Waveform Processing Modules for CVALID

Rob Russell

September 7th, 2006

1 Introduction

The CVALID processing program is used for generating VALID formatted output based on the range information extracted from raw waveform data. However, the type of output produced by CVALID was previously limited to the hard coded range calculation implemented in C++. Recent modifications allow the range, the energies, and a fourth user-defined field to be computed externally. The program used to do these computations outside of the CVALID program will be called a **module** in this document.

The only requirement for the use of a particular programming language when writing a module is that the language must have the capability to open files in the Linux file system and read in binary data. This implies that a processing module can be written in C, C++, FORTRAN, Perl, Python, IDL, Matlab, Octave, etc.

2 Overview

2.1 Connection and Synchronization

In order to pass data between CVALID and a module, a form of inter-process communication is used that utilizes FIFOs. A FIFO is a kind of bi-directional pipe that is represented in the file system with a file entry (like "`/tmp/myfifo`", for example). A FIFO is opened by two processes: one **reader** and one **writer**. A FIFO is not limited to a single reader/writer, but at least one reader and writer must be attached to a FIFO before it can be used. If a FIFO is opened without a corresponding reader or writer also attached, the call to open the FIFO will block until such a condition exists.

To synchronize the data passed from CVALID to a module, two FIFOs are used per module. One FIFO is used for data output from CVALID, while the second is used for data input to CVALID. Waveform data is passed from CVALID to a module, and then the module must pass the computed values (range, transmit and receive energies, and a fourth variable) back to CVALID. Failure to return the necessary values to CVALID from the module will result in a deadlock: CVALID will wait forever expecting a result (or until the connection between the module and the FIFO is broken, which results in a `SIGPIPE`). Synchronization is ensured because the module and CVALID both block while waiting to receive data from each other.

2.2 Capability and Limitations

Up to ten separate modules can be used to process any given raw data file at a time. The memory usage does not increase based on the number of external modules used (except for the memory needed to run each additional process, which should be minimal if written correctly). Using a module to compute range measurements does not appreciably increase the processing time when using a compiled C program (as in the included examples), but an interpreted language may certainly lengthen processing time.

The names of the FIFOs are hard-coded in CVALID, so programming a module must follow the defined naming convention. Also, each module must use its own pair of FIFOs, and must not share the same FIFOs as another module that will be running concurrently.

3 Configuring CVALID to use Modules

CVALID must be informed that modules are to be used for external computations instead of the built-in range computation. To do this, the **.ini** file must be modified (usually called "**process.ini**"). Below is an excerpt from an **.ini** file.

```
# The processing options
[processing]
# '0' for normal valid output using cvalid range calculation
# '1' for testing "window" mode (tx & rx in the first window)
# '2' for external module mode
procType=0
# The number of modules to be used
numExternal=0
# The names of the modules
externProg0=
externProg1=
externProg2=
externProg3=
externProg4=
externProg5=
externProg6=
externProg7=
externProg8=
externProg9=
```

Use of the labels `procType`, `numExternal`, and `externProgX` are explained in the following text.

3.1 procType Label

The `procType` label can have one of the following three values:

0 : The internal range calculation is used
1 : The range is calculated only using the first captured window (not normally used)
2 : One or more external modules will be specified and used

If '2' is selected, then at least one `externProgX` must also be specified.

3.2 numExternal Label

This label specifies the number of modules to use. This value can range from 0 to 10. Note: if the `procType` value is 0, then this value is ignored.

3.3 externProgX

Each of the ten labels (externProg[0-9]) specify what modules are to be used. It must be noted that the **order in which the modules are listed is important**. The first module corresponds to the first FIFO pair; the second corresponds to the second FIFO pair - and so on. It becomes obvious that each module must be aware of the order in which it is called, and programmed accordingly.

For example, if I have a module called `exampleModule` and I want to use it as the second module in my **.ini** file, I would assign it like so:

```
externProg1=exampleModule
```

When writing `exampleModule`, I would have to specify that it uses the **second** FIFO when coding it. If I switch the position of `exampleModule` with whatever module I have specified in `externProg0`, a deadlock will occur while CVALID is waiting for the first pair of FIFOs to be opened.

3.4 Example

Here is an example excerpt from an **.ini** file when using two modules:

```
# The processing options
[processing]
# '0' for normal valid output using cvalid range calculation
# '1' for testing "window" mode (tx & rx in the first window)
# '2' for external module mode
procType=2
# The number of modules to be used
numExternal=2
# The names of the modules
externProg0=firstReturn
```

```
externProg1=lastReturn
externProg2=
externProg3=
externProg4=
externProg5=
externProg6=
externProg7=
externProg8=
externProg9=
```

This would produce two valid files, named with the extensions “firstReturn.vld” and “lastReturn.vld”. Each valid output file uses the corresponding name as read from the `externProgX` variables as extensions.

4 Writing a Module

There are only a few important things to know before writing a module to connect with CVALID : the data output from CVALID, the variables expected by CVALID, the names of the FIFOs that are used for communication and how to use them.

4.1 Input and Output Data Fields

CVALID produces the following fields as output for a module to produce a range, transmit and receive energies, and a fourth user-defined field.

Field #	Bytes	Type	Name
1	4	unsigned int	nTx
2	4	unsigned int	nRx
3	nTx	char	txDat
4	nRx	char	rxDat
5	8	double	hPos1
6	8	double	hPos2
7	4	unsigned long	ts1H
8	4	unsigned long	ts1L
9	4	unsigned long	ts2H
10	4	unsigned long	ts2L
11	8	double	gain
12	8	double	offset

Note: The number of bytes shown for each type could change if a non 32-bit system and compiler are used.

After a FIFO is opened, this data will be written to it, with the fields ordered numerically. Here is a description of the fields:

1. `nTx` - the number of samples in the transmit segment
2. `nRx` - the number of samples in the receive segment
3. `txData` - the actual sample data of the transmit segment
4. `rxData` - the actual sample data of the receive segment
5. `hPos1` - the first horizontal position variable (see the Acqiris documentation for more info)
6. `hPos2` - the second horizontal position variable (see the Acqiris documentation for more info)
7. `ts1H` - the high bits of the first timestamp (see the Acqiris documentation for more info)
8. `ts1L` - the low bits of the first timestamp (see the Acqiris documentation for more info)
9. `ts2H` - the high bits of the second timestamp (see the Acqiris documentation for more info)
10. `ts2L` - the low bits of the second timestamp (see the Acqiris documentation for more info)
11. `gain` - the gain value used during this particular acquisition (see the Acqiris documentation for more info)
12. `offset` - the voltage offset used during this particular acquisition (see the Acqiris documentation for more info)

The size of fields `txDat` and `rxDat` depend on the values `nTx` and `rTx`, which are read in first. The instructions for reconstructing a waveform using the data presented here can either be found in the Acqiris documentation or seen in the example range calculation modules that are now provided with CVALID.

As previously noted, CVALID will block and wait after writing this data to the FIFO, expecting the following data as written by the module:

Field #	Bytes	Type	Name
1	8	double	range
2	8	double	txE
3	8	double	rxE
4	8	double	custom

All four fields must be written to the FIFO by the module, or else CVALID will block and wait forever. The field descriptions are as follows:

1. `range` - the calculated range
2. `txE` - the transmit energy

3. `rxE` - the receive energy

4. `custom` - a custom variable that will be placed in the VALID output in word 11. In the examples provided with CVALID, the width of the return pulse is always placed in this field.

4.2 Using FIFOs

Each FIFO is represented by a file entry in the filesystem. A script in the `/src/external/` directory of the CVALID code repository called `mkallfifos.sh` will automatically create the FIFOs used by CVALID. If the script is missing, they can also be created manually with the shell command `mkfifo` (see the man pages). The format of the FIFOs are:

```
/tmp/vldoutX
/tmp/vldinX
```

with the postfix "X" being a number ranging from 0 to 9. CVALID writes *to* the "vldout" FIFOs and reads data *from* the "vldin" FIFOs. Conversely, a module reads data from the "vldout" FIFOs and writes data to the "vldin" FIFOs. When opening the FIFOs within a module, care must be taken to ensure that each FIFO is opened with the proper read/write permissions so that deadlock does not occur.

5 Example Module

The following portions of the example module "rangeExample.c" are explained to clarify the process of writing a module. For the entire code listing, please see the Appendix or check the `/src/external/` directory in the CVALID code repository.

5.1 Handling a SIGPIPE

When CVALID is finished using the module, it will generally close each opened FIFO that the module is using. In order to handle this gracefully (in the C language), we must install a signal handler to catch the `SIGPIPE` signal that is emitted when this occurs.

Different languages will have different methods for handling signals. If handling the signal is not done, then the module may have simply exited; however, catching the signal and closing

5.2 Opening the FIFOs

One of the next steps should include opening the FIFOs that will be used:

Each of the two pointers should probably be checked to ensure that the FIFOs indeed exist (as shown in the entire code listing). It must be noted

Algorithm 1 SIGPIPE handling

```
struct sigaction sa; // action to handle the SIGPIPE signal
// Set up the signal handler.
// When the connection to cvalid is broken, the signal will call handler()
memset( &sa, 0, sizeof(sa));
sa.sa_handler = &handler;
sigaction (SIGPIPE, &sa, NULL);
```

Algorithm 2 `Opening the FIFOs`

```
FILE * fifo_in = fopen("/tmp/vldout0", "r");
FILE * fifo_out = fopen("/tmp/vldin0", "w");
```

again that opening each FIFO with the proper read/write options is essential to prevent deadlock.

5.3 Reading Data from CVALID

The data written to `fifo_in` by `CVALID` should next be read in. Although not shown here, it is probably wise to check that the values read in for the number of transmit and return pulse samples are within known limits, so that dynamic memory allocation using bogus values isn't done.

5.4 Range Calculation

Now that the data has been read, the actual range calculation (as well as calculations for the other three variables) can be done. No part of those calculations are shown here; please refer to the actual code for details on how that might be done.

There are three examples that come with CVALID for doing range calculations. They are:

- rangeExample - produces the same range information as the native range calculation in CVALID
- firstReturn - determines the range based on the first return found
- lastReturn - determines the range based on the last return

The `firstReturn` and `lastReturn` examples have not been tested thoroughly, so the results may or may not be acceptable. `rangeExample` has been shown to produce identical results as the native range calculation.

Algorithm 3 Reading data

```
    fread( &numTxSamps, sizeof(numTxSamps), 1, fifo_in);
    fread( &numRxSamps, sizeof(numRxSamps), 1, fifo_in);
    // Allocate the TX and RX buffers according to the sizes that we just read in.
    tx = malloc(numTxSamps);
    rx = malloc(numRxSamps);
    // Clear the buffers
    memset(tx,0,numTxSamps);
    memset(rx,0,numRxSamps);
    // Read in the waveforms from the FIFO
    fread( (void*)tx, numTxSamps, 1, fifo_in);
    fread( (void*)rx, numRxSamps, 1, fifo_in);
    // Read in the remaining variables
    fread( (void*)&horpos1, sizeof(horpos1), 1, fifo_in);
    fread( (void*)&horpos2, sizeof(horpos2), 1, fifo_in);
    fread( (void*)&ts1H, sizeof(ts1H), 1, fifo_in);
    fread( (void*)&ts1L, sizeof(ts1L), 1, fifo_in);
    fread( (void*)&ts2H, sizeof(ts2H), 1, fifo_in);
    fread( (void*)&ts2L, sizeof(ts2L), 1, fifo_in);
    fread( (void*)&gain, sizeof(gain), 1, fifo_in);
    fread( (void*)&offset, sizeof(offset), 1, fifo_in);
```

5.5 Writing Data to CVALID

The next step should be returning the range, transmit and receive energies, and the fourth custom field back to CVALID. This can be done like so:

Algorithm 4 Returning variables

```
    fwrite( (void*)&range, sizeof(range), 1, fifo_out);
    fwrite( (void*)&txenergy, sizeof(txenergy), 1, fifo_out);
    fwrite( (void*)&rxenergy, sizeof(rxenergy), 1, fifo_out);
    fwrite( (void*)&pulseWidth, sizeof(pulseWidth), 1, fifo_out);
    // Since fwrite() is buffered, you MUST flush it or you will get a deadlock!
    fflush(fifo_out);
```

The use of `fwrite()` is being buffered; therefore, the buffers **must be flushed** after writing data to the FIFO. Failure to do this will leave CVALID in a waiting state, which will result in a deadlock. Different I/O libraries or languages may or may not buffer similarly.

5.6 Finishing Up

When cvalid is finished writing data to the FIFOs, it closes them, which triggers the `SIGPIPE` signal. In order to exit the module gracefully, use of a global flag to indicate that the main loop of the module should be exited can be set in the signal handler, `handler()`. After the FIFOs are closed, there is really nothing left to do but exit the module (and perhaps free any `malloc`'d memory, etc).

One final **important note: the module should immediately return (run in the background) when called so that deadlock does not occur.** This can easily be accomplished by creating a startup script containing the name of the executable with an ampersand afterward:

```
    myModule &
```

The small script above would be saved with the name that would be used in the **process.ini** file ("myModuleScript", for example). This means that the module **"myModule**" would not be named in **process.ini**; the name of the script, "myModuleScript", should be specified.

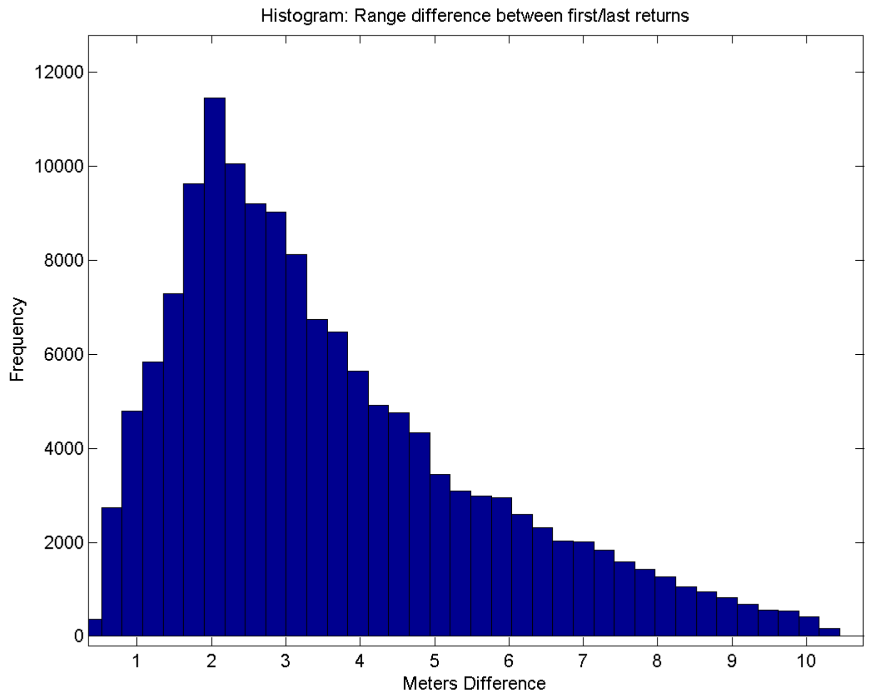
6 Applications + Notes

The three modules already mentioned (in 5.4) are implemented and available to be used with CVALID. When using `firstReturn` and `lastReturn` on a single data file recorded near the Yakutat, AK forelands, we can see that the difference between the first and last returns indicate features of the vegetation, with the most common separation between returns being about 2 meters.

In this example, the majority of the range measurements only show a single return (about 93.5%), with the last 6.5% having the following number of detected returns:

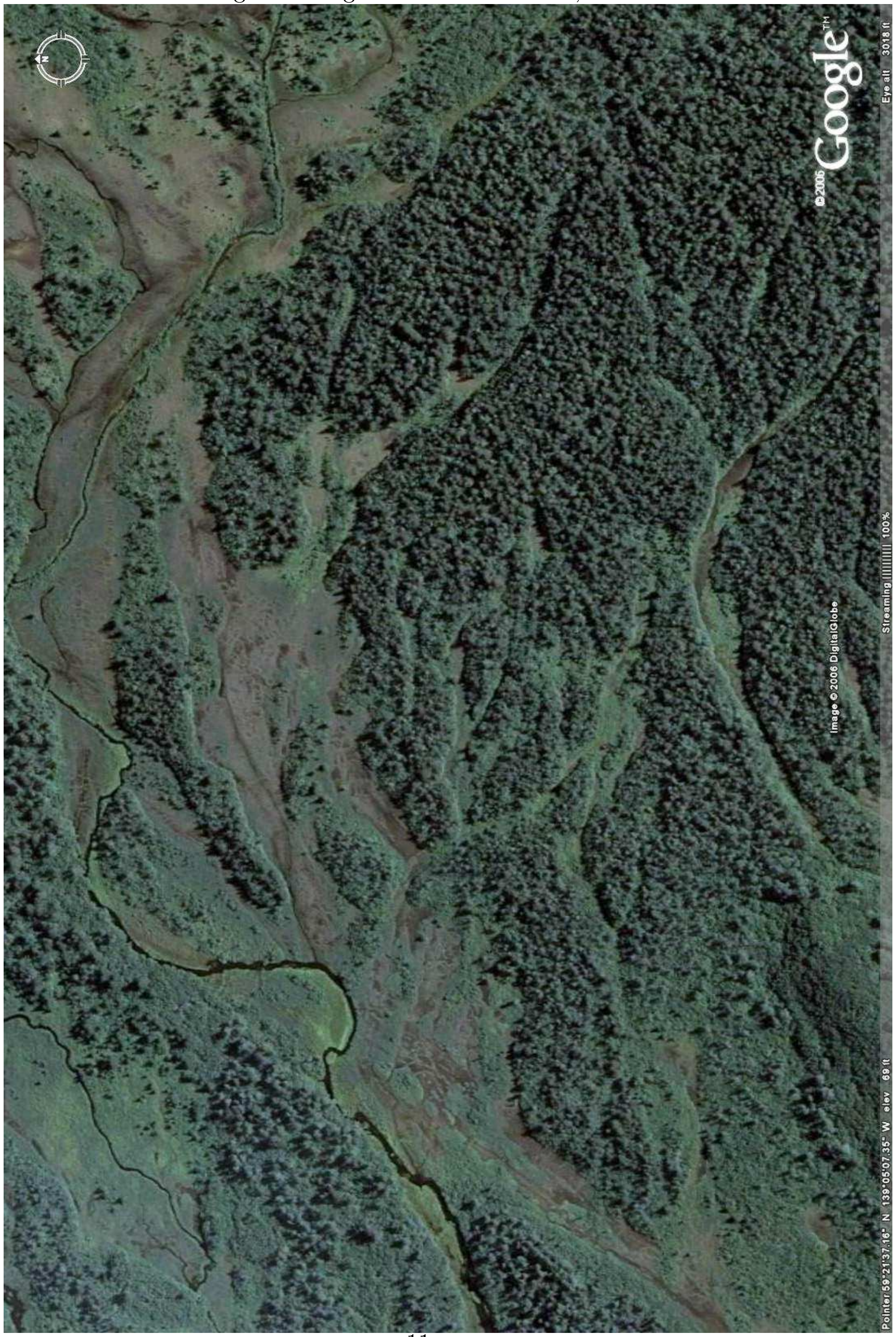
- 91.2% have 2 detected returns
- 9.3% have 3 detected returns
- $<$ 1% have 4 or 5 detected returns

Figure 1: Histogram of differences



More detailed analysis could be done using just these three modules; however, the `firstReturn` and `lastReturn` modules should be more thoroughly tested before being used for final products. The data used for this eample was taken over an area with scattered trees and brush, as shown in the following image.

Figure 2: Vegetation near Yakutat, AK


©2006 Google™
Image © 2006 DigitalGlobe
Streaming |||||||||| 100%
Pointer 59°21'37.16" N 139°05'07.35" W elev 69 ft
Eye alt 3018 ft

7 Appendix

The complete code of "rangeExample.c" can be found here, as well as in the CVALID code repository.

```
/** rangeExample.c
  *
  * Rob Russell
  * August 2006
  *
  * A module that reads in waveform data from a pipe and calculates the range.
  * This program uses the same method that is currently implemented in the cvalid
  * executable program, so the output using this 'processing module' should be identica
  * to the output of the native cvalid output.
  *
  * Please read the comments to understand how this works.
  */
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <signal.h>
#define MAX_SAMPLES 100000
// Prototype for the function that does the actual range calculation.
// If this example were used as the basis for a different 'processing module',
// this function could be the starting point.
void get_range(unsigned int,unsigned int,char*,char*,double, double, unsigned long, uns
// A flag to tell the main loop to exit when the pipe is broken
int EXITME = 0;
// The signal handler for a broken pipe
void handler(int signumber)
{
    EXITME = 1;
}
int main(char argc, char **argv)
{
// These variables are to be read in from the FIFO.
// The must be declared with the types shown (or equivalents in different languages)
unsigned int numTxSamps;        // Number of TX samples recorded
unsigned int numRxSamps;        // Number of RX samples recorded
char *tx;                       // Transmit waveform
char *rx;                       // Return waveform
double horpos1;                 // Horizontal position 1
double horpos2;                 // Horizontal position 2
unsigned long ts1H;             // High bits of 1st timestamp
unsigned long ts1L;             // Low bits of 1st timestamp
unsigned long ts2H;             // High bits of 2nd timestamp
unsigned long ts2L;             // Low bits of 2nd timestamp
double gain;                    // gain read from digitizer
double offset;                  // offset read from digitizer

// Related to the signal handler for SIGPIPE
struct sigaction sa;            // action to handle the SIGPIPE signal

// Set up the signal handler.
// When the connection to cvalid is broken, the signal will call handler() and end this
memset( &sa, 0, sizeof(sa));
sa.sa_handler = &handler;
sigaction (SIGPIPE, &sa, NULL);

// Other variables
int lastTxSamps = 0;
int lastRxSamps = 0;
int alreadyInitTx = 0;
int alreadyInitRx = 0;
// Open the FIFOs.  They must be created first by the cvalid program.
// The FIFOs can be opened exactly like a file, so ANY language that
// is able to handle files can also interface with cvalid.
// The 'vldout' FIFO relays output from cvalid to this module;
// 'vldin' is where we will send our computed values.

FILE * fifo_in = fopen("/tmp/vldout0", "r");
FILE * fifo_out = fopen("/tmp/vldin0", "w");
if (fifo_in == NULL || fifo_out == NULL)
{
    printf("One of the named FIFOs do not exist, exiting.\n");
    exit(1);
}
EXITME = 0;
    while( !EXITME ) // When the FIFO is closed by cvalid, this loop will exit
    {
    // Now read in the various waveform related data.
    // NOTE: The order in which these are read in is critical!
    // The ordering should follow the order in which they have been declared at the
    // top of this code - that is the same order in which they are written to the FIFO.

    fread( &numTxSamps, sizeof(numTxSamps), 1, fifo_in);
    fread( &numRxSamps, sizeof(numRxSamps), 1, fifo_in);

    // Allocate the TX and RX buffers according to the sizes that we just read in.
    // Clear them as well.  Also impose a limit so that bogus values don't cause proble
```

```
if (numTxSamps < 1 || numTxSamps > MAX_SAMPLES || numRxSamps < 1 || numRxSamps > MA
{
    printf("Too many/not enough samples requested in processing module, exiting...\
    exit(1);
}

// To prevent a new malloc for each waveform, only do it if the number of samples c
if (numTxSamps != lastTxSamps)
{
    if (alreadyInitTx) free (tx);
    tx = malloc(numTxSamps);
    alreadyInitTx = 1;
}
if (numTxSamps != lastTxSamps)
{
    if (alreadyInitRx) free(rx);
    rx = malloc(numRxSamps);
    alreadyInitRx = 1;
}
memset(tx,0,numTxSamps);
memset(rx,0,numRxSamps);
// Read in the waveforms from the FIFO
fread( (void*)tx, numTxSamps, 1, fifo_in);
fread( (void*)rx, numRxSamps, 1, fifo_in);
// Read in the remaining variables
fread( (void*)&horpos1, sizeof(horpos1), 1, fifo_in);
fread( (void*)&horpos2, sizeof(horpos2), 1, fifo_in);
fread( (void*)&ts1H, sizeof(ts1H), 1, fifo_in);
fread( (void*)&ts1L, sizeof(ts1L), 1, fifo_in);
fread( (void*)&ts2H, sizeof(ts2H), 1, fifo_in);
fread( (void*)&ts2L, sizeof(ts2L), 1, fifo_in);
fread( (void*)&gain, sizeof(gain), 1, fifo_in);
fread( (void*)&offset, sizeof(offset), 1, fifo_in);

// Now calculate a range, etc.
// cvalid expects these to be of type 'double' so they are declared as such.
// Of course, they are truncated to floats later when the actual 'valid'
// output is created.

double range=0;
double rxenergy=0;
double txenergy=0;
double pulseWidth=0;

get_range(numTxSamps, numRxSamps, tx, rx, horpos1, horpos2, ts1H, ts1L, ts2H, ts2L,
    // Return our Range, TX energy, RX energy, and another float that can be user defin
    // to cvalid via the FIFO. This user-defined float is placed in word 11 in the vali
    // In this example, that value will be the pulse width so as to imitate the
    // standard cvalid output.

    fwrite( (void*)&range, sizeof(range), 1, fifo_out);
    fwrite( (void*)&txenergy, sizeof(txenergy), 1, fifo_out);
    fwrite( (void*)&rxenergy, sizeof(rxenergy), 1, fifo_out);
    fwrite( (void*)&pulseWidth, sizeof(pulseWidth), 1, fifo_out);

    // Since fwrite() is buffered, you MUST flush it or you will get a deadlock!
    fflush(fifo_out);
    }
free(tx);
free(rx);
printf("Exiting rangeExample external program.\n");
// We are finished, so close the FIFOs
fclose(fifo_in);
fclose(fifo_out);
// End
return(0);
}
void get_range(unsigned int numTxSamps,unsigned int numRxSamps,char *tx,char *rx,double
{
    // The actual range calculation is done here.
    // This is the same as the calculation done in cvalid, so the output
    // should produce identical results.
    int minTxVal = 100000;
    int minRxVal = 100000;
    int minTxValPos = 0;
    int minRxValPos = 0;
    int minTxSampNum =0;
    int maxTxSampNum = 0;
    int minRxSampNum =0;
    int maxRxSampNum = 0;
    long numTxSamples = 0;
    long numRxSamples = 0;
    double xmtenergy = 0;
    double rcvenergy = 0;
    double txAmplTot = 0;
    double rxAmplTot = 0;
    int txInt[numTxSamps];
    int rxInt[numRxSamps];
    double rxCentroidPos = 0;
    double txCentroidPos = 0;
    long long timeStamp1, timeStamp2, tempStamp;
long long tsDiff;
double txCtVal;
double rxCtVal;
minTxVal = 256;
minRxVal = 256;

// Get the minimums

int i;
for (i = 0; i < numTxSamps; i++)
{
    txInt[i] = ((int)tx[i]) + 128;
    if ( txInt[i] < minTxVal)
    {
        minTxVal = txInt[i];
        minTxValPos = i;
    }
}
for (i = 0; i < numRxSamps; i++)
{
    rxInt[i] = ((int)rx[i]) + 128;
    if ( rxInt[i] < minRxVal)
    {
        minRxVal = rxInt[i];
        minRxValPos = i;
    }
}
minTxSampNum = minTxValPos;
maxTxSampNum = minTxValPos;
minRxSampNum = minRxValPos;
maxRxSampNum = minRxValPos;
// Reconstruct the timestamps
timeStamp1 = ts1H;
tempStamp = timeStamp1<<32;
timeStamp1 = tempStamp + (unsigned long)ts1L;
timeStamp2 =ts2H;
tempStamp = timeStamp2<<32;
timeStamp2 = tempStamp + (unsigned long)ts2L;
tsDiff = (timeStamp2 - timeStamp1);

// Set the threshold values
txCtVal = ((double)(256 - minTxVal) * 0.25)+ (double)minTxVal; // the byte value of
rxCtVal = ((double)(256 - minRxVal) * 0.25)+ (double)minRxVal; // the byte value of
while( (txInt[minTxSampNum]  < txCtVal) && (minTxSampNum > 0) )
{
    minTxSampNum--;
```

```
}
// Sub-sample value (left)
double txPercentByte = (double)(txCtVal-txInt[minTxSampNum+1])/(double)(txInt[minTx
double txFracValue = (double)(minTxSampNum+1) - txPercentByte;
//right
while ( (txInt[maxTxSampNum] < txCtVal) && (maxTxSampNum < 159) )
{
    maxTxSampNum++;
}
// Sub sample position (right)
double txPercentByteR = (double)(txCtVal-txInt[maxTxSampNum-1])/(double)(txInt[maxT
double txFracValueR = (double)(maxTxSampNum-1) + txPercentByteR;

// left
while( (rxInt[minRxSampNum] < rxCtVal) && (minRxSampNum > 0) )
{
    minRxSampNum--;
}
// Sub-sample value (left)
double rxPercentByte = (double)(rxCtVal - rxInt[minRxSampNum+1])/(double)(rxInt[min
double rxFracValue = (double)(minRxSampNum +1) - rxPercentByte;
//right
while( (rxInt[maxRxSampNum] < rxCtVal) && (maxRxSampNum < 159) )
{
    maxRxSampNum++;
}
// Sub sample value (right)
double rxPercentByteR = (double)(rxCtVal - rxInt[maxRxSampNum-1])/(double)(rxInt[ma
double rxFracValueR = (double)(maxRxSampNum -1) + rxPercentByteR;

// Get the width of (possibly) saturated signals
int minSatNumRx = minRxValPos;
int maxSatNumRx = minRxValPos;

int minSatNumTx = minTxValPos;
int maxSatNumTx = minTxValPos;

while( (rxInt[minSatNumRx] <= rxInt[minRxValPos]) && (minSatNumRx > 0) )
{
    minSatNumRx--;
}
while( (rxInt[maxSatNumRx] <= rxInt[minRxValPos]) && (maxSatNumRx < 159) )
{
    maxSatNumRx++;
}
while( (txInt[minSatNumTx] <= txInt[minTxValPos]) && (minSatNumTx > 0) )
```

```
{
    minSatNumTx--;
}
while( (txInt[maxSatNumTx] <= txInt[minTxValPos]) && (maxSatNumTx < 159) )
{
    maxSatNumTx++;
}
int satRxWidth = maxSatNumRx - minSatNumRx;
int satTxWidth = maxSatNumTx - minSatNumTx;
xmtenergy = 0;
txAmplTot = 0;

// Energy calculations
for (i = minTxSampNum+1; i < maxTxSampNum; i++)
{
    xmtenergy = xmtenergy + (256 - txInt[i]);
    txAmplTot = txAmplTot + i*(256 - txInt[i]);
    numTxSamples++;
}
numTxSamples = numTxSamples + 2;
xmtenergy = xmtenergy + txPercentByte*(256.0-txCtVal) + txPercentByteR*(256.0-txCtV
txAmplTot = txAmplTot + txFracValue*(256.0-txCtVal) +txFracValueR*(256.0-txCtVal) ;
rcvenergy = 0;
rxAmplTot = 0;

for (i = minRxSampNum+1; i < maxRxSampNum; i++)
{
    rcvenergy = rcvenergy + (256 - rxInt[i] );
    rxAmplTot = rxAmplTot + i*(256 - rxInt[i]);
    numRxSamples++;
}
numRxSamples = numRxSamples + 2;
rcvenergy = rcvenergy + (rxPercentByte+rxPercentByteR)*(256.0-rxCtVal);
rxAmplTot = rxAmplTot + rxFracValue*(256.0-rxCtVal) + rxFracValueR*(256.0-rxCtVal);
rxCentroidPos = ((double)rxAmplTot/(double)rcvenergy);
txCentroidPos = ((double)txAmplTot/(double)xmtenergy);

//  Temp variables
double c = 0.5 *2.99792458e8;
double range2 = 0;
double dig_rate = 5.e-10;
double time_offset = 0;
double tsd = (double)tsDiff;
double fact = 1.e-12;
double hp1 = horpos1;
double hp2 = horpos2;
```

```
    time_offset = fact * tsd - hp1 + hp2;
    // New (11/10/05) range calculation using the constant fraction leading edge method
    range2 = (c) * (time_offset + dig_rate * ( (double)(rxFracValue - txFracValue) ) );
    // 9/14/05 : Added a check for a saturated return pulse.  If received, then adjust
    // The value of 0.0062 and subtracting 5 came from investigating range walk (S. Man

    // New compenstaions for saturated transmit and return pulses when using the consta
    // method of range calculation
    if (satRxWidth >3 )
    {
        satRxWidth = satRxWidth - 3;
        range2 = range2 + (0.026 * (double)satRxWidth);
    }
    if (satTxWidth >3)
    {
        satTxWidth = satTxWidth -3;
        range2 = range2 - (0.026 * (double)satTxWidth);
    }

    // Set range, energies
    *range = range2;
    *txenergy = xmtenergy;
    *rxenergy = rcvenergy;
    // Set the pulseWidth value
    *pulseWidth = (double)(maxRxSampNum - minRxSampNum);
}
```