Synchronised Multi-channel Recording

Use of the SPx library to record multiple channels of radar video into the same recording file

Summary

The standard SPx recording process was designed to record a single stream of radar video data into each recording file. With careful design and use of a non-public function of the SPx library it is possible record multiple radar video channels, synchronised, into the same recording file. This application note describes how to use the SPx library to perform multi-channel recording in this way. Note that Cambridge Pixel's RDR application is still the recommended solution for multi-channel recording.

Introduction

It may be desirable to record a number of separate radar video streams into a single recording file. For example, to ensure synchronisation is maintained on replay or simply to ease file management. Under normal operation, the standard SPx recording process ("SPxProRecord") writes data from a single radar stream into its output file. However, by setting up multiple input PIMs in a particular way, making use of callback functions and calling the main SPx recording function directly it is possible to record all of the radar streams into the same file.

Recording Design

An overview of the multi-channel recording design is shown in Figure 1. For each channel to be recorded there should exist a separate SPx source, RIB and PIM and process chain.

Null processes may be used as the processes at the end of each input chain. The null process does not actually effect the radar video data but is a standard SPx process and allows a "return handler" (i.e. callback) function to be installed. The return handler is called each time the process is presented with new data by its driving PIM. The same return handler should be installed for each input channel.

Within the return handler the code should detect which channel has called the handler and then use the "SPxProRecordReturn()" function to write the return data to disk. The channel index is supplied to this function as an argument.

A top-level SPxProRecord process needs to be present within the main application, even though it doesn't need to be connected to any of the input channel processing chains. This SPxProRecord process is passed to the SPxProRecordReturn() function as an argument and controls the recording filename, the compression mode and whether recording is actually active or not.

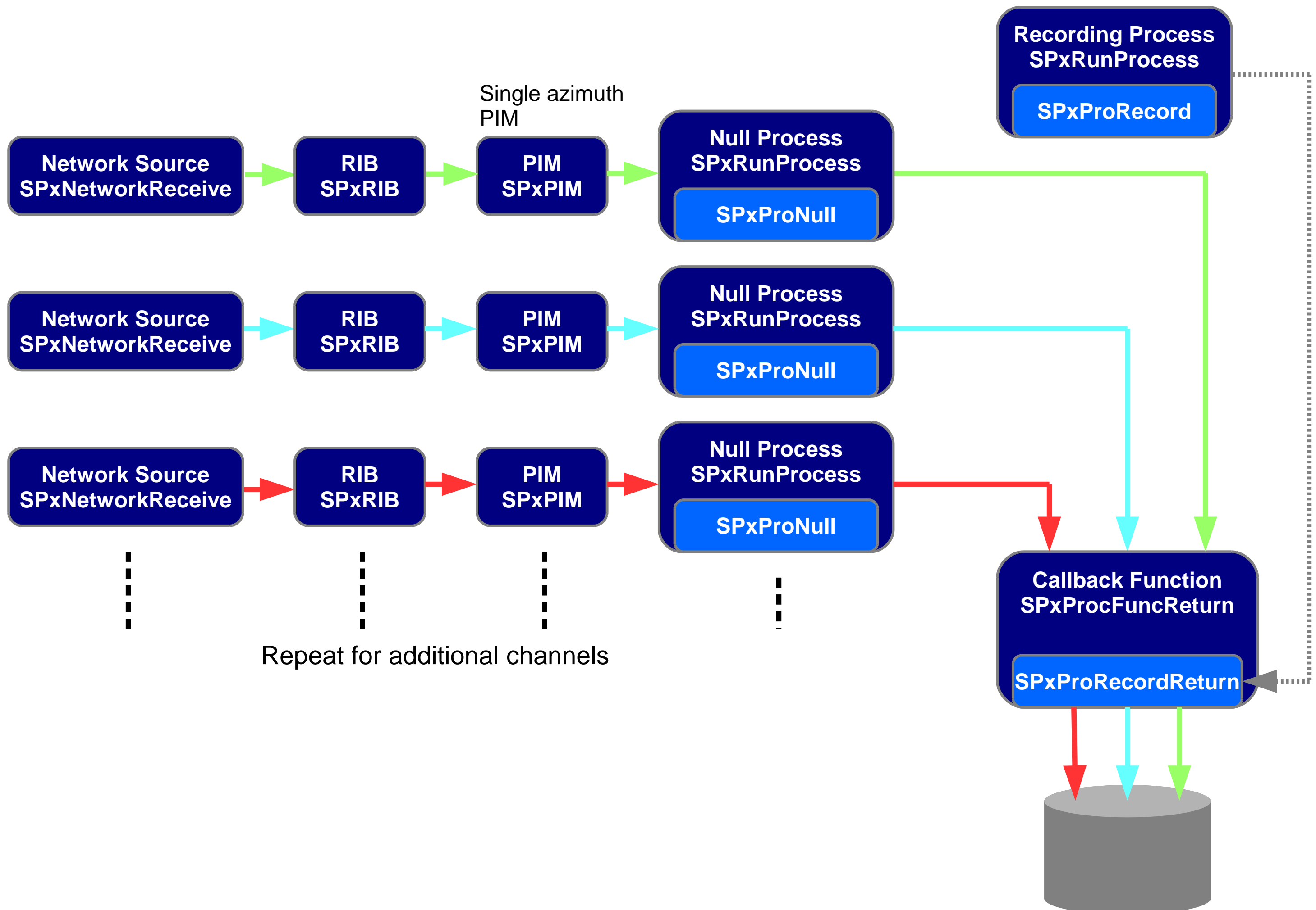

Figure 1: Overview of the multi-channel recording architecture

This design is scalable, as more input channels can simply be added by creating additional input chains and linking them to the same return handler.

Implementation Details

Installing a return handler on an SPx process and direct calling of the SPxProRecordReturn() function are non-standard but perfectly valid ways of using the SPx library. As such, the details are not documented in the standard SPx user manuals.

A standard SPxProRecord process needs to be created and the normal recording process initialisation function needs to be called. The recording process does not need to be created with any input processes or PIMs, for example:

```
SPxProRecordInit();
m_recPro = new SPxRunProcess(SPxProRecord,NULL,NULL,NULL,0);
```

The input PIMs, for each recording channel, should each be defined as single-azimuth PIMs, for example:

```
m_recPimA = new SPxPIM(m_ribA, 2048, 1,
                       SPX_PIM_RAN_SUBSAMPLE,SPX_PIM_AZI_OVERWRITE,
                       SPX_PIM_INPUT(1),2);
```

Each of the PIMs should then feed a null process:

```
m_nullProA = new SPxRunProcess(SPxProNull, NULL, m_recPimA);
```

Each null process should have the same return handler installed upon it. The SPxRunProcess class provides a function to set this return handler function, which may be called as follows:

```
m_nullProA->SetReturnHandler(writeReturn, &userArg);
```

The first argument is a pointer to the return handler function to be installed and the second argument is then a pointer to an argument that gets passed to this return handler. The prototype of the return handler (called "writeReturn" in the example above) is defined, in SPxProcess.h, as:

```
typedef void (*SPxProcFuncReturn)(void *userArg,   /* User arg. */
                          SPxRunProcess *rp,        /* Run process. */
                          unsigned int firstAzi,    /* First azi index. */
                          unsigned int numAzis);    /* Number of azis. */
```

Where:

userArg is the user argument, set when the handler is installed

rp is a pointer to the instance of SPx run process that called the handler

firstAzi is the index of the first azimuth that the handler has been called for

numAzis is the number of azimuths

It is important to note that the run process that invokes the handler, and is passed as the second argument to SPxRunProcess::SetReturnHandler(), is the null process of the input channel not the global recording process. The user argument must be used to pass in information about a) the global recording process and b) the channel number. The details of this will depend on how the application is written, but one approach might be to pass the "this" pointer of the parent object as the user argument and then let the handler function identify the recording process and channel index. Another approach could be to create a data structure that contains both pieces of information and pass this.

The body of the return handler needs to be written and the exact details will very much depend on the wider design of the recording application. However, the section below describes what the return handler needs to do, as a minimum.

It's advisable to update the "table of contents" (TOC) in the recording file, as the process progresses. The TOC is held at the start of the recording file and contains information mapping recording time to byte positions in the recording file. By keeping the TOC regularly updated the program may support simultaneous writing to- and replaying from- the same recording file. The TOC update interval is set, in seconds, via the recording process's "TocWriteIntervalSecs" parameter and a value of 10 seconds is suggested. For example:

```
m_recPro->SetParamValueI("TocWriteIntervalSecs", 10);
```

Return Handler (Callback)

The return handler is invoked with a user argument, a pointer to the run process that actually called it and information about the azimuth (within the driving PIM), as described above. Given a pointer to the run process that invoked the callback, it is straightforward to access the driving PIM:

```
SPxPIM *pim = rp->GetInputPIM();
```

Once the PIM has been identified, the radar return for a given azimuth index within that PIM can be accessed as follows:

```
SPxReturn *rtnPtr = pim->GetStoreMemory()[firstAzi];
```

The azimuth index ("firstAzi" in the example call above) may simply be the third calling argument of the return handler. Because the driving PIM is a single azimuth wide, the return handler is called for each return and is provided with data for a single return. Furthermore, the azimuth index should always be 0 (zero). This can be checked-for within the return handler code:

```
if (firstAzi != 0)
{
        /* Handle error. */
        return;
}
```

Note: even though the PIM azimuth value will always be zero in this case the true azimuth index is still set within the SPxReturn header.

The return handler ultimately uses the globally available SPxProRecordReturn() function to write the incoming radar data returns to disk. The function has the following prototype (declared in "SPxProcess.h"):

```
SPxErrorCode SPxProRecordReturn(SPxRunProcess *rp,
                                SPxReturn *rtn,
                                UCHAR channelIndex=0);
```

This function needs to be supplied with a pointer to the global recording process, a pointer to the current return to write, and given a unique index number for each channel that calls it.

Crucially, the same return handler is invoked by each of the null processes that installed it, using the SPxRunProcess::SetReturnHandler() function, each time they receive new data from their driving PIM. Since the same function is called by each of the input channels, it needs some way to distinguish which channel actually called it, so that this information may be passed onto the SPxProRecordReturn() function.

The exact details of how to retrieve a pointer to the global recording process and how to determine the current channel will depend on the structure and design of the overall application. Such details are therefore beyond the scope of this document; however, one approach could be to pass a pointer to a parent object as the user argument when calling the SPxRunProcess::SetReturnHanlder() function. Assuming that this parent object has oversight of the recording process and each of the null processes, this will then provide the necessary link to the information that SPxProRecordReturn needs. For example:

```
/* Cast the user argument to the actual data type. */
MyClass *obj = static_cast<MyClass *>(userArg);

/* Make sure the object contains the recording process */
if(obj->m_recPro)
{
   /* Record this return using the ID for this channel. */
    SPxProRecordReturn(obj->m_recPro, rtnPtr, obj->m_chanIndex);
}
```

A full example return handler is given below:

```
void MyClass::returnHandler(void *userArg,
                            SPxRunProcess *rp,
                            unsigned int firstAzi,
                            unsigned int numAzis)
{
   /* Get our object. */
   MyClass *obj = static_cast<MyClass *>(userArg);
   if (obj == NULL)
   {
       /* Handle error. */
       return;
   }
   /* Get the PIM. */
   SPxPIM *pim = rp->GetInputPIM();
   if (pim == NULL)
   {
       /* Handle error. */
       return;
   }
   /* We only have a single azimuth PIM so do a sanity check
      on firstAzi. */
   if (firstAzi != 0)
   {
       /* Handle error. */
       return;
   }
   /* Get the address of the input return and data. */
   SPxReturn *rtnPtr = pim->GetStoreMemory()[firstAzi];
   if (obj->m_recPro)
   {
       /* Record this return using the ID for this channel. */
       SPxProRecordReturn(obj->m_recPro, rtnPtr, obj->m_chanIndex);
   }
}
```

Replaying Files

Replaying files which contain data from multiple channels requires the software to filter its output based on the channel index. Fortunately, this is largely handled automatically by the SPxRadarReplay source object.

The first step is to create a replay source, with no PIM:

```
m_fileSrc = new SPxRadarReplay(NULL);
```

Now, for each channel create a separate RIB and PIM. Add the RIB for each channel to the replay source, using the SPxRadarReplay::AddChannel() function, specify the same channel index that was used for the channel during recording:

```
m_fileSrc->AddChannel(m_chanIndex, m_rib);
```

The replay source will automatically direct each packet to the correct RIB based in its channel index, and then onto the attached PIM and downstream processing, e.g. network distribution.

< End of document >