JR3

DSP-BASED
FORCE SENSOR RECEIVERS

SOFTWARE AND INSTALLATION
MANUAL

***JR3,* INC.**
22 HARTER AVE.
WOODLAND, CA 95776

(530) 661-3677x4
(530) 661-3701 (fax)
jr3@jr3.com (e-mail)

5907E

Warranty Provisions

JR3, Inc. warrants this product to be in good working order for a period of one year from the date of its purchase as a new product. Should this product fail to perform properly at any time within that one year period, ***JR3,*** Inc. will, at its option, repair or replace this product at no cost except as set forth in this warranty. Replacement parts or products will be furnished on an exchange basis only. Replaced parts and/or products become the property of ***JR3,*** Inc. No warranty is expressed or implied for products damaged by accident, abuse, natural or personal disaster, or unauthorized modification.

Warranty Limitations

All warranties for this product expressed or implied, including merchantability and fitness for a purpose, are limited to a one year duration from date of purchase, and no warranties, express or implied, will apply after that period.

If this product does not perform as warranted herein, owner's sole remedy shall be repair or replacement as provided above. In no event will ***JR3,*** Inc. be liable to any purchaser for damages, lost revenue, lost wages, lost savings, or any other incidental or consequential damages arising from purchase, use, or inability to use this product, even if ***JR3,*** Inc. has been advised of such damages.

Product performance is affected by system configuration, software, the application, customer data, and operator control of the system among other factors. The specific functional implementation by users of the product will vary. Therefore, the suitability of this product for a specific application must be determined by the customer and is not warranted by ***JR3,*** Inc..

This manual is as complete and factual as possible at the time of printing, however, the information in this manual may have been updated since that time, without notice.

Copyright Notice

Printed in the United States of America.

Table of Contents

JR3's DSP-based Force Sensor Receivers
Software and Installation Manual Overview

This manual covers the setup and operation of ***JR3*** DSP-based force sensor receivers. The main part of the manual covers that information which is common to the different receivers. Information unique to a particular receiver is covered in an appendix for that receiver.

This overview contains three sections. The first discusses the layout of the manual, the second, Getting Started, makes a few suggestions about getting results quickly. The third section gives an overview of the ***JR3*** DSP architecture.

SOFTWARE AND INSTALLATION MANUAL LAYOUT

JR3's DSP-based force sensor receivers contain a dual-ported RAM that the user and ***JR3*** DSP can both read and write. The ***JR3*** DSP writes current force and moment data into this RAM. The user can then read this data from the RAM. This manual primarily consists of a description of that shared area. The data structure declarations are formatted in 'C' style code. The first section, **Data Locations and Definitions**, starts on page 7 and contains the variable declarations and descriptions. The second section, **Data Structure Definitions**, starts on page 19 and contains the data type declarations. The data declarations in both sections are shown in the style of the 'C' computer language. There is a short 'C' Language Primer starting on page 27.

The next section, Command Definitions, starts on page 29 and contains descriptions of the commands which can be given to the ***JR3*** DSP. These commands, along with various data locations are used to alter the tasks performed by the DSP. Immediately following is the Examples section on page 41.

The main part of the manual ends with a summary table for the data locations and the commands, a glossary of terms, and a brief discussion of performance issues. The appendices contain specific data on the different versions of the ***JR3***'s DSP-based force sensor receivers.

GETTING STARTED

The best place for the reader to start in this manual is with the architectural overview which follows this section. It discusses the capabilities of the ***JR3*** DSP-based force sensor receivers. Second the reader should look at the appendix which is appropriate to the particular receiver he has. This will show how to interface to the receiver from a hardware and software perspective.

If the reader has little or no experience with the 'C' computer language, he should look next at the 'C' Language Primer on page 27. After brushing up on 'C', browse the **Data Locations and Definitions** section and the **Data Structure Definitions** section. Finally the Command Definitions section and the accompanying examples show how to execute commands.

When trying to communicate with the ***JR3*** DSP for the first time, reading the copyright statement at offset 0x0040, and the **count_x** variable, at offset 0x00ef, which is discussed on page 14, provides a quick way to see if the user can read from the shared address space. Writing a value into the FX offset variable, at 0x0088, and then reading that variable back can provide a quick way to see if writing to the shared address space is successful.

ARCHITECTURAL OVERVIEW

The ***JR3*** DSP-based force sensor receivers utilize the latest technology to provide 6 degree-of-freedom (6DOF) force and moment data at very high bandwidths. Employing an Analog Devices ADSP-2105, a 10 Mips digital signal processing chip, the ***JR3*** system can provide decoupled and digitally filtered data at 8 kHz per channel. This data rate is an order of magnitude faster than previously available. The receivers have been optimized to work with ***JR3***'s force moment sensors containing onboard electronics.

JR3 DSP-based receivers are available for several interfaces, with the IBM-AT bus and VMEbus being just two examples. These boards share a common architecture. The architecture consists of a dual-ported RAM, to which the host and the ***JR3*** DSP can both read and write. This RAM allows the host to read data from the ***JR3*** DSP with very little overhead. It also allows the host to reconfigure the ***JR3*** DSP, on the fly, by writing configuration commands to the RAM.

The receiver board contains circuitry to receive the serial digital data transmission from the sensor, as well as circuitry to monitor and adjust the power supply voltage to the sensor. The automatic remote power supply adjustment means that the sensor cable requirements are very forgiving. Long, small gage wires can be used with success. This means ***JR3***'s newest sensors, with onboard electronics, no longer need stiff expensive cables.

The dual-ported address space consists of 16k 2-byte words. The first 8k of this space is RAM, while the remaining 8k consists of status registers and other features. The majority of the user activity takes place in the first 256 words of the shared address space. The location of the variables in the shared address space is documented in the Data Locations and Definitions section this manual (pg. 7). Details of reading and writing to and from the shared address space vary among the different receivers and are documented separately for each in an appendix to this manual.

The ***JR3*** DSP is used to process the raw data transmitted from the sensor. The ***JR3*** DSP performs several functions. These include: offset removal, data decoupling, saturation detection, digital low-pass filtering, force and moment vector magnitude calculation, peak detection, rate calculation, coordinate translation and rotation, and threshold monitoring.

The raw data from the sensor is passed through a decoupling matrix and offsets are removed. This process removes sensor cross-coupling as well as tare loads. One by-product of this process is that it becomes difficult to determine if the analog-to-digital converter (ADC) in the force moment transducer is saturated. If the ADC is saturated, feedback loops using the force data will become unstable. To alert the user of this condition, the ***JR3*** DSP monitors the raw sensor data and indicates when the sensor data is approaching saturation, and when it is saturated.

The decoupled data is passed through cascaded low-pass filters. Each succeeding filter is calculated 1/4 as often, and has a cutoff frequency of 1/4 of the preceding filter. The cutoff frequency of a filter is 1/16 of the sample rate for that filter. For a typical sensor with a sample rate of 8 kHz, the cutoff frequency of the first filter would be 500 Hz. The following filters would cutoff at 125 Hz, 31.25 Hz, 7.81 Hz, etc. The delay through the filter is approximately equal to:

$$\text{Delay} \cong \frac{1}{\text{Cutoff Frequency}}$$

Therefore the delay through the 500 Hz filter would be:

$$\frac{1}{500\text{Hz}} \cong 2\text{ms}$$

Since the delay varies with the frequency of the data, this value, while providing a good estimate, is not exact. For a better description of filter properties, see the accompanying graphs showing filter response.

Force and moment vector magnitudes (vectors) are calculated from each data set. Therefore, there are vectors calculated from the unfiltered decoupled data, as well as from each of the sets of filtered data. The unfiltered vectors are calculated at 1/2 the bandwidth of the unfiltered data. The vectors calculated from the filtered data are calculated at 4 times the filter cutoff frequency, or 1/4 as often as the filter is calculated.

Any single data set (8 items) can be monitored for peaks and valleys. The data is monitored at full bandwidth, and if a new minimum or maximum is detected it is stored. The minimums and maximums can be read and/or reset under user control. Any single data set (8 items) can be used for rate calculations. At a user set interval, the first derivative of a data set is also calculated and reported.

The user can apply sensor coordinate transformations to set any sensor axes origin and orientation. This will, for example, allow the user to align the force sensor axes with the user's tool axes, which can greatly simplify the mathematics needed for force control or monitoring.

The ***JR3*** DSP can be configured by the user to monitor thresholds. The ***JR3*** DSP can toggle a bit, should any data cross a user specified threshold value. These load envelopes allow the user to monitor several load trip conditions with very little overhead.

The following graphs show the characteristics of the digital filters. The response of the first 3 filters are shown. The outputs of the filters vary slightly as the frequency of the filter decreases due to the cascaded nature of the filters. The data for the filters after the third are essentially the same as filter #3 and therefore they are not shown. The only difference for filters #4 - #6 is that the frequency axis would need to be divided by four for each succeeding filter from the graphs for filter #3

Fig 1: Filter Characteristics for Filter1 with Sensor Data at 8 kHz

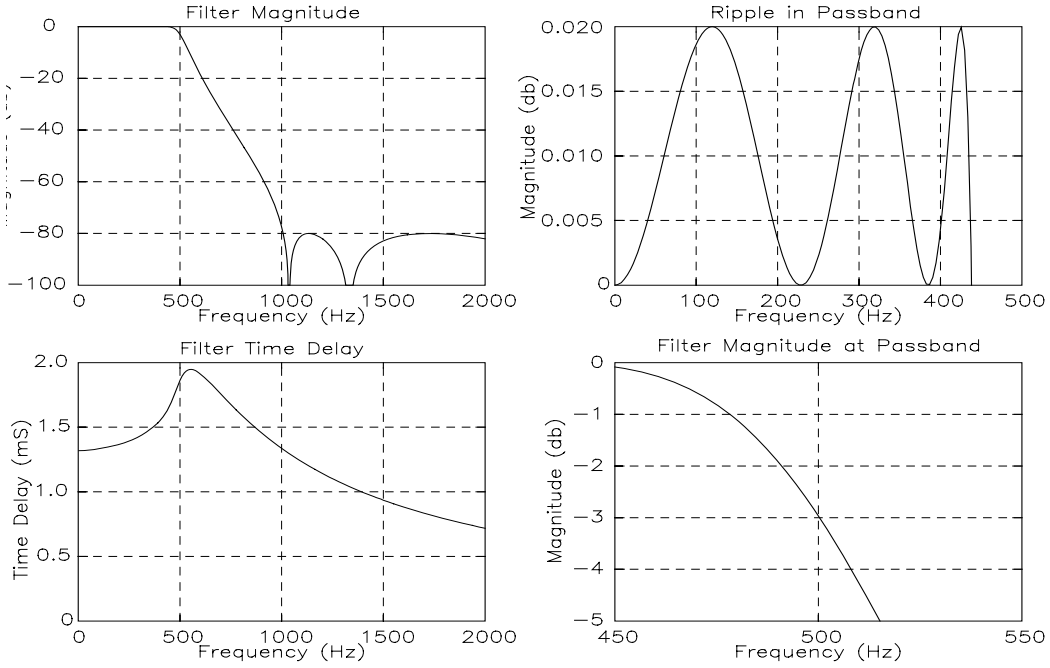


Fig 2: Filter Characteristics for Filter2 with Sensor Data at 8 kHz

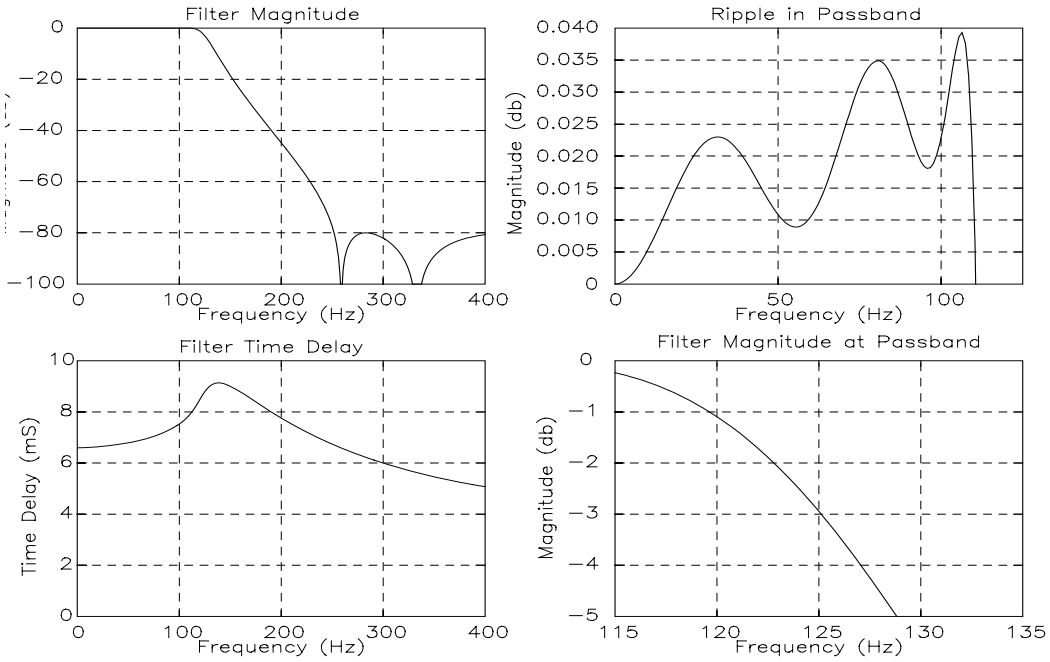
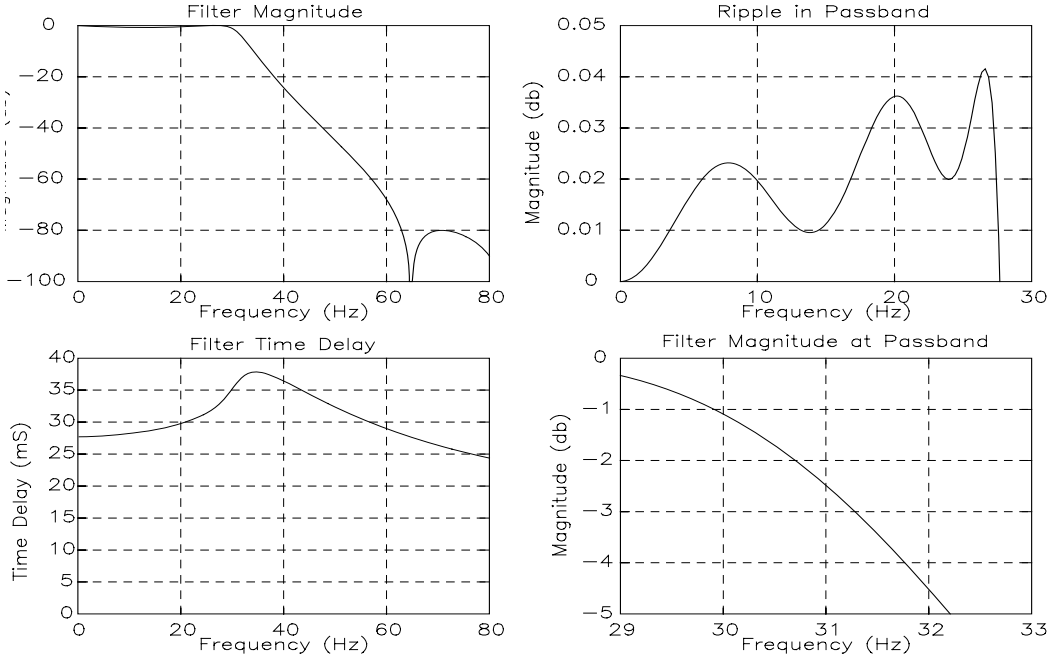


Fig 3: Filter Characteristics for Filter3 with Sensor Data at 8 kHz



JR3's DSP-based Force Sensor Receivers
Data Locations and Definitions

The following information details the memory interface of ***JR3***'s DSP-based force sensor receivers. The data structure declarations are in 'C' style code. The supporting structure declarations follow the data structure declaration. The declarations are in a courier typeface and are indented to distinguish them from their descriptions. Areas marked as reserved should not be written by the user. See the table on page 45 for a graphical layout of the variables. There is also a glossary of some of the terms on page 47.

```
/*

          For the following structure definitions:

          int:                signed 16-bit value
          unsigned int:       unsigned 16-bit value

          bit fields are shown with the lsb first.

*/
```

FORCE_SENSOR_DATA

The force_sensor_data structure shows the layout of information on the ***JR3*** DSP receiver card. The offsets shown are offsets into the ***JR3*** DSP's address space and they are for a 16-bit wide data space. Therefore, in an environment where memory is addressed 8-bits at a time, the offsets would need to be doubled.

```
struct  force_sensor_data
{
```

RAW_CHANNELS

Raw_channels is the area used to store the raw data coming from the sensor.

```
          raw_channel[16]     raw_channels        /*  offset 0x0000  */
```

COPYRIGHT

Copyright is a null terminated ASCII string containing the ***JR3*** copyright notice.

```
          int[0x0018]         copyright;          /*  offset 0x0040  */
          int[0x0008]         reserved1;          /*  offset 0x0058  */
```

SHUNTS

Shunts contains the sensor shunt readings. Some ***JR3*** sensors have the ability to have their gains adjusted. This allows the hardware full scales to be adjusted to potentially allow better resolution or dynamic range. For sensors that have this ability, the gain of each sensor channel is measured at the time of calibration using a shunt resistor. The shunt resistor is placed across one arm of the resistor bridge, and the resulting change in the output of that channel is measured. This measurement is called the shunt reading, and is recorded here. If the user has changed the gain of the sensor, and made new shunt measurements, those shunt measurements can be placed here. The ***JR3*** DSP will then scale the calibration matrix such so that the gains are again proper for the indicated shunt readings. If shunts is 0, then the sensor cannot have its gain changed. For details on changing the sensor gain, and making shunts readings, please see the sensor manual. To make these values take effect the user must call either command (5) use transform # (pg. 33) or command (10) set new full scales (pg. 38).

```
six_axis_array      shunts;             /*  offset 0x0060  */
int[2]              reserved2;          /*  offset 0x0066  */
```

DEFAULT_FS

Default_FS contains the full scale that is used if the user does not set a full scale.

```
six_axis_array      default_FS;         /*  offset 0x0068  */
int                 reserved3;          /*  offset 0x006e  */
```

LOAD_ENVELOPE_NUM

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 envelopes has been initialized.

```
int       load_envelope_num;  /*  offset 0x006f  */
```

MIN_FULL_SCALE

Min_full_scale is the recommend minimum full scale.

These values in conjunction with max_full_scale (pg. 9) helps determine the appropriate value for setting the full scales. The software allows the user to set the sensor full scale to an arbitrary value. But setting the full scales has some hazards. If the full scale is set too low, the data will saturate prematurely, and dynamic range will be lost. If the full scale is set too high, then resolution is lost as the data is shifted to the right and the least significant bits are lost. Therefore the maximum full scale is the maximum value at which no resolution is lost, and the minimum full scale is the value at which the data will not saturate prematurely. These values are calculated whenever a new coordinate transformation is calculated. It is possible for the recommended maximum to be less than the recommended minimum. This comes about primarily when using coordinate translations. If this is the case, it means that any full scale selection will be a compromise between dynamic range and resolution. It is usually recommended to compromise in favor of resolution which means that the recommend maximum full scale should be chosen.

WARNING: Be sure that the full scale is no less than 0.4% of the recommended minimum full scale. Full scales below this value will cause erroneous results.

```
six_axis_array      min_full_scale;     /*  offset 0x0070  */
int                 reserved4;          /*  offset 0x0076  */
```

TRANSFORM_NUM

Transform_num is the transform number that is currently in use. This value is set by the ***JR3*** DSP after the user has used command (5) use transform # (pg. 33).

```
int                 transform_num;      /*  offset 0x0077  */
```

MAX_FULL_SCALE

Max_full_scale is the recommended maximum full scale. See min_full_scale (pg. 9) for more details.

```
six_axis_array      max_full_scale;     /*  offset 0x0078  */
int                 reserved5;          /*  offset 0x007e  */
```

PEAK_ADDRESS

Peak_address is the address of the data which will be monitored by the peak routine. This value is set by the user. The peak routine will monitor any 8 contiguous addresses for peak values. (ex. to watch filter3 data for peaks, set this value to 0x00a8).

```
int                 peak_address;       /*  offset 0x007f  */
```

FULL_SCALE

Full_scale is the sensor full scales which are currently in use. Decoupled and filtered data is scaled so that +/- 16384 is equal to the full scales. The engineering units used are indicated by the units value discussed on page 16. The full scales for Fx, Fy, Fz, Mx, My and Mz can be written by the user prior to calling command (10) set new full scales (pg. 38). The full scales for V1 and V2 are set whenever the full scales are changed or when the axes used to calculate the vectors are changed. The full scale of V1 and V2 will always be equal to the largest full scale of the axes used for each vector respectively.

```
force_array         full_scale;         /*  offset 0x0080  */
```

OFFSETS

Offsets contains the sensor offsets. These values are subtracted from the sensor data to obtain the decoupled data. The offsets are set a few seconds (< 10) after the calibration data has been received. They are set so that the output data will be zero. These values can be written as well as read. The ***JR3*** DSP will use the values written here within 2 ms of being written. To set future decoupled data to zero, add these values to the current decoupled data values and place the sum here. The ***JR3*** DSP will change these values when a new transform is applied. So if the offsets are such that FX is 5 and all other values are zero, after rotating about Z by 90°, FY would be 5 and all others would be zero.

```
six_axis_array      offsets;            /*  offset 0x0088  */
```

OFFSET_NUM

Offset_num is the number of the offset currently in use. This value is set by the ***JR3*** DSP after the user has executed the use offset # command (pg. 34). It can vary between 0 and 15.

```
int                 offset_num;         /*  offset 0x008e  */
```

VECT_AXES

Vect_axes is a bit map showing which of the axes are being used in the vector calculations. This value is set by the ***JR3*** DSP after the user has executed the set vector axes command (pg. 37).

```
vect_bits vect_axes;          /*  offset 0x008f  */
```

FILTER0

Filter0 is the decoupled, unfiltered data from the ***JR3*** sensor. This data has had the offsets removed.

```
force_array         filter0;            /*  offset 0x0090  */
```

FILTER1 - FILTER6

These force_arrays hold the filtered data. The decoupled data is passed through cascaded low pass filters. Each succeeding filter has a cutoff frequency of 1/4 of the preceding filter. The cutoff frequency of filter1 is 1/16 of the sample rate from the sensor. For a typical sensor with a sample rate of 8 kHz, the cutoff frequency of filter1 would be 500 Hz. The following filters would cutoff at 125 Hz, 31.25 Hz, 7.813 Hz, 1.953 Hz and 0.4883 Hz.

```
force_array         filter1;            /*  offset 0x0098  */
force_array         filter2;            /*  offset 0x00a0  */
force_array         filter3;            /*  offset 0x00a8  */
force_array         filter4;            /*  offset 0x00b0  */
force_array         filter5;            /*  offset 0x00b8  */
force_array         filter6;            /*  offset 0x00c0  */
```

RATE_DATA

Rate_data is the calculated rate data. It is a first derivative calculation. It is calculated at a frequency specified by the variable rate_divisor (pg. 12). The data on which the rate is calculated is specified by the variable rate_address (pg. 12).

```
force_array         rate_data;          /*  offset 0x00c8  */
```

MINIMUM_DATA

MAXIMUM_DATA

Minimum_data & maximum_data are the minimum and maximum (peak) data values. The ***JR3*** DSP can monitor any 8 contiguous data items for minimums and maximums at full sensor bandwidth. This area is only updated at user request. This is done so that the user does not miss any peaks. To read the data, use either the read peaks command (pg. 40), or the read and reset peaks command (pg. 39). The address of the data to watch for peaks is stored in the variable peak_address (pg. 10). Peak data is lost when executing a coordinate transformation or a full scale change. Peak data is also lost when plugging in a new sensor.

```
force_array         minimum_data;       /*  offset 0x00d0  */
force_array         maximum_data;       /*  offset 0x00d8  */
```

NEAR_SAT_VALUE
SAT_VALUE

Near_sat_value & sat_value contain the value used to determine if the raw sensor is saturated. Because of decoupling and offset removal, it is difficult to tell from the processed data if the sensor is saturated. These values, in conjunction with the error and warning words (pg. 14), provide this critical information. These two values may be set by the host processor. These values are positive signed values, since the saturation logic uses the absolute values of the raw data. The near_sat_value defaults to approximately 80% of the ADC's full scale, which is 26214, while sat_value defaults to the ADC's full scale:

$$sat\_value = 32768 - 2^{(16 - \text{ADC bits})}$$

```
int       near_sat_value;               /*  offset 0x00e0  */
int       sat_value;                    /*  offset 0x00e1  */
```

RATE_ADDRESS
RATE_DIVISOR

RATE_COUNT

Rate_address, rate_divisor & rate_count contain the data used to control the calculations of the rates. Rate_address is the address of the data used for the rate calculation. The ***JR3*** DSP will calculate rates for any 8 contiguous values (ex. to calculate rates for filter3 data set rate_address to 0x00a8). Rate_divisor is how often the rate is calculated. If rate_divisor is 1, the rates are calculated at full sensor bandwidth. If rate_divisor is 200, rates are calculated every 200 samples. Rate_divisor can be any value between 1 and 65536. Set rate_divisor to 0 to calculate rates every 65536 samples. Rate_count starts at zero and counts until it equals rate_divisor, at which point the rates are calculated, and rate_count is reset to 0. When setting a new rate divisor, it is a good idea to set rate_count to one less than rate divisor. This will minimize the time necessary to start the rate calculations.

```
int                rate_address;       /*  offset 0x00e2  */
unsigned int       rate_divisor;       /*  offset 0x00e3  */
unsigned int       rate_count;         /*  offset 0x00e4  */
```

COMMAND_WORD2
COMMAND_WORD1
COMMAND_WORD0

Command_word2 through command_word0 are the locations used to send commands to the ***JR3*** DSP. Their usage varies with the command and is detailed later in the Command Definitions section (pg. 29). In general the user places values into various memory locations, and then places the command word into command_word0. The ***JR3*** DSP will process the command and place a 0 into command_word0 to indicate successful completion. Alternatively the ***JR3*** DSP will place a negative number into command_word0 to indicate an error condition. Please note the command locations are numbered backwards. (I.E. command_word2 comes before command_word1).

```
int       command_word2;               /*  offset 0x00e5  */
int       command_word1;               /*  offset 0x00e6  */
int       command_word0;               /*  offset 0x00e7  */
```

COUNT1 - COUNT6

Count1 through count6 are unsigned counters which are incremented every time the matching filters are calculated. Filter1 is calculated at the sensor data bandwidth. So this counter would increment at 8 kHz for a typical sensor. The rest of the counters are incremented at 1/4 the interval of the counter immediately preceding it, so they would count at 2 kHz, 500 Hz, 125 Hz etc. These counters can be used to wait for data. Each time the counter changes, the corresponding data set can be sampled, and this will insure that the user gets each sample, once, and only once.

```
unsigned int       count1;             /*  offset 0x00e8  */
unsigned int       count2;             /*  offset 0x00e9  */
unsigned int       count3;             /*  offset 0x00ea  */
unsigned int       count4;             /*  offset 0x00eb  */
unsigned int       count5;             /*  offset 0x00ec  */
unsigned int       count6;             /*  offset 0x00ed  */
```

ERROR_COUNT

Error_count is a running count of data reception errors. If this counter is changing rapidly, it probably indicates a bad sensor cable connection or other hardware problem. In most installations error_count should not change at all. But it is possible in an *extremely* noisy environment to experience occasional errors even without a hardware problem. If the sensor is well grounded, this is probably unavoidable in these environments. On the occasions where this counter counts a bad sample, that sample is ignored.

```
unsigned int        error_count;        /*  offset 0x00ee  */
```

COUNT_X

Count_x is a counter which is incremented every time the ***JR3*** DSP searches its job queues and finds nothing to do. It indicates the amount of idle time the ***JR3*** DSP has available. It can also be used to determine if the ***JR3*** DSP is alive. See the Performance Issues section on pg. 49 for more details.

```
unsigned int        count_x;            /*  offset 0x00ef  */
```

WARNINGS
ERRORS

Warnings & errors contain the warning and error bits respectively. The format of these two words is discussed on page 21 under the headings warnings_bits and error_bits.

```
warning_bits        warnings; /*  offset 0x00f0  */
error_bits          errors;             /*  offset 0x00f1  */
```

THRESHOLD_BITS

Threshold_bits is a word containing the bits that are set by the load envelopes. See load_envelopes (pg. 17) and thresh_struct (pg. 23) for more details.

```
int                 threshold_bits;     /*  offset 0x00f2  */
```

LAST_CRC

Last_crc is the value that shows the actual calculated CRC. CRC is short for cyclic redundancy code. It should be zero. See the description for cal_crc_bad (pg. 21) for more information.

```
int       last_CRC;           /*  offset 0x00f3  */
```

EEPROM_VER_NO
SOFTWARE_VER_NO

EEProm_ver_no contains the version number of the sensor EEProm. EEProm version numbers can vary between 0 and 255. Software_ver_no contains the software version number. Version 3.02 would be stored as 302.

```
int       eeprom_ver_no;              /*  offset 0x00f4  */
int       software_ver_no;            /*  offset 0x00f5  */
```

SOFTWARE_DAY
SOFTWARE_YEAR

Software_day & software_year are the release date of the software the ***JR3*** DSP is currently running. Day is the day of the year, with January 1 being 1, and December 31, being 365 for non leap years.

```
int       software_day;               /*  offset 0x00f6  */
int       software_year;              /*  offset 0x00f7  */
```

SERIAL_NO
MODEL_NO

Serial_no & model_no are the two values which uniquely identify a sensor. This model number does not directly correspond to the ***JR3*** model number, but it will provide a unique identifier for different sensor configurations.

```
unsigned int        serial_no;          /*  offset 0x00f8  */
unsigned int        model_no; /*  offset 0x00f9  */
```

CAL_DAY
CAL_YEAR

Cal_day & cal_year are the sensor calibration date. Day is the day of the year, with January 1 being 1, and December 31, being 366 for leap years.

```
int       cal_day;                    /*  offset 0x00fa  */
int       cal_year;          /*  offset 0x00fb  */
```

UNITS
BITS
CHANNELS

Units is an enumerated **read only** value defining the engineering units used in the sensor full scale. The meanings of particular values are discussed in the section detailing the force_units structure on page 22. The engineering units are setto customer specifications during sensor manufacture and cannot be changed by writing to Units.

Bits contains the number of bits of resolution of the ADC currently in use.

Channels is a bit field showing which channels the current sensor is capable of sending. If bit 0 is active, this sensor can send channel 0, if bit 13 is active, this sensor can send channel 13, etc. This bit can be active, even if the sensor is not currently sending this channel. Some sensors are configurable as to which channels to send, and this field only contains information on the channels available to send, not on the current configuration. To find which channels are currently being sent, monitor the Raw_time fields (pg. 19) in the raw_channels array (pg. 7). If the time is changing periodically, then that channel is being received.

```
force_units         units;              /*  offset 0x00fc  */
int                 bits;               /*  offset 0x00fd  */
int                 channels; /*  offset 0x00fe  */
```

THICKNESS

Thickness specifies the overall thickness of the sensor from flange to flange. The engineering units for this value are contained in units (pg. 16). The sensor calibration is relative to the center of the sensor. This value allows easy coordinate transformation from the center of the sensor to either flange.

```
int       thickness;                  /*  offset 0x00ff  */
```

LOAD_ENVELOPES

Load_envelopes is a table containing the load envelope descriptions. There are 16 possible load envelope slots in the table. The slots are on 16 word boundaries and are numbered 0-15. Each load envelope needs to start at the beginning of a slot but need not be fully contained in that slot. That is to say that a single load envelope can be larger than a single slot. The software has been tested and ran satisfactorily with 50 thresholds active. A single load envelope this large would take up 5 of the 16 slots. The load envelope data is laid out in an order that is most efficient for the ***JR3*** DSP. The structure is detailed later in the section showing the definition of the le_struct structure (pg. 23).

```
le_struct[0x10]     load_envelopes;     /*  offset 0x0100  */
```

TRANSFORMS

Transforms is a table containing the transform descriptions. There are 16 possible transform slots in the table. The slots are on 16 word boundaries and are numbered 0-15. Each transform needs to start at the beginning of a slot but need not be fully contained in that slot. That is to say that a single transform can be larger than a single slot. A transform is 2 * no of links + 1 words in length. So a single slot can contain a transform with 7 links. Two slots can contain a transform that is 15 links. The layout is detailed later in the section showing the definition of the transform structure (pg. 26).

```
transform[0x10]     transforms;         /*  offset 0x0200  */
```

Data Structure Definitions

RAW_CHANNEL

The raw data is stored in a format which facilitates rapid processing by the ***JR3*** DSP chip. The raw_channel structure shows the format for a single channel of data. Each channel takes four, two-byte words.

RAW_TIME

Raw_time is an unsigned integer which shows the value of the ***JR3*** DSP's internal clock at the time the sample was received. The clock runs at 1/10 the ***JR3*** DSP cycle time. ***JR3***'s slowest DSP runs at 10 Mhz. At 10 Mhz raw_time would therefore clock at 1 Mhz.

RAW_DATA

Raw_data is the raw data received directly from the sensor. The sensor data stream is capable of representing 16 different channels. Channel 0 shows the excitation voltage at the sensor. It is used to regulate the voltage over various cable lengths. Channels 1-6 contain the coupled force data Fx through Mz. Channel 7 contains the sensor's calibration data. The use of channels 8-15 varies with different sensors.

```
struct  raw_channel
{
          unsigned int        raw_time;
          int                 raw_data;
          int[2]              reserved;
};
```

FORCE_ARRAY

The force_array structure shows the layout for the decoupled and filtered force data.

```
struct  force_array
{
          int       fx;
          int       fy;
          int       fz;
          int       mx;
          int       my;
          int       mz;
          int       v1;
          int       v2;
};
```

SIX_AXIS_ARRAY

The six_axis_array structure shows the layout for the offsets and the full scales.

```
struct  six_axis_array
{
          int       fx;
          int       fy;
          int       fz;
          int       mx;
          int       my;
          int       mz;
};
```

VECT_BITS

The vect_bits structure shows the layout for indicating which axes to use in computing the vectors. Each bit signifies selection of a single axis. The V1x axis bit corresponds to a hex value of 0x0001 and the V2z bit corresponds to a hex value of 0x0020. Example: to specify the axes V1x, V1y, V2x, and V2z the pattern would be 0x002b. Vector 1 defaults to a force vector and vector 2 defaults to a moment vector. It is possible to change one or the other so that two force vectors or two moment vectors are calculated. Setting the changeV1 bit or the changeV2 bit will change that vector to be the opposite of its default. Therefore to have two force vectors, set changeV1 to 1.

```
struct  vect_bits
{
          unsigned  fx : 1;
          unsigned  fy : 1;
          unsigned  fz : 1;
          unsigned  mx : 1;
          unsigned  my : 1;
          unsigned  mz : 1;
          unsigned  changeV2 : 1;
          unsigned  changeV1 : 1;
          unsigned  reserved : 8;
};
```

WARNING_BITS

The warning_bits structure shows the bit pattern for the warning word. The bit fields are shown from bit 0 (lsb) to bit 15 (msb).

XX_NEAR_SAT

The xx_near_sat bits signify that the indicated axis has reached or exceeded the near saturation value.

```
struct  warning_bits
{
          unsigned  fx_near_sat : 1;
          unsigned  fy_near_sat : 1;
          unsigned  fz_near_sat : 1;
          unsigned  mx_near_sat : 1;
          unsigned  my_near_sat : 1;
          unsigned  mz_near_sat : 1;
          unsigned  reserved : 10;
};
```

ERROR_BITS
XX_SAT
MEMORY_ERROR
SENSOR_CHANGE

The error_bits structure shows the bit pattern for the error word. The bit fields are shown from bit 0 (lsb) to bit 15 (msb). The xx_sat bits signify that the indicated axis has reached or exceeded the saturation value. The memory_error bit indicates that a problem was detected in the on-board RAM during the power-up initialization. The sensor_change bit indicates that a sensor other than the one originally plugged in has passed its CRC check. This bit latches, and must be reset by the user.

SYSTEM_BUSY

The system_busy bit indicates that the ***JR3*** DSP is currently busy and is not calculating force data. This occurs when a new coordinate transformation, or new sensor full scale is set by the user. A very fast system using the force data for feedback might become unstable during the approximately 4 ms needed to accomplish these calculations. This bit will also become active when a new sensor is plugged in and the system needs to recalculate the calibration CRC.

CAL_CRC_BAD

The cal_crc_bad bit indicates that the calibration CRC has not calculated to zero. CRC is short for cyclic redundancy code. It is a method for determining the integrity of messages in data communication. The calibration data stored inside the sensor is transmitted to the ***JR3*** DSP along with the sensor data. The calibration data has a CRC attached to the end of it, to assist in determining the completeness and integrity of the calibration data received from the sensor. There are two reasons the CRC may not have calculated to zero. The first is that all the calibration data has not yet been received, the second is that the calibration data has been corrupted. A typical sensor transmits the entire contents of its calibration matrix over 30 times a second. Therefore, if this bit is not zero within a couple of seconds after the sensor has been plugged in, there is a problem with the sensor's calibration data.

WATCH_DOG
WATCH_DOG2

The watch_dog and watch_dog2 bits are sensor, not processor, watch dog bits. Watch_dog indicates that the sensor data line seems to be acting correctly, while watch_dog2 indicates that sensor data and clock are being received. It is possible for watch_dog2 to go off while watch_dog does not. This would indicate an improper clock signal, while data is acting correctly. If either watch dog barks, the sensor data is not being received correctly.

```
struct  error_bits
{
          unsigned  fx_sat : 1;
          unsigned  fy_sat : 1;
          unsigned  fz_sat : 1;
          unsigned  mx_sat : 1;
          unsigned  my_sat : 1;
          unsigned  mz_sat : 1;
          unsigned  reserved : 4;
          unsigned  memory_error : 1;
          unsigned  sensor_change : 1;
          unsigned  system_busy : 1;
          unsigned  cal_crc_bad : 1;
          unsigned  watch_dog2 : 1;
          unsigned  watch_dog : 1;
};
```

FORCE_UNITS

Force_units is an enumerated value defining the different possible engineering units used.

0 - lbs_in-lbs_mils:	lbs, inches * lbs, and inches * 1000
1 - N_dNm_mmX10:	Newtons, Newtons * meters * 10, and mm * 10
2 - dkgF_kgFcm_mmX10:	kilograms-force * 10, kilograms-Force * cm, and mm * 10
3 - klbs_kin-lbs_mils:	1000 lbs, 1000 inches * lbs, and inches * 1000

```
enum force_units
{
          lbs_in-lbs_mils,
          N_dNm_mmX10,
          dkgF_kgFcm_mmX10
          klbs_kin-lbs_mils
          reserved_units_4
          reserved_units_5
          reserved_units_6
          reserved_units_7
};
```

THRESH_STRUCT

Thresh_struct is the structure showing the layout for a single threshold packet inside of a load envelope. Each load envelope can contain several threshold structures.

DATA_ADDRESS

Data_address is the address of the data for that threshold. Each threshold can look at any piece of data. While the obvious filtered and unfiltered data can be monitored, it is also possible to monitor the raw data, the rate data, the counters, or the error and warning words.

THRESHOLD

Threshold is the value at which, if the data is above or below, the bits will be set. Therefore, for a greater than equal threshold (GE), if the data value is above the threshold value, the bit pattern will be OR'ed into the threshold variable at 0x00f2.

BIT_PATTERN

Bit_pattern contains the bits that will be set if the threshold value is met or exceeded.

```
struct    thresh_struct
{
          int       data_address;
          int       threshold;
          int       bit_pattern;
};
```

LE_STRUCT

Le_struct is the structure showing the layout of a load envelope packet. This structure shows 4 thresholds, but the thresholds can in fact be laid end to end for as many thresholds as needed. If there are more than four thresholds the load envelope will overlap the succeeding load envelope. This is acceptable as long as the user realizes this and does not try to use the succeeding load envelope. The thresholds need to be arranged with the greater than or equal thresholds (GE) first and the less than or equal (LE) thresholds next.

LATCH_BITS

Latch_bits is a bit pattern which shows which bits the user wants to latch. The latched bits will not be reset once the threshold which set them is no longer true. In that case, the user must reset them using the reset_bit command.

NUMBER_OF_GE_THRESHOLDS

NUMBER_OF_LE_THRESHOLDS

These values specify how many GE thresholds there are and how many LE thresholds there are. The GE thresholds are first, and are followed by the LE thresholds.

```
struct    le_struct
{
          int                 latch_bits;
          int                 number_of_ge_thresholds;
          int                 number_of_le_thresholds;
          thresh_struct[4]    thresholds;
          int                 reserved  ;
};
```

Fig 4: Load Envelope Structure

Address	data	
0	latch Pattern	Pattern of bits which latch
1	no of GE	number of >= thresholds
2	no of LE	number of <= thresholds
3	data addr	data address for GE #1
4	threshold	threshold for GE #1
5	bit pattern	bit pattern for GE #1
6	data addr	data address for GE #2
7	threshold	threshold for GE #2
8	bit pattern	bit pattern for GE #2
	...	
Ge#*3	data addr	data address for last GE
ge#*3+1	threshold	threshold for last GE
ge#*3+2	bit pattern	bit pattern for last GE
ge#*3+3	data addr	data address for LE #1
ge#*3+4	threshold	threshold for LE #1
ge#*3+5	bit pattern	bit pattern for LE #1
	...	
	data addr	data address for last LE
	threshold	threshold for last LE
	bit pattern	bit pattern for last LE

LE - **L**ess than or **E**qual threshold
GE - **G**reater than or **E**qual threshold

LINK_TYPES

Link_types is an enumerated value showing the different possible transform link types. The end transform packet is put at the end of the transform chain. The translate and rotate types are used to translate the sensor axes origin and orientation. The negate all axes type makes all axes negative. It is used to convert from the default robot point of view to the tool point of view.

0 - end transform packet
1 - translate along X axis (TX)
2 - translate along Y axis (TY)
3 - translate along Z axis (TZ)
4 - rotate about X axis (RX)
5 - rotate about Y axis (RY)
6 - rotate about Z axis (RZ)
7 - negate all axes

```
enum link_types
{
          end_x_form,
          tx,
          ty,
          tz,
          rx,
          ry,
          rz,
          neg
};
```

TRANSFORM

Transform is a structure which is used to describe a transform. A transform is made up of successive links. The links can be in any order. Each link is two words. The first word is the type, and the second word is the amount to transform. The types are detailed in the description for link_types (pg. 25). The amount to translate is specified in the engineering length units described by the **units** variable (pg. 16). The amount to rotate is scaled such that:

$$\text{amount} = \text{degrees} \cdot \frac{32768}{180} \quad \text{or} \quad \text{amount} = \text{radians} \cdot \frac{32768}{\pi}$$

These units work very nicely, because a 16 bit integer maps exactly into 0-2pi range. The translation units however, can cause a problem if extraordinarily long translations are necessary, because the absolute value of the translation must be ≤ 32767. The solution, is to break a very long translation into two or more shorter translations in the same direction but with lengths ≤ 32767. The negate all axes type must have a non-zero amount specified, the value specified for amount is not significant.

```
struct links
{
          link_types          link_type
          int                 link_amount;
};

struct transform
{
          links[8]            link;
};
```

Fig 5: Transform Structure

Address	data	
0	link # 1 - type	Link #1 packet
1	link # 1 - amount	
2	link # 2 - type	Link #2 packet
3	link # 2 - amount	
	...	
(n-1)*2	link #n - type	Link #n packet
(n-1)*2+1	link #n - amount	
N*2	0	end_x_form packet
n*2+1	don't care	

‘C’ Language Primer

The data declarations in the “**Data Locations and Definitions**“ and “**Data Structure Definitions**“ sections, are shown in the style of the ‘C’ computer language. ‘C’ was chosen because it is the closest thing to a universal language currently in wide spread usage. But for those fortunate enough to have never worked with ‘C’, this is a short tutorial in reading ‘C’ style data declarations.

The symbols /* and */ are used as the begin and end comment delimiters. The semicolon ;, is used as a statement terminator, and the comma is used to separate items in a list. Hexadecimal numbers are declared by appending a prefix of 0x. So to indicate the hexadecimal value of 16 we would write 0x0010. Like wise, 157 would be 0x009d. When declaring a variable, the variable’s type is listed first, followed by the name of the variable. So to declare a variable of type int, with the name sat_value, the following would be used:

```
int sat_value;
```

Int is the basic signed integer type in ‘C’. The size of an int can vary depending on the machine for which the ‘C’ compiler has been written. In our case the natural size for an int is 16 bits, since the ADSP-2105 processes information 16 bits at a time. Therefore an int can take values ranging from -32768 to 32767. The unsigned int data type is the same size as an int, but has no sign. Therefore, the following:

```
unsigned int countx;
```

would declare a variable called countx, which could take on the value of from 0 to 65535.

‘C’ has the ability to bundle data together into structures. A struct declaration defines a data structure. After defining a struct, a variable can be declared with the type of the struct, and then individual fields of the struct can be accessed. To declare a force array structure the following could be used:

```
struct force_structure
{
          int fx;
          int fy;
          int fz;
};
```

This declares a struct named force_structure with three int fields named: fx, fy, and fz. These fields would be arranged in memory with fx at the lowest, or first, address and fz at the highest, or last, address. To define a variable using a struct, the struct name is used for the variable type in the variable declaration.

```
force_structure filter0;
```

would declares a variable called filter0, of type force_structure. Using the earlier declaration of force_structure, it would have three fields named: fx, fy and fz.

A data array in 'C' is declared by appending [x] to the variable type in a normal data declaration. The x, in [x] indicates the number of elements contained in the array. So,

```
force_structure[0x0010] force_data;
```

would indicate we had a variable called force_data, which contained 16 (0x0010 in hexadecimal) elements of type force_structure. Therefore force_data would consist of 48 (16 * 3) ints. This can be thought of as 16 different sets of force data laid end to end in memory.

Finally 'C' has a data structure type called bit fields, which allows the easy manipulation of individual bits or groups of bits in a variable. Bit fields are indicated by appending :x to the element name in a struct definition. The x in :x indicates the number of bits used by the element. 'C' allows the bit order to specified by the compiler writer, we define them to be declared from lsb to msb. That is if you were to encode our bit field as a binary number, the first bit field listed would have the least weight, and the last bit the most.

```
struct test_bits
{
          fx_bit : 1;
          mx_bit : 1;
          reserved : 14;
}
```

This declares a struct where the lsb is to be considered the fx_bit, the second bit is the mx_bit, and the top 14 bits are reserved.

Command Definitions

EXAMPLE COMMAND (0)

Calling Parameters:

Data_1	This is the name of the data to modify prior to setting command_word_0
Command_Word_0	0x0000

Returned Variables:

Data_2	This is the name of the data which is modified by this command.
Command_Word_0	0 (Command_Word_0 is set to zero to indicate that the command has successfully completed)

Execution time:

40 µS	(average)
125 µS	(maximum)

Notes:

The example command shows the layout of the other commands in this section. For the calling parameters and the returned variables, the first column shows the name of the variable. The second column shows the value which the variable is set to or the value which is to be returned.

If the user needed to execute this example command, the first step would for the user to modify Data_1 as needed. The user would then write the command number to Command_Word_0. In this case that would be 0x0000. The user would then monitor Command_Word_0 until it was changed to 0 by the ***JR3*** DSP. At that point the command would have been successfully executed and the user could examine the value returned in Data_2.

MEMORY READ (1)

Calling Parameters:

Command_Word_1	Address to read
Command_Word_0	0x0100

Returned Variables:

Command_Word_2	Data read from address
Command_Word_0	0

Execution time:

40 µS	(average)
125 µS	(maximum)

Notes:

The memory read command is of no use in standard operation. Memory can be read directly by the host. This command is only used for testing purposes, and to read the internal memory of the DSP.

MEMORY WRITE (2)

Calling Parameters:

Command_Word_2	Data to write
Command_Word_1	Address to write
Command_Word_0	0x0200

Returned Variables:

Command_Word_2	Data read from address
Command_Word_0	0

Execution time:

40 µS	(average)
125 µS	(maximum)

Notes:

The memory write command is used to write to a section of the DSP memory. It is only needed if there is a chance the DSP may modify the location at the same time the user is trying to write to it. This is primarily true of the latching bits of the warning, error and threshold bit pattern locations. This command also returns the value that was present in the location before it wrote to the location.

BIT SET (3)

Calling Parameters:

Command_Word_2	Bit map of bits to set
Command_Word_1	Address in which bits will be set
Command_Word_0	0x0300

Returned Variables:

Command_Word_2	Data read from address
Command_Word_0	0

Execution time:

40 µS	(average)
125 µS	(maximum)

Notes:

The bit set command is used to easily set single or multiple bits of a word in memory. It is only needed if there is a chance the DSP may modify the location at the same time the user is trying to write to it. This is primarily true of the latching bits of the warning, error, and threshold bit pattern locations. This command also returns the value that was present in the location before it wrote to the location.

BIT RESET (4)

Calling Parameters:

Command_Word_2	Bit map of bits to reset
Command_Word_1	Address in which will be reset
Command_Word_0	0x0400

Returned Variables:

Command_Word_2	Data read from address
Command_Word_0	0

Execution time:

40 µS	(average)
125 µS	(maximum)

Notes:

The bit reset command is used to easily reset single or multiple bits of a word in memory. It is only needed if there is a chance the DSP may modify the location at the same time the user is trying to write to it. This is primarily true of the latching bits of the warning, error and threshold bit pattern locations. This command also returns the value that was present in the location before it wrote to the location.

USE TRANSFORM # (5)

Calling Parameters:

transforms	Setup appropriate slot of the transform table
Command_Word_0	0x050?,

where "?" is the transform slot to use and can be 0x0 to 0xf.

Returned Variables:

Command_Word_0	0
Transform_Num	new current transform number

Execution time:

3 - 10 mS (average)

The time this command takes is directly related to the number of links in the transform. The time is approximately 2.25 ms + 0.75 ms for each link.

Notes:

Before executing the use transform command, a transform must be setup in the appropriate slot in the transform table. If the user does not require multiple transforms to be stored in local memory, just place the transform into slot 0 (@ 0x200) and then place 0x0500 into command_word_0. This command takes several milliseconds to execute, so the system_busy bit will be set in the error_bits word while it is executing. Also, as with the other commands, command_word_0 will be reset to 0 at the completion of this command.

USE OFFSET # (6)

Calling Parameters:

Command_Word_0	0x060?,

where " ? " is the offset number to use and can be 0x0 to 0xf.

Returned Variables:

Command_Word_0	0
offset_num	new current offset # from command_word_0
offsets	set to new value from offset table

Execution time:

150 µS	(average)
225 µS	(maximum)

Notes:

The ***JR3*** DSP is capable of storing 16 different offsets. This might be needed when switching to and from different tooling, or might be needed for different tooling orientations. The use offset command sets the current offset #. It also loads the offsets from the offset table. Subsequent set offsets commands or reset offsets commands will write the new offsets into this entry of the table. If the user does not require multiple offsets to be stored locally, then this command would never be called and offset #0 would always default to the working offset. If the user wants to set the current offset # without loading the offsets from the table, just write the desired offset # directly to the current offset_num location (@ 0x008e).

SET OFFSETS (7)

Calling Parameters:

offsets	set to desired offsets
Command_Word_0	0x0700

Returned Variables:

Command_Word_0	0
current offset table entry	values from the offsets variable

Execution time:

150 µS	(average)
225 µS	(maximum)

Notes:

The set offsets command takes the values from the offsets variable (@ 0x0088) and uses it for the current sensor offsets as well as storing them in the offset table under the current offset_num entry. This command is not really necessary because simply changing the values in the offsets variable will achieve the same results. The command is desirable because the ***JR3*** DSP can take up to 2 milliseconds to notice that the values have changed, and to start using them. This command speeds up the process considerably. If a 2 ms delay in changing offsets is not a problem, then simply placing the new offsets into the offsets array will suffice.

RESET OFFSETS (8)

Calling Parameters:

Command_Word_0	0x0800

Returned Variables:

Command_Word_0	0
offsets	set so that the output data will be 0
current offset table entry	values from the offsets variable

Execution time:

150 µS	(average)
225 µS	(maximum)

Notes:

The reset offsets command takes the data from filter2 and then sets the offsets such that this data will be 0 after the offset change. The command updates the values in the offsets variable (@ 0x0088) as well as storing the offsets in the offset table under the current offset_num entry. This command is not really necessary because the user can calculate and change the offset values as needed, but this command makes the chore somewhat simpler.

SET VECTOR AXES (9)

Calling Parameters:

Command_Word_0	0x09??

where "??" is a bit map describing the axes to use.
V1x = 1, V1y = 2, V1z = 4, V2x = 8, V2y = 16, V2z = 32
V1 is force vector, V2 is moment vector = 0
V1 is force vector, V2 is force vector = 64
V1 is moment vector, V2 is moment vector = 128

Returned Variables:

Command_Word_0	0
vect_axes	same bit map passed in command_word_0
full_scale	full scales for V1 and V2 are set equal to the largest component axis.

Execution time:

40 µS	(average)
125 µS	(maximum)

Notes:

The set vector axes command allows the user to set which axes are used for calculating the V1 (vector 1) and V2 (vector 2) vector resultants. There are two vectors calculated. They can be calculated from either forces or moments. Normally V1 is calculated from the forces and V2 from the moments. But it is also possible to set them both to be either force vectors or moment vectors.

SET NEW FULL SCALES (10)

Calling Parameters:

full_scale of fx-mz	Set to desired full scale
Command_Word_0	0x0A00

Returned Variables:

Command_Word_0	0

Execution time:

3 - 10 mS	(average)

The time the set new full scales command takes is directly related to the number of links in the current transform. The time is approximately 3.4 ms + 0.75 ms for each link.

Notes:

The set new full scales command takes several milliseconds to execute, so the system_busy bit will be set in the error_bits word while it is executing. Also, as with the other commands, command_word_0 will be reset to 0 at the completion of this command.

This command calculates a new coordinate transform as part of its normal processing. So if the user has changed the variable transform_num (pg. 9) or has changed the current slot of the transform table, this command will alter the coordinate transform. This side effect can be useful, or an unexpected surprise, so please be aware of it.

Please refer to the section on minimum_full_scale (pg. **9**) for details on choosing an appropriate full scale.

READ AND RESET PEAKS (11)

Calling Parameters:

peak_address	points to data to watch for peaks
Command_Word_0	0x0B00

Returned Variables:

Command_Word_0	0
minimums	minimum values seen since last peak reset
maximums	maximum values seen since last peak reset

Execution time:

40 µS	(average)
125 µS	(maximum)

Notes:

The read and reset peaks command takes the peak values which the ***JR3*** DSP stores internally, and copies those values to the variables minimums (@ 0x00d0) and maximums (@ 0x00d8). It then resets the internal minima and maxima to be the same as the current data. The minima and maxima operate in this fashion so that the user will not miss any minima or maxima.

\

READ PEAKS (12)

Calling Parameters:

peak_address	points to data to watch for peaks
Command_Word_0	0x0C00

Returned Variables:

Command_Word_0	0
minimums	minimum values seen since last peak reset
maximums	maximum values seen since last peak reset

Execution time:

40 μS	(average)
125 μS	(maximum)

Notes:

The read peaks command takes the peak values which the ***JR3*** DSP stores internally, and copies those values to the variables minimums (@ 0x00d0) and maximums (@ 0x00d8). The minima and maxima operate in this fashion so that the user will not miss any minima or maxima. To reset the minima and maxima see command (11) read and reset peaks (pg. 39).

Examples

Following are some examples of using ***JR3***'s DSP-based force sensor receivers. The examples are shown in a pseudo-code. We will use ## to indicate that the remainder of a line is a comment. We need to define several functions:

```
readData (addr)
```

which returns the 16 bit value which is at addr in the DSP's address,

```
writeData (addr, data)
```

which writes data to addr in the DSP's address space and,

```
write     stuff to write
```

which displays '*stuff to write*' to the user's terminal.

EXAMPLE #1 - GET DSP SOFTWARE VERSION

This example will write the currently executing DSP software version number. This value is stored at offset 0x00f5.

```
Write 'The Software Version # is '
Write (readData (0x00f5) / 100)
```

EXAMPLE #2 - GET SCALED FORCE DATA FOR FX

This example retrieves the full scale for force X (@ 0x0080) and applies it to the current force X data from filter #2 (@ 0x00a0).

```
Write 'Current FX from filter 2 is '
Write (readData (0x00a0) / 16384 * readData (0x0080))
```

EXAMPLE #3 - RESET OFFSETS

This example uses the reset offsets command (pg. 36) to make the sensor data of all axes equal to zero. This has the effect of removing the tare weight. It writes the value 0x0800, which is command 8, to the 0x0e7 location, which is command_word0 (pg. 13)

```
writeData (0x00e7, 0x0800)
```

EXAMPLE #4 - SET OFFSET, FORCE Z

This example sets the offset for the FZ axis. It does this by reading the current FZ data from filter 2 (@ 0x00a2) and the current FZ offset (@ 0x008a), and summing them. This value summed with the desired offset is then written back into the FZ offset location.

```
writeData (0x008a, readData (0x00a2) + readData (0x008a) - 20)
```

When setting offsets using the method shown above we need to rely on the DSP noticing that we have changed the offset. The DSP only looks for this change in offsets one in every 16 samples from the sensor. So if the sensor is sending data at 8 kHz, the DSP may take up to 2 mS to notice the changed offset. For many applications 2 mS is too slow, so we need to invoke command (7) set offsets (pg. 35) after writing the desired offsets to the DSP's address space to speed up the process. We do this by writing 0x0700, which is command 7, to the 0x0e7 location, which is command_word0

```
## first set the FZ offset to 20
writeData (0x008a, readData (0x00a2) + readData (0x008a) - 20)

## now write the rest of the offsets if desired
          ...

## finally, invoke the set offsets command
writeData (0x00e7, 0x0700)
```

EXAMPLE #5 - SET VECTOR AXES

This example will set the axes used for the vector calculations. By default the vectors V1 and V2 are a force and a moment vector respectively, each using all three axes. To change the axes used for V1 and V2 use command (9) set vector axes (pg. 37). The following example will set up V1 to use Fx and Fy and V2 to use Fx, Fy and Fz. It does this by writing 0x097b to the 0x0e7 location, which is command_word0 (pg. 13)

```
writeData (0x00e7, 0x097b)
```

The value 0x097b is constructed by adding the command value 0x0900 to the axes bits needed for V1 (1 = V1x, 2 = V1y), the axes bits needed for V2 (8 = V2x, 16 = V2y, 32 = V2z) and the bit needed to make V2 uses forces instead of moments (64 = V1 and V2 forces). 1+2+8+16+32+64 = 123 (0x007b).

EXAMPLE #6 - USE COORDINATE TRANSFORMATION

This example shows how to use a simple coordinate transform. This transform will simply rotate about the Y axis by -180°. We will be using transform table entry 0. First write the transform type for rotate Y (5) to the first entry in the transform table (0x0200). Then write the amount to transform, -180°, to the second entry. This value is encoded as -32768 as discussed in the transform section on page 26. To indicate that this is the last transform in the list, the list is terminated with a 0. Finally use command (5) use transform # (pg. 33).

```
writeData (0x0200, 5)         ## rotate about Y axis
writeData (0x0201, -32768)    ## rotate -180 degrees
writeData (0x0202, 0)         ## last transform in list
writeData (0x00e7, 0x0500)    ## use transform #0
```

EXAMPLE #7 - USE COMPLEX COORDINATE TRANSFORMATION

This example shows how to use a complex coordinate transform. This transform will start with a rotation about the Z axis by 45º. We will be using transform table entry 2. First write the transform type for rotate Z (6) to the first entry in the transform table (0x0220). Then write the amount to transform, 45º, to the second entry. This value is encoded as 8192 as discussed in the transform section on page 26. Second, we will translate along the Z axis by ½ of the sensor thickness. This will put the coordinate axis on the flange of the sensor. Then, to indicate that this is the last transform in the list, the list is terminated with a 0. Finally use command (5) use transform # (pg. 33) to effect the new coordinate transform.

```
writeData (0x0220, 6)          ## rotate about Z axis
writeData (0x0221, 8192)       ## rotate 45 degrees
writeData (0x0222, 3)          ## translate along Z axis

## translate ½ sensor thickness
writeData (0x0223, readData (0x00ff) / 2)

writeData (0x0224, 0)          ## last transform in list
writeData (0x00e7, 0x0502)     ## use transform #2
```

EXAMPLE #8 - USE A LOAD ENVELOPE

This example shows how to prepare and use a load envelope with multiple thresholds. We will be using 5 thresholds on two different axes. We will setup a positive and negative limit on Fx and Fy of 1/4 of full scale. We will setup these thresholds to each trigger two bits. We will latch one of the bits. We will also setup a threshold on the force vector at 1/2 full scale. Finally we read the threshold bits and then reset them.

```
## Start by putting the appropriate value into load envelope #2
writeData (0x0120, 0xff00)    ## latch top 8 bits
writeData (0x0121, 3)         ## we have 3 GE thresholds
writeData (0x0122, 2)         ## we have 2 LE thresholds

## first Greater or Equal threshold
writeData (0x0123, 0x0090)    ## addr of filter0 fx
writeData (0x0124, 4096)      ## 1/4 of full-scale
writeData (0x0125, 0x0101)    ## use bits 0 and 8

## second Greater or Equal threshold
writeData (0x0126, 0x0091)    ## addr of filter0 fy
writeData (0x0127, 4096)      ## 1/4 of full-scale
writeData (0x0128, 0x0202)    ## use bits 1 and 9

## last Greater or Equal threshold
writeData (0x0129, 0x0096)    ## addr of filter0 v1
writeData (0x012a, 8192)      ## 1/2 of full-scale
writeData (0x012b, 0x1010)    ## use bits 4 and 12

## first Less or Equal threshold
writeData (0x012c, 0x0090)    ## addr of filter0 fx
writeData (0x012d, -4096)     ## -1/4 of full-scale
writeData (0x012e, 0x0404)    ## use bits 2 and 10

## starting with this threshold we will spill into slot #3
## last Less or Equal threshold
writeData (0x012f, 0x0091)    ## addr of filter0 fy
writeData (0x0130, -4096)     ## -1/4 of full-scale
writeData (0x0131, 0x0808)    ## use bits 3 and 11

writeData (0x006f, 2)         ## we are using LE slot #2

write 'The current threshold status is '
write (readData (0x00f2))

## Now reset the latch bits
writeData (0x00e5, 0xff00)    ## reset top 8 bits
writeData (0x00e6, 0x00f2)    ## address of threshold bits
writeData (0x00e7, 0x0400)    ## command 4 - reset bits
```

Table 1: Summary of *JR3* DSP Data locations

	0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07
0x00	ch0time	ch0data			ch1time	ch1data		
0x08	ch2time	ch2data			ch3time	ch3data		
0x38	chEtime	chEdata			chFtime	chFdata		
0x40	'C '	'o '	'p '	'y '	'r '	'i '	'g '	'h '
0x48	't '	' '	'J '	'R '	'3 '	', '	' '	'I '
0x50	'n '	'c '	' '	'1 '	'9 '	'9 '	'4 '	0
0x58								
0x60	shunt fx	shunt fy	shunt fz	shunt mx	shunt my	shunt mz		
0x68	def fs fx	def fs fy	def fs fz	def fs mx	def fs my	def fs mz		load env #
0x70	min fs fx	min fs fy	min fs fz	min fs mx	min fs my	min fs mz		xForm #
0x78	max fs fx	max fs fy	max fs fz	max fs mx	max fs my	max fs mz		peak addr
0x80	fs fx	fs fy	fs fz	fs mx	fs my	fs mz	fs v1	fs v2
0x88	ofs fx	ofs fy	ofs fz	ofs mx	ofs my	ofs mz	ofs #	vect axes
0x90	f0 fx	f0 fy	f0 fz	f0 mx	f0 my	f0 mz	f0 v1	f0 v2
0x98	f1 fx	f1 fy	f1 fz	f1 mx	f1 my	f1 mz	f1 v1	f1 v2
0xa0	f2 fx	f2 fy	f2 fz	f2 mx	f2 my	f2 mz	f2 v1	f2 v2
0xa8	f3 fx	f3 fy	f3 fz	f3 mx	f3 my	f3 mz	f3 v1	f3 v2
0xb0	f4 fx	f4 fy	f4 fz	f4 mx	f4 my	f4 mz	f4 v1	f4 v2
0xb8	f5 fx	f5 fy	f5 fz	f5 mx	f5 my	f5 mz	f5 v1	f5 v2
0xc0	f6 fx	f6 fy	f6 fz	f6 mx	f6 my	f6 mz	f6 v1	f6 v2
0xc8	rate fx	rate fy	rate fz	rate mx	rate my	rate mz	rate v1	rate v2
0xd0	min fx	min fy	min fz	min mx	min my	min mz	min v1	min v2
0xd8	max fx	max fy	max fz	max mx	max my	max mz	max v1	max v2
0xe0	near sat	sat	rate addr	rate div	rate count	comm 2	comm 1	comm 0
0xe8	count 1	count 2	count 3	count 4	count 5	count 6	errors	count x
0xf0	warning	error	threshold	crc	rom ver #	ver no	ver day	ver year
0xf8	serial	model	cal day	cal year	units	bits	chans	thickness

0x100-0x1ff - Load envelope table (threshold monitoring), 16 entries
0x200-0x2ff - Transform table (translations and rotations), 16 entries

Description of table entrys, see text for full description and missing entries:

ch0time, ch0data	time last data for channel 0 was received, last data received for raw channel 0
shunt fx,...	shunt reading for fx channel
def fs fx,...	sensor default full scale
min fs fx,...	min full scale, at which the data will not have the lsb zero filled
max fs fx,...	max full scale, at which the data will not have the lsb truncated
fs fx,...	full scale value for fx, when fx = 16384 this is the equivalent engineering units
load env #	number of currently active load envelope
xForm #	number of the transform currently in use
peak addr	addr of the data used in finding the maxima and minima
ofs fx,...	current offset value for fx
ofs #	number of the offset currently in use
vect axes	bit map for the axes which are being used for calculating the vectors
f0 fx,f0 fy,...	decoupled, unfiltered data
f1 fx,...	fx from filter 1
rate fx,...	rate calculation for fx
min fx,..., max fx,...	minimum peak (valley) value for fx, maximum peak value for fx
near sat, sat	raw value which sets near sat bit in warning word, and sat bit in the error word
rate addr	address of data used for calculating the rate data
rate div	rate divisor, the number of samples between rate calculations
rate count	this counter counts up to rate div, and then the rates are calculated
comm2,...	command word 2, 1 and 0. Area used to send commands to ***JR3*** DSP
count1,...	counter for filter #1, 1 count = 1 filter iteration
errors	a count of data reception errors
warning, error, threshold	warning word, error word, threshold monitoring word (load envelopes)
rom ver no	version no. of data stored in sensor EEProm
ver no, ver day	software version # that the ***JR3*** DSP is running, ***JR3*** DSP software release date
serial, model	sensor serial number, and sensor model number
cal day	last calibration date of the sensor
units	engineering units of full scale, 0 is lbs, in-lbs and in*1000, 1 is Newtons, ...
bits	number of bits in sensor ADC
chans	bit map of channels the sensor is capable of sending
thickness	the thickness of the sensor

Table 2: Summary of *JR3* DSP commands

Command Name	Command Words				Other Data Sent	Data Returned
	0 hi	0 lo	1	2		
mem read	1	-	addr	-	-	cw2 = data read
mem write	2	-	addr	data	-	cw2 = data read
bit set	3	-	addr	bits	-	cw2 = data read
bit reset	4	-	addr	bits	-	cw2 = data read
use transform	5	xform #	-	-	xform table	transform_num
use offset	6	ofs #	-	-	-	ofs_num, offsets
set offset	7	-	-	-	offsets	-
reset offset	8	-	-	-	-	offsets
set vec axes	9	axes	-	-	-	vect_axes
set full-scale	10	-	-	-	full_scale fx-mz	-
read and reset peak	11	-	-	-	-	min and max
read peak	12	-	-	-	-	min and max

Column Descriptions:

Command Name is the name of the command.

Command Words are located at 0x00e5 - 0x00e7.

0 hi is the upper byte of command word 0.

0 lo is the lower byte of command word 0.

1 and 2 are command word 1 and 2.

Other Data Sent indicates which other data needs to be setup before command word 0 is written

Data Returned indicates which data is affected by the command

For all commands, write command_word_0 last. Command_word_0 will return 0 or negative to indicate command completion. For further details of each command see text.

Glossary of Terms

0x0406	Hexadecimal notation for the decimal number 1030
ADC	Analog to Digital Converter: This is the device in the sensor which converts the analog voltage produced by the strain gages into a digital signal used by the ***JR3*** DSP.
Bit Fields	Data fields in which individual bits have meanings distinct from other bits in the same data field.
Byte	An eight bit data value.
Fx, Fy, Fz	Force X, Force Y and Force Z
GE	Greater than or Equal to.
LE	Less than or Equal to.
lsb	Least Significant Bit: bit 0, or the bit with a binary value of 1. The lower case distinguishes it from LSB. (see LSB).
LSB	Least Significant Byte: In a multi-byte data value, this signifies the byte with the least value. To use a decimal example, the least significant digit of 1234 is 4. The uppercase distinguishes it from lsb.
LSW	Least Significant Word: see LSB
msb	Most Significant Bit: see lsb
MSB	Most Significant Byte: see LSB
MSW	Most Significant Word: see LSB
Mx, My, Mz	Moment X, Moment Y and Moment Z
Robot point of view	The ***JR3*** force moment sensor axes are oriented so that they form a right hand rule when the tool side of the sensor is considered to be fixed and loads are applied to the robot side. We call this the robot point of view.
Tool point of view	The tool point of view is when the robot side of the force sensor is considered to be fixed, and loads are applied to the tool side of the sensor. When using this point of view the ***JR3*** sensor will appear to have a left hand coordinate system. Use the negate transform type to adjust this if needed.
V1, V2	Abbreviations for Vector 1 and Vector 2.
WORD	A 16-bit data value.

Performance Issues

The ***JR3*** DSP-based force sensor receiver uses the very high performance ADSP-2105 digital signal processor. But even this chip has a finite amount of processing power. Therefore some compromises have been made in computing some of the data values. This section details the frequency at which the different data items are calculated. Detailing these timing factors should help the user interface to the data.

All calculation frequencies are slaved to the data rate of the sensor. The standard sensor data rate is 8 kHz. For the rest of this discussion calculation frequencies will be discussed in terms of fractions of sensor bandwidth. So 1/2 bandwidth would be 4 kHz for an 8 kHz sensor.

Sensor data is decoupled and the offsets are removed at full sensor bandwidth. Filter1 and Peak data are also calculated at full sensor bandwidth. The Rate data is calculated at the user specified fraction of sensor bandwidth of 1/x, where x can be 1 to 65536. So with a fraction of 1/1 the rate data could be calculated at full sensor bandwidth. The saturation status bits are monitored at full sensor bandwidth, while all other data is calculated at less than sensor bandwidth.

The load envelope thresholds are monitored at 1/4 sensor bandwidth. The filtered data is calculated such that each filter is calculated at 1/4 the bandwidth of the preceding filter.

Filter 1 - 1/1 bandwidth
Filter 2 - 1/4 bandwidth
Filter 3 - 1/16 bandwidth
Filter 4 - 1/64 bandwidth
Filter 5 - 1/256 bandwidth
Filter 6 - 1/1024 bandwidth

The vectors for each data set are calculated from that data set. The vectors themselves are not filtered, but they are calculated from filtered data.

Vectors for Filter 0 - 1/2 bandwidth
Vectors for Filter 1 - 1/4 bandwidth
Vectors for Filter 2 - 1/16 bandwidth
Vectors for Filter 3 - 1/64 bandwidth
Vectors for Filter 4 - 1/256 bandwidth
Vectors for Filter 5 - 1/256 bandwidth
Vectors for Filter 6 - 1/1024 bandwidth

Because the user has the ability to impose a varying processing load on the DSP, it is possible for the user to hang the DSP if too much is asked from it. The two primary culprits are the rates command, and the load envelopes. A secondary influence is the processing time taken away from the DSP by the host when the host reads data. If the host is sitting in a tight timing loop waiting for command_word_0 to change to 0, or is otherwise making heavy usage of the DSP's local bus by reading or writing to it, the DSP can be slowed down. This will not be an issue for doing normal data reads, but a very fast host in a tight loop could cause problems.

The variable count_x (pg. 14) can be used to gage how busy the processor is. Count_x increments every time the DSP searches its job queues and finds nothing to do. To be sure that all tasks are being completed, we should see at least one count on count_x for each data packet. Therefore, for an 8 kHz sensor data rate, we would want to see count_x changing at least 8000 times per second.

As of software version 2.0, each count of count_x represents approximately 5.4 uS of free time. So for every count above the 8,000 Hz baseline, we can impose approximately 5.4 uS more load per second on the processor. Each iteration of the rate routine takes approximately 3.3 uS, while each additional load envelope threshold takes approximately 0.6 uS.

Example 1: For an 8 kHz sensor, the configuration under test is showing count_x changing at about 20,000 Hz. Therefore we have approximately (20,000 Hz - 8,000 Hz) * 5.4 uS = 64.8 mS/second available. How many more load envelope thresholds can we process? Each threshold takes 0.6 uS * 2,000 Hz = 1.2 mS / second, so 64.8 / 1.2 = 54 more thresholds could, at best, be calculated.

Example 2: For an 8 kHz sensor, a test at ***JR3*** of a system with the rates calculating at 1/800 bandwidth and with no load envelope thresholds, showed count_x changing at 30,500 Hz. So if the rates were changed to 1/1 bandwidth how many load envelope thresholds could be calculated? (30,500 Hz - 8,000 Hz) * 5.4 uS = 121.5 mS / second available. We need 799/800 * 8,000 more rates at 3.3 uS per rate gives 26.4 mS / second for the rates, which leaves 121.5 - 26.4 = 95.1 mS / second. From Ex. 1 the thresholds are 1.2 mS / second, which gives 95.1/1.2 = 79 thresholds maximum.

These examples are not meant to recommend the use of more than the 50 thresholds suggested on pg. 17. But, with the proper testing in the user's application, more thresholds could probably be used, especially if rate calculations are not needed.

JR3's DSP-based Force Sensor Receiver Card for the VMEbus

This appendix describes the setup and operation of the Force Sensor Receiver for the VMEbus. It covers only those aspects of the card that differ from other members of ***JR3***'s DSP-based force sensor receiver family.

FORM FACTOR

The receiver card plugs into a single wide (4H) double high (6U), VMEbus backplane slot. The receiver uses only the P1 connector. The receiver interfaces to the sensor through an 8-pin (RJ-45) modular jack on the front panel. Also included is an indicator lamp which shows green if the sensor is plugged in and powered up. The lamp shows red if the sensor is not plugged in, and is off if the sensor power is off.

POWER

The receiver requires no external power. It draws power directly from the VMEbus. The board uses the following voltages and currents:

5V - 870 mA typical
12V - 25 mA typical (w/o sensor)
-12V - 5 mA typical (w/o sensor)

The sensor will also draw anywhere from 200 to 400 mA from the +12V, and possibly 100 mA from the -12V.

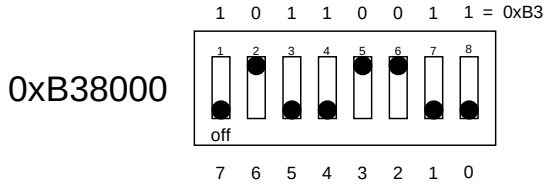
SOFTWARE INTERFACE

The ***JR3*** DSP receiver's address space is mapped directly into the VME A24 address space. Therefore, the offsets given in the section describing the memory interface need to be multiplied by 2 and then added to the base address of the card as selected in the next section. The ***JR3*** DSP receiver is then simply accessed as memory on the VMEbus.

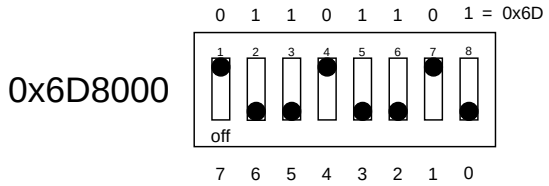
ADDRESS SELECTION

The VMEbus receiver goes into a 32k-byte block in the VME's A24 address space. The board can reside in the upper half of any 64k-byte block. The receiver adheres to the VMEbus A24 and D16 specifications with one exception: The board will respond to 8-bit data accesses with a Bus Error. The board responds to 4 address modifier codes: 0x39, 0x3a, 0x3d, 0x3e.

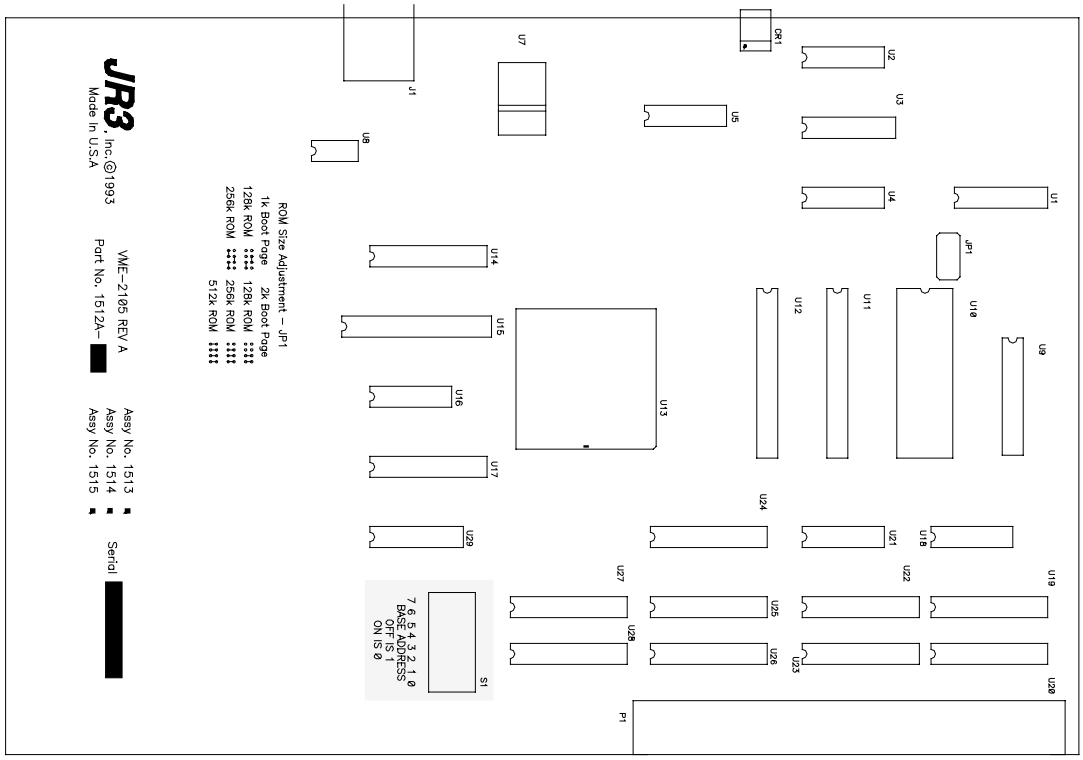
The architecture of the board from the VMEbus perspective, is 32k bytes laid out as 16k 2-byte words. The base address is selected by dip switches on the receiver. The dip switches are labeled on the silk screen as 7 through 0. These switches correspond to address bits A23 through A16. These switches make the bit take the value of 1 when they are off and 0 when they are on. Example: to set a base address of 0xB38000 the switches would be set to:



to the set the address to 0x6D8000:



Outline Drawing Showing Address DIP Switches for *JR3* VMEbus, DSP-based receiver card



JR3's DSP-based Force Sensor Receiver Card for the ISA (IBM-AT) bus.

This appendix describes the setup and operation of the Force Sensor Receiver for the ISA bus. It covers only those aspects of the card that differ from other members of ***JR3***'s DSP-based force sensor receiver family.

FORM FACTOR

The receiver card plugs into a 16 bit connector on the ISA bus. The receiver uses both the 62-pin and the 36-pin connectors. The receiver interfaces to the sensor through an 8-pin (RJ-45) modular jack on the back panel.

POWER

The receiver requires no external power. It draws power directly from the ISA bus. The board uses the following voltages and currents:

5V - 650 mA typical
12V - 25 mA typical (w/o sensor)
-12V - 5 mA typical (w/o sensor)

The sensor will also draw anywhere from 200 to 400 mA from the +12V, and possibly as much as 100 mA from the -12V.

SOFTWARE INTERFACE

The architecture of the DSP receiver board from the ISA bus perspective is two 16-bit wide registers. The address register is at I/O addresses zero and one relative to the base address. The data register is at addresses two and three relative to the base address. Therefore if the board base address is 0x314, then the 16-bit port at 0x314 is the address port, while the 16-bit port at 0x316 is the data port.

To read a data value from the DSP's address space, first write the address of the desired data to the address port, then read the data from the data port. Writing data is done in an analogous manner.

The ISA receiver board also has a block transfer ability. To read or write data from consecutive addresses, simply write the address of the first data to the address register, then read or write each succeeding data value from or to the data register. The only caveat with this operation is that only the bottom 8 bits of the address will update. Therefore if the address 0xfe were written to the address register, the first data read would be from address 0xfe, the second from 0xff, and third from 0x00.

The following routine will read data from the board. The address should be in register AX, the data will be returned in AX.

```
MOV       dx, 0x314 ; Board base address is 0x314
OUT       dx, ax              ; Write addr to addr reg
ADD       dx, 2               ; Data reg is 2 above addr reg
IN        ax, dx              ; Read data from board
```

The following is a 'C' version of data read.

```
#include  <dos.h>
#define   baseAddr  0x314
int       getData(int addr)
{
          outport(baseAddr, addr);
          return inport(baseAddr+2);
}
```

The following routine will write data to the board. The address should be in register AX, the data in register BX.

```
MOV       dx, 0x314 ; Board base address is 0x314
OUT       dx, ax              ; Write addr to addr reg
ADD       dx, 2               ; Data reg is 2 above addr reg
MOV       ax, bx              ; Put the data into ax
OUT       dx, ax              ; Write data to the board
```

The following is a 'C' version of data write.

```
#include  <dos.h>
#define   baseAddr 0x314
int       putData(int addr, int data)
{
          outport(baseAddr, addr);
          outport(baseAddr+2, data);
}
```

The following will read a block of data from the board. The address of the beginning of the data block should be in register AX. Registers ES:DI should point to a buffer where the data block will be stored. Register CX should be the number of words of data to transfer. This code uses the INSW instruction which is not available on an 8086/8088. But since this board must go into a 16-bit expansion slot, the processor should be an 80286 or above.

```
MOV       dx,0x314            ; Board base address is 0x314
AND       ax,$3FFF            ; AX should only have 14 bits
OUT       dx,ax               ; Write the address
ADD       dx,2                ; Point to data port
PUSHF                         ; Save flags
CLD                           ; Clear the direction flag
CLI                           ; Hold interrupts
REP       INSW                ; Do the data transfer
POPF                          ; Restore the flags
```

The following will write a block of data to the board. The destination address of the data block should be in register AX. Registers DS:SI should point to a buffer where the data block is stored. Register CX should be the number of words of data to transfer. This code uses the OUTSW instruction which is not available on an 8086/8088. But since this board must go into a 16-bit expansion slot, the processor should be an 80286 or above.

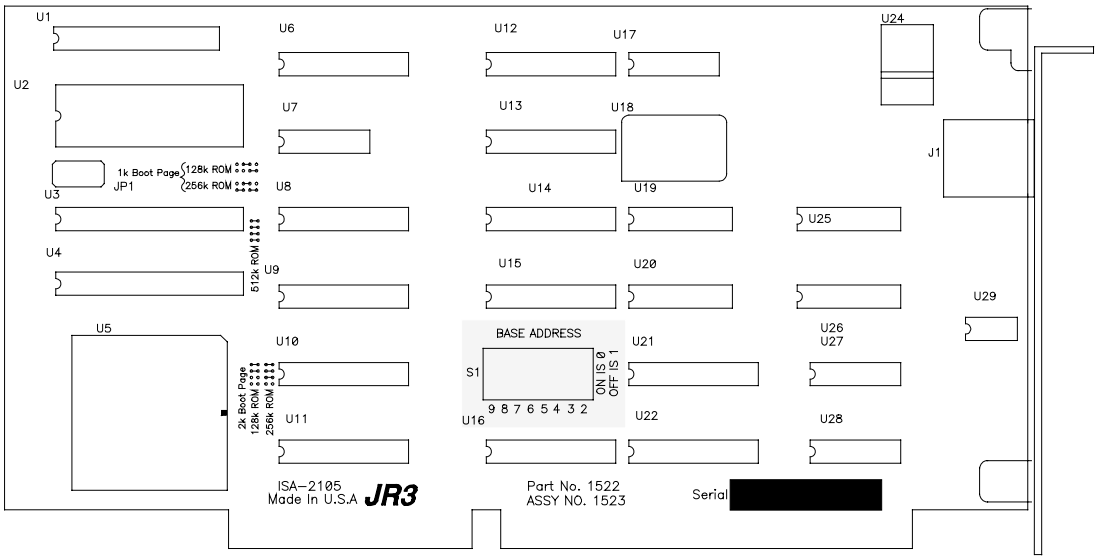
```
MOV       dx,0x314          ; Board base address is 0x314
AND       ax,$3FFF          ; AX should only have 14 bits
OUT       dx,ax             ; Write the address
ADD       dx,2              ; Point to data port
PUSHF                       ; Save flags
CLD                         ; Clear the direction flag
CLI                         ; Hold interrupts
REP       OUTSW             ; Do the data transfer
POPF                        ; Restore flags
```

ADDRESS SELECTION

The ISA bus receiver goes into 4 consecutive I/O addresses. The board can reside in any 4 byte wide space from addresses 0x000 to 0x3ff. The base address is selected by dip switches on the receiver. The dip switches are labeled on the silk screen as 9 through 2. These switches correspond to address bits A9 through A2. These switches make the bit take the value of 1 when they are off and 0 when they are on. Example: to set a base address of 0x314 the switches would be set to:

to the set the address to 0x260:

Outline Drawing Showing Address DIP Switches for *JR3* ISA bus, DSP-based Receiver Card



JR3's DSP-based Force Sensor Receiver Card for the Stäubli UNIVAL Robot Controller.

This appendix describes the setup and operation of the Force Sensor Receiver for the Stäubli UNIVAL controller. It covers only those aspects of the card that differ from other members of ***JR3***'s DSP-based force sensor receiver family.

FORM FACTOR

The receiver card plugs into a standard Stäubli UNIVAL controller slot. This slot is similar to a VMEbus slot except that the card is 220mm tall instead of 160 mm tall. The receiver uses only the P1 connector. The receiver interfaces to the sensor through an 8-pin (RJ-45) modular jack on the front panel. Also included is an indicator lamp which shows green if the sensor is plugged in and powered up. The lamp shows red if the sensor is not plugged in, and is off if the sensor power is off.

POWER

The receiver requires no external power. It draws power directly from the Stäubli UNIVAL controller. The board uses the following voltages and currents:

5V - 870 mA typical
12V - 25 mA typical (w/o sensor)
-12V - 5 mA typical (w/o sensor)

The sensor will also draw anywhere from 200 to 400 mA from the +12V, and possibly 100 mA from the -12V.

SOFTWARE INTERFACE

The ***JR3*** DSP receiver's address space is mapped directly into the Stäubli UNIVAL A24 address space. Therefore, the offsets given in the section describing the memory interface need to be multiplied by 2 and then added to the base address of the card as selected in the section on address selection (pg. 59). The ***JR3*** DSP receiver is then simply accessed as memory on the Stäubli UNIVAL bus.

The following are some example programs to show basic interfacing to the ***JR3*** DSP receiver for the Stäubli UNIVAL bus. They are excerpted from the file getdata.val available from ***JR3***.

This program must be run to initialize the software environment for the ***JR3*** DSP receiver.

```
; To set a new base address change the two most significant
; digits in the following hex number to agree with the value
; set by the base address switch on the JR3 DSP receiver board.
; i.e. if switches set to ^Hb0
    !JR3.base = ^Hb08000
; Enable 'Z' instruction set
    parameter terminal = ^H50001
```

This program shows how to read a single data value from the ***JR3*** DSP receiver:

```
.program JR3.read.port (!data.addr, data)
    data = setr (!zpeekw (!JR3.base+!data.addr*2))
    IF data > 32767 THEN
        data = data - 65536
    END
```

This program shows how to write a single data value to the ***JR3*** DSP receiver:

```
.program JR3.write.port (!data.addr, data)
    local !data
    IF data > 32767 THEN
        !data = 32767
    ELSE
        IF data < -32768 THEN
            !data = 32768
        ELSE
            IF data < 0 THEN
                !data = !seti (data + 65536)
            ELSE
                !data = !seti (data)
            END
        END
    END
    zpokew (!JR3.base+!data.addr*2) = !data
```

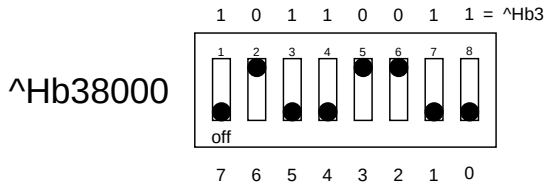
The following code will return the current value of FX in engineering units

```
    CALL JR3.read.port (^H90, tmp1)
    CALL JR3.read.port (^H80, tmp2)
    fx = tmp1 / 16384 * tmp2
```

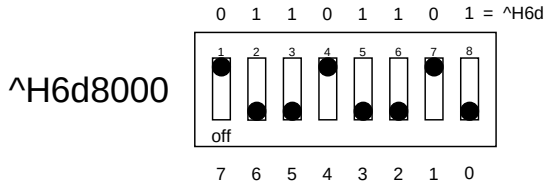
ADDRESS SELECTION

The Stäubli UNIVAL receiver goes into a 32k-byte block in the Stäubli UNIVAL's A24 address space. The board can reside in the upper half of any 64k-byte block. The receiver adheres to the VMEbus A24 and D16 specifications with one exception: The board will respond to 8-bit data accesses with a Bus Error. The board responds to 4 address modifier codes: 0x39, 0x3a, 0x3d, 0x3e.

The architecture of the board from the Stäubli UNIVAL controller perspective is 32k bytes laid out as 16k 2-byte words. The base address is selected by dip switches on the receiver. The dip switches are labeled on the silk screen as 7 through 0. These switches correspond to address bits A23 through A16. These switches make the bit take the value of 1 when they are off and 0 when they are on. Example: to set a base address of ^Hb38000 the switches would be set to:

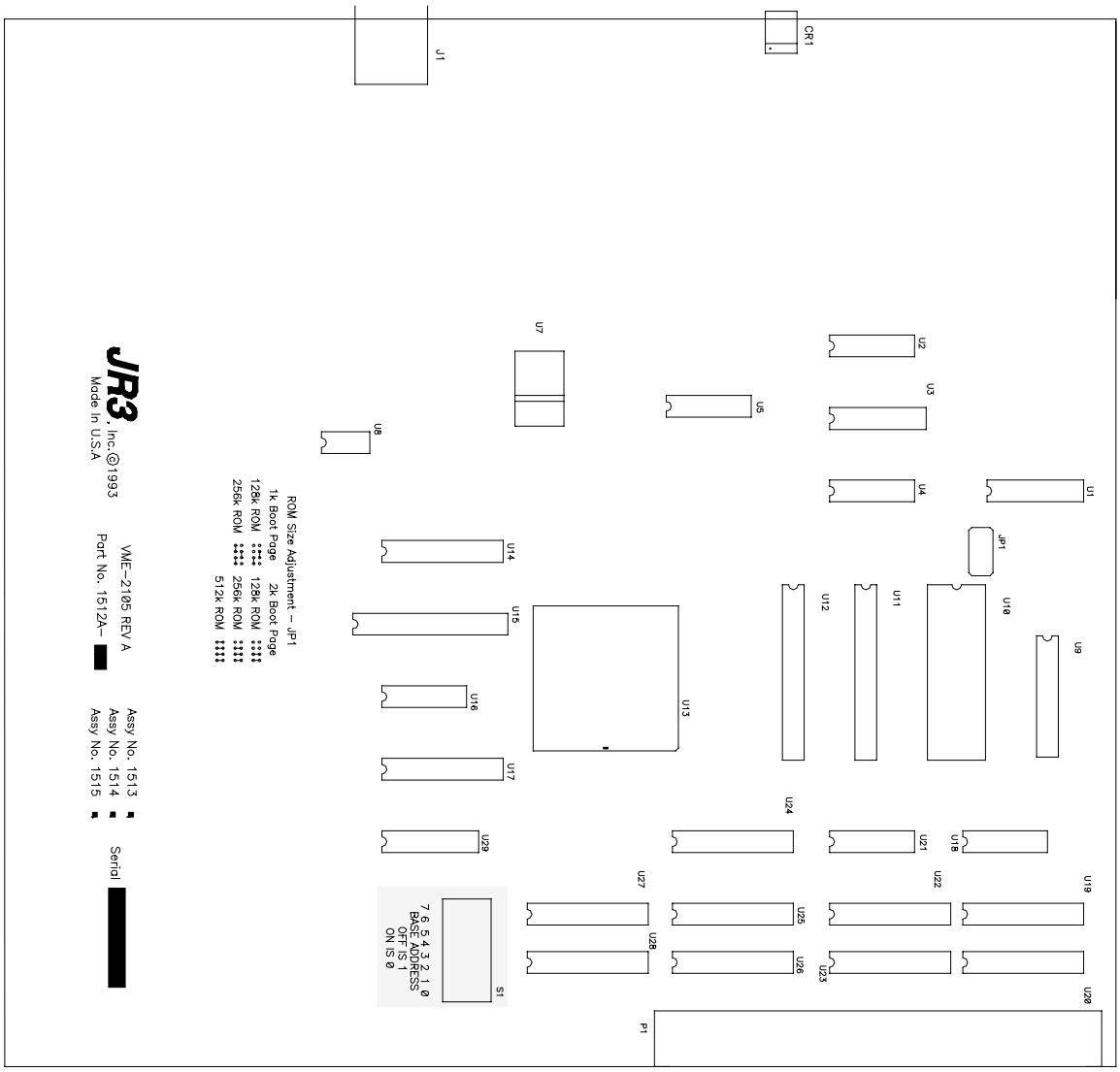


to the set the address to ^H6d8000:



Choosing a base address is somewhat more complicated than setting it. It is impossible for ***JR3*** to recommend a suitable address for all applications but if the user has no knowledge of his memory map, ***JR3*** suggests trying ^Hb38000.

Outline Drawing Showing Address DIP Switches for *JR3* Stäubli UNIVAL, DSP-based receiver card



JR3's DSP-based Force Sensor Receiver Card for the PCI bus

This appendix describes the setup and operation of the Force Sensor Receiver for the PCI bus. It covers only those aspects of the card that differ from other members of ***JR3***'s DSP-based force sensor receiver family.

FORM FACTOR

The receiver card plugs into a standard length PCI Bus slot. It is a 5V, 33 MHz, 32-Bit PCI card. It does **not** support 3.3 Volt or 66 MHz operation. The receiver interfaces to the sensor through an 8-pin (RJ-45) modular jack on the back panel. Also included are two indicator led. The first led shows green if the DSP has successfully booted. The second led shows green if the sensor is plugged in and powered up.

POWER

The receiver requires no external power. It draws power directly from the PCI bus. The board uses the following voltages and currents:

5V - 870 mA typical
12V - 25 mA typical (w/o sensor)
-12V - 5 mA typical (w/o sensor)

The sensor will also draw anywhere from 200 to 400 mA from the +12V, and possibly 100 mA from the -12V.

SOFTWARE INTERFACE

The ***JR3*** PCI DSP receiver's address space is mapped directly into the PCI data address space. The 16 bit wide DSP memory is mapped into the 32 bit wide PCI bus. Therefore, the offsets given in the section describing the memory interface need to be multiplied by 4 and then added to the base address of the card + 0x6000 hex. The ***JR3*** DSP receiver is then simply accessed as memory on the PCI bus.

ADDRESS SELECTION

The PCI bus is a plug and play bus, there is no user configuration necessary. The receiver goes into a 512k-byte block in the PCI's address space. The receiver responds to 4 command types: 0x6 Memory Read, 0x7 Memory Write, 0xa Configuration Read and 0xb Configuration Write.

CODE DOWNLOAD

The ***JR3*** PCI DSP receiver's card code must be downloaded from the host. After any reset, and before the card can be used, it must have code downloaded. C Source to perform this function is available from ***JR3***