PVRScope

User Manual

Filename	:	PVRScope.User Manual
Version	:	PowerVR SDK REL_17.1@4658063a External Issue
Issue Date	:	07 Apr 2017
Author	:	Imagination Technologies Limited

Contents

1. Introduction

1.1. Document Overview

This document is intended as a reference for PVRScope library and provides a number of examples of its use. The document covers key topics including how to access the performance counters in PowerVR hardware and send user defined information to PVRTune.

Note: For more information about PVRTune, consult the "PVRTune User Manual".

1.2. Library Overview

PVRScope is a utility library which has two functionalities, namely:

- **PVRScopeStats**: PVRScope is used to access the hardware performance counters in PowerVR hardware.
- **PVRScopeComms**: PVRScope allows an application to send user defined information to PVRTune via PVRPerfServer as counters, timing data, and marks, or as editable data that can be passed back to the application.

PVRScope is supplied in the following files:

- `PVRScopeStats.h`: This is the header file defining the PVRScopeStats functionality.
- `PVRScopeComms.h`: This is the header file defining the PVRScopeComms functionality.
- `PVRScopeDeveloper.lib`: This is the PVRScope library file.

1.2.1. Library Compatibility

Currently, PVRScope libraries installed on Windows machines only function with Visual Studio 2010 and later versions.

1.2.2. PVRScopeStats Limitations

The following are the limitations to PVRScopeStats:

- Only one instance of PVRScopeStats may communicate with PVRScopeServices (a driver library) at any given time. If a PVRScopeStats-enabled application attempts to communicate with PVRScopeServices at the same time as another such application, or at the same time as PVRPerfServer, conflicts can occur that may make performance data unreliable. In this case, PVRPerfServer can be run with the `--disable-hwperf` flag.
- Performance counters can only be read on devices whose drivers have been built with hardware profiling enabled. This configuration is the default in most production drivers due to negligible overhead.
- Performance counters contain the average value of that counter from the last time the counter was interrogated.

Multithreading

- PVRScopeStats is not re-entrant. Ensure that only a single thread calls into PVRScope at any point in time.

1.2.3. PVRScopeComms Limitations

The following are limitations of PVRScopeComms:

- PVRPerfServer must be running on the host device as an intermediary between the PVRScopeComms-enabled application and PVRTune.
- The available data types for Remotely Editable Library items are: `Boolean`, `Enumerator`, `Float`, `Integer` and `String`.

Multithreading

- PVRScopeComms is re-entrant only if the `SSPSCommsData` pointer is unique per thread that is simultaneously executing inside a PVRScope function.
- If a single pointer is used (perhaps protected via locks) then everything would work except if Custom Timing Data is sent, as each thread's activity would then be garbled together.
- Depending on your needs, a single pointer could be a good choice for Custom Counters and a Remotely Editable Library.
- For Marks and Custom Timing Data, a common solution is for each thread to call `pplInitialise(…)` to create a connection, meaning that each can submit custom timing data, and have each show up as a parallel timeline in the PVRTune GUI. If a thread pool is in use, another solution could be to create a connection per "job", leading to the PVRTune GUI showing a timeline per job rather than per thread.

2. PVRScopeStats

PVRScopeStats is used to access the performance counters in PowerVR hardware via a driver library called PVRScopeServices.

2.1. Initialisation

To initialise PVRScopeStats:

1. Include the PVRScope header file.

```
#include "PVRScopeStats.h"
```

2. Define the control structures.

```
// Defines
#define NO GROUP CHANGE 0xffffffff

// Internal control data
SPVRScopeImplData         *PVRScopeStatsData;

// Counter information
SPVRScopeCounterDef       *counterDefinitions;
unsigned int               numCounters;

// Counter reading data
unsigned int               activeGroup;
unsigned int               selectedGroup;
bool                       activeGroupChanged;
SPVRScopeCounterReading    counterReading;
```

3. Initialise communication with PVRScopeServices.

```
PVRScopeInitialise(PVRScopeStatsData)
```

4. Set up the counter data structures.

```
PVRScopeGetCounters(*PVRScopeStatsData,
                    numCounters,
                    counterDefinitions,
                    counterReading)
```

Note: PVRScopeFindStandardCounter is a function that makes setting up counter data structures more efficient. Rather than listing all the counters available, it presents an indexed list of common counters.

2.2. Reading Counter Data

To update counter data:

1. Set `activeGroup` to one of the following:

- A new Group ID, if the next counters to be read are in a different group.
- `0xffffffff` if no group change is required.

2. Read the current counter values and set the group for the next update:

```
PVRScopeReadCounters (PVRScopeStatsData, &counterReading, activeGroup)
PVRScopeSetGroup (PVRScopeStatsData, &counterReading, activeGroup)
```

It is to be noted that there are some limitations to the `PVRScopeReadCounters` and `PVRScopeSetGroup` functions. The functions have to be called regularly to allow PVRScope to track the latest hardware performance data. If `psReading` is not `NULL`, PVRScope will also calculate and return counter values to the application. There are two use cases for calling the function:

- A 3D application rendering a performance HUD, e.g., the on-screen graphs in PVRScopeStats Example (see Appendix A). Such an application should call this function at least once per frame in order to gather new counter values. If slower HUD updates are desired, `psReading` may be `NULL` until a new reading is required, in order to smooth out values across longer time periods, perhaps for a number of frames.
- A standalone performance monitor (e.g., PVRMonitor) or logging application. Such an application should idle and regularly wake up to call this function. Suggested rates are 100Hz (10ms delays) or 200Hz (5ms delays). If counter updates are required at a lower rate, set `psReading` to `NULL` on all calls except when new counter values are desired (see Appendix A).

Note: Although the sampling period is configurable, the recommended sampling interval is 16ms. Setting a smaller sampling period may result in "noisy" data, whilst a longer sampling period will result in some counter events being omitted as the driver has a limited amount of data that can be stored in its circular buffer until old values are overwritten.

2.3. Outputting Counter Data

To output counter data, perform the following steps in a loop from `i=0` to `i=numCounters`:

1. Check if the counter has been updated and only perform Steps 2 through 5 if it has.

```
if(i < counterReading.nValueCnt)
{
```

2. Check if the counter is given as a percentage.

```
bool isPercentage = counterDefinitions[i].bPercentage;
```

3. Retrieve the name of the counter.

```
std::string counterName = counterDefinitions[i].pszName
```

4. Retrieve the value of the counter.

```
float counterValue = counterReading.pfValueBuf[i]
```

5. Output the value of the counter. For example using `printf`:

```
// If it is a percentage output the % symbol
    if(isPercentage)
        printf("%s : %f%%\", counterName, counterValue);
    else
        printf("%s : %f",   counterName, counterValue);
}
```

2.4. Shutdown

To shutdown PVRScopeStats call the shutdown functions as follows:

```
PVRScopeDeInitialise(*PVRScopeStatsData, *counterDefinitions, *counterReading);
```

For a more in depth example, see Appendix A.

3. PVRScopeComms

PVRScopeComms allows an application to send user defined information to PVRTune via PVRPerfServer, both as counters, timing data, and marks, or as editable data that can be passed back to the application.

3.1. Initialisation

To initialise PVRScopeComms:

1. Include the PVRScope header file.

```
#include "PVRScopeComms.h"
```

2. Create a name for the timeline as it should appear in PVRTune.

```
std::string threadName = "Main Thread"
```

3. Initialise the communication with PVRTune.

```
// Init Comms
SSPSCommsData* PVRScopeComms;
PVRScopeComms = pplInitialise(threadName.c_str(),
                              (unsigned int)threadName.length());
```

4. Repeat Steps 2 and 3 for each additional timeline which should appear in PVRTune.

3.2. Sending a Custom Mark

To send a custom mark:

1. Create a title for the mark.

```
// Create the custom mark
std::string customMarkTitle = "Custom Mark";
```

2. Send the mark to PVRTune.

```
pplSendMark(*PVRScopeComms,
             customMarkTitle.c_str(),
            (unsigned int)customMarkTitle.length());
```

3.3. Sending a Custom Counter

A counter is an accumulator that is used by PVRTune to only display the per-frame and per-second rates (i.e., rate of change values) of that accumulator. A counter should only increment, meaning it must never reset or decrease, but it may wrap. Furthermore, a counter cannot display absolute values. The following exemplify counter usage in PVRTune:

- **Cars being rendered**: Increment counter every time a car is rendered or, alternatively, increment once per frame for the number of cars that will be rendered. PVRTune will calculate the 'cars rendered per frame' and 'cars rendered per second'.
- **Healing (health received)**: Every time a character receives health, increment the counter by the quantity. PVRTune will generate 'healing per second' and 'healing per frame'.

3.3.1. Additional Initialisation Steps

To send a custom counter, a number of additional initialisation steps must be performed:

1. Set up an enumerator of possible custom counters. This contains one entry per counter and an entry containing the total number of counters.

```
enum PVRScopeCustomCounters
{
    eARBITRARY COUNTER,
    eNUM_CUSTOM_COUNTERS
};
```

2. Define a custom counter array.

```
SSPSCommsCounterDef PVRScopeCustomCounterArray[eNUM_CUSTOM_COUNTERS];
```

3. Set the name of each counter.

```
std::string customCounterName[eNUM_CUSTOM_COUNTERS] = "";
for(unsigned int i = 0; i < (unsigned int)eNUM_CUSTOM_COUNTERS; ++i)
{
    switch (i)
    {
        case eARBITRARY COUNTER:
            customCounterName[i] = "Arbitrary Counter";
            break;
        default:
            customCounterName[i] = "Unnamed counter";
            break;
    }
    PVRScopeCustomCounterArray[i].pszName = customCounterName[i].c_str();
    PVRScopeCustomCounterArray[i].nNameLength = customCounterName[i].length();
}
```

4. Define an array of integers to hold the values as they are updated for each frame.

```
unsigned int newValueArray[eNUM_CUSTOM_COUNTERS] = { 0 };
```

5. Submit the array of counters.

```
pplCountersCreate(*PVRScopeComms, PVRScopeCustomCounterArray, eNUM_CUSTOM_COUNTERS);
```

3.3.2. Updating and Sending the Counter Value

To update and send the counter value, perform the following:

1. Set a new value for any counter that needs updating. For example, updating `eARBITRARY_COUNTER` to increment by a value of `2`:

```
int newCounterValue = 2;
newValueArray[eARBITRARY_COUNTER] += newCounterValue;
```

2. Send the list of updated values to PVRTune.

```
pplCountersUpdate(*PVRScopeComms, newValueArray);
```

3.4. Sending and Receiving Remotely Editable Library

The following data types can be sent to PVRTune for editing: `Boolean`, `Enumerator`, `Float`, `Integer` and `String`. All of these items are stored as instances of `SSPSCommsLibraryItem` with `eType` stating the data type of the item and `pData` containing the actual item. The name of the item, as it appears in PVRTune, is based on the value of `pszName`. Full stops placed in `pszName` can be used to create categories and sub-categories which PVRTune displays as a foldable tree view. For example:

- Category names:

```
Category-A.SubCategory-A.Item-1
Category-A.SubCategory-A.Item-2
Category-A.SubCategory-B.Item-1
Category-B.SubCategory-A.Item-1
Category-B.SubCategory-A.Item-2
```

- Foldable tree view:

```
Category-A.
        SubCategory-A.
             Item-1
             Item-2
        SubCategory-B.
             Item-1
Category-B.
        SubCategory-A.
             Item-1
             Item-2
```

The process of sending and receiving editable data consists of three steps, namely initialising the editable data, sending the data to PVRTune at initialisation time, and then retrieving updates to the data. The first two steps are usually performed simultaneously and, as such, the examples identified next follow this approach.

3.4.1. Additional Initialisation Steps

In order to send editable data, a number of additional initialisation steps must be performed:

1. Set up enumerators of possible editable items, one for each data type. From this, determine the total number of editable items.

```
enum PVRScopeEditableFloats
{
                eARBITRARY EDITABLE FLOAT,
                eNUMBER OF EDITABLE FLOATS
};
enum PVRScopeEditableStrings
{
                eARBITRARY_EDITABLE_STRING,
                eNUMBER OF EDITABLE STRINGS
};
const unsigned int numEditableItems = (unsigned int)eNUMBER OF EDITABLE FLOATS
                                    + (unsigned int)eNUMBER_OF_EDITABLE_STRINGS;
```

2. Create the editable item array.

```
SSPSCommsLibraryItem editableItemArray[numEditableItems];
```

3. Initialise the editable items by giving them default values.

```
for(unsigned int i = 0; i < numEditableItems; i++)
{
    // Set some arbitrary float data
    float arbitraryFloat = 3.5f;
    float arbitraryString = "Arbitrary String";

    // Create an editable float object
    SSPSCommsLibraryTypeFloat scopeFloat;
    editableItemArray[i].pszName  = arbitraryString;
    editableItemArray[i].eType    = eSPSCommsLibTypeFloat;
    scopeFloat.fCurrent           = arbitraryFloat;
    scopeFloat.fMin               = 0.0f;
    scopeFloat.fMax               = 200.0f;

    // Add the float object to the editable item
    editableItemArray[i].itemData = (const char*)&scopeFloat;
    editableItemArray[i].nDataLength = sizeof(scopeFloat);
}
```

4. Send the editable items to PVRTune.

```
pplLibraryCreate(*PVRScopeComms, editableItemArray, numEditableItems)
```

3.4.2. Retrieving Updated Data

To retrieve updated data:

1. Create a number of temporary variables to hold the updated data.

```
// Used to determine the length of strings
unsigned int itemLength(0);

// Used to determine which item in 'editableItemArray' the updated refers to
unsigned int itemNum(0)

// A pointer to the edited item
const char* itemData(NULL);

// A float to store the returned value in
float arbitraryFloat = 0.0f;
```

2. In a loop, retrieve each dirty item.

```
while(pplLibraryDirtyGetFirst(*PVRScopeComms, itemNum, itemLength, &itemData))
{
```

3. Determine the items type based on its item number (based on the enumerators created in Step 1).

```
    if(itemNum < eNUMBER_OF_EDITABLE_FLOATS)
    {
```

4. Retrieve the editable item. For example, retrieving a float:

```
        SSPSCommsLibraryTypeFloat* newItemData = (SSPSCommsLibraryTypeFloat*)itemData;
        arbitraryFloat = newItemData->fCurrent;
    }
}
```

Note: Retrieving a string is a special case. The string should be created in advance and retrieved as follows:

```
arbitraryString.assign(itemData, itemLength);
```

5. Use the retrieved data to update the application.

3.5. Sending Custom Timing Data

PVRTune has the ability to display custom timing data similar to that displayed for the TA and 3D tasks. This custom timing data appears in the timeline created using `pplInitialise(…)`. Each block of the custom timing data is started with a call to `pplSendProcessingBegin(…)` and ended with a call to `pplSendProcessingEnd(…)`.

Note: These calls cannot be nested.

3.5.1. Starting a Timing Block

To start a timing block, complete the following instructions:

1. Determine the current frame:

```
unsigned int currentFrame = 0;
```

2. Give the timing block a name:

```
std::string funcName = __func__;
```

3. Call `pplSendProcessingBegin(…)`.

```
pplSendProcessingBegin(*PVRScopeComms,
                       funcName.c_str(),
                       (unsigned int)funcName.length(),
                       currentFrame);
```

3.5.2. Ending a Timing Block

To end a timing block, call `pplSendProcessingEnd(…)` as follows:

```
pplSendProcessingEnd(*PVRScopeComms);
```

3.6. Shutdown

To shutdown PVRScopeComms, call the shutdown function as follows:

```
pplShutdown(PVRScopeComms);
```

For more in-depth examples, see Appendix B.

4. Contact Details

For further support, visit our forum:
http://forum.imgtec.com

Or file a ticket in our support system:
https://pvrsupport.imgtec.com

To learn more about our PowerVR Graphics Tools and SDK and Insider programme, please visit:
http://www.powervrinsider.com

For general enquiries, please visit our website:
http://imgtec.com/corporate/contactus.asp

Appendix A. PVRScopeStats Example Code

```
/*!*****************************************************************************************
*
* This example outputs the values of the hardware counters found in Group 0
* to Android Logcat once a second for 60 seconds, and consists of five steps:
*
* 1. Define a function to initialise PVRScopeStats
* 2. Initialise PVRScopeStats
* 3. Set the active group to 0
* 4. Read and output the counter information for group 0 to Logcat
* 5. Shutdown PVRScopeStats
*
*******************************************************************************************/

#include <android/log.h>

// Few macro definitions here
#define LOGV(...)   android_log_print(ANDROID_LOG_VERBOSE, "PVRScope",   VA_ARGS  )
#define LOGD(...) __android_log_print(ANDROID_LOG_DEBUG  , "PVRScope", __VA_ARGS__)
#define LOGI(...) __android_log_print(ANDROID_LOG_INFO   , "PVRScope", __VA_ARGS__)
#define LOGW(...)   android_log_print(ANDROID_LOG_WARN   , "PVRScope",   VA_ARGS  )
#define LOGE(...)   android_log_print(ANDROID_LOG_ERROR  , "PVRScope",   VA_ARGS  )
#define NUMBER_OF_LOOPS 60


#include <unistd.h>
#include <string.h>
#include <jni.h>
#include "PVRScopeStats.h"

// Step 1. Define a function to initialise PVRScopeStats
bool PSInit(SPVRScopeImplData **ppsPVRScopeData, SPVRScopeCounterDef **ppsCounters,
SPVRScopeCounterReading* const psReading, unsigned int* const pnCount)
{
	//Initialise PVRScope
	const EPVRScopeInitCode eInitCode = PVRScopeInitialise(ppsPVRScopeData);

	if(ePVRScopeInitCodeOk == eInitCode)
	{
		LOGI("Initialised services connection.\n");
	}
	else
	{
		LOGE("Error: failed to initialise services connection.\n");
		*ppsPVRScopeData = NULL;
		return false;
	}

	//Initialise the counter data structures.
	if (PVRScopeGetCounters(*ppsPVRScopeData, pnCount, ppsCounters, psReading))
	{
		LOGI("Total counters enabled: %d.", *pnCount);
	}

	return true;
}

int main()
{
	//Internal control data
	SPVRScopeImplData *psData;

	//Counter information (set at uint time)
	SPVRScopeCounterDef           *psCounters;
	unsigned int                  uCounterNum;

	//Counter reading data
	unsigned int                  uActiveGroupSelect;
	bool                          bActiveGroupChanged;
	SPVRScopeCounterReading       sReading;


	// Step 2. Initialise PVRScopeStats
```

```
        if (PSInit(&psData, &psCounters, &sReading, &uCounterNum))
                LOGI("PVRScope up and running.");
        else
                LOGE("Error initializing PVRScope.");

        //Print each and every counter (and its group)
        LOGI("Find below the list of counters:");
        for(int i = 0; i < uCounterNum; ++i)
        {
                LOGI("    Group %d %s", psCounters[i].nGroup, psCounters[i].pszName);
        }

        // Step 3. Set the active group to 0
        bActiveGroupChanged = true;
        uActiveGroupSelect = 0;

        unsigned int sampleRate = 100;
        unsigned int index = 0;

        unsigned int i = 0;
        while (i < NUMBER OF LOOPS)
        {
                // Ask for the active group 0 only on the first run.
                if(bActiveGroupChanged)
                {
                        PVRScopeSetGroup(psData, uActiveGroupSelect);
                        bActiveGroupChanged = false;
                }

                ++index;
                if (index < sampleRate)
                {
                        // Sample the counters every 10ms. Don't read or output it.
                        PVRScopeReadCounters(psData, NULL);
                }
                else
                {
                        ++i;
                        index = 0;

                        // Step 4. Read and output the counter information for group 0 to
                        Logcat
                        if(PVRScopeReadCounters(psData, &sReading))
                        {
                                // Check for all the counters in the system if the counter has a
                                value on the given active group and ouptut it.
                                for(int i = 0; i < uCounterNum; ++i)
                                {
                                        if(i < sReading.nValueCnt)
                                        {
                                                //Print the 3D Load
                                                if (strcmp(psCounters[i].pszName, "GPU task load:
                                                    3D core") == 0)
                                                {
                                                        LOGI("%s : %f\%", psCounters[i].pszName,
                                                             sReading.pfValueBuf[i]);
                                                }
                                                //Print the TA Load
                                                else if (strcmp(psCounters[i].pszName, "GPU task
                                                          load: TA core") == 0)
                                                {
                                                        LOGI("%s : %f\%", psCounters[i].pszName,
                                                             sReading.pfValueBuf[i]);
                                                }
                                        }
                                }
                        }
                }

                //Poll for the counters once a second
                usleep(10 * 1000);
        }

        // Step 5. Shutdown PVRScopeStats
        PVRScopeDeInitialise(&psData, &psCounters, &sReading);
}
```

Appendix B. PVRScopeComms Example Code

B.1. Sending a Custom Mark

```
/*!*****************************************************************************************
*
* This example code consists of four steps:
*
* 1. Setup PVRScopeComms
* 2. Create a Custom Mark
* 3. Send the Custom Mark to PVRTune
* 4. Shutdown PVRScopeComms
*
*******************************************************************************************/

#include "PVRScopeComms.h"

// Step 1. Setup PVRScopeComms
SSPSCommsData* PVRScopeComms;
std::string timelineTitle = "Example";
PVRScopeComms = pplInitialise(timelineTitle,
                              (unsigned int)timelineTitle.length());

// Step 2. Create a Custom Mark
std::string customMarkTitle = "Custom Mark";

// Step 3. Send the Custom Mark to PVRTune
if(!pplSendMark(*PVRScopeComms,
                customMarkTitle.c str(),
               (unsigned int)customMarkTitle.length()))
{
    // Error handling goes here
}
// Step 4. Shutdown PVRScopeComms
pplShutdown(PVRScopeComms);
```

B.2. Sending Custom Timing Data

```
/*!*****************************************************************************************
*
* This example code consists of four steps:
*
* 1. Setup two functions to time
* 2. Setup PVRScopeComms
* 3. Call the two functions to end the Custom Timing Data to PVRTune
* 4. Shutdown PVRScopeComms
*
*******************************************************************************************/

#include "PVRScopeComms.h"

// Step 1. Setup two functions to time

// Function that pretends to do some work
void Foo(SSPSCommsData* comms, unsigned int frameNum)
{
    std::string funcName =   func  ;
    pplSendProcessingBegin(*comms, funcName, (unsigned int)funcName.length(), frameNum);

    // Pretend to do some work
    sleep(1);

    pplSendProcessingEnd(*comms);
}

// Function that pretends to do a bit more work
void Bar(SSPSCommsData* comms, unsigned int frameNum)
{
    std::string funcName = __func__;
    pplSendProcessingBegin(*comms, funcName, (unsigned int)funcName.length(), frameNum);
```

```
    // Pretend to do some work
    sleep(2);

    pplSendProcessingEnd(*comms);
}


// Step 2. Setup PVRScopeComms
SSPSCommsData* PVRScopeComms;
std::string timelineTitle = "Example";
PVRScopeComms = pplInitialise(timelineTitle,
                              (unsigned int)timelineTitle.length());

// Step 3. Send Custom Timing Data to PVRTune
unsigned int currentFrame = 0;
for(unsigned int i = 0; i < 100; i++)
{
    Foo(PVRScopeComms, currentFrame);
    Bar(PVRScopeComms, currentFrame);

    currentFrame++;
}

// Step 4. Shutdown PVRScopeComms
pplShutdown(PVRScopeComms);
```

B.3. Sending a Custom Counter

```
/*****************************************************************************************************
*
* This example code consists of six steps:
*
* 1. Setup PVRScopeComms
* 2. Create a Custom Counter
* 3. Submit the Custom Counter to PVRTune
* 4. Send an initial value for that counter to PVRTune
* 5. Update the counter once and send the updated counter to PVRTune
* 6. Shutdown PVRScopeComms
*
*****************************************************************************************************/

#include "PVRScopeComms.h"

// Setup an enum of custom counters to make life easier later
enum PVRScopeCustomCounters
{
    eARBITRARY COUNTER,
    eNUM CUSTOM COUNTERS
};

// Step 1. Setup PVRScopeComms
SSPSCommsData* PVRScopeComms;
std::string timelineTitle = "Example";
PVRScopeComms = pplInitialise(timelineTitle,
                              (unsigned int)timelineTitle.length());

// Step 2. Create a Custom Counter
std::string customCounterName = "";
SSPSCommsCounterDef PVRScopeCustomCounterArray[eNUM CUSTOM COUNTERS];
for(unsigned int i = 0; i < (unsigned int)eNUM CUSTOM COUNTERS; ++i)
{
    switch (i)
    {
    case eARBITRARY COUNTER:
        customCounterName = "Arbitrary Counter";
        break;
    default:
        customCounterName = "Unnamed counter";
        break;
    }
    PVRScopeCustomCounterArray[eARBITRARY COUNTER].pszName = customCounterName.c str();
    PVRScopeCustomCounterArray[eARBITRARY COUNTER].nNameLength = customCounterName.length();
}

// Step 3. Submit the Custom Counter to PVRTune
if(!pplCountersCreate(*PVRScopeComms, PVRScopeCustomCounterArray, eNUM CUSTOM COUNTERS))
{
    // Error handling goes here
}

// Step 4. Send an initial value for that counter to PVRTune
int newCounterValue = 0;
unsigned int newValueArray[eNUM CUSTOM COUNTERS];
for(unsigned int i = 0; i < (unsigned int)eNUM_CUSTOM_COUNTERS; ++i)
{
    newValueArray[eARBITRARY_COUNTER] = newCounterValue;
}
// Submit counters for transmission
pplCountersUpdate(*PVRScopeComms, newValueArray);

// Step 5. Update the counter once and send the updated counter to PVRTune
newCounterValue = 100;
for(unsigned int i = 0; i < (unsigned int)eNUM CUSTOM COUNTERS; ++i)
{
    newValueArray[eARBITRARY COUNTER] = newCounterValue;
}
// Submit counters for transmission
pplCountersUpdate(*PVRScopeComms, newValueArray);

// Step 6. Shutdown PVRScopeComms
pplShutdown(PVRScopeComms);
```

B.4. Sending and Retrieving Editable Data

```
/*****************************************************************************************************
*
* This example code consists of six steps:
*
* 1. Create an array of editable items
* 2. Setup PVRScopeComms
* 3. Create default values for the editable items
* 4. Submit the array of editable items to PVRTune
* 5. Retrieve the updated items from PVRTune
* 6. Shutdown PVRScopeComms
*
*****************************************************************************************************/

#include <stdlib>
#include "PVRScopeComms.h"

// Step 1. Create an array of editable items
enum PVRScopeEditableFloats
{
    eARBITRARY EDITABLE FLOAT,
    eNUMBER OF EDITABLE FLOATS
};
enum PVRScopeEditableStrings
{
    eARBITRARY EDITABLE STRING,
    eNUMBER OF EDITABLE STRINGS
};
const unsigned int numEditableItems = (unsigned int)eNUMBER_OF_EDITABLE_FLOATS
                                    + (unsigned int)eNUMBER_OF_EDITABLE_STRINGS;
SSPSCommsLibraryItem editableItemArray[numEditableItems];

// Step 2. Setup PVRScopeComms
SSPSCommsData* PVRScopeComms;
std::string timelineTitle = "Example";
PVRScopeComms = pplInitialise(timelineTitle,
                              (unsigned int)timelineTitle.length());

// Step 3. Create default values for the editable items
float arbitraryFloat = 3.5f;
float arbitraryString = "Arbitrary String";
for(unsigned int i = 0; i < numEditableItems; ++i)
{
    // If the editable item is a float
    if(i < eNUMBER OF EDITABLE FLOATS)
    {
        // Create the float
        SSPSCommsLibraryTypeFloat scopeFloat;
        switch(i)
        {
        case eARBITRARY EDITABLE FLOAT:
            // Set its starting values
            editableItemArray[i].pszName = "Arbitrary Float";
            editableItemArray[i].eType   = eSPSCommsLibTypeFloat;
            scopeFloat.fCurrent          = arbitraryFloat;
            scopeFloat.fMin              = 0.0f;
            scopeFloat.fMax              = 200.0f;
            break;
        default:
            return false;
        }

    // Add the float object to the editable item
    editableItemArray[i].itemData = (const char*)&scopeFloat;
    editableItemArray[i].nDataLength = sizeof(scopeFloat);
    }

    else  // The editable item is a string (because we're only using strings and floats)
    {
        switch(i - eNUMBER_OF_EDITABLE_FLOATS)
        {
        case eARBITRARY EDITABLE STRING:
            // Set its starting balues
            editableItemArray[i].pszName = "Arbitrary String";
            editableItemArray[i].eType = eSPSCommsLibTypeString;
```

```
            editableItemArray[i].itemData = arbitraryString.c_str();
            editableItemArray[i].nDataLength = arbitraryString.length();
            break;
        default:
            return false;
        }
    }
    editableItemArray[i].nNameLength = (unsigned int) strlen(editableItemArray[i].pszName);
}

// Step 4. Submit the array of editable items to PVRTune
if(!pplLibraryCreate(*PVRScopeComms, editableItemArray, numEditableItems))
{
                // Error handling goes here
}

// Sleep for a bit so we have time to receive some data from PVRTune
Sleep(1000);

// Step 5. Retrieve the updated items from PVRTune
unsigned int itemNum(0);
unsigned int itemLength(0);
const char* itemData(NULL);
while(pplLibraryDirtyGetFirst(*PVRScopeComms, itemNum, itemLength, &itemData))
{
    // Update the item referred to by 'itemNum' with 'data'
    if(itemNum < eNUMBER OF EDITABLE FLOATS)
    {
        switch(itemNum)
        {
        case eARBITRARY_EDITABLE_FLOAT:
        {
            SSPSCommsLibraryTypeFloat* newItemData = (SSPSCommsLibraryTypeFloat*)itemData;
            arbitraryFloat = newItemData->fCurrent;
            break;
        }
        default:
        {
            // Handle unknown float
            break;
        }
        }
    }
    else
    }
        switch(itemNum-eNUMBER OF EDITABLE FLOATS)
        {
        case eARBITRARY_EDITABLE_STRING:
        {
            arbitraryString.assign(itemData, itemLength);
            break;
        }
        default:
        {
            // Handle unknown string
            break;
        }
    }
 }

// Step 6. Shutdown PVRScopeComms
pplShutdown(PVRScopeComms);
```