FastMultiWavePort:
A new processor for the LeCroy XStream Scopes

APPLICATION BRIEF
LAB WM785

February, 2011

Summary

FastMultiWavePort was constructed as the result of customer requests concerning rapid access to acquisition data and/or the need for rapid calculations (of a proprietary nature) to be fed back into the scope software.This application brief shows an example of how this feature is used.

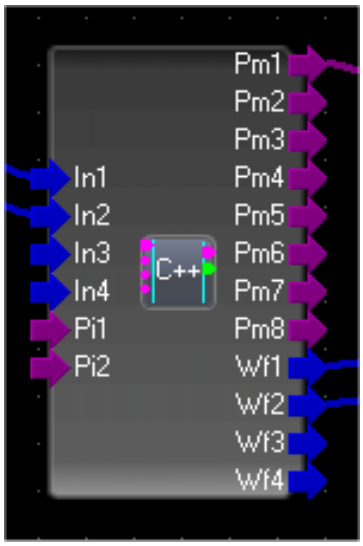

Figure 1: The processing web icon for The FastMultiWavePort processor

FastMultiWavePort:

The FastMultiWavePort processor allows another windows application to access waveform data from the Xstream processing and either re-inject results into the processing stream as parameter values or waveforms, or simply deal with the results of calculation independently of the scope software. In this later scenario it allows fast access to the scope's waveform data. All this is accomplished by sharing common memory between the scope and user application.

This new processor is based on the existing FastWavePortMath and is similar in many ways to the existing processor. FastMultiWavePort adds the following functionality:

- Support for 4 waveform inputs instead of 1 waveform input.
- Support for 8 parameter outputs compared to
- Support for batching an entire sequence of waveforms before the exchange between the scope and the client application (to reduce the overhead incurred for each exchange)
- Since multiple inputs and outputs are "designed in" there is no issue for synchronization of inputs and outputs.
- Client application decides how much named memory is mapped and for which named memory spaces, as opposed to fixed 80Mbyte allocation per waveform.
- The data structures for both waveform and parameters support multiple results for a single output pin (i.e. multiple waveform "segments" and multi-valued parameters are supported.)

Figure 1 shows the processing web icon for the FastMultiWavePort processor showing the 8 parameter outputs and 4 waveform outputs

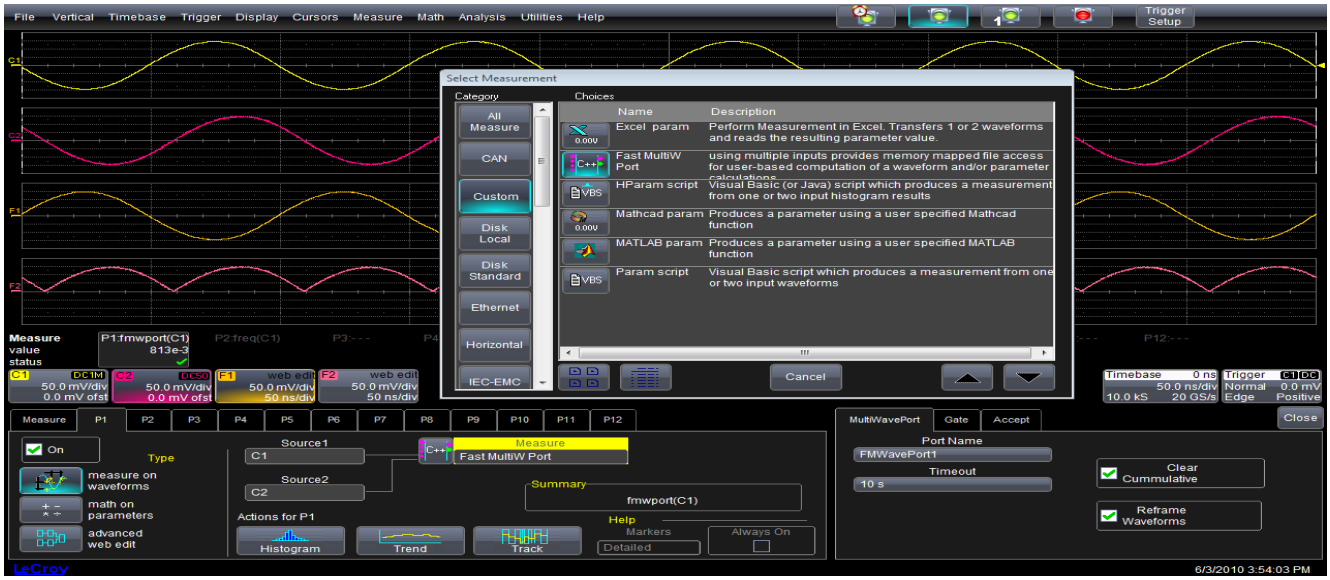

Figure 2: The measure setup showing the FastMultiWavePort setup.

FastMultiWavePort can be accessed either from the measure setup (see Figure 2) or via the processing web (Figure 3) . In either case it is only selectable from the custom measurement category. The choice of which setup method to use depends on what outputs you wish to use. If you use the measurement setup you will have access to the waveform via the shared memory and be able to return a single parameter to the scope. Using the processing web allows the user to access any of the 8 parameters and 4 waveform outputs.

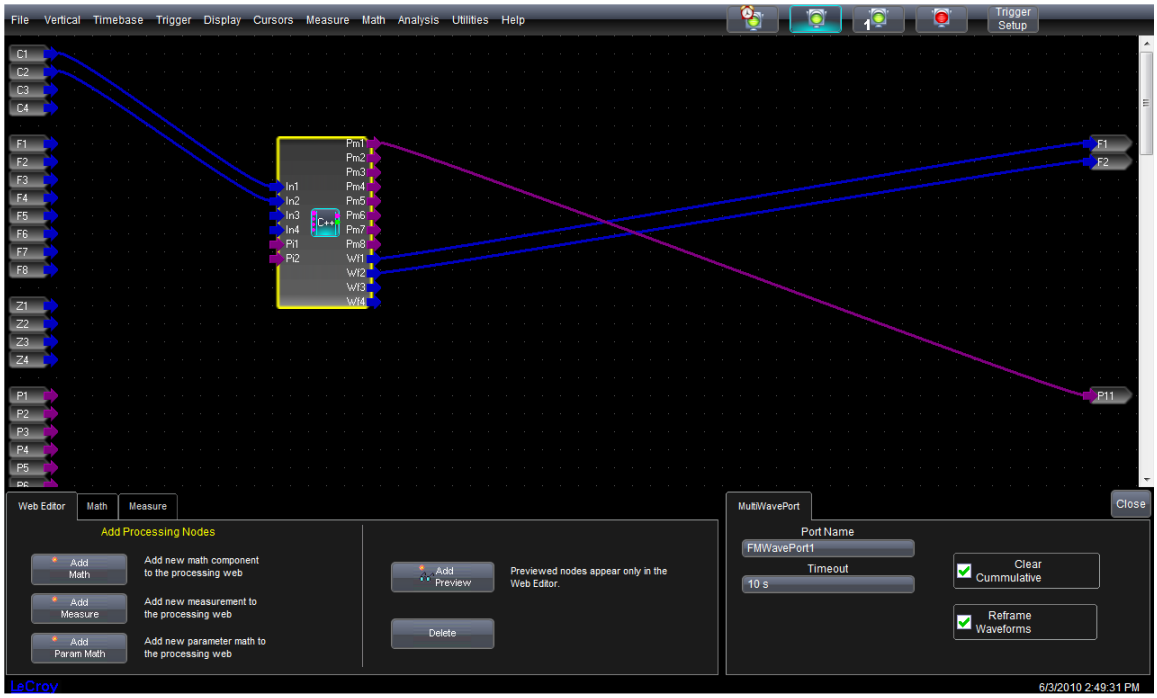

Figure 3: FastMultiWavePort processor in the processing web

Sharing memory to achieve mutual access to processed data

Like its predecessor, the FastMultiWavePort processor relies on a standard Windows feature called "Creating Named Shared Memory"

Multiple waveforms are stored in the shared memory with an overall descriptor similar to the one used for the FastWavePortMath processor, but which has been slightly modified, and is supplied in the "MultiWavePort.h" header file. There is also a "mini-descriptor" block which contains incremental information to characterize each additional waveform data block.

Similarly a descriptor header has been provided for parameter results, and a protocol for supplying parameter values.

The client application is responsible for creating the "named memory" spaces which are used to communicate between processes. The base name of the mapped memory is specified in the control variable of the processing function (in the scope) and by default is "FMWavePort1". This is shown in Figure 3. The FastMultiWavePort section of the manual contains complete details of the stored waveform and header structure.

From this point we will be using a C++ programming example from the manual repeated at the end of this application brief. This example sets up a parameter (P1) measuring a correlation coefficient of the waveforms in channels 1 and 2. It also inverts channel 1 and takes the absolute value of channel 2 and returns them to the scope to be output as function traces F1 and F2.

The application program should be compiled offline and then executed on the scope first. Start the scope application. The FastMultiWavePort processor is setup using the processing web as shown in Figure 3.

Once the application is running it will provide an output screen as shown in Figure 4 reporting the processing status.

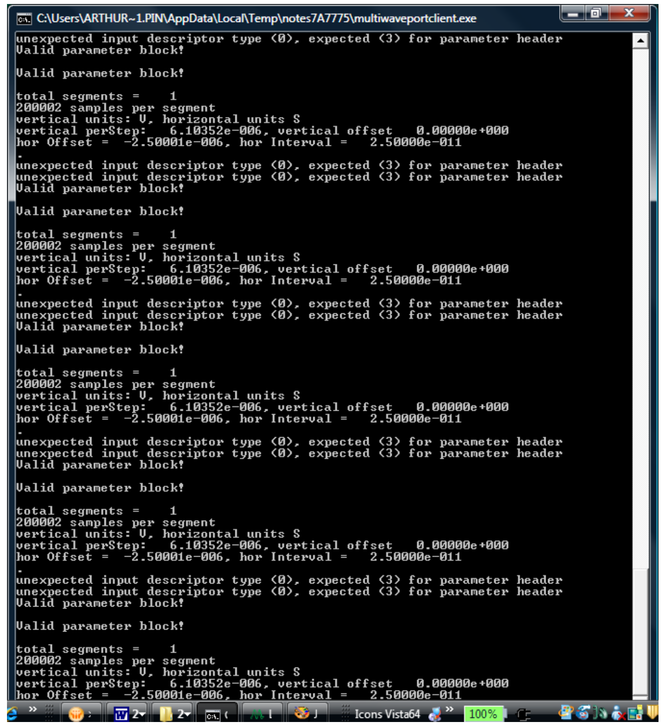

Figure 4: The application program output screen showing program status.

The scope output is shown in Figure 5.

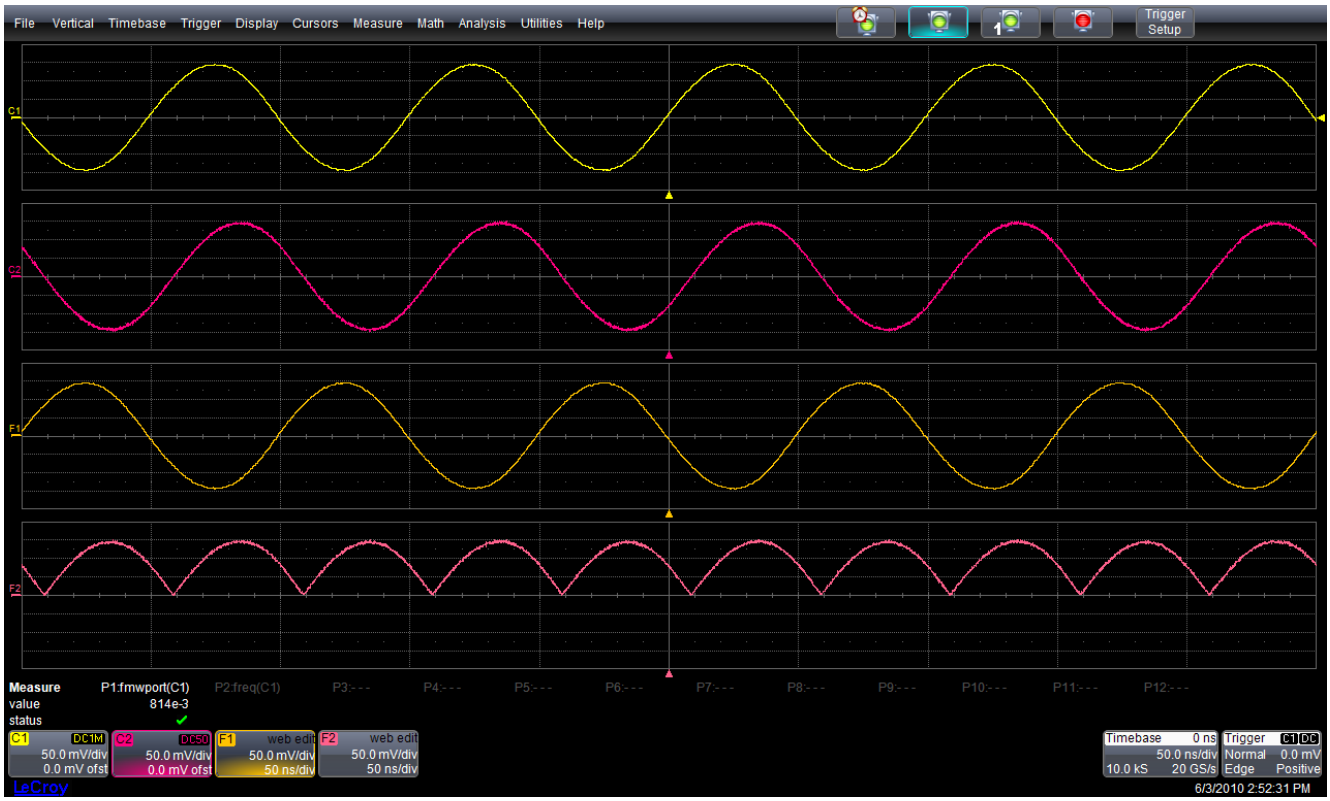

Figure 5: The FastMultiWavePort output showing the function traces F1 and F2 as well as the parameter P1.

Programming example:

```
//-------------------------------------------------------------------------------------
// FastWavePortClient.cpp : Defines the entry point for the console application.
//
// Prototype C++ client application for "Fast Wave Port' Math Processor
//
// Anthony Cake, March 2003
//
// Compatibility:
//      Microsoft Visual C++ 6.0, 7.1, 8.0
//      MinGW 'gcc' based compiler (free download from http://www.mingw.org/)
//          Compile with: mingw32-c++ -o fastWavePortClient.exe fastwaveportclient.cpp
//-------------------------------------------------------------------------------------

#include "windows.h"
#include "MultiPortClient.h"
#include <stdio.h>
#include <math.h>

//-------------------------------------------------------------------------------------
// FastWavePort header, describes various properties of the waveform passed to the user-processing
// function. Also used to carry the properties of the processed waveform back to the DSO.
//
//-------------------------------------------------------------------------------------
// Pentium read-time-stamp-counter instruction.
//#pragma warning(disable:4035)
#ifndef _WIN64
inline unsigned __int64 read64bitCounter()
{
   __asm rdtsc
}
#endif
//-------------------------------------------------------------------------------------

bool checkInt16WformHeader(CWaveDescHeader* descHeader)
{
          return (descHeader->descType == _int16WformDescriptor && descHeader->headerSize ==
CWaveDescHeaderSize);
}
//-------------------------------------------------------------------------------------
bool checkPmHeader(CParameterDescHeader* descHeader)
{
          return (descHeader->descType == _parameterDescriptor && descHeader->headerSize ==
CParameterDescHeaderSize);
}

//-------------------------------------------------------------------------------------
```

```
// The buffer size is 80MB (40,000,000 samples, stored as short integers) plus 0x1000 bytes for the header.
const unsigned long HEADER_SIZE = 0x1000;
const unsigned long MEM_MAP_FILE_SIZE_WAVEFORM = 80000000 + CWaveDescHeaderSize;
        // = 40MSamples, or 80MBytes
const unsigned long MEM_MAP_FILE_SIZE_PARAMETER = 1000 * CParameterValueSize +
CParameterDescHeaderSize;    // some space

int main(int argc, char* argv[])
{
   // names based on 'FastWavePort1' name defined in Processor setup.
   char szMapFileName1[]                = "FMWavePort1FileIn1";
        char szMapFileName2[]                = "FMWavePort1FileIn2";

        // additional parameter input pins (July 2009)
   char szMapFileNameP1[]               = "FMWavePort1FileInParam1";
        char szMapFileNameP2[]               = "FMWavePort1FileInParam2";

        char szMapFileName3[]                = "FMWavePort1FileOut1";   // first parameter output
        char szMapFileName4[]                = "FMWavePort1FileOut2";   // second parameter output
   char szMutexDataAvailableName[]      = "FMWavePort1MutexDataAvailable";
   char szMutexProcessingCompleteName[]    = "FMWavePort1MutexProcessingComplete";

   // Associate shared memory file handle value.
      HANDLE m_hMMFile1 = CreateFileMapping (INVALID_HANDLE_VALUE, NULL, PAGE_READWRITE, 0,
MEM_MAP_FILE_SIZE_WAVEFORM, szMapFileName1);
      if(m_hMMFile1 == 0)
      {
         printf("Unable to create file1 mapping\n");
         return 0;
      }

   // Map a view of this file for writing.
      short *m_lpMMFile1 = (short *)MapViewOfFile (m_hMMFile1, FILE_MAP_ALL_ACCESS, 0, 0,   0);
      if(m_lpMMFile1 == 0)
      {
         printf("Unable to map view of file1\n");
         return 0;
      }
   // Associate shared memory file handle value.
      HANDLE m_hMMFile2 = CreateFileMapping (INVALID_HANDLE_VALUE, NULL, PAGE_READWRITE, 0,
MEM_MAP_FILE_SIZE_WAVEFORM, szMapFileName2);
      if(m_hMMFile2 == 0)
      {
         printf("Unable to create file2 mapping\n");
         return 0;
      }
```

```
   // Map a view of this file for writing.
      short *m_lpMMFile2 = (short *)MapViewOfFile (m_hMMFile2, FILE_MAP_ALL_ACCESS, 0, 0,   0);
      if(m_lpMMFile2 == 0)
      {
         printf("Unable to map view of file2\n");
         return 0;
      }

   // Associate shared memory file handle value for 1st parameter input.
      HANDLE m_hMMFileParam1 = CreateFileMapping (INVALID_HANDLE_VALUE, NULL,
PAGE_READWRITE
                                    , 0,  MEM_MAP_FILE_SIZE_PARAMETER, szMapFileNameP1);
      if(m_hMMFileParam1 == 0)
      {
         printf("Unable to create param file1 mapping\n");
         return 0;
      }

   // Map a view of this file for writing.
      short *m_lpMMFileParam1 = (short *)MapViewOfFile (m_hMMFileParam1, FILE_MAP_ALL_ACCESS
                                    , 0, 0,   0);
      if(m_lpMMFileParam1 == 0)
      {
         printf("Unable to map view of param file1\n");
         return 0;
      }

   // Associate shared memory file handle value for 2nd parameter input.
      HANDLE m_hMMFileParam2 = CreateFileMapping (INVALID_HANDLE_VALUE, NULL,
PAGE_READWRITE
                                    , 0,  MEM_MAP_FILE_SIZE_PARAMETER, szMapFileNameP2);
      if(m_hMMFileParam2 == 0)
      {
         printf("Unable to create param file2 mapping\n");
         return 0;
      }

   // Map a view of this file for writing.
      short *m_lpMMFileParam2 = (short *)MapViewOfFile (m_hMMFileParam2, FILE_MAP_ALL_ACCESS
                                    , 0, 0,   0);
      if(m_lpMMFileParam2 == 0)
      {
         printf("Unable to map view of param file2\n");
         return 0;
      }
   // Associate shared memory file handle value.
      HANDLE m_hMMFile3 = CreateFileMapping (INVALID_HANDLE_VALUE, NULL, PAGE_READWRITE, 0,
MEM_MAP_FILE_SIZE_PARAMETER, szMapFileName3);
```

```
    if(m_hMMFile3 == 0)
    {
       printf("Unable to create file3 mapping\n");
       return 0;
    }

  // Map a view of this file for writing.
    short *m_lpMMFile3 = (short *)MapViewOfFile (m_hMMFile3, FILE_MAP_ALL_ACCESS, 0, 0,   0);
    if(m_lpMMFile3 == 0)
    {
       printf("Unable to map view of file3\n");
       return 0;
    }
   // Associate shared memory file handle value.
    HANDLE m_hMMFile4 = CreateFileMapping (INVALID_HANDLE_VALUE, NULL, PAGE_READWRITE, 0,
MEM_MAP_FILE_SIZE_PARAMETER, szMapFileName3);
    if(m_hMMFile4 == 0)
    {
       printf("Unable to create file4 mapping\n");
       return 0;
    }
 // Map a view of this file for writing.
    short *m_lpMMFile4 = (short *)MapViewOfFile (m_hMMFile4, FILE_MAP_ALL_ACCESS, 0, 0,   0);
    if(m_lpMMFile3 == 0)
    {
       printf("Unable to map view of file4\n");
       return 0;
    }
  // create/open events used for synchronization
  // if the client app. was run before the scope then these events will be created, if the scope was run first then
these events
  // will just be opened
    HANDLE m_hDataAvailable = CreateEvent(NULL, FALSE, FALSE /* initial state */,
szMutexDataAvailableName);
    HANDLE m_hProcessingComplete = CreateEvent(NULL, FALSE, FALSE /* initial state */,
szMutexProcessingCompleteName);
    if(m_hDataAvailable == 0 || m_hProcessingComplete == 0)
    {
       printf("Unable to open events\n");
       return 0;
    }

        //  install appropriate descriptors for two waveform results
                {
                        CWaveDescHeader* descWaveformHeader[2];
                        descWaveformHeader[0] = (CWaveDescHeader *) m_lpMMFile1;
                        descWaveformHeader[1] = (CWaveDescHeader *) m_lpMMFile2;
                        for(int i = 0; i < 2; ++i)
```

```
                {
                        descWaveformHeader[i]->descType = _int16WformDescriptor;
                        descWaveformHeader[i]->descVersion = 1;
                        descWaveformHeader[i]->headerSize = CWaveDescHeaderSize;
                        descWaveformHeader[i]->windowSize = MEM_MAP_FILE_SIZE_WAVEFORM;
                }


                // two parameter inputs need preparation (if processor is to use them)
                CParameterDescHeader* descInputParamHeader[2];

                descInputParamHeader[0] = (CParameterDescHeader *) m_lpMMFileParam1;
                descInputParamHeader[1] = (CParameterDescHeader *) m_lpMMFileParam2;

                for(int i = 0; i < 2; ++i)
                {
                        descInputParamHeader[i]->descType = _parameterDescriptor;
                        descInputParamHeader[i]->descVersion = 1;
                        descInputParamHeader[i]->headerSize = CParameterDescHeaderSize;
                        descInputParamHeader[i]->windowSize =
MEM_MAP_FILE_SIZE_PARAMETER;
                }

                // two parameter outputs for starters ... up to 8 are possible
                CParameterDescHeader* descParamHeader[2];

                descParamHeader[0] = (CParameterDescHeader *) m_lpMMFile3;
                descParamHeader[1] = (CParameterDescHeader *) m_lpMMFile4;

                for(int i = 0; i < 2; ++i)
                {
                        descParamHeader[i]->descType = _parameterDescriptor;
                        descParamHeader[i]->descVersion = 1;
                        descParamHeader[i]->headerSize = CParameterDescHeaderSize;
                        descParamHeader[i]->windowSize =  MEM_MAP_FILE_SIZE_PARAMETER;
                }


        }

  // main loop ... intereacting with the (FastMultiWavePort) processor in the scope
        int loopCount = 0;
    while(1)
    {
                if(loopCount % 64 == 0)
                        printf("\nWaiting.");
                else
                        printf(".");
```

```
                ++loopCount;

        // wait an infinite amount of time for data to be available
            DWORD waitSuccess = WaitForSingleObject(m_hDataAvailable, INFINITE);
            // look for the input parameter output block which should be ready-made ... but may
                        // or may not be valid (may have inputs connected or not)

                        int             numInputParameterValues[2] = {0, 0};
                        CParameterValue* myInputValuePointer[2];
                        myInputValuePointer[0] =
(CParameterValue*)&m_lpMMFileParam1[CParameterDescHeaderSize/sizeof(short)];
                        myInputValuePointer[1] =
(CParameterValue*)&m_lpMMFileParam2[CParameterDescHeaderSize/sizeof(short)];
                        CParameterDescHeader* descInputParamHeader[2];
                        descInputParamHeader[0] = (CParameterDescHeader *) m_lpMMFileParam1;
                        descInputParamHeader[1] = (CParameterDescHeader *) m_lpMMFileParam2;


                        // check to see if there are input parameters in the two input pins
                        bool    validInputParameterBlock[2] = {false, false};
                        for(int i = 0; i < 2; ++i)
                        {
                                lecDescType descType = descInputParamHeader[i]->descType;
                                int descVersion = descInputParamHeader[i]->descVersion;
                                int windowSize = descInputParamHeader[i]->windowSize;
                                int headerSize = descInputParamHeader[i]->headerSize;
                                if(descType == _parameterDescriptor && descVersion == 1 &&
headerSize == CParameterDescHeaderSize)
                                {
                                        validInputParameterBlock[i] = true;
                                        if(descInputParamHeader[i]->flags == LStatus_Valid)
                                        {
                                                // we're going to get some values  there
                                                printf("\nValid input parameter %1 block!\n", i + 1);
                                                int numValues = descInputParamHeader[i]->numValues;
                                                printf("\nThere are %1d values\n", numValues);
                                        }
                                }
                                else
                                {
                                        printf("\nunexpected input descriptor type (%d), expected (%d)
for parameter header", descInputParamHeader[i]->descType, _parameterDescriptor);
                                        myInputValuePointer[i] = NULL;
                                }
                        }

            // look for the parameter output block which should be ready-made ... but invalidated
```

```
                int             numParameterValues[2] = {0, 0};
                CParameterValue* myValuePointer[2];
                myValuePointer[0] =
(CParameterValue*)&m_lpMMFile3[CParameterDescHeaderSize/sizeof(short)];
                myValuePointer[1] =
(CParameterValue*)&m_lpMMFile4[CParameterDescHeaderSize/sizeof(short)];
                CParameterDescHeader* descParamHeader[2];
                descParamHeader[0] = (CParameterDescHeader *) m_lpMMFile3;
                descParamHeader[1] = (CParameterDescHeader *) m_lpMMFile4;


                // check some stuff to see if it's what and where we expect it
                bool    validParameterBlock[2] = {false, false};
                for(int i = 0; i < 2; ++i)
                {
                        lecDescType descType = descParamHeader[i]->descType;
                        int descVersion = descParamHeader[i]->descVersion;
                        int windowSize = descParamHeader[i]->windowSize;
                        int headerSize = descParamHeader[i]->headerSize;
                        if(descType == _parameterDescriptor && descVersion == 1 &&
headerSize == CParameterDescHeaderSize)
                        {
                                validParameterBlock[i] = true;
                                descParamHeader[i]->flags = LStatus_Valid;      // we're going to
put something there
                                printf("\nValid parameter block!\n");
                        }
                        else
                        {
                                printf("\nunexpected descriptor type (%d), expected (%d) for
parameter header", descParamHeader[i]->descType, _parameterDescriptor);
                                myValuePointer[i] = NULL;
                        }
                }

        // print the first few bytes of the input waveform
           CWaveDescHeader *descHeader = (CWaveDescHeader *) m_lpMMFile1;
                CWaveDescMiniHeader* descMiniHeader;
           CWaveDescHeader *descHeader2 = (CWaveDescHeader *) m_lpMMFile2;
                CWaveDescMiniHeader* descMiniHeader2;

                if(checkInt16WformHeader(descHeader) &&
checkInt16WformHeader(descHeader2))
                {
                        // protect against unterminated C-string ... jam a null char into the last
available spot
                        descHeader->verUnit[47] = '\0';
                        descHeader->horUnit[47] = '\0';
```

```
                descHeader2->verUnit[47] = '\0';
                descHeader2->horUnit[47] = '\0';

                int numSegments1 = descHeader->numSegments;
                int numSegments2 = descHeader2->numSegments;

                if (numSegments1 != numSegments2)
                {
                        printf("\n mismatch in number of segments = %4d != %4d\n",
numSegments1, numSegments2);
                }
                int numSegmentsCommon = min(numSegments1, numSegments2);

                printf("\ntotal segments = %4d\n", numSegments1);
                printf("%6d samples per segment\n", descHeader->numSamples);
                printf("vertical units: %s, horizontal units %s\n", descHeader->verUnit,
descHeader->horUnit );
                printf("vertical perStep: %14.5e, vertical offset %14.5e\n", descHeader-
>verGain, descHeader->verOffset );

                double horOffset = descHeader->horOffset;

                printf("hor Offset = %14.5e, hor Interval = %14.5e\n", descHeader-
>horOffset, descHeader->horInterval);

                // start and end of domain over which waveform spans. (note:
numIntervals = numSamples - 1)
                int     numSamples = descHeader->numSamples;
                int     numSamples2 = descHeader2->numSamples;

                int numSamplesCommon = min(numSamples, numSamples2);

                // we set the domain to the common range, associated with the horizontal
domain of input 1
                double startOfDomain = descHeader->horOffset;
                double endOfDomain = descHeader->horOffset + descHeader-
>horInterval * (numSamplesCommon - 1);

                ////////////////////////////////////////////////////////////////////////////////
                // Here as an example we will calculate the Pearson Product-Moment
Coefficient
                //
                ////////////////////////////////////////////////////////////////////////////////
                if(validParameterBlock[0])
                {
                        strncpy(descParamHeader[0]->verUnit, "", 47);// set the output
units to none
```

```
                                        strncpy(descParamHeader[0]->horUnit, "S", 47); // set the
horizontal output units to seconds
                                        descParamHeader[0]->verResolution = 0.01;

                                        descParamHeader[0]->horResolution = descHeader-
>horResolution; // same as input resolution
                                }

                                short *m_lpWaveform = (short*)&m_lpMMFile1[CWaveDescHeaderSize /
sizeof(short)];
                                short *m_lpWaveform2 = (short*)&m_lpMMFile2[CWaveDescHeaderSize
/ sizeof(short)];

                                for(int segment = 0; segment < numSegments1; ++segment)
                                {
                                        if(segment > 0)
                                        {
                                                // for segments after the first one, some information may
(and probably does) change.
                                                descMiniHeader =
(CWaveDescMiniHeader*)m_lpWaveform;
                                                m_lpWaveform += CWaveDescMiniHeaderSize /
sizeof(short);
                                                descMiniHeader2 =
(CWaveDescMiniHeader*)m_lpWaveform2;
                                                m_lpWaveform2 += CWaveDescMiniHeaderSize /
sizeof(short);

                                                numSamples = descMiniHeader->numSamples;
                                                numSamples2 = descMiniHeader2->numSamples;
                                                numSamplesCommon = min(numSamples,
numSamples2);
                                                startOfDomain = descMiniHeader->horOffset;
                                                endOfDomain = descMiniHeader->horOffset +
descHeader->horInterval * (numSamplesCommon - 1);

                                                // check incomming hor offset and trigger time
                                                double horOffset = descMiniHeader->horOffset;
                                                lecTimeStamp trigTime = descMiniHeader->trigTime;

                                                int segmentIndex = descMiniHeader->segmentIndex;
                                        }
                                        // // Use this code to assure yourself there's something
interesting in the first few points of data
                                        //for(int i = 0; i < 4; ++i)
                                        //{
                                        //      printf("%f (%4d),", (m_lpWaveform[i] * descHeader-
>verGain) + descHeader->verOffset, m_lpWaveform[i]);
```

```
//		printf("%f (%4d,", (m_lpWaveform2[i] * descHeader2->verGain) + descHeader2->verOffset, m_lpWaveform2[i]);
//}
//printf("\n");

// as an example ... compute the mean of all data values, while computing the abs value of the
// integer values for the waveform in-place
	double resultValue = 0.0;
	if (numSamplesCommon > 1)
	{
		double sumX = 0.0;
		double sumY = 0.0;
		for(int i = 0; i < numSamplesCommon; ++i)
		{
			double xValue = (m_lpWaveform[i] * descHeader->verGain) + descHeader->verOffset;
			sumX += xValue;
			double yValue = (m_lpWaveform2[i] * descHeader2->verGain) + descHeader2->verOffset;
			sumX += xValue;
			sumY += yValue;

		}
		double meanX = sumX / (double) numSamplesCommon;
		double meanY = sumY / (double) numSamplesCommon;
		// second pass ... sorry, but we're not trying to be subtle
		double sumdXdY = 0.0;
		double sumdX = 0.0;
		double sumdY = 0.0;
		double sumdXdX = 0.0;
		double sumdYdY = 0.0;
		for(int i = 0; i < numSamplesCommon; ++i)
		{
			double xValue = (m_lpWaveform[i] * descHeader->verGain) + descHeader->verOffset;
			double yValue = (m_lpWaveform2[i] * descHeader2->verGain) + descHeader2->verOffset;
			double dX = xValue - meanX;
			double dY = yValue - meanY;
			sumdXdY += (dX * dY);
			sumdXdX += (dX * dX);
			sumdYdY += (dY * dY);
			sumdX += dX;
			sumdY += dY;
		}
		double varianceX = sumdXdX / (double)numSamplesCommon;
```

```
                                                double varianceY = sumdYdY /
(double)numSamplesCommon;
                                                double varianceProduct =  varianceX * varianceY;

                                                if(varianceProduct > 0.0)
                                                        resultValue = sumdXdY /
((double)(numSamplesCommon - 1) * sqrt(varianceProduct));

                                                // now modify both input waveforms in place.
                                                for(int i = 0; i < numSamplesCommon; ++i)
                                                {
                                                        m_lpWaveform[i] = -(m_lpWaveform[i]);
                                                        m_lpWaveform2[i] = abs(m_lpWaveform2[i]);
                                                }
                                        }

                                // for this example ... we install the average as a parameter result for
each segment
                                        if(validParameterBlock[0] && myValuePointer[0] != NULL)
                                        {
                                                // for each parameter calculated (and there could have
been more than 1 per waveform segment)
                                                myValuePointer[0]->value = resultValue;
                                                myValuePointer[0]->startOfRegion = startOfDomain;
                                                myValuePointer[0]->endOfRegion = endOfDomain;
                                                myValuePointer[0]->flags = LStatus_Valid;
                                                // move forward to the next parameter value
                                                ++myValuePointer[0];
                                                ++numParameterValues[0];
                                        }
                                        // advance to end of waveform
                                        m_lpWaveform += numSamples;
                                        m_lpWaveform2 += numSamples2;
                                }
                                descParamHeader[0]->numValues = numParameterValues[0];
                                char* endOfValues = (char*)myValuePointer[0];
                                sprintf(endOfValues,"end of values");


                        }
                        else
                                printf("\nunexpected descriptor type (%d), expected (%d)", descHeader-
>descType, _int16WformDescriptor);

        // use to flag that the output is not valid, increasing performance when
        // it is not necessary to read data back into the DSO
           //descHeader->flags |= LStatus_Invalid;
```

```
            // flag that processing is complete
                SetEvent(m_hProcessingComplete);
        }

    return 0;
}
```