motion_stop({Motion, motion_name}: motion);
motion_set_playback_speed(percent_of_normal);
```

The functions motion_pause() and motion_stop() can be passed either the handle returned by motion_play() or motion_play_file(), or can be passed the symbolic name of the motion defined in motions.inc.

The function motion_set_playback_speed() will adjust the actual playback speed, normally 100 (percent). This allows you to make Pleo look sluggish or peppy, for example. Note that this setting will remain in effect until Pleo is reset or power cycled, so be sure to set it back to normal when appropriate.

You can query the state of the motion system with these functions:

```
bool: motion_is_playing({Motion, motion_name}: motion);
motion_get_playback_speed();
```

The function motion_is_playing() can be passed a handle or a motion name, from which it will return true if the specified motion file is playing. Alternatively, if passed Motion_Any, it will return true if any motion file is playing, regardless of what file is playing

Property System

The property system serves three main functions:

- communication between LifeOS and Pawn
- communication between the different Pawn VMs
- persistence across power cycles

Properties used for communication between LifeOS and Pawn are defined by LifeOS and called system properties. Pawn developers can also define properties for use in their scripts, called user properties. The user properties need to be declared in the Ugobe Project File (upf), while system properties are defined in the properties.inc include file. Most system properties should be considered read-only, although this is not enforced. Changing the values of system properties in script can have unpredictable results.

A property consists of a name and a value. The name is stored as an enumeration value and also referred to as the property id. The value of a property can range from

-0x3FFFFFFF to 0x3FFFFFFF (-1,073,741,823 to 1,073,741,823).

A total of 224 properties can be defined in LifeOS 1.1

Currently, the following system properties are defined:

```
  property_none: name used to specify no property
        property_min: the number of the lowest defined
property                            (useful for loops)
  property_arousal: not used by LifeOS
  property_valence: not used by LifeOS
  property_stance: not used by LifeOS
  property_age: not used by LifeOS
  property_damage: not used by LifeOS
  property_energy: not used by LifeOS
  property_health: not used by LifeOS
  property_mood: not used by LifeOS
  property_command: currently active command
  property_command_status: current command status
            (per command_status_enum)
  property_layer: current layer of the current animation
  property_probability: not used by LifeOS
  property_motion: currently active motion
  property_command_pending: currently pending command (if
any)
  property_behavior_status: current behavior vm status
            (per command_status_enum)
  property_script_status: alias for
property_behavior_status
  property_fatigue: not used by LifeOS
  property_direction: not used by LifeOS
  property_pose: not used by LifeOS
  property_sequence: not used by LifeOS
  property_platform: the platform LifeOS is executing on.
Pleo=4
  property_behavior: the currently active behavior
  property_script: alias for property_behavior
  property_speed: not used by LifeOS
  property_neutral: not used by LifeOS
  property_pickedup: not used by LifeOS
  property_stand: not used by LifeOS
  property_liedown: not used by LifeOS
  property_trick_step: not used by LifeOS
  property_trick_id: not used by LifeOS
  property_sound: currently active sound
  property_drive: currently active drive
  property_motion_time: time left for this motion (in ms)
  property_attn_track_weight: minimum size of an object to
be tracked
  property_attn_hold_flags: bitmask representing which
touch sensors
            are held
  property_cam_img_progress: current status of the camera
system
  property_attn_track_mindist: minimum distance an object
must move
            before a tracking event is generated
  property_attn_track_move: allow LifeOS to move the neck
for
            tracking (0 or 1)
  property_attn_p2p_timeout: time after which pleo
considers
            a p2p conversation abandoned
  property_max: one higher than the last system property
            (useful for loops)
  property_limit: maximum value of a property name (65535)
```

The value of a property can be set and retrieved by the native functions get_property and set_property respectively. For brevity, set and get aliases have been defined.

Persistence

The property system provides persistence by allowing properties to be saved to a file on either dataflash or SD card to be loaded at a later time. There is no option to selectively save or load properties; all properties are saved and all existing properties are replaced on load.

Leaky Integrators

The property system provides a mechanism to have property values change over time without having to explicitly update their values, by using Leaky Integrators. A leaky integrator can be specified with the native function property_set_leak. A leaky integrator has a delta and an interval allowing flexibility in how much a property will increment or decrement as well as how often this happens. A maximum and minimum can also be set. 32 Leaky integrators can be defined in LifeOS 1.1

Reporters

To be notified when a property crosses a threshold or changes by a certain amount, the property system provides the Reporter mechanism. This is primarily useful if you wish to receive notification when system properties are updated, or when a property modified by a leaky integrator reaches a specific value. LifeOS notifies the sensor VM of these events by calling into on_property, if defined. Reporters are set by calling the add_reporter native. LifeOS 1.1 supports up to four reporters.

Drive system

Like every life form, Pleo's actions are controlled by a number of competing drives. These drives control Pleo's behavior at the highest level; examples are hunger, fatigue and being social, but the developer can define arbitrary drives making Pleo into a thrill seeker or always in need of light, etc.

Every drive has a value, and the drive with the highest value is the one that wins and becomes active, meaning it is now in control of Pleo's actions.

Adding Drives

Adding drives happens from script using the drive_add function, which takes two trigger values aside from the drive name and the evaluation interval.

The trigger values dictate when pleo will actively try to work to:

- bring the drive evaluation down (eval > high trigger),
- maintain the current level (low trigger < eval < high trigger)
- bring the drive evaluation up (eval < low trigger)

'eval' is the value returned from the Drive Evaluators. Every Drive must have a drive evaluator defined in the Main VM, by the name of <DriveName>_eval(). If a drive is added to control Pleo's nutritional needs called 'hunger', an evaluator called hunger_eval must exist. A drive evaluator can be very simple and return the value of a single variable, or can be a complex function taking into account many factors regarding Pleo's internal state as well as the environment. Calling drive_set_value can also explicitly set drive evaluation values.

When a drive is first added, a call into its init function is made. For the example of the hunger drive, this would be hunger_init(). When the drive system is unloaded, hunger_exit() is called for any cleanup. When the active drive is switched from one drive to another, the respective deactivate and activate functions are called. For our 'hunger' drive example these functions would be called hunger_activate() and hunger_deactivate().

Behaviors

Once a drive is the winner and in control of the system, it can exercise this control by picking a new behavior. A behavior is a separate script running in the behavior VM. Every drive is associated with one or more behaviors. Behaviors are added by calling the behavior_add function. Like drives, behaviors have evaluation functions. Because behaviors have IDs rather than names, all behaviors for a drive have the same evaluator function. For the hunger example, this would be hunger_behavior_eval(), which passes the behavior ID for the behavior to be evaluated. Like drives, the behavior evaluation value can also be explicitly set from any part of the system by calling behavior_set_value.

Example:

```
public hunger_behavior_eval(behavior_id)
{
            switch(behavior_id)
            {
                  case scr_eat_grass:
                  {
                        if
(get(property_grass_available) == 1)
                        {
                              return 50;
                        }
                        return -50;
                  }
                  default:
                  {
                        return 0;
                  }
            }
}
```

which will return 50 for the eat_grass behavior if there is grass, but -50 if there is no grass. For any other behavior evaluation it will return 0, such that they will lose from eat_grass when grass is around, but win from eat_grass when there is no grass. This function can easily be extended for more complicated circumstances for the other hunger-related behaviors (look for food, or beg for it), as well as additional complexity for eat_grass (whether or not Pleo has eaten anything but grass in the last 3 days may affect his desire to eat more of it, for example).

The eat_grass behavior script can now be written and added to the Ugobe Project File:

```
<resources>

     ...
           <script path="${SCRIPTS}/eat_grass.p" />
</resources>
```

Behavior scripts are very similar to the main.p script, in that they have three public funtions: init(), main() and close(). Unlike the main.p script, the main function of a behavior script is called only once. To prevent Pleo from having nothing to do, it is recommended to have a main() function that does not terminate by itself. When LifeOS activates a new behavior, the current one is interrupted, close() is called and the next behavior is loaded.

Sensors System

Each sensor in Pleo has a unique identifier, rules about when it is triggered, and different ranges of values that it may have. In this section we detail each sensor.

There are two types of sensors in Pleo: ‘raw’ and ‘derived’ sensors. Raw sensors are those sensors that are associated directly with a hardware sensor. For example, battery level, touch sensors, etc. Derived sensors are ‘virtual’ sensors, which use a combination of the raw sensors and other information. These include picked up, abused, etc.

Raw sensors

The following is a list of all the sensors in Pleo, taken from the ${pdk}/include/pleo/sensors.inc include file:

```
enum sensor_name {
  SENSOR_BATTERY
  SENSOR_IR
  SENSOR_HEAD
  SENSOR_CHIN
  SENSOR_BACK
  SENSOR_LEFT_LEG
  SENSOR_RIGHT_LEG
  SENSOR_LEFT_ARM
  SENSOR_RIGHT_ARM
  SENSOR_TAIL
  SENSOR_FRONT_LEFT
  SENSOR_FRONT_RIGHT
  SENSOR_BACK_LEFT
  SENSOR_BACK_RIGHT
  SENSOR_CARD_DETECT
  SENSOR_WRITE_PROTECT
  SENSOR_LIGHT
  SENSOR_OBJECT
  SENSOR_MOUTH
  SENSOR_SOUND_DIR
  SENSOR_LIGHT_CHANGE
  SENSOR_SOUND_LOUD
  SENSOR_TILT
  SENSOR_TERMINAL
  SENSOR_USB_DETECT
  SENSOR_WAKEUP
  SENSOR_BATTERY_TEMP
  SENSOR_SHAKE
  SENSOR_SOUND_LOUD_CHANGE
  SENSOR_BEACON
  SENSOR_BATTERY_CURRENT
  SENSOR_PACKET
  SENSOR_EDGE_IN_FRONT
  SENSOR_EDGE_ON_LEFT
  SENSOR_EDGE_ON_RIGHT
  SENSOR_OBJECT_IN_FRONT
  SENSOR_OBJECT_ON_LEFT
  SENSOR_OBJECT_ON_RIGHT
  SENSOR_TOUCH_TAP
  SENSOR_TOUCH_HOLD
  SENSOR_TOUCH_RELEASE
  SENSOR_TOUCH_PETTED
  SENSOR_ABUSE
  SENSOR_PICKED_UP
  SENSOR_TRACKABLE_OBJECT
  SENSOR_EDGE
  SENSOR_TOUCH_TAP_HOLD
  SENSOR_JOINT_STUCK
  SENSOR_JOINT_UNSTUCK
  SENSOR_TIMER
  SENSOR_MSG_RECEIVED
  SENSOR_MSG_GONE
  SENSOR_PLOG
};
```

Sensor values can be read with the following functions:

```
sensor_get_value(sensor_name: sensor);
```

for scalar values, or:

```
sensor_read_data(sensor_name: sensor, data[], length =
sizeof data);
```

to read an array of data for those sensors that deal with data arrays (IR, camera, terminal, or sound for example).

SENSOR_BATTERY

Sensor ID: 2

Value range: 0 – 100

Trigger condition: incremental change (by 25)

Description: The SENSOR_BATTERY is used to read the current level of the battery. It is normalized to a value range of 0-100, where 100 is full-charge, 50 is half-charge and 0 is no charge.

There are two critical battery levels that have special meaning:

- The first special battery level indicates when the battery is low enough to prohibit motions that take a lot of power from playing.
- The second special battery level is when Pleo ought to go to sleep because he cannot play any motions. This is when Pleo should enter the sleep pose, turn off unneeded peripherals – including motors – and wait to be charged.
    - Pleo's firmware will automatically shutdown when the SENSOR_BATTERY value goes below 11. If you are running Pleo's default personality, it will animate Pleo to a sleeping position and shutdown when the value goes below 23.

SENSOR_IR

Sensor ID: 3

Value range: 0 – 64

Trigger condition: any received data

Description: The SENSOR_IR is used to indicate reception of valid NEC-format IR data that does not match the beacon or object detection codes; this can be from the use of the general purpose IR data mechanism in Pleo, or from the use of an NEC-format remote control.

The value of the sensor is the number of lines of data in the IR receive buffer, where a line is defined to be a sequence of one or more characters delimited by an ASCII linefeed or NULL character. If the IR data comes from an NEC-format remote control, the line will be the 4 digit hex string representing the key pressed.

SENSOR_HEAD

Sensor ID: 6

Value range: 0, 1

Trigger condition: change to state of touch sensor (touched or released)

Description: The SENSOR_HEAD reflects the state of the touch sensor under the skin of Pleo's head.

SENSOR_CHIN

Sensor ID: 7

Value range: 0, 1

Trigger condition: change to state of touch sensor (touched or released)

Description: The SENSOR_CHIN reflects the state of the touch sensor under the skin on Pleo's chin.

SENSOR_BACK

Sensor ID: 8

Value range: 0, 1

Trigger condition: change to state of touch sensor (touched or released)

Description: The SENSOR_BACK reflects the state of the touch sensor under the skin of Pleo's back, near the neck.

SENSOR_LEFT_LEG

Sensor ID: 9

Value range: 0, 1

Trigger condition: change to state of touch sensor (touched or released)

Description: The SENSOR_LEFT_LEG reflects the state of the touch sensor under the skin on Pleo's left leg.

SENSOR_RIGHT_LEG

Sensor ID: 10

Value range: 0, 1

Trigger condition: change to state of touch sensor (touched or released)

Description: The SENSOR_RIGHT_LEG reflects the state of the touch sensor under the skin of Pleo's right leg.

SENSOR_LEFT_ARM

Sensor ID: 11

Value range: 0, 1

Trigger condition: change to state of touch sensor (touched or released)

Description: The SENSOR_LEFT_ARM reflects the state of the touch sensor under the skin of Pleo's left arm.

SENSOR_RIGHT_ARM

Sensor ID: 12

Value range: 0, 1

Trigger condition: change to state of touch sensor (touched or released)

Description: The SENSOR_RIGHT_ARM reflects the state of the touch sensor under the skin of Pleo's right arm.

SENSOR_TAIL

Sensor ID: 13

Value range: 0, 1

Trigger condition: change to state of touch sensor (touched or released)

Description: The SENSOR_TAIL reflects the state of the touch sensor under the skin near Pleo's tail, on it's torso, not actually on it's tail as the name implies.

SENSOR_FRONT_LEFT

Sensor ID: 14

Value range: 0, 1

Trigger condition: change to state of foot switch (touching surface / not touching)

Description: The SENSOR_FRONT_LEFT reflects the state of the foot switch on the bottom of Pleo's front left foot.

SENSOR_FRONT_RIGHT

Sensor ID: 15

Value range: 0, 1

Trigger condition: change to state of foot switch (touching surface / not touching)

Description: The SENSOR_FRONT_RIGHT reflects the state of the foot switch on the bottom of Pleo's front right foot.

SENSOR_BACK_LEFT

Sensor ID: 16

Value range: 0, 1

Trigger condition: change to state of foot switch (touching surface / not touching)

Description: The SENSOR_BACK_LEFT reflects the state of the foot switch on the bottom of Pleo's back left foot.

SENSOR_BACK_RIGHT

Sensor ID: 17

Value range: 0, 1

Trigger condition: change to state of foot switch (touching surface / not touching)

Description: The SENSOR_BACK_RIGHT reflects the state of the foot switch on the bottom of Pleo's back right foot.

SENSOR_CARD_DETECT

Sensor ID: 18

Value range: 0, 1

Trigger condition: change to state of SD card insertion – inserted = 1, removed = 0

Description: The SENSOR_CARD_DETECT indicates the state of the SD card slot; 1 = a card is inserted, 0 = no card is present.

SENSOR_WRITE_PROTECT

Sensor ID: 19

Value range: 0, 1

Trigger condition: change to state of write protect (which occurs on card insertion / removal)

Description: The SENSOR_WRITE_PROTECT reflects the state of the write protect switch on the inserted SD card.

SENSOR_LIGHT

Sensor ID: 21

Value range: 0-255

Trigger condition: Triggers when the value crosses, in either direction, values of 30 and 150.

Description: The SENSOR_LIGHT indicates the current absolute ambient light level seen by Pleo's camera module in its snout. Lower values indicate a darker environment.

SENSOR_OBJECT

Sensor ID: 23

Value range: 0-100

Trigger condition: Triggers when the value crosses, in either direction, the values of 10 and 40.

Description: The SENSOR_OBJECT value indicates the confidence that an object is in front of Pleo. A value of 100 indicates there is definitely something there; a value of 0 indicates that nothing is there. The trigger occurs when it is likely, in normal environments, that either an object has just appeared in front of Pleo or has just disappeared. Note that the value is not a measurement of distance, as it depends to a significant amount on the reflectivity and texture of the object in question, as well as, to a lesser degree, the amount of ambient IR light signals.

SENSOR_MOUTH

Sensor ID: 24

Value range: 0, 1

Trigger condition: change to state of the mouth IR interrupter.

Description: The SENSOR_MOUTH indicates the presence or absence of an IR-opaque object in Pleo's mouth: for example, Pleo's leaf.

SENSOR_SOUND_DIR

Sensor ID: 26

Value range: -90 to +90, or -128

Trigger condition: triggers when change of direction of sound is detected

Description: The SENSOR_SOUND_DIR triggers when an unambiguously measurable sound source is detected; the direction in degrees relative to Pleo is returned as the value (-90 is to the left, 0 is straight ahead, +90 is to the right). If loud sound is detected but its direction cannot be determined, the value will change to -128.

Pleo's sound direction sensor works best with relatively long sounds, such as spoken words, rather than claps or finger snaps.

SENSOR_LIGHT_CHANGE

Sensor ID: 27

Value range: -127 to +127

Trigger condition: triggers when the value changes to being greater than +30 (it has become brighter) or less than -30 (it has become darker)

Description: The SENSOR_LIGHT_CHANGE detects relative changes to the current ambient light level.

SENSOR_SOUND_LOUD

Sensor ID: 28

Value range: 0 to 100

Trigger condition: triggers when loudness goes from < 30 to > 40, or > 40 to < 30

Description: The SENSOR_SOUND_LOUD detects the edges of loud sounds that stand out from the background sound level.

SENSOR_TILT

Sensor ID: 29

Value range: 0 to 6 (see enum tilt_name in sensors.inc)

Trigger condition: triggers when the tilt sensor transitions to a new position

Description: The SENSOR_TILT detects the orientation of Pleo's torso in 3-space (independent of the angular position of Pleo's neck). Values returned are:

- TILT_NONE = 0 – no orientation known
- TILT_ON_FEET = 1 – feet are oriented downwards with respect to torso; does not mean that the foot switches are depressed – Pleo could be being held up in the air
- TILT_LEFT_SIDE = 2 – on left side
- TILT_RIGHT_SIDE = 3 – on right side
- TILT_ON_NOSE = 4 – front of torso is pointed downwards
- TILT_ON_TAIL = 5 – aft-end of torso is pointed downwards
- TILT_ON_BACK = 6 – feet are pointed upwards with respect to torso

SENSOR_TERMINAL

Sensor ID: 30

Value range: 0 to 64

Trigger condition: triggers when the user types one or more character on the monitor interface, including just a carriage return

Description: The SENSOR_TERMINAL triggers when a line has been entered; the value of the sensor is the number of characters received. Use the sensor_read_data() function to retrieve the characters in your script.

SENSOR_USB_DETECT

Sensor ID: 32

Value range: 0, 1

Trigger condition: triggers when a valid USB connection to the host is established or when a connection is removed

Description: The SENSOR_USB_DETECT reflects the state of the USB system.

SENSOR_WAKEUP

Sensor ID: 33

Value range: 0, 1

Trigger condition: triggers when the user depresses or releases the wakeup button, located near the SD card slot and USB connector on Pleo's belly

Description: The SENSOR_WAKEUP can be used in script for any purpose. When running Pleo's default personality, the wakeup button is used to adjust Pleo's sound volume.

SENSOR_BATTERY_TEMP

Sensor ID: 34

Value range: 0 to 100

Trigger condition: triggers when the value, which is Pleo's battery temperature in degrees C, crosses the values of 51 and 55

Description: The SENSOR_BATTERY_TEMP is used to monitor the state of Pleo's battery; very heavy use of animation, especially when the ambient room temperature is high, can result in overheating of the battery; Pleo's firmware will automatically shutdown the system when the battery temperature reaches 58 C. You may choose to reduce activity when the SENSOR_BATTERY_TEMP triggers, in order to avoid further heating that may lead to such a shutdown.

SENSOR_SHAKE

Sensor ID: 36

Value range: 0 to 255

Trigger condition: triggers when the shake frequency goes from below 75 to above 150 or vice versa.

Description: The SENSOR_SHAKE is used to detect Pleo being shaken, for example, while sleeping.

SENSOR_SOUND_LOUD_CHANGE

Sensor ID: 37

Value range: -127 to +127

Trigger condition: triggers when a large dynamic change in ambient noise level occurs

Description: The SENSOR_SOUND_LOUD_CHANGE is used to detect large, interesting changes in the ambient noise level. The long-term average noise level is monitored. This sensor's value is the difference between the short-term noise level and the long term.

SENSOR_BEACON

Sensor ID: 38

Value range: 0 to 255

Trigger condition: triggers when a beacon is received from another Pleo

Description: The SENSOR_BEACON occurs when one Pleo receives a valid beacon from another Pleo; the value of the sensor is the lower 8 bits of that Pleo's electronic serial number, which the beacon system uses as a semi-unique identifier.

You can read the one byte unique identifier plus 3 bytes of user-defined payload using the sensor_read_data() function. Note that if you set the length parameter of the sensor_read_data() function to 1 rather than 4, the data byte you read back will be this Pleo's own id rather than the other Pleo's.

You can write (modify) the outgoing beacon payload using the sensor_write_data() function. By setting the length parameter to 0, you can turn off the beacon system. Otherwise, the up to 3 data bytes you pass in will be broadcast approximately every 2 seconds from this Pleo over IR using the NEC consumer remote control protocol.

SENSOR_BATTERY_CURRENT

Sensor ID: 39

Value range: 0 to 5000

Trigger condition: none

Description: The SENSOR_BATTERY_CURRENT returns the number of milliamps being consumed from the battery by all electronics, including motors, in Pleo.

SENSOR_PACKET

Sensor ID: 40

Value range: 0 to 400

Trigger condition: triggers when a matching packet (according to the current packet filter criteria) is received from the NXP head processor by the main ARM processor

Description: The SENSOR_PACKET gives script programmers direct access to data received from the head processor. See the Packet System section for more details.

SENSOR_EDGE_IN_FRONT

Sensor ID: 101

Value range: 0, 1

Trigger condition: triggers when edge detected or lost

Description: The SENSOR_EDGE_IN_FRONT outputs 1 when an edge (of a table, for example) is detected in front of Pleo; it outputs 0 when the edge disappears. Pleo's firmware does this automatically whenever the object detect sensor is enabled (which it is by default), and the neck is straight and pointing downwards at a predefined angle. The angle limits can be changed using set_object_edge_params().

SENSOR_EDGE_ON_LEFT

Sensor ID: 102

Value range: 0, 1

Trigger condition: triggers when edge detected or lost

Description: The SENSOR_EDGE_ON_LEFT works in a similar manner to SENSOR_EDGE_IN_FRONT, except that it occurs when the neck is turned to Pleo's left.

SENSOR_EDGE_ON_RIGHT

Sensor ID: 103

Value range: 0, 1

Trigger condition: triggers when an edge is detected or lost

Description: The SENSOR_EDGE_ON_RIGHT works in a similar manner to SENSOR_EDGE_IN_FRONT, except that it occurs when the neck is turned to Pleo's right.

SENSOR_OBJECT_IN_FRONT

Sensor ID: 104

Value range: 0, 1

Trigger condition: triggers when an object is detected or lost

Description: The SENSOR_OBJECT_IN_FRONT works in a similar manner to SENSOR_EDGE_IN_FRONT, except that it only occurs when the neck is straight and pointed above a predefined angle; this use of neck vertical angle automatically distinguishes between flat walking surfaces and obstacles or walls.

SENSOR_OBJECT_ON_LEFT

Sensor ID: 105

Value range: 0, 1

Trigger condition: triggers when an object is detected or lost

Description: The SENSOR_OBJECT_ON_LEFT is similar to SENSOR_OBJECT_IN_FRONT except that it only occurs when the neck is pointed to Pleo's left side by a predefined angle.

SENSOR_OBJECT_ON_RIGHT

Sensor ID: 106

Value range: 0, 1

Trigger condition: triggers when an object is detected or lost

Description: The SENSOR_OBJECT_ON_RIGHT is similar to SENSOR_OBJECT_IN_FRONT except that it only occurs when the neck is pointed to Pleo's right side by a predefined angle.

SENSOR_TOUCH_TAP

Sensor ID: 107

Value range: sensor_id of the touch sensor involved

Trigger condition: triggers when touch sensor has been touched and released

Description: The SENSOR_TOUCH_TAP triggers when a touch sensor has been touched and released after a short period of time; the value of the sensor returned is the ID of the touch sensor that was tapped.

SENSOR_TOUCH_HOLD

Sensor ID: 108

Value range: sensor_id of the touch sensor involved

Trigger condition: triggers when held for a sufficient period of time

Description: The SENSOR_TOUCH_HOLD is similar to SENSOR_TOUCH_TAP, except that it distinguishes for you the difference between a short tap vs. a longer hold.

SENSOR_TOUCH_RELEASE

Sensor ID: 109

Value range: sensor_id of the touch sensor involved

Trigger condition: triggers when touch sensor, which was being held, has been released

Description: The SENSOR_TOUCH_RELEASE follows a corresponding SENSOR_TOUCH_HOLD.

SENSOR_TOUCH_PETTED

Sensor ID: 110

Value range: enum petted_name

Trigger condition: triggers when petting has been detected

Description: The SENSOR_TOUCH_PETTED detects when a series of one or more touch sensors are triggered, indicating that the user is petting Pleo. The value of the sensor indicates in what manner the pet is occurring, or that it has stopped.

```
enum petted_name  {
   PETTED_NONE             =  0,
   PETTED_MIN              =  1,
   PETTED_STOPPED          =  1,
   PETTED_BACKARSE         =  2,
   PETTED_ARSEBACK         =  3,
   PETTED_HEADBACK         =  4,
   PETTED_BACKHEAD         =  5,
   PETTED_HEADBACKARSE     =  6,
   PETTED_ARSEBACKHEAD     =  7,
   PETTED_MAX
};
```

SENSOR_ABUSE

Sensor ID: 111

Value range: 0, 1

Trigger condition: triggers when abuse starts or concludes

Description: The SENSOR_ABUSE detects when Pleo's joints are being moved by some external force (e.g., the user has grabbed Pleo by a leg and is moving it back and forth).

SENSOR_PICKED_UP

Sensor ID: 112

Value range: 0, 1

Trigger condition: triggers when Pleo is either picked up or put down

Description: The SENSOR_PICKED_UP sensor detects when Pleo is set down on or lifted up from a surface.

SENSOR_TRACKABLE_OBJECT

Sensor ID: 113

Value range: 0-9

Trigger condition: triggers when a matching object is found or lost

Description: The SENSOR_TRACKABLE_OBJECT detects the location of an object that matches the current colormap, and also detects when such an object goes away. If the value is 0, the object is gone; if the value is 1-9, it indicates the direction in Pleo's field of view that the object is detected; the numbering system is oriented like a keypad.

SENSOR_EDGE

Sensor ID: 119

Value range: none

Trigger condition: none

Description: The SENSOR_EDGE is a bundle sensor used to enable/disable all sensors related to edge detection.

SENSOR_TOUCH_TAP_HOLD

Sensor ID: 120

Value range: none

Trigger condition: none

Description: The SENSOR_TOUCH_TAP_HOLD is a bundle sensor used to enable/disable all touch-related sensors.

SENSOR_TIMER

Sensor ID: 124

Value range: 0..65535

Trigger condition: periodically triggers when timer expires

Description: The SENSOR_TIMER triggers every time the timer expires; the value is the amount of time that has passed since the last trigger.   The native function set_timer_interval() can be used to set the number of milliseconds for the trigger period.

SENSOR_MSG_RECEIVED

Sensor ID: 125

Value range: 0..255

Trigger condition: triggers when a new Pleo's beacon has been received

Description: The SENSOR_MSG_RECEIVED is triggered when a new Pleo has been detected by means of its IR beacon. The value returned is the 8 bit Pleo ID of the Pleo that transmitted the beacon. See also SENSOR_BEACON and SENSOR_MSG_GONE.

SENSOR_MSG_GONE

Sensor ID: 126

Value range: 0..255

Trigger condition: triggers when a timeout occurs

Description: The SENSOR_MSG_GONE triggers when contact has been lost with a Pleo that was previously detected. The value is the 8 bit Pleo ID of the lost Pleo. See also SENSOR_MSG_RECEIVED and SENSOR_BEACON.

SENSOR_PLOG

Sensor ID: 128

Value range: none

Trigger condition: none

Description: The SENSOR_PLOG is a virtual sensor used to enable or disable the Pleo Log (plog).

Sound System

LifeOS provides the ability to play back sounds in a variety of ways:

Referenced from a motion file

Referenced in a sound command file

Directly from script

Just as in the Motion System, many Sound System functions return or accept as parameters a Sound file handle, or a special value representing any sound file – Sound_Any.

Some functions can accept a file name (minus the .usf extension), or a sound file enumeration consisting of "snd_" prefixed to the root of the source sound file name, and listed automatically in sounds.inc. For example:

```
/****************************************************\
 *                     WARNING                      *
 *     This file was automatically generated by     *
 * enumgen.py.  Do not edit directly or put under   *
 *                 source control.                  *
\****************************************************/

enum sound_name {
  snd_none               =     0,
  snd_min                =  4096,
  snd_yawn                =  4096,
  snd_max
};
```

Functions to start playing a sound include:

```
Sound: sound_play(sound_name: sound, bool: interrupt =
false);
Sound: sound_play_file(const string[]);

sound_command(command_name: name, bool: interrupt = false);
```

The difference between these functions is whether to play a sound by symbolic name or file name. The special function sound_command() will start the named lookup table, which is used by the system to randomly select and play one of many sound files.

To stop a specific sound, call the following function with the name of the sound file enumeration or the handle returned by one of the play functions, or Sound_Any to stop any currently playing sound:

```
bool: sound_stop({Sound,sound_name}: sound = Sound_Any);
```

To check whether a specific sound file, or any sound file, is playing, use this:

```
bool: sound_is_playing({Sound,sound_name}: sound =
Sound_Any);
```

The loudness of sound playback can be adjusted between 10% and 200% using the sound_set_volume() function; the current volume can of course be queried. 100% is the normal playback volume.

```
sound_get_volume();
sound_set_volume(volume);
```

The playback speed can also be modified, over a range of 25% to 200%, where 100% is normal playback speed.

```
sound_get_speed();
sound_set_speed(speed);
```

Pleo-to-Host Connectivity

When programming Pleo, it is most convenient to connect Pleo directly to a development machine. This can be done through the serial port or the USB port. In order to test applications, it is also necessary to use SD Cards.

See the *Pleo Monitor* documentation for more details on the possible connectivity options available.

Packet System

Introduction

Pleo includes two ARM7 processors: the main Atmel AT91SAM7S256 in the body of Pleo, and an NXP LPC2103 in the head. They are connected through a UART interface. The communications protocol between them is a packet-based system, the underlying transport being based on RFC 1662.

In Pleo software versions 1.0.x there is a mechanism to send packets from Pawn – executing on the Atmel - to the NXP. But there is no mechanism to receive packets from the NXP in Pawn.

This document describes an extension available in the Pleo 1.1.x software that allows the sending and receiving of packets to and from the NXP.

Background

The NXP in Pleo handles the following components:

- Audio input, including loudness, direction detection and raw audio data capture.
- Camera interface, including light levels, color tracking and raw image data capture.
- IR interface, including object detection, IR 'beacon' and generic IR communications.
- Mouth IR interrupter, which detects when something is in his mouth.
- Serial RAM interface, which allows storing and retrieving information to the SRAM in the head.
- Head and Chin capacitive touch sensors.

The packet system between the NXP and Atmel is designed to be as concise as possible, but still be recognizable to humans, so we use alpha characters as packet designators. The following packet types are defined:

- 'a': audio
- 'c': camera
- 'e': echo
- 'i': IR
- 'l': log
- 'm': mouth IR
- 'p': packet statistics
- 's': serial ram
- 't': touch sensors

- 'v': version string
- 'x': error string

These codes are used in both directions. For example, to disable the camera system, you send the packet “cd”, where the 'c' represents the Camera module, and 'd' represents disable.

Pleo 1.0.x

Sending

In the Pleo 1.0.x software release, you can send arbitrary packets from the Pleo Monitor, using the 'pkt send' command. For example, you can perform the following command:

```
> pkt send ?
```

and you will receive (assuming you have enabled the NXP log type via a 'log enable nxp') output similar to the following:

```
> INFO: nxp: 7361 - Nov 15 2007 21:34:25
```

From Pawn script you can use the function **monitor_exec** (see Util.inc) to perform any monitor command. Note that access rights are in effect, so you may need to first execute an 'access' command to gain access to the monitor commands you wish to execute.

For example, in your Pawn script, you can do the following:

```
monitor_exec(“pkt send ?”);
```

If you have a terminal program connected to Pleo, you will see the same output as the example above that was done directly from the monitor.

Receiving

This feature is useful if you wish to set parameters of components on the NXP, but you cannot receive packets from the NXP in Pleo 1.0.

Pleo 1.1.x

In Pleo 1.1.x we have added a new 'virtual' sensor named SENSOR_PACKET. This sensor can be associated with any of the packet types listed above.

Sending

In Pleo 1.1.x, the same method is used for sending data to the NXP as described above for Pleo 1.0.

There is also an new alternate method to send packets to the NXP. Instead of using the monitor_exec call, you can now build your own packets in a buffer, and send them to the head using the generic sensor_write_data function. For example, to get the version string back from the NXP, you can do the following:

```
sensor_write_data(SENSOR_PACKET, “?”);
```

This will send the buffer passed as-is directly to the packet system.

NOTE: If this is an unrecognized packet, an error packet ('x') will be returned. So it may be useful to always filter the error packet to catch any errors you may be producing.

Receiving

When one of these NXP packets is received by the low-level firmware, the data associated with it will be stored in a special area of memory, and the SENSOR_PACKET sensor triggered, resulting in a call to the sensors on_sensor function, if any.

For example, here is a sensor script that will listen for any touch packets from the NXP:

```
// include sensor functions and types
#include <Sensor.inc>
public init()
{
    // set the packet filter to send us touch
events
    sensor_set_config(SENSOR_PACKET,
sensor_config_packet_filter, PACKET_TOUCH);
}

public on_sensor(time, sensor_name:sensor, value)
{
    switch (sensor)
    {
          case SENSOR_PACKET:
          {
                new buffer[32];
                sensor_read_data(SENSOR_PACKET,
buffer);
```

```
                printf("== SENSOR_PACKET:
length:%d, data:%s ==\n", value, buffer);
          }
          default:
                printf("Unhandled sensor event:
%d\n", sensor);
    }
    return true;
}
```

In this example, we note the following:

In the init function, we tell the firmware what packets we would like to receive using the sensor_set_config function. The SENSOR_PACKET is a sensor_name: defined in pleo/sensors.inc. The sensor_config_packet_filter is a sensor_config:, also defined in pleo/sensors.inc. And the PACKET_TOUCH is a packet_type: defined in pleo/sensors.inc.

1. In the on_sensor function, which is called every time a sensor is triggered, we handle the SENSOR_PACKET trigger. This trigger is set whenever a packet that we have registered for (or set the filter for) is received by the underlying firmware. The value passed is the length of the packet received.
2. Within the SENSOR_PACKET handler, we use the sensor_get_data function to retrieve the actual packet data. From this point on we can examine the data to extract the information we care about. Note in this case we simply print the result to the monitor.

Notes:

- You must be careful about how you extract the data in Pawn. Remember that the [] operator is cell-based. If the data is packed in the resultant buffer, you must use the {} operator.
- Currently, all data returned is unpacked, meaning we are wasting some memory, and that you must ensure that your buffer is declared large enough to hold your largest expected packet.

Appendices

File Formats

There are a number of formats used throughout the Pleo software system This Appendix will detail these in order to allow third-party developers to write their own tools that can generate resource data usable by Pleo.

The types include:

- Sound Files, USF
- Motion Files, both CSV and UMF
- Command Files, both CSV and UCF
- Script, .AMX
- Ugobe Project Files (UPF)
- Ugobe Resource Files (URF)

Sound Files

USF is the native sound file format for Pleo sounds. The USF file starts with a header with the following structure:

```
#define SOUND_FILE_ID "UGSF"
#define SOUND_FILE_EXT "usf"

struct SOUD_FILE_HEADER
{
      // = "UGSF" = UGobe Sound File
      int8 usf_id[4];
      // = 0 (old format), = 1 if we include the sf_name
(current)
      uint8 sf_ver;
      // original file name, minus the extension
      int8 sf_name[32];
      // specify things about data that follows
      SFINFO info;
      // normally 8
      uint8 bits_per_sample;
      // normally 1
      uint8 num_channels;
      // normally 11000
      uint16 samples_per_second;
      // number of times to loop this sound clip; 0 =
forever
      uint16 loop_count;
      // 0xFFFFFFFF = just read to end of file...
      uint32 num_samples;
};
```

where SFINFO is:

```
struct SFINFO
{
    // 0 = uncompressed PCM; 1 = ADPCM
    uint8 comp_type : 4;
    uint8 dont_pitch_shift : 1;
    uint8 reserved : 3;
};
```

The header is followed by the raw PCM data or the ADPCM data, depending on comp_type flag setting in the SFINFO byte.

Note the ADPCM does not follow the IMA specification. It uses the same delta tables, but we store only the top 8 bits of audio data, since the current sound output can only go up to 10-bits of resolution.

Motion Files

Motion files contain instructions on how to move each joint over time. They are sometimes also called animation files. Binary UMF files are generated from text-based CSV files, typically exported from 3ds Max.

```
#define UMF_JOINTS 15

#define JID_RE 0x4552 /* "Right Elbow" */
#define JID_LE 0x454C /* "Left Elbow" */
#define JID_RS 0x5352 /* "Right Shoulder" */
#define JID_LS 0x534C /* "Left Shoulder" */
#define JID_RK 0x4b52 /* "Right Knee" */
#define JID_LK 0x4b4c /* "Left Knee" */
#define JID_RH 0x4852 /* "Right Hip" */
#define JID_LH 0x484c /* "Left Hip" */
#define JID_TS 0x5354 /* "Torso" */
#define JID_HT 0x5448 /* "Horizontal Tail" */
#define JID_VT 0x5456 /* "Vertical Tail" */
#define JID_HN 0x4e48 /* "Horizontal Neck" */
#define JID_VN 0x4e56 /* "Vertical Neck" */
#define JID_EY 0x5945 /* "Eyes" */
#define JID_JW 0x574a /* "Jaw" */
#define JID_SD 0x4453 /* "Sound Channel" */
#define JID_TR 0x5254 /* "Transition Data" */
#define JID_FS 0x5346 /* "Foot switch data" */
#define JID_HD 0x4448 /* "Head data" */

#define UMF3_ID "UGMF"
#define UMF3_VER 3
#define SIZE_UMF3_NAME 32

struct UMF3_HEADER {
    /* UGMF*/
    uint8 umf_id[4];
    /* 3 */
    uint8 umf_version;
    /* original base name of this file. Sans
extension*/
```

```
    uint8 umf_name[SIZE_UMF3_NAME];
    /* number of joints in the data */
    uint8 num_joints;
    /* 1 = degrees; other values are deprecated */
    uint8 angle_range;
    /* units of time in milliseconds */
    uint8 timebase_ms;
    /* number of MotionVectors in this motion file */
    uint32 num_vectors;
    /* end time for this animation, in units of
timebase_ms */
    uint16  end_time;
    // NOTE: we store exactly num_joints uint16s for
following table in the file; it is larger than that
below to simplify runtime memory management
    /* list of jids, in order to match data */
    uint16 joints[UMF_JOINTS+4];
};
```

After the header we have a list of motion vectors sorted in time order.

```
#define UMF3_GOOD 0xF00D
#define UMF3_DONE 0xDEAD

// run time motion tags:
#define UMF3_RUN  0xEA75  // this has been sent to
motor control
#define UMF3_END  0x0000  // this is done executing

struct  UMF3_MOTION_VECTOR
{
    /* set to 0xF00D to mark the start; 0xDEAD to
mark end of file, just for safety. */
    uint16 motion_tag;
    uint16 joint_ids;
    uint16 start_time;
    uint16 goal_time;
    sint16 velocity;
    sint16 position;
};
```

Command Files

Command files describe when a motion or sound can play, based on associated properties. Command Files can be of Sound or Motion type. The file formats are identical – the oly difference is that Motin Command files contain a property_motion key, whereas Sound Commands contain a property_sound key.

```
#define UCF_ID "UGCF"
#define UCF_VER 4
#define SIZE_COMMAND_NAME 32

struct UCF_HEADER {
    sint8 signature[4];
    sint16 version;
    sint8 cmd_name[SIZE_COMMAND_NAME];
    sint16 property_count;
    sint16 entry_count;
};
```

Following the header, there are *property_count* * *entry_count* tuples, each of which contains a property id, property type, and value.

```
struct TAG_ENTRY {
    uint16 id;
    uint16 type;
    uint16 value
};
```

The property type is one of these:

```
enum TAG_TYPE {
    PROP_TYPE_NONE    = 0,  // invalid
    PROP_TYPE_VALUE = 1, //matches exactly
    PROP_TYPE_MIN   = 2, //must be over
    PROP_TYPE_MAX   = 3, //must be under
};
```

Resource Files

A Ugobe Resource File (URF) is a collection of sound files, motion files, command files, script files and the user properties defined in a project. Combining resources makes copying and transportation of Pleo 'applications' much easier.

The general layout of a URF looks like this:

- Header
- Resource 1
- Resource 2
- Resource 3
- …
- Resource N
- Sound Index
- Motion Index
- Command Index
- Script Index
- Property Index

The URF file starts with a simple header, which contains a signature, the file format version, the build version and a list of offsets to the various Indices (also referred to as Table Of Contents, or TOC).

Byte offset	Data	Description
0	‘UGRF’	Resource file tag or signature
4	VVvv	File format version. VV is the major version and vv is the minor version.
8	tttt	Time and date this URF was built. Uses the Python time.time function.
12	xxxx	Subversion revision ID, or the value given in the <options/version /> element in the project file.
16	‘UGSF’	Signature of Sound Index
20	xxxx	Offset to start of Sound Index within this URF. Relative to start of file.
24	‘UGMF’	Signature of Motion Index
28	xxxx	Offset to start of Motion Index within this URF. Relative to start of file.
32	‘UGCF’	Signature of Command Index
36	xxxx	Offset to start of Command Index within this URF. Relative to start of file.
40	‘ AMX’	Signature of Script Index
44	xxxx	Offset to start of Script Index within this URF. Relative to start of file.
48	‘PROP’	Signature of Property Index
52	xxxx	Offset to start of Property Index within this URF. Relative to start of file.
56	‘SIZE’	Signature of Size field
60	xxxx	Size of this file, minus trailing Adler32 tag and value

The order of the various index offsets can, in theory, vary; that is, they may come in any order. In is recommended, however, that they be in this order.

Following the header are all the resources for this application, possibly aligned on some block size, in order to speed lookup and loading.

After all the resources are the Indices for each resource type. Each index starts with the same signature as that in the header, and a length field that is the size, in bytes, of this particular index.

Byte offset	Data	Description
0	'UGSF'	Resource file tag or signature
4	xxxx	Size of this index/TOC, in bytes.

This is followed by a list of entries, one for each resource in the body of the URF. It is arranged like so:

Byte offset	Data	Description
0	xxxx	Beginning offset for this resource, in bytes, from the beginning of the URF file
4	xxxx	Length or size of this resource, in bytes.
8	ssss	Name of this resource. This field is always 32 characters in length. It may or may not be null-terminated.

After the Indices, there is one more tag and value for the calculated Adler32 values for this file. The Adler32 is used to verify that the URF is intact and has not been corrupted. The Adler32 value does NOT include the ADLR tag or value itself.

Byte offset	Data	Description
0	'ADLR'	Signature for the Adler32 value
4	xxxx	Adler32 checksum value for the whole URF, minus this value and the signature

Document Revision History

Revision	Date	Comment
0.1		*Initial version*
0.2		*Add note about .exe tools change*
0.3		*Formatting updates. Add note about OSX build tools.*
0.4	*May 07, 2008*	*Add prerequisites section*
0.5	*May 30, 2008*	*Formatting updates*
0.6	*June 5, 2008*	*Some clarifications and simplifications*
0.7	*June 11, 2008*	*Formatting updates, template changes*
0.8	*July 09, 2008*	*Added Property and Drive chapters*
0.9	*July 29, 2008*	*Added Sensors and Sound chapters*
1.0	*July 30, 2008*	*Added Build Tools section. Added URF file description.*
2.0	*August 15, 2008*	*Formatting updates, template revisions*