JR3 DSP Logger C++ interface source description March 1, 2021

Purpose:

This document is intended to describe the PCI interface code portion of the JR3 DSP logger. The target audience is users who would like to leverage the code, or implement in a language other than C++.

Proficiency (or at least exposure) to a high-level language will help understand the source, or at least help correlate the explanations to the code:

- Best background: C, C++
- Next-best: C#, Java, Perl, Python

This document will not discuss the GUI code.

Relevant files:

In this document, the term "the code," it refers to collectively:

- JR3PCILIB.cpp: the PCI card interface C++ source code.
- JR3PCIIoctls.h and JR3PCILIB.h: "supporting" code for JR3PCILIB.cpp.

When you see "5907 document" it refers to file _5907e_rec_man.pdf, which describes the interface model for the PCI card.

Variable and function naming conventions:

File JR3PCILIB.cpp and the supporting files contain work by a previous author who created a test utility for the JR3 PCI cards. Thus you will see two coding conventions.

Generally, variable names will begin with a variable type in lower-case, with the variable name in upper and lower case.

Types are:

Type		Example
h	handle	hJr3PciDevice
i	integer	iDataField
pus	pointer to unsigned short	pusForceSensorData
ul	unsigned long integer	ulOffset
us	unsigned short integer	usRdData

struct “force_sensor_data” (“fsd”):

A struct is a composite data type in C and C++. It occupies a contiguous area in memory. It contains typed fields which may be accessed similarly to simple variables.

In our use, the PCI channel memory is copied to force_sensor_data on demand. Subsequently any field in force_sensor_data may be used like a variable.

force_sensor_data generally follows the PCI card data map found in the 5907 document, Table 1: Summary of *JR3* DSP Data locations, as well as the individual field definitions in *JR3*'s DSP-based Force Sensor Receivers Data Locations and Definitions.

The following force_sensor_data struct fields differ from the 5907 document. In the document, these are created from smaller structs of individual bits. Due to difficulty defining these structs in Microsoft Visual C++, and because the JR3 logger does not use these fields, they have been defined as follows:

- short near_sat_value;
- short sat_value;
- short vect_bits_word;
- unsigned short warnings;
- unsigned short errors;

Using these field types, bitwise operators can be employed to operate on the individual bits.

PCI card detection and basic verification:

Note: The JR3 PCI card driver must be installed and operating to detect and enumerate PCI cards.

The JR3 driver supports up to four PCI cards. It enumerates the cards as "JR3PCI1," "JR3PCI2," "JR3PCI3" and "JR3PCI4."

The basic steps to ensuring a PCI card is present and working correctly are:

1. Attempt to establish connection to a preset PCI card name via a file handle.
2. Read 'copyright' data at offset 0x0040 and validate copyright data.
3. Write a pattern to a specific address in PCI memory, then read the value back.

Function "DiscoverCards()"

Synopsis:

1. Iterate through the four possible PCI card names
2. Call OpenHandleByIndex to attempt to open a handle to the current PCI card.
3. Call GetSupportedChannels to determine the number of channels this PCI card supports.
4. Validate the number of channels.
5. Iterate through each channel to:
   a. VerifyChannelCopyright
   b. VerifyReadWrite

The logger creates a long integer "ulReturnCode" to encode the presence and status of each possible PCI card. This is entirely a convention for our code to simplify passing the status to the GUI.

1w2x3y4z

w, x, y and z are the status for channels 1, 2, 3 and 4 respectively. The status is a digit which decodes to:

0 : Failed to open a communication handle (card probably absent)
1, 2, 4 : Number of channels (card present)

5 : Cannot determine number of channels
6 : Cannot verify copyright
7 : Card detected, failed read/write/verify.

Function "OpenHandleByName()"

Synopsis: Attempt to open communication to a designated PCI card using a file handle.

The syntax of this function may be cryptic, but it is well-commented so that even those not familiar with C++ can understand the arguments passed to "CreateFile()."

Ultimately, hJr3PciDevice will either be a valid file handle or will be "INVALID_HANDLE_VALUE."

Function "OpenHandleByIndex()"

Synopsis: Attempt to open communication to a designated PCI card using the card's index (the last digit of the card name; JR3PCI1 is index 1).

The card names are in array pciCardNames. Because arrays being at index 0 and our cards begin at index 1, the array index is "iCardIndex – 1."

Once the card name is extracted from the array, OpenHandleByIndex() calls OpenHandleByName() passing the card name as argument.

Function "CloseFileHandle()"

Synopsis: Closes file handle hJr3PciDevice if it is open.

Function "ReadStructure()"

Synopsis: Reads the content of the PCI card channel specified by argument "ulChannelIndex" to struct "fsd".

There are really two “ReadStructure()” functions/methods. This is because C++ permits function “overloading,” where the same function name may be used as long as the arguments differ.

ReadStructure(HANDLE hPciDevice, ULONG ulChannelIndex) takes two arguments, the handle to the PCI card, and the PCI card channel number. This is a more generic way to invoke this function.

```
bool ReadStructure(ULONG ulChannelIndex)
{
   return ReadStructure(hJr3PciDevice, ulChannelIndex);
}
```

This ReadStructure() implementation accepts one argument, the PCI card channel number. It then calls the more generic function, adding the handle “hJr3PciDevice” to the channel number argument. This works great for the JR3 logger because we only have one handle.

```
    ULONG ulNumWords = sizeof(fsd) / sizeof(short);
```

This line calculates an unsigned long integer of the number of words to be transferred from the PCI memory to the struct. It calculates the size of the force_sensor_data struct, divides that by the size of a short integer, resulting in unsigned long NumWords.

```
    short* pusForceSensorData = (short*)& fsd;
```

This code establishes a pointer to struct “fsd.” Transfering data via pointers is very efficient.

```
    for (ULONG ulOffset = 0; ulOffset < ulNumWords; ulOffset++)
    {
       pusForceSensorData[ulOffset] = ReadWord(hJr3PciDevice,
            (UCHAR)ulChannelIndex, ulOffset);
    }
```

This “for” loop reads data from the PCI card (notice the “hJr3PciDevice” PCI card handle) to the fsd struct one word at a time. Each iteration of the loop passes three variables to function ReadWord(), which locates the data, reads it, and returns it.

ReadWord(HANDLE hPciDevice, UCHAR ucChannel, ULONG ulOffset):

This is another intricate piece of C/C++ code, I will explain the gist.

Using arguments hPciDevice (handle to PCI card), ucChannel, and ulOffset, the code calculates the address of the memory word in the specified card, reads the word, and returns the word.

The syntax is obviously quite involved. But that is the gist.

Using force_sensor_data fields:

The logger reads the entire channel memory to the data structure, then accesses the variable(s) using standard structure syntax.

ReadStructure() reads a specified channel memory to the data structure, using C++ command “ReadWord().”

Once the data structure contains the card memory data, any variable can be accessed using common structure syntax.

The data structure is defined in file JR3PCILIB.h (for convenience the data structure definition portion of JR3PCILIP.n is included as Appendix A).

To read any sensor parameter:

1. Read PCI memory to the data structure (“ReadStructure()”)
2. Access one or more variables from the structure (GetDataField()).

Each variable in the structure is accessed as follows:

structure_name.group.field

or

structure_name.field

The structure instance is “fsd.”

To access Filter 0 Fx value: fsd.filter0.fx

To access Filter 1 Fz value: fsd.filter1.fz

Signed and Unsigned integers:

Document _5907e_rec_man.pdf specifies which fields are signed integers and which are unsigned. The only unsigned fields in the logger code are:

fsd.serial_no

fsd.units

Appendix A: Sensor data structure

```
// JR3 force/torque sensor data definition. For more information see sensor and
// hardware manuals.
typedef struct force_sensor_data
{
     // Raw_channels is the area used to store the raw data coming from the
sensor
     // See raw_channel struct definition
     struct raw_channel raw_channels[16];    // offset    0 0x00

     // JR3 copyright notice and reserved address 1
     // short copyright[0x0018];                // offset   64 0x40
     // short reserved1[0x0008];                // offset   88 0x58
     short copyright[0x000d];               // offset   64 0x40
     short reserved1[0x0013];               // offset   77 0x4d


     // Shunts contains the shunt readings. This is only used when the sensor
     // enables GAINS adjustments. Not used with this model, so its value must
     // read ALWAYS 0 (zero).
     struct six_axis_array shunts;          // offset   96 0x60
     short reserved2[2];                    // offset  102 0x66

     // Default full scale: used when other full scale is not set by user.
     struct six_axis_array default_FS;      // offset  104 0x68
     short reserved3;                       // offset  110 0x6e

     // Load_envelope_num is the load envelope number that is currently in use.
     // This value is SET BY THE USER after one of the load envelops has been
```

```
      // initialized.
      short load_envelope_num;                // offset  111 0x6f

      // Recommended minimum full scale (see manual pag.9).
      // This is the value at which the data will not saturate prematurely.
      struct six_axis_array min_full_scale;   // offset  112 0x70
      short reserved4;                        // offset  118 0x76

      // Transform_num is the transform number that is currently in use. This
value
      // is SET BY JR3 DSP after the user used command(5) ... see manual (pag.35).
      short transform_num;                    // offset  119 0x77

      // Recommended maximum full scale (see manual pag.9).
      // This is the maximum value at which no resolution is lost.
      struct six_axis_array max_full_scale;   // offset  120 0x78
      short reserved5;                        // offset  126 0x7e

      // Address of the data that will be monitored by the peak routine.
      // This value is SET BY THE USER, to check the 8 contiguous addresses.
      short peak_address;                     // offset  127 0x7f

      // Current full scale used by the sensor (see manual page 10).
      // usually it is recommended to compromise in favor of resolution which
means
      // that the recommended maximum full scale SHOULD BE CHOSEN.
      struct force_array full_scale;          // offset  128 0x80

      // These are the sensor offsets. They are subtracted from the sensor data to
      // obtain the decoupled data (the output data will be then zero).
      // To set the future decoupled data to zero add this values to the current
      // decoupled data and place the the sum here.
      struct six_axis_array offsets;          // offset  136 0x88

      // This is the current offset. This is SET BY THE JR3 DSP ... (pag.10)
      short offset_num;                       // offset  142 0x8e

      // Bit map showing which of the axis are being used in the vector
calculations
      // This value is SET BY THE JR3 DSP after ... (pag. 11)
      // struct vect_bits vect_axes;
      // PKN taking some liberty here to try to expedite making the logger work.
      // Changing the vect_bits to a short.
      short vect_bits_word;                   // offset  143 0x8f

      // Unfiltered and decoupled data (i.e, with the offsets removed) from the
      // JR3 sensor
      struct force_array filter0;             // offset  144 0x90
```

```
      // Each of following arrays hold the filtered data. The decoupled data
passes
      // trought a cascade of low pass filters, each having a cutoff frequency 1/4
      // of the succeeding filter. Filter 1 has a cutoff frequency of 1/16 of the
      // sample rate from the sensor: 500Hz for a typical sensor with a sample
rate
      // of 8KHz. The rest of the filters would cutoff at 125Hz, 31.25Hz, 7.813Hz,
      // 1.953 Hz and 0.4883Hz.
      struct force_array filter1;
      struct force_array filter2;
      struct force_array filter3;
      struct force_array filter4;
      struct force_array filter5;
      struct force_array filter6;

      // Calculated rate data, first derivative calculation. Calculated at a
      // frequency specified by variable_rate_divisor and calculated on the data
      // specified by rate_address.
      struct force_array rate_data;

      // The following arrays hold the minimum and maximum (peak) data values.
      // The JR3 DSP monitors 8 contiguous data items for MIN and MAX values at
full
      // sensor bandwidth. User  must request for area update. The address of the
      // data to watch for peaks is specified by peak_address.
      // Peak data is lost when executing coordinate transformation, full scale
      // change and when a new sensor is plugged in.
      struct force_array minimum_data;
      struct force_array maximum_data;

      // This values are used to determine if the raw sensor is saturated. The
decou-
      // pling process (offset removal) makes it difficult to say from the
processed
      // data if the sensor is saturated. Also watch for error and warning words.
      // These values may be SET BY THE USER, and the defaults are:
      // 80% of ADC full scale for near_sat_value (26214) and
      // ADC ful scale for sat_value (32768 - 2^(16 - ADC bits)).
      short near_sat_value;
      short sat_value;

      // Definition for rate calculations:
      // Rate_address - address of data used for calculations (8 contiguous)
      // Rate_divisor - Determines how often rate is calculated: 1 for rate
      //                calculation at full sensor bandwith, 0 for calculation
      //                every 65536 samples ... (100 for calc. every 100 samples)
      // Rate count   - Counts from zero until rate_divisor, at wich the rate is
      //                          calculated: rate_count resets then to zero and
...
```

```
      // Hint: When setting new rate_divisor set rate_count to rate_divisor-1.
This
      // will speed up the begeening of rate calculations.
      short rate_address;
      unsigned short rate_divisor;
      unsigned short rate_count;

      // These areas are used to send commands to the JR3 DSP. The DSp answers
with
      // a zero (0) when the command was successful and with a negative value to
      // indicate an error.
      short command_word2;
      short command_word1;
      short command_word0;

      // These values are incremented every time the matching filters are
calculated.
      // These values can be used to wait for data, i.e, the user should read data
      // after count change to ensure that he reads data just once.
      unsigned short count1;
      unsigned short count2;
      unsigned short count3;
      unsigned short count4;
      unsigned short count5;
      unsigned short count6;

      // This value counts data reception errors. If it is changing rapidly it
means
      // that there is some hardware or cabling error. In normal situation it
should
      // not change at all. It is nevertheless possible to have some activity in
      // EXTREMELY NOISY environments: in those cases (not meaning hardware
problems)
      // the sampled data is ignored.
      unsigned short error_count;

      // When the JR3 DSP searches it job queue and find nothing to do this
counter
      // is incremented. it is an indication of the amount of time the DSP was
      // available (doing nothing). It can also be used to see if the DSP is
alive.
      unsigned short count_x;

      // Warnings and errors contain the warning and error bits ... (pag. 22)
      // struct warning_bits warnings;
      // struct error_bits errors;
      unsigned short warnings;
      unsigned short errors;
```

```
      // Contains the bits that are set by the load envelops ... (pages 17 & 22)
      short threshold_bits;

      // Actual calculated CRC. It should be zero ... (pag. 22)
      short last_crc;

      // EEPROM number and software version
      short eeprom_ver_no;
      short software_ver_no;

      // Release date of the software: day of the year from 1 (1/1) to 365 (31/12)
for
      // non leap years.
      short software_day;
      short software_year;

      // Serial number and model number: they identify the sensor. Actually the
model
      // number does not correspond to JR3 model number but provides a unique
      // identifier for different sensor configurations.
      unsigned short serial_no;
      unsigned short model_no;

      // Calibration date: day from 1 (1/1) to 366 (31/12) for leap years.
      short cal_day;
      short cal_year;

      // Units defines the units used in this sensor full scale.
      // enum force_units units;
      unsigned short units;

      // Number of bits of the ADC currently in use.
      short bits;

      // Bit field that specifies the channels the current sensor can send.
      short channels;

      // Specifies the overall thickness of the sensor.
      short thickness;


      // Table containing the load envelope descriptions. See le_struct ... (pag.
25)

      struct le_struct load_envelopes[0x10];



      // Table containing the transform descriptions. See transform struct
(pag.28).
```

```
      struct transform transforms[0x10];

} force_sensor_data;
```

Appendix B: Data structure enums

Enum names chosen to reasonably close to the field names used in document _5907e_rec_man.pdf.

```
enum Field {
      raw_fx = 4, raw_fy = 8, raw_fz = 12, raw_mx = 16, raw_my = 20, raw_mz = 24,
      def_fs_fx = 104, def_fs_fy, def_fs_fz, def_fs_mx, def_fs_my, def_fs_mz,
      min_fs_fx = 112, min_fs_fy, min_fs_fz, min_fs_mx, min_fs_my, min_fs_mz,
      max_fs_fx = 120, max_fs_fy, max_fs_fz, max_fs_mx, max_fs_my, max_fs_mz,
      fullscalefx = 128, fullscalefy, fullscalefz, fullscalemx, fullscalemy,
fullscalemz,
      offsetfx = 136, offsetfy, offsetfz, offsetmx, offsetmy, offsetmz,
      filter0fx = 144, filter0fy, filter0fz, filter0mx, filter0my, filter0mz,
      filter1fx = 152, filter1fy, filter1fz, filter1mx, filter1my, filter1mz,
      filter2fx = 160, filter2fy, filter2fz, filter2mx, filter2my, filter2mz,
      filter3fx = 168, filter3fy, filter3fz, filter3mx, filter3my, filter3mz,
      filter4fx = 176, filter4fy, filter4fz, filter4mx, filter4my, filter4mz,
      filter5fx = 184, filter5fy, filter5fz, filter5mx, filter5my, filter5mz,
      filter6fx = 192, filter6fy, filter6fz, filter6mx, filter6my, filter6mz,
      serial = 248,
      units = 252,
      channels = 254
};
```

Variable and function/method descriptions and explanations:

C++ syntax can be confusing. While this document will not teach you C++, reasonable effort will be made to explain in simple terms where possible.

pciCardNames[4]: An array used to correlate possible PCI card names (assigned by the PCI card driver) to a simple index.

Example:
pciCardNames[0] = "JR3PCI1"
pciCardNames[1] = “JR3PCI2"
pciCardNames[3] = "JR3PCI3"
pciCardNames[4] = "JR3PCI4"

HANDLE hJr3PciDevice : Communication to the PCI card is conducted via a handle (much like a file handle). This variable is declared here much like a global variable which many functions utilize.

force_sensor_data fsd : "force_sensor_data" is a data structure ("struct") . It corresponds to the PCI card data in _5907e_rec_man.pdf Table 1: Summary of *JR3* DSP Data locations. The C++ code reads the memory channel for the specified sensor to the struct, then accesses individual fields via the struct member variables.