G-Rex Programming Manual

Mariss Freimanis (Geckodrive Inc.)
Steve Hardy (Xarach Controls)
September 3, 2005

This manual describes the programming interface for the Geckodrive G-REX series of motion controllers. The G-REX is field-programmable, meaning that its functionality can be changed and upgraded at any time, subject to limitations of the hardware. This manual pertains to revision 3.0 of the standard (Geckodrive) reference design configuration.

The G-Rex uses an FPGA for hardware step pulse generation and I/O control. It also has an embedded high-speed RISC type processor to handle real-time closed-loop computations. The G101 appears as an 8-bit peripheral and requires an on-board MCU for control and communication. The MCU socket is pinned-out for Rabbit Semiconductor RCM3600 / RCM3700 series core modules but it's not restricted to them. Any MCU that can interface to 8-bit data, 4-bit address, /CS /RD, /WR and INT is suitable.

The G101 generates 2^16 evenly spaced CW and CCW frequencies in each of 8 ranges and reads quadrature inputs at a 1MHz rate for 6 axis simultaneously. These frequencies can be updated 2,048 times a second. In addition there are 16 general-purpose outputs (0 to 24VDC, 100mA), 22 general-purpose inputs (5V, filtered, OVP), 4 8-bit analog inputs (0 to 5V), 4 8-bit analog outputs (0 to 5VDC, 10mA) and a USB interface.

The table of contents entries are in red for G101 write registers and in blue for G101 read registers along with their hex addresses. This is to facilitate quick look-up of these registers in this manual.

G101 Programming Overview:

The G101 is an 8-bit peripheral board that integrates all the features necessary for a complete multi-axis motion control system. The G101 is designed to be controlled by a microcomputer and has an interface socket to accept an MCU board. This socket (CN6) is a 40-position 2X20 .1" dual-row connector.

Though any MCU can be used with the G101, the MCU socket (CN6) is pinned-out to accept Rabbit Semiconductor RCM-3600 series and RCM-3700 series MCU modules directly. If these are used, Geckodrive can provide the firmware for FPGA configuration and USB support. See http://rabbitsemiconductor.com/ for more information on these MCUs.

The G101 uses a fairly standard primary MCU interface:

1 8-bit bi-directional data bus
2 4-bit address bus
3 /RD, /WR and /CS
4 /INT external interrupt service request, constant rate
5 /INT_AUX external interrupt, can be variable rate

The G101 also has several secondary interfaces. These are:

1) FPGA configuration JTAG interface. The G101 uses a Xilinx XC3S50 FPGA and depends on the MCU to configure the FPGA with a Geckodrive supplied configuration file. The configuration interface accommodates 2.5V to 5V bus interface for these configuration interface signals. Please see http://www.xilinx.com for details. These signals are:

1 TMS
2 TDI
3 TDO
4 TCK
5 DONE

2) USB interface. The G101 has an on-board USB connector (type B) and uses a Future Technology Devices Inc. FT245BM USB interface IC. This is the main external communication interface unless the MCU used has its own interface, in which case the on-board one can be ignored. The FT245BM shares the bi-directional data bus with the Xilinx FPGA. The interface is:

1 8-bit bi-directional data bus
2 /RD (separate from the FPGA /RD)
3 WR (separate from the FPGA /WR) Note: The USB WR is asserted as a '1', not a '0'.
4 /TxE (FIFO transmitter empty)
5 /RxF (FIFO receiver full)

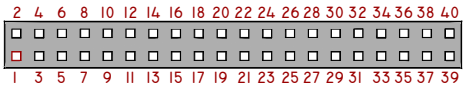
Please see http://www.ftdichip.com/Products/FT245BM.htm for full details on the USB interface.

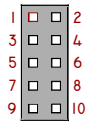
3) Miscellaneous. This interface provides:

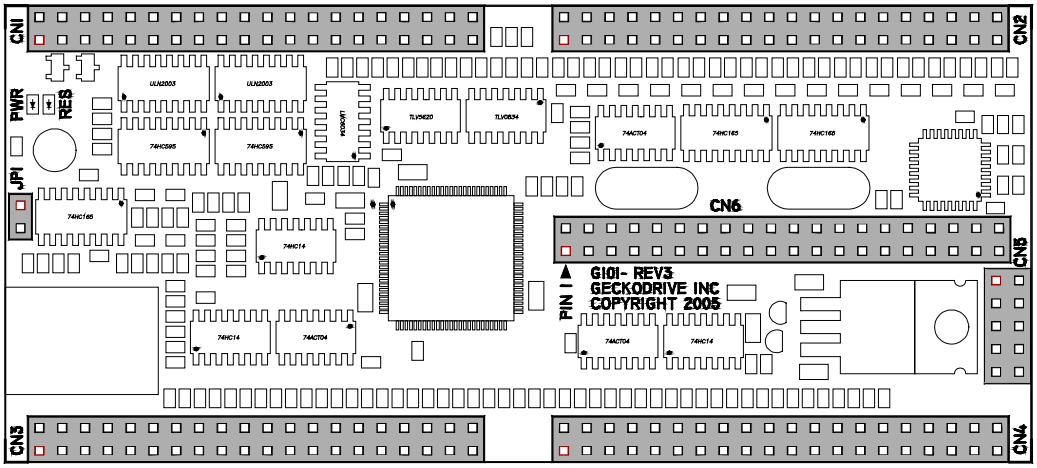
1 Power supply ground and 5VDC
2 External SDRAM backup battery supply
3 Hard Reset
4 All other uncommitted pins to the MCU interface connector

G101 Connector Description:

The G101 has 6 headers used for I/O connections. They are grouped according to function and numbered CN1 through CN6. JP4 is a 2-pin header used to enable the USB port to power the G101. The figure below shows the location of these headers and the pin numbering.







Connector CN1: Output Group			
Pin Function	**Pin #**		**Pin Function**
5 to 24VDC	1	2	Output 16
5 to 24VDC	3	4	Output 15
5 to 24VDC	5	6	Output 14
5 to 24VDC	7	8	Output 13
5 to 24VDC	9	10	Output 12
5 to 24VDC	11	12	Output 11
5 to 24VDC	13	14	Output 10
5 to 24VDC	15	16	Output 9
5 to 24VDC	17	18	Output 8
5 to 24VDC	19	20	Output 7
5 to 24VDC	21	22	Output 6
5 to 24VDC	23	24	Output 5
5 to 24VDC	25	26	Output 4
5 to 24VDC	27	28	Output 3
5 to 24VDC	29	30	Output 2
5 to 24VDC	31	32	Output 1
Op-Amp Vcc	33	34	Analog Out 1
5 VDC	35	36	Analog Out 2
GND	37	38	Analog Out 3
GND	39	40	Analog Out 4

Table 1. CN1: Output Group

Connector CN2: Input Group			
Pin Function	**Pin #**		**Pin Function**
GND	1	2	Analog In 1
GND	3	4	Analog In 2
GND	5	6	Analog In 3
GND	7	8	Analog In 4
GND	9	10	Input 1
GND	11	12	Input 2
GND	13	14	Input 3
GND	15	16	Input 4
GND	17	18	Input 5
GND	19	20	Input 6
GND	21	22	Input 7
GND	23	24	Input 8
GND	25	26	Input 9
GND	27	28	Input 10
GND	29	30	Input 11
GND	31	32	Input 12
GND	33	34	Input 13
GND	35	36	Input 14
GND	37	38	Input 15
GND	39	40	Input 16

Table 2. CN2: Input Group

Connector CN3: Axis X, Y, Z Group			
Pin Function	**Pin #**		**Pin Function**
+USB	1	2	GND
-USB	3	4	GND
X-LIMIT SW	5	6	GND
X-DIRECTION	7	8	5 VDC
X-STEP PULSE	9	10	GND
X-ENC CH_A	11	12	5 VDC
X-ENC CH_B	13	14	GND
X-ENC CH_I	15	16	5 VDC
Y-LIMIT SW	17	18	GND
Y-DIRECTION	19	20	5 VDC
Y-STEP PULSE	21	22	GND
Y-ENC CH_A	23	24	5 VDC
Y-ENC CH_B	25	26	GND
Y-ENC CH_I	27	28	5 VDC
Z-LIMIT SW	29	30	GND
Z-DIRECTION	31	32	5 VDC
Z-STEP PULSE	33	34	GND
Z-ENC CH_A	35	36	5 VDC
Z-ENC CH_B	37	38	GND
Z-ENC CH_I	39	40	5 VDC

Table 3. CN3: Axis X, Y, Z Group

Connector CN4: Axis A, B, C Group			
Pin Function	**Pin #**		**Pin Function**
GND	1	2	GND
5 VDC	3	4	5 VDC
A-LIMIT SW	5	6	GND
A-DIRECTION	7	8	5 VDC
A-STEP PULSE	9	10	GND
A-ENC CH_A	11	12	5 VDC
A-ENC CH_B	13	14	GND
A-ENC CH_I	15	16	5 VDDC
B-LIMIT SW	17	18	GND
B-DIRECTION	19	20	5 VDC
B-STEP PULSE	21	22	GND
B-ENC CH_A	23	24	5 VDC
B-ENC CH_B	25	26	GND
B-ENC CH_I	27	28	5 VDC
C-LIMIT SW	29	30	GND
C-DIRECTION	31	32	5 VDC
C-STEP PULSE	33	34	GND
C-ENC CH_A	35	36	5 VDC
C-ENC CH_B	37	38	GND
C-ENC CH_I	39	40	5 VDC

Table 4. CN4: Axis A, B, C Group

Connector CN6: Microprocessor Group			
Pin Function	**Pin #**		**Pin Function**
DB 7	1	2	DB 6
DB 5	3	4	DB 4
DB 3	5	6	DB 2
DB 1	7	8	DB 0
FPGA TMS	9	10	FPGA TCK
N.C.	11	12	A 0
A 1	13	14	A 2
A 3	15	16	N.C.
USB TxE	17	18	USB WR
USB RxF	19	20	USB /RD
FPGA TDI	21	22	FPGA TDO
N.C.	23	24	N.C.
/CS	25	26	RES B
FPGA DONE	27	28	/INT MAIN
/INT AUX	29	30	N.C.
N.C.	31	32	/IOWR
/IORD	33	34	N.C.
N.C.	35	36	N.C.
N.C.	37	38	GND
5 VDC	39	40	GND

Table 6. CN6: Microprocessor Group

Connector CN5: Miscellaneous Group			
Pin Function	**Pin #**		**Pin Function**
CN6 pin 36	1	2	CN6 pin 37
CN6 pin 31	3	4	CN6 pin 34
CN6 pin 24	5	6	CN6 pin 35
Cn6 pin 23	7	8	CN6 pin 30
5 VDC	9	10	GND

Table 5. CN5: Miscellaneous Group

I/O Hardware Interface Overview:

The G101 uses 6 I/O interface circuit blocks for I/O filtering and buffering. These are shown in the 6 figures below:

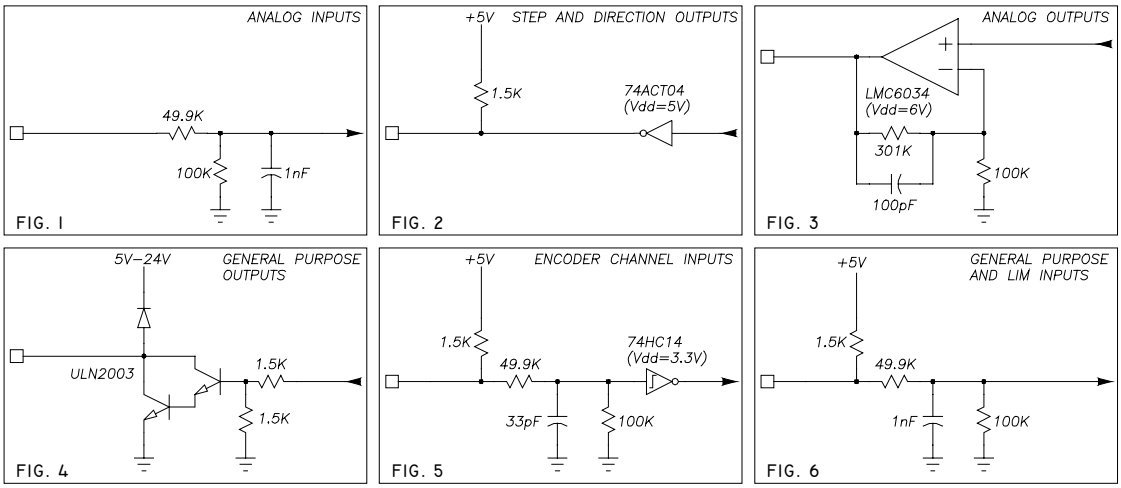


Fig. 1 Analog Inputs:

The G101 has 4 Analog to Digital inputs. The ADCs use a 3.3V power supply voltage so the analog 0 to +5V input range must be attenuated to a 0 to +3.3V range. The 49.9K and 100.0K resistors form the voltage divider to scale down the input range. The size of the resistors serves to protect the CMOS ADCs from damage for up to +/- 50V input voltages. The 1nF capacitor low-pass filters the inputs to limit noise.

Fig. 2 Step and Direction Outputs:

The 74ACT04 Step and Direction drivers are specified to +/-24 mA sink and source current capability while maintaining TTL voltage level limits. The input threshold is 1.5V, which nicely matches the FPGA 0V and 3.3V output levels. This makes these outputs suitable for driving moderate length cables to the motor drive Step and Direction inputs.

Fig. 3 Analog Outputs:

The G101 buffers the DACs with an LMC6034 CMOS quad op-amp. The op-amp level-shifts and amplifies the Analog Outputs to a 0V to 5V scale. It is recommended to have an external power supply voltage of 6VDC or more for the op-amp for other than the lightest of loads. The op-amp power supply is available on CN1 pin 33. If not, connect CN1 pin 33 to the adjacent +5VDC pin 35.

Fig. 4 General Purpose Outputs:

The G101 uses ULN2003A output drivers for the General Purpose Outputs. The outputs are rated at 100 mA maximum current and at a voltage of +5V to +24V. The external +5V to +24V power supply connects to any of the CN1 connector pins marked ‘5 to 24 VDC’. The ULN2003A drivers have integral collector to power supply ‘catch’ diodes, so inductive loads can be driven directly.

Fig. 5 Encoder Channel Inputs:

The Encoder Channel Inputs accept 5V logic inputs. These inputs have a 1.5K pull-up resistors and attenuate and filter the signals to 3.3V logic levels. The inputs are low-pass filtered and then 'squared-up' by the Schmidt trigger 74HC14 inverters. The maximum input frequency is 1 MHz.

The Encoder Channel inputs are intended for quadrature encoders but can also function as Up / Down counter inputs or as direct general-purpose inputs.

Encoders and Step Motors

The encoders also can be paired with step motors. There are several improvements that result when a step motor is mounted with an encoder. In the simplest case the encoder can verify the step motor has moved to the commanded location or give an indication back to the microprocessor the motor has stalled. If a motor does stall, the encoder counter contents can be used to restore the motor to the originally requested position after the cause of the stall is cleared.

With almost no increase in complexity, the G101 can be programmed to use the Axis Lead / Lag registers. These registers indicate how much torque load the motor the motor is under while it is running or even stopped. If a 10-microstep drive is used and the motor is mounted with a 500-line encoder, then motor load can be resolved to 2.5% accuracy. Better yet, the Axis Lead / Lag registers natively return a 'percentage of available torque being used' information. As step motor speed increases, the motor's available torque decreases. What is of interest is to always know how much (what percentage) of torque is being utilized. If it approaches 100% then a stall is impending; the Axis Lead / Lag registers give warning before the motor stalls, while there is still time to take corrective action.

The most complex solution is to close the loop on the step motor and the design of the G101 has taken that into consideration. Doing so turns a step motor into a high pole-count AC servo motor and all sorts of performance advantages occur then. The motor can accelerate / decelerate at the maximum available torque rate, low speed and mid-band resonances are suppressed and the servo 'stiffness' becomes equal to ¼ the encoder line-count.

A further enhancement would be to take advantage of the step motor's inverse speed-torque characteristic, that being the slower you go the more torque you have available. The microprocessor could adaptively slow the motor if it were to have an impending overload. This would cause the motor to 'back-up' along its speed-torque curve to where more torque is available, reducing the 'percentage of usable torque being used' value. In effect, the motor would become unstallable. The design of the G101 intrinsically insures a multi-dimensional vector move (constant contouring) path will be honored. If an overload causes a slowdown, the intended path would still be followed at a slower rate. Remove the overload and the speed along the path will be restored to the original value.

Other Encoder Input Uses

If a motor is mounted with an encoder, the motor drive can be disabled (free-wheeled) and then the motor can be manually turned with a hand-wheel. The encoder counter will keep track of the position, which then can be read-out as a DRO (digital read-out). This information can be read-back by the controlling PC, (digitizing) cause the axis motor to move to the new position, restore to the original position, etc.

Fig. 6 General Purpose Inputs:

These inputs are impedance protected, and bandwidth limited to reduce the effect of noise. In some cases, additional capacitance to ground may be required for very noisy inputs, such as when a limit switch is routed in the same multi-conductor cable as the motor driver outputs.

Since there is a pull-up resistor on each input, the conventional arrangement is to have a switch contact short the input to ground, or have an open-collector output driving the input. Push-pull drivers (CMOS or TTL) should be capable of switching between +5V and ground

G-REX FPGA Design and Features:

The FPGA chip is the core of the G-REX design, and accounts for its flexibility. An FPGA is a programmable logic device. Unlike some other PLDs, an FPGA is generally based on a static RAM configuration. This is volatile storage, and needs to be re-loaded at each power-up. This is one of the roles of the supporting MCU.

Free FPGA synthesis and programming tools are available from Xilinx, so you are free to devise your own hardware implementation. Since this requires fairly specialized skills, Geckodrive provides a reference design in binary form, which is suitable for direct loading into the FPGA using an implementation of Xilinx's JTAG programmer (see Xilinx application note XAPP058 at http://direct.xilinx.com/bvdocs/appnotes/xapp058.pdf).

The rest of this manual refers specifically to the reference design; in particular version/release 1.0. Geckodrive reserves the right to make modifications and improvements to this design.
The design incorporates the following major elements:

1 Step/direction pulse generator

2 Quadrature decoder, counters

3 White Heat sequencer

4 Peripheral I/O

"White Heat" is the code name for a dedicated sequencer (or "coprocessor") which runs within the FPGA. This is a true processor, including conditional branch, arithmetic and logical operations. Owing to limited amounts of memory in the FPGA, a White Heat program is limited to 1024 instructions. It is a Harvard architecture processor, therefore there is a separate program and data address space. The data space is divided into two sub-spaces: working set and shared memory MCU interface. There is another address space which is used to access the other hardware within the FPGA (peripherals).

White Heat is described in detail in its own chapter, on page 25. It is fully programmable without requiring a rebuild of the FPGA configuration, however the reference design from Geckodrive includes a default White Heat program within the FPGA reference configuration, so there is no need for the MPU to deal with White Heat. The following section makes some reference to White Heat, since it is used to mediate between the MCU and the general purpose digital and analog I/O.

G-REX FPGA Register Overview

To conserve address space, the G-REX uses 6 or 12-deep byte-wide or word-wide shift registers for data bus reads or writes of certain register locations. If a multi-byte register address is accessed, it must be read from or written to 6 or 12 times within an interrupt period. It does not matter if 8-bit value read / writes are successive or not so long as the address is read / written the required number of times. It is also OK not to access a multi-byte

register if its contents are not needed. This of course does not apply to single-byte registers. 16-bit (word) data (LSB, MSB) **must** be written / read LSB first, then MSB, with no intervening access between the LSB and MSB.

Example: It may be necessary to know the contents of the Axis Input for the Z axis (axis number '2'). Reading I/O address "0x8" gives the X axis contents, the 2nd read from "0x8" returns the Y axis contents, the 3rd read returns the needed Z axis. Continuing on, the 4th , 5th and 6th reads would return the A, B and C axis results. A further read will start over from the X axis.

Contents read from the registers usually reflect their state at the instant of the beginning of the current interrupt cycle. Contents written to the registers will apply simultaneously at the beginning of the next interrupt cycle. The exceptions to this are the global control register (0) and step clock override, which take effect immediately upon being written, without waiting for the next interrupt cycle.

Address	Function	Bytes	Ref Page	Initial Value
0x0	Global Control	1	9	0x13
0x1				
0x2	Step Clock Override (LSB)	1	10	0xFF
0x3	Step Clock Override (MSB)	1		0xFF
0x4	White Heat Address (LSB)	1	11	undefined
0x5	White Heat Address (MSB)	1		undefined
0x6	White Heat Data bit 8	1	12	undefined
0x7	White Heat Data bit 0-7	1		undefined
0x8	Counter Control (LSB)	12	13	0x00
0x9	Counter Control (MSB)	12		0x00
0xA				
0xB				
0xC				
0xD	Step Generator Control	6	14	0x00
0xE	Axis Step Pulse Rate (LSB)	6	15	0x00
0xF	Axis Step Pulse Rate (MSB)	6		0x00

Table 7. Write Register Summary

Address	Function	Bytes	Ref Page
0x0	Global Status	1	16
0x1	FPGA Configuration Vers.	1	17
0x2			
0x3	White Heat Read Data	1	17
0x4			
0x5			
0x6			
0x7			
0x8	Axis Inputs	6	17
0x9			
0xA	Axis Counters (LSB)	6	18
0xB	Axis Counters (MSB)	6	
0xC			
0xD	Axis Lead/Lag	6	18
0xE			
0xF			

Table 8. Read Register Summary

Global Control Register (1-byte, WR, Address 0x0):

This register is normally written during initialization; it sets:

1) The external interrupt rate.
2) The secondary watchdog timer enable.
3) The Analog to Digital Converters to either differential or single-ended mode operation.
4) White Heat sequencer control flags

External interrupt rate
(bits 1,0)

The G-REX has 4 settable interrupt rates (INT rate) ranging from 256 Hz to 2048 Hz. The INT rate sets the update rate for all inputs, outputs and the upper limit at which step pulse frequencies can be updated. Lower INT rates allow the microprocessor controlling the G-REX to execute more instructions between interrupts at the expense of a slow update rate. Higher INT rates decrease the G-REX response time at the expense of fewer instructions that the microprocessor can execute between interrupts.

At power-up, INT and INT_AUX run at the same rate. Normally, there is no need to change this, however writing to the step clock override register allows the INT_AUX rate to slow down relative to INT. INT_AUX defines the step pulse update rate; everything else is synchronized to the INT rate, which is always constant, and greater than or equal to INT_AUX.

Secondary watchdog timer enable
(bit 3)

If enabled, this timer ensures that the microprocessor writes 6 axis with Step Pulse Rate data per INT_AUX cycle. If not, the secondary watchdog timer will hold the G-REX in RESET until answered by the microprocessor. This is to prevent the G-REX from outputting step pulses or enabling outputs in the event of a microprocessor program crash.

Analog to Digital Converter differential or single-ended mode
(bit 4)

The G-REX has the analog to digital converters default to single-ended mode, in other words there are 4 separate ADC inputs. The ADC mode bit specifies whether the ADC is to operate in single-ended or differential mode. Single-ended mode means that there are four independent inputs, referenced to ground. Differential means that ADC inputs 0,1 and 2,3 are used in pairs. In differential mode, there are still four readings available. The readings, for each pair AB, are A-B and B-A. Whichever reading would be negative is set to zero. For more details about differential mode, and general characteristics of the ADC, please refer to the Texas Instruments datasheet for the TLV0834 ADC.

White Heat Control Flags
(bits 7,6,5)

Three bits in this register control White Heat. If using the default White Heat program, then these three bits must all be set to 1's immediately before enabling the normal MCU interrupt. The default is 0's, however if not enabled then the general purpose I/O, ADC and DAC will not function. When reprogramming White Heat, it should be disabled by setting bit 7 to zero, then re-enabled once the program is written.

Bit	Initial Value	Description
1,0	11	Primary Interrupt Rate: 00 = 2048 Hz 01 = 1024 Hz 10 = 512 Hz 11 = 256 Hz
2	0	Reserved
3	0	Secondary Watchdog: 0 = disabled 1 = enabled
4	1	ADC Mode: 0 = differential 1 = single-ended
5	0	White Heat 32kHz Interrupt: 0 = disable 1 = enabled
6	0	White Heat RAM Double Buffer 0 = disable 1 = enable
7	0	White Heat Execute Flag 0 = disable 1 = enable

Table 9. Global Control Register

Step Clock Override (1-word, WR, Address 0x2, 0x3):

This register forms a 16-bit unsigned integer and therefore must be written LSB first followed by MSB next with no intervening reads or writes to any other addresses. The 16-bit Step Clock Override register globally controls the step rate generator clock. The effective clock rate for all axis step rate generators is equal to:

Clock = 2^{23} * (Step Clock Override + 1) / 2^{16}
Or, in terms of the basic, constant, INT rate:
INT_AUX = INT * (Step Clock Override + 1) / 2^{16}

On hard reset, this register is set to 0xFFFF (65,535), making the step clock and the INT rate equal to their expected set values.

There is one exception to the above rule, and that is if the step clock override is written as 0x0000, then the INT_AUX interrupt is completely halted (rather than continuing at 1/65536 of the normal rate, as implied by the above formulae).

One purpose of the Step Clock Override is to implement a somewhat limited impending motor stall detect and correct if the step motors are equipped with encoders. Impending stall is detected by the motor lag angle when the encoders are connected to the G-REX quadrature encoder inputs. When this lag angle approaches the stall value, the Step Clock Override value is decreased, slowing the motor to a speed where it has sufficient torque for the overload. Because the INT_AUX rate decreases proportionally as well, no action has to be taken by the microprocessor in executing the motion control algorithm. All 2D or 3D motion will progress along the expected vector path, albeit more slowly than programmed until the overload is removed.

Another use for the step clock override is to allow synchronization with an external source. For example, and external device may be sending motion increment commands at exactly 1000Hz. In this case, it would be convenient to slow down the normal update rate from 1024Hz to 1000Hz in order to remain in sync. The MCU may even implement a digital phase-locked loop so that the clocks are kept synchronized over long periods in spite of slight variations. This use of the step clock override is obviously not compatible with the use described in the previous paragraph.

NOTE: if the microprocessor firmware is going to modify the step clock override from its power-up setting, then both interrupt lines (INT and INT_AUX) will need to be serviced individually. This means that the MCU will need two separate interrupt inputs, or will need to OR the interrupts together then vector to the appropriate service routine. In this case, the INT_AUX service routine must handle only the axis step rate commands. All other activity must be handled by the INT service routine.

Example:

The INT and INT_AUX rates are both set to 1024 Hz and a motor has a velocity of 3072 steps per second. The motor moves 3 steps during the interrupt period. Writing 0x7FFF to the Step Clock Override register results in the INT_AUX rate becoming 512 Hz and the motor velocity becoming 1544 steps per second, (exactly half of each original value). The motor still moves 3 steps per INT_AUX period but at one half the original speed, or 1.5 steps per 1/1024 second period.

This is a very simple and effective method of adaptively avoiding motor stall due to overload. The limitations are all axes are affected proportionally, and the update rate slows down in the same proportion. Changing the step clock override does not affect the timing for any other functions including ADC, DAC or digital I//O, since these are keyed to the (constant) INT rate. See the **Axis Lead / Lag registers** section for how motor lag is detected. Unlike all other register settings, this setting becomes immediately effective upon writing the MSB (address 0x3).

White Heat Address (1-word WR, address 0x4,0x5):

This register pair sets the next program or data space access address. The memory space is much larger than the address bus between the MCU and FPGA could directly access. Thus, the access address needs to be set indirectly via this register pair.

From the MCU's point of view, the White Heat program address space is comprised of 4096 x 9-bit words. The data space is 2048 x 8-bit words. From White Heat's perspective, the program space is 1024 x 36-bit words, and the data space is 1024 x 16-bit words. This distinction is elaborated in the section on White Heat (page 25). This section will only consider the MCU's view.

The White Heat address register contains the target address in the address space selected by the most significant bits of this register, as outlined below:

Bit	Description
11 - 0	Initial Address: 0 – 4095 for program space, 0 – 2047 for data space
14 - 12	Reserved, set to 0.
15	White Heat Address Space Select 0 = data 1 = program

Table 10. White Heat Address Register

Once an address (and space select) is written to this register pair, the next data written to the White Heat Data register will be stored at this address. The address will be automatically incremented by 1 after each write. Thus, this register only needs to be reset when non-contiguous data is being written.

In general, when writing to the program space, White Heat should be disabled by writing a 0 to bit 7 of the global control register. It should only be re-enabled when the programming is completed. The program will automatically start at location 1 (*not* zero) when re-enabled.

White Heat Data (multi-byte WR, address 0x6, 0x7):

In order to store data in White Heat's program or data memory, an initial address is stored in the White Heat address register (above). Then, the desired data is written to this register. When the MSB is written (register 0x7), then the data is stored and the address counter incremented.

Storing to Program Space:

The register at 0x6 is only used when storing to the program space, since the program space requires a 9th bit. This is the only exception to the rule that the LSBs are written to the even register, and the MSBs to the odd register. The 9th bit is stored in bit 0 of register 0x6. The other bits should be stored as zeros. After storing the 9th bit (which is the most significant bit of the 9-bit word), the 8 LSBs are written to register 0x07. At this point, the data is stored in the program memory.

Storing to Data Space:

This is simpler than program space. Since the data is only in the form of 8-bit words, each byte is simply written to register 0x07; register 0x6 is not modified.

It is expected that data space writes will be the most frequent, since programming only needs to be performed at initialization time. The RCM37xx MCU has a useful instruction for rapidly writing (and reading) the White Heat data space. The following assembler code illustrates the technique:

```
ld      hl,0x0000
ioe  ld  (0xFF04),hl                    ; Set addr to 0, space=data
ld      hl,<source block to store to W.H.>
ld      de,0xFF07
ld      bc,12                           ; bytes to store
ioe  ldisr                              ; store all!
ld      hl,0xFF03
ld      de,<dest block to get W.H. readback>
ld      bc,12
ioe  lsidr                              ;read all!
```

Counter Control (12-word, WR, Address 0x8, 0x9):

These registers form a 16-bit value and therefore must be written LSB first followed by MSB next with no intervening reads or writes to any other addresses. If any axis value is to be changed, then all 12 axis Counter Control registers must be written to. The first LSB, MSB writes the X-axis external counter setting, the subsequent writes affect the Y, Z, A, B and C axis external counters in that order. The same sequence applies to the internal counter settings for X,Y,Z,A,B and C.

These registers are grouped into two sets of 6 words. The first set of 6 words applies to the 'external counters', and the second set applies to the 'internal counters', as described below.

Each axis has an associated up-down ‘external counter’, which can be used for axis position feedback, or unrelated up-down counting tasks. Each counter can run in one of several modes, as set by this register. The ‘external counter’ connects to the external signals on each axis channel ‘A’ and ‘B’ inputs.

In addition, there is an internal counter for each axis, operated in effect by the internal command outputs of that axis. Thus there are a total of 12 counters. The counter-pairs for each axis can be combined in a number of ways. Only the combined result for each axis may be read by the microprocessor, so only 6 values may be read.

The following table describes the bit fields in each word value written to this register:

Bit	Initial Value	Description
11 - 0	0x000	Counter Increment. Signed 12-bit value
13,12	00	Counter Mode: 00 = quadrature counter 01 = tachometer, CH_A input 10 = tachometer, CH_B input 11 = step/direction, CH_A is step, CH_B is direction
15,14	00	Counter Options: 00 = disable counting 01 = count on rising edge 10 = count on falling edge 11 = count on both rising and falling edges

Table 11. Counter Control Register

Counter Increment
(bits 11-0)

In most cases, this should be set to '1' for the external counters. It can also be set to '-1' (0xFFF) to make the encoder count in reverse, as would be required if the A and B channel inputs were swapped over. Setting this to other values also has uses, which are described in detail in the lead/lag register description (page **Error! Bookmark not defined.**). The internal counters would normally be set to 0 increment, meaning that the command step/direction output for the axis has no effect on the counter readback. If using the lead/lag register, e.g. in feedback loop applications, then a non-zero value is appropriate.

Counter Mode
(bits 13,12)

Most encoders work in quadrature mode, hence the 00 option will be used. The 01 and 10 modes are indended for frequency counter or tachometer devices, where there is only a single direction of motion or rotation, or where the direction is not important. The step/direction emulation modes may be useful during test, where the

G-REX's own step/direction outputs may be looped back into the encoder inputs; or the G-REX may be programmed to track an existing step/direction signal from another device.

Warning: for correct operation, the counter mode must be set to 00 (quadrature) for the internal counters. If not, then the results in the lead/lag and counter readback registers will be invalid.

Counter Options:
(bits 15,14)

The interpretation of this field depends on the counter mode selected. In quadrature mode, this field is ignored (but should be set to zeros). In either of the tachometer modes, the bit field is interpreted as follows:

00 - disable counting
01 - count rising edges
10 - count falling edges
11 - count both rising and falling edges
.
The last of these effectively doubles the count rate. The tachometer modes, as the name implies, count only in one direction; they only add the counter increment, never subtract it. In the step/direction counter mode, the counter options are interpreted as follows:

00 - Step on falling edge, direction negative if high.
01 - Step on rising edge, direction negative if high.
10 - Step on falling edge, direction positive if high.
11 - Step on rising edge, direction positive if high.

Step Generator Control (6-byte, WR, Address 0xD):

These registers control the axis step pulse generators. Exactly 6 bytes should be written to this location to program all 6 axis. The first write programs the X axis, then the Y, Z, A, B and C axis in that order. Normally these registers are set during initialization, and not touched after that.

The following table describes the bit fields in each word value written to this register:

Bit	Initial Value	Description
2-0	000	Step Pulse Frequency Range and Resolution: 000 = 4.194MHz, 128Hz 100 = 262kHz, 8Hz 001 = 2.097MHz, 64Hz 101 = 131kHz, 4Hz 010 = 1.048MHz, 32Hz 110 = 65.5kHz, 2Hz 011 = 524kHz, 16Hz 111 = 32.7kHz, 1Hz
5-3	000	Step/Direction Outputs Mode: 000 = step/direction, 50% duty cycle, rising edge active 001 = step/direction, 50% duty cycle, falling edge active 010 = step/direction, 25% duty cycle, either edge active 011 = step/direction, 25% duty cycle, either edge active 100 = quadrature, step is CH_A, direction is CH_B 110 = direct, step and direction become general-purpose outputs (bit0, bit15) 101, 111 = reserved, do not use.
7,6	00	Reserved, set to 00

Table 12. Step Generator Control Register

The axis step pulse generator produces 32,767 clockwise and 32,768 counter-clockwise evenly spaced frequencies. The Step Generator Control register selects one of eight full-scale frequency ranges for the step pulse rates. The step pulse frequency resolution is equal to the selected range divided by 32,768.

Normally bits 3,4,5 are set to '010' for most opto-coupled step drivers such as the G201. Bits 0,1,2 are set to select the desired full-scale step pulse frequency range at initialization. The 32.768 kHz range is best for full-step and half-step motor drives. The 131.072 kHz range is best for 10-microstep drives and the 2.097152 MHz range is best for 125-microstep drives.

Axis Step Pulse Rate (6-word, WR, Address 0xE,0xF):

These registers form a 16-bit 2's complement signed integer and therefore must be written LSB first followed by MSB next with no intervening reads or writes to any other addresses. All 6 axes must be written to every INT_AUX period. If the secondary watchdog timer (page **Error! Bookmark not defined.**) is enabled, failing to write to all 6 axes will result in the G-REX entering a reset state and lighting the 'FAULT' indicator.

Note: if your machine has fewer than 6 axes (a common case) then it doesn't really matter what is written to the unused axis step pulse rate registers. However it is recommended that the MCU writes 0 to these axes. Alternatively, the unused axis registers can be set to "direct" mode which allows them to be used as additional digital outputs.

The first LSB, MSB byte-pair writes to the X-axis, subsequent writes go to the Y, Z, A, B and C axis. The Step Pulse Rates for all axis becomes effective at the next INT_AUX period.

The Axis Step Pulse Rate in combination with the selected Step Pulse Frequency Range sets the motor velocity and forms the basis for all motion control algorithms used in the microprocessor. At a minimum, the microprocessor must write 12 bytes to addresses "0xE" and "0xF" every INT_AUX period.

Here are some examples of setting the Step Pulse Rate, and the resulting velocity:

0x0000	(0)	Zero speed
0x0001	(1)	Minimum forward speed
0x7FFF	(32767)	Maximum forward speed
0x8000	(-32768)	Maximum reverse speed
0x8001	(-32767)	Almost maximum reverse speed; reverse of the maximum forward speed
0xFFFF	(-1)	Minimum reverse speed

The exact step pulse rate is given by the formula:

Frequency = Step Pulse Frequency Range * (Step Pulse Rate / 32768)

Example: A step pulse frequency of 100kHz is needed. The Step Pulse Frequency Range is set to 131.072 kHz. (100 kHz / 131.072 kHz) times 32,768 is 25,000 or "0x61A8" for CW, "0x9E58" for CCW.

All the 65,535 step pulse rates (CW and CCW) available in a selected range are evenly spaced frequencies. In the above example, the next lower frequency from 100 kHz would be 99.996 kHz, the next higher frequency would be 100.004 kHz. For that range (0 to 131.072 kHz), the frequency resolution is 4 Hz. For a 10-microstep drive, the lowest non-zero speed would be 0.12 RPM while the highest speed would be 3,932.04 RPM.

Axis Step Pulse Rate WR Order:

1	WR Address 0xE:	X axis Step Pulse Rate LSB
2	WR Address 0xF:	X axis Step Pulse Rate MSB
3	WR Address 0xE:	Y axis Step Pulse Rate LSB
4	WR Address 0xF:	Y axis Step Pulse Rate MSB
5	WR Address 0xE:	Z axis Step Pulse Rate LSB
6	WR Address 0xF:	Z axis Step Pulse Rate MSB
7	WR Address 0xE:	A axis Step Pulse Rate LSB
8	WR Address 0xF:	A axis Step Pulse Rate MSB
9	WR Address 0xE:	B axis Step Pulse Rate LSB
10	WR Address 0xF:	B axis Step Pulse Rate MSB
11	WR Address 0xE:	C axis Step Pulse Rate LSB
12	WR Address 0xF:	C axis Step Pulse Rate MSB

Global Status Register (1-byte RD, Address 0x00):

This register may be read at any time. It is a bit-field, as specified by the following table:

Bit	Description
0	If set, this bit indicates there was a counter fault. Such a fault may occur if there is an invalid state transition detected on the counter inputs. For example, both inputs changing at the same time for a quadrature counter. This is an OR of all individual axis counter fault flags. The individual axis may be determined by reading the axis flags (below) within the same interrupt cycle. This flag is automatically cleared as soon as it is read by the MCU. It is set whenever any axis counter fault occurs, and is latched until read.
1	This reads back the state of the A2D_SARS pin (Successive Approximation Register).
2	This reflects the state of the internal clock multiplier lock indicator. It should always be read as a '1'. If not, then the MCU should reset the FPGA. Note that after reset, it may take up to 1ms for the clock multiplier to lock on. The MCU should thus wait for 1ms after hard resetting the FPGA before checking this status bit.
3	This bit, if set, indicates there was at least one low->high transition of one or more of the axis index inputs since the last time this register was read. This bit is automatically reset when this register is read. If set, the MCU may read the individual axis input registers to find out which axis (or axes) detected an index pulse.
4	This bit is identical the the above, except that it indicates high->low transitions.
6,5	Reserved
7	This indicates the state of the main FAULT output. It is set if there was a watchdog timeout. If this bit is read as 1 by the MCU, the MCU should hard reset the FPGA.

Table 13. Global Status Register

Bits 0, 3 and 4 remain latched until this register is read. After being read, they are automatically set back to zero.

Firmware Version Info (1-byte RD, Address 0x1):

This register allows the MCU to interrogate the FPGA to find out which version and release it is currently running. This will read back as 0x10 for the version described in this manual. The version number is in the most significant 4 bits, and the release in the least significant 4 bits.

It is intended that the version number will increment when there is an incompatible change in the FPGA configuration, whereas the release number will increment if there is a change which is backward compatible within the same version. In any case, the MCU can interrogate these bits to ensure that the expected configuration is loaded. Note that if this register reads back as 0x00 or 0xFF, then there may have been a problem configuring the FPGA. In this case, the MCU should try reloading the configuration.

White Heat Read Data (multi-byte RD, Address 0x3):

Although White Heat has a separate program and data space, only the data space contents may be read by the MCU. The program space data cannot be altered except by the MCU, which is why it is not necessary to read it back. As for writing data, the read address is set by writing to registers 0x4 and 0x5. The address is shared for reads and writes, and is incremented by either.

The procedure for reading the data space is to set the White Heat address register (which must have bit 15 set to 0 to indicate data space), then read as many contiguous bytes from this register as required. Example RCM37xx code is given on page **Error! Bookmark not defined.**.

Axis Inputs (6-byte RD, Address 0x8):

This is a shift register. It must be read exactly 6 times per interrupt cycle in order to retrieve all axis data. The contents are a bit field, which reflects the state of the counter inputs, the index input and the counter fault signal. The first byte read will be for axis X, followed by Y, Z, A, B, C in succession.

Bit	Description
0	This bit is a snapshot of the state of the channel A input of the encoder, at the start of the current interrupt cycle.
1	This is a snapshot of the channel B input of the encoder.
2	This is a snapshot of the index input of the encoder.
3	If 1, then there was a counter fault condition. This bit is set only for the interrupt cycle in which it was detected. It is cleared at the start of the next cycle (unless there was another fault). A counter fault occurs if, in the given counter mode, there is an invalid transition. Generally, this indicates a bad connection or an encoder which is exceeding its rated speed.
4	If 1, then there was at least one low->high transition of the index input for this axis, in the immediately preceding cycle. This bit is set only for the interrupt cycle in which it was detected. It is cleared at the start of the next cycle (unless there was another index event). In any cycle, it this bit is set, then the global status register will latch the appropriate indicator bit to a '1', until the status register is read. Thus, it is always possible to catch an edge of the index input, however the timing resolution will only be accurate to the millisecond, if this register is polled every cycle.
5	This is identical to the above, but detects high->low transitions.
7,6	Reserved.

Table 14. Axis Input Register

Axis Counters (6-word RD, Address 0xA, 0xB):

The Axis Counter Registers must be read exactly 6 times (for each low then high byte) in each interrupt cycle in order to obtain the current counter reading. The value is interpreted as a 16-bit quantity which wraps around on overflow. The first word read will be the value for the X axis, the second for Y, etc.

Each axis may be configured with a counter pair. The counter pair accepts count events from external pins as well as the step generator output. The two counters (external and internal) are summed together when read out at this address. Thus, it is not possible to obtain independent readings of both counters in the pair. The following counter modes are useful:

External Increment	Internal Increment	Usage
-1	0	Independent external 'down' counter
0	+1	Readback/confirmation of net step output
-1	+1	Differential mode with 1:1 ratio between step output and encoder feedback. In other words, the encoder has exactly 1/4 the number of cycles per revolution, or lines engraved on the encoder wheel, as the motor has microsteps per revolution. Note that the encoder cycles per rev. is multiplied by 4 to get the actual counts per rev.
-1024	+1024	As above, except scaled for optimum resolution in the lead/lag detector.
+1024	+1024	As above, except compensates for one of the motor phases, or the encoder phases, being wired in reverse.
-512	+1024	As above, except the encoder has two counts for every microstep of the motor. For example, a 4000-count encoder with a 2000 microstep motor.
-1000	+1024	Differential mode, for the typical case of a 2000 microstep/rev motor, and a 512 line encoder wheel directly mounted on the motor shaft.

Table 15. Useful Axis Counter Settings

Axis Lead / Lag Registers (6-bytes RD, Address 0xD):

This register should be read 6 times at each interrupt if differential counter mode is being used to track command vs. position error i.e. lead/lag.

The axis Lead/Lag registers will indicate how much a motor is ahead or behind of where it is commanded if the motors are equipped with encoders. When used with encoder equipped step motors, the Lead/Lag registers can accurately show what percentage of available torque the load is demanding. This can be used as a 'load' readout directly or, in more sophisticated applications, be used to close the loop on a step motor to turn it into a true servo.

For a meaningful value, the relevant axes should be set to differential counter mode, with the increment values scaled so that the minimum value magnitude is approximately 256 (assuming a 1024Hz interrupt rate). The lead/lag from other axes (not in differential mode) should be ignored.

This value is computed as the sum of the most significant 7 bits in the 16-bit differential counter, sign extended. The sum is accumulated to 20 bits (since there are 8192 sums in a 1ms interrupt period). Only the 8 MSBs of the 20-bit sum are read in this register.

The 20-bit accumulator is zerod at the start of each period, and then 8192 sums of a value between -64 and +63 (the 7 MS bits of the counter) are added. The maximum result magnitude is $2^6 \times 2^{13} = 2^{19}$, which cannot overflow a 20-bit accumulator.

For application in 'unstallable step motor' systems, the allowable lead/lag will be +/-20 counts with 1:1 encoder to step output. Any higher than this represents a definite motor stall, since the electrical to mechanical phase angle cannot exceed 180 degrees under any circumstance.

Since this register determines an average lead/lag over the 1ms interval, it is desirable to obtain the best resolution. At non-negligible motor speeds, some additional information (in the form of precise edge timing) is available to increase measurement resolution. In order to take advantage of this, an appropriate step increment (scaling) should be programmed in. For example, suppose the +/-20 count raw measurement is to be readable as a +/-80 count in this register (thus gaining an extra 2 bits of resolution). To achieve this, a value of 1024 is programmed into the increment for both internal and external counter. A lead/lag of the limiting value of 20 counts is magnified by this factor, 1024, becoming 20480 in the differential counter. Since the 7 MSBs of the counter are used, this value is effectively divided by 2^9, giving 40. The 20-bit accumulator then sums 40 times 8192, giving 327680. Taking just the 8 MSBs of this effectively divides by 4096, giving 80, which is the desired result.

The initial magic factor of 1024 was selected so that the available raw differential count (20) was magnified to take advantage of the full 16 bits of the differential counter. There is no need to use a power of 2. For example, 1280 would give a final lead/lag register range of +/-100. In cases where the encoder is not exactly 1:1 with the step output, different increment values are used for internal and external counts.

In general, if the step output is ***S*** per revolution, and the encoder is ***E*** counts/rev, then divide both by their largest common factor, giving ***S'*** and ***E'*** respectively. Then multiply both by an integer (giving ***S"*** and ***E"***) which makes ***E"*** reasonably close to 1024 (for a 2000 step/rev motor). If the resulting values are too disparate, or ***S"*** ends up being over 2048, then reduce the common factor until they 'fit'. This will result in a loss of resolution, but should still work. Note that the ***S"*** value should be programmed into the external counter increment, and the ***E"*** value into the internal counter increment. Example: ***S***=2000, ***E***=4096. Then ***S***=2*2*2*2*5*5*5, ***E***=2^{12}. Eliminate common factors of 2 giving ***S'***=125, ***E'***=2^8=256. Scaling up (by 4) gives ***S"***=500, ***E"***=1024. Checking, over one rev there will be 2000***E"*** step increments, and 4096***S"*** encoder increments, which are numerically equal, hence correct.]

Note that the 'extra bits of resolution' are only meaningful when the motor is rotating. At rest, there is no timing information available to allow interpolation of the average lead/lag angle, and the measured value will always be one of a limited fixed set. It may be argued that truncating the 9 LSBs of the differential counter at each summation into the 20-bit accumulator will lead to biased results. In practice, it is equivalent to a half-step offset, which may be assumed by the MCU. In any case, the mechanism for deciding the initial lead/lag phase angle is not specified. This angle may be anything up to +/-90 degrees depending on initial load conditions. Thus, the MCU will need to add in a compensating offset in any case, once it determines the 'neutral' point.

G-REX I/O Peripherals:

This is the general term for the auxiliary I/O devices, as outlined on this page. These are all accessed via White Heat, rather than through direct register addressing. For details, see page 33.

Digital to Analog Outputs:

The G-REX has four on-board 8-bit Digital to Analog converters. The DACs are scaled to output over a 0 to 5VDC scale. The DAC outputs are buffered by an LM6034 quad op-amp and have a 5kHz low-pass -3db corner frequency.

Writing “0x00” results in an analog output voltage of 0VDC and writing “0xFF” results in an output voltage of 4.98VDC (5V times 255 / 256). The output voltage appears on the Analog Output pins on the following INT period.

Analog to Digital Inputs:

The G-REX has four on-board 8-bit Analog to Digital converters. The ADCs are scaled to input over a 0 to 5VDC scale. The ADC inputs are low-pass filtered and have a –3db corner frequency of 5kHz. The input impedance is 1.5 kilo-Ohms.

Reading “0x00” is in an analog input voltage of 0VDC and reading “0xFF” is in an input voltage of 4.98VDC (5V times 255 / 256). The input read from these registers reflects the Analog Input voltages present on the physical pins in the previous INT period.

Digital Outputs:

16 general purpose open collector outputs are available, accessed via White Heat shared memory. A logic ‘1’ turns an output pin ‘on’ because they are open-collector Darlington transistors driving a pull-up load (ULN2003). The latency is 2 INT periods; a write to an output will change the physical output pin at the start of the INT period after next. There is no read back of the output pin state so shadow output registers should be maintained by the microprocessor.

General Purpose and Axis Inputs:

There are 6 axis LIM input pins and the 16 general-purpose input pins of the G-REX. A read from this address will reflect the state of the physical input pins at the start of the previous INT period. The input bits are non-inverted; a bit will read as a ‘1’ if the input has 5VDC on it, a ‘0’ will be read if the input is at 0VDC. The input pins have pull-up resistors on them so unused or inactive inputs will read a logical ‘1’.

The inputs have 1.5K pull-up resistors to 5VDC and are meant to be operated with SPST switch contacts to ground or with open-collector devices. The inputs are filtered and protected against a +/- 50V input range.

Although 6 of the inputs, denoted LIM, are dedicated to their own axis, there is actually no difference between these inputs and any of the "uncommitted" inputs. They can be used for the intended purpose (axis limit switches or fault conditions) or as additional inputs. This is really a function of the user interface software.

Using the G-REX:

Theory of Operation:

The G-REX relies on digital integration of velocity (step pulse frequency) with time (interrupt period) to calculate the distance moved. It is not necessary to count step pulses to track axis position at any time. This is perfectly accurate because the same oscillator clock that generates the step pulse rate also generates the time period 'ticks' (interrupt period) for which the step pulse frequency applies.

For instance, if a velocity of 1,024 step pulses per second is written to an Axis Step Pulse Rate register (0xE, 0xF) and if the Global Control register (0x0) is set to produce an INT rate of 1,024 interrupts per second, then the axis will move 1 increment of motion per INT period. Position is calculated by summing the velocity (Axis Step Pulse Rate) word to an axis position register in the MCU every INT period.

Any change in velocity caused by clock oscillator drift is precisely matched by a compensating change of the INT period time; the ratio of step pulses per INT period is independent of the clock oscillator frequency.

Acceleration and deceleration is accomplished by increasing or decreasing the axis velocity word every INT period. In the simplest case, a constant acceleration value is added to or subtracted from the velocity word every INT period and then that summed velocity is written to the Axis Step Pulse Rate register of the G-REX. Summing that velocity to an axis position register every INT period updates the axis position. This information is then used by the user's motion control algorithm to calculate subsequent velocity words to complete the axis move.

Actual implementations work best with an interrupt service routine (ISR) invoked by the INT signal working in tandem with a main background program. The ISR feeds the G-REX's FPGA pulse engine using a simple, tightly written motion algorithm while the background program 'crunches numbers' for the next vector segment. These parameters are passed to the ISR as it completes one vector and is ready to process the next one.

Practical implementations are based on the concept of 'accelerate, run at speed, decelerate and stop'. **Please see the example shown on page 24.** Multi-axis motion is based on treating moves as 1 to 6-dimension vectors. For example, a 3-D vector has an X,Y,Z starting coordinate and another X,Y,Z end coordinate. The background program computes the vector 3-D length and the vector X,Y,Z velocity components from the programmed vector velocity. These are then passed to the ISR routine.

The ISR runs a single dimension vector 'accel, run, decel and stop', all the while multiplying the resulting velocity / per INT stream by the individual X,Y,Z velocity components. These are then passed on the G-REX Step Pulse Rate registers. The result is motion in 3-D space at a constant velocity independent of the vector's 3-D direction. Having a microprocessor like the RCM-3610 that can do 16-bit hardware multiplication is almost a necessity.

Oftentimes it is needed to move along a path formed by a long series of concatenated vectors of various lengths and directions. Usually these vectors form a linear piecewise approximation of an arbitrarily curved path. It is also usually undesirable to have the motion along that path start and stop on every concatenated vector segment. In that case the solution is to have the actual path 'blend' from one vector to the next at the vector nodes. This is sometimes called 'constant-contouring' and can become extremely complex using various look-ahead schemes. The G-REX lends itself to an extremely simple and unique solution to this problem by post-processing the velocity / per INT stream through a moving average filter. The result is a constant vector velocity along the path that automatically adapts itself; the faster the programmed velocity, the more 'rounded' the actual path becomes. Another advantage is the moving average filter can be switched on and off anywhere along the path as many times as needed; 'on' for constant contouring or 'off' where a sharp vector direction change is required.

INT Rate Considerations:

INT Rate and Latency:

Choosing an INT rate is a balance between update latency against microprocessor clock cycles per interrupt period. The latency between an input to the G-REX and an output in response is equal to 2 INT periods. Increasing the INT rate decreases this latency time.

INT Rate and Staircasing:

The INT rate directly affects the acceleration and deceleration 'staircasing'. Staircasing refers to the magnitude of step pulse frequency change and number of discrete step pulse frequencies issued during acceleration / deceleration. Step pulse frequencies can only change at the INT rate. The lower the INT rate, the larger and longer these stair case 'steps' are. Each change in step pulse frequency imposes a torque demand of the motor as it accommodates to the speed change. If the INT rate is made too low, this torque demand begins to adversely affect motor performance.

INT Rate and Fractional Steps:

Another consideration is the size of the axis position registers maintained by the microprocessor. The **Global Control Register** can set the INT rate from 256 Hz to 2048 Hz while the **Step Generator Control Register** can be set the step pulse frequency from a 1 Hz resolution to a 128 Hz resolution.

If the INT rate is set to 2048 Hz and the step pulse frequency range is set to a 1 Hz resolution, writing 0x0001 to the **Axis Step Pulse Rate Register** will result in a step pulse output of 1 increment of motion every second. In that one second 2048 INT periods have occurred and that 0x0001 velocity has been summed to the microprocessor axis position register for a count of 2048. It takes 11 bits to hold this total. Looked at a different way, each INT period accumulates 1/2048 of an increment of motion. This is the fractional step portion of the axis position register. If 4 bytes of axis position is used, the position register has 11 of its 32 bits taken by this fractional portion, leaving 21 bits for actual, measurable, increments of motion. If each increment translated to 0.0001 inch (0.0025mm) on the machine axis, this represents a full-scale range of 209 inches, or 5.3m.

At the other extreme, if the INT rate is set to 256 Hz and step pulse frequency range is set to the 128 Hz resolution then there would be a minimum of one measurable step per two interrupt periods. A velocity of 0x0001 would give 1/2 step pulse per INT period, 0x0002 would give 1 step pulses per INT period and so on. A 4-byte axis position register at a 0.0001" or 0.0025 mm resolution would give an excessive working range of about 3.3 miles or 5.3 kilometers respectively! Not many mechanisms are built on that scale. This would also suffer from the effects of excessively coarse step pulse frequency staircaising and long input to output latency.

The table on the following page shows the number of fractional bits required versus frequency range resolution and INT rate

Step Pulse Frequency Resolution (Hz)	Interrupt Rate (Hz)			
	256	512	1024	2048
1 Hz	1	2	3	4
2 Hz	2	3	4	5
4 Hz	3	4	5	6
8 Hz	4	5	6	7
16 Hz	5	6	7	8
32 Hz	6	7	8	9
64 Hz	7	8	9	10
128 Hz	8	9	10	11

Table 16. Fractional Bits in Position Counter

A Rabbit Semiconductor RCM-37xx (CN6 native pin-out) executes at an average of 140nS per instruction resulting in about 1,700 to 15,000 instructions per INT period assuming a 50% ISR time utilization, which is adequate for the task.

If the fractional bits derived from the above table is called ***N***, then all vector move distances must be multiplied by 2^{N} before being submitted to the ISR motion control algorithm. Naturally, a shift left by ***N*** amounts to the same thing.

Step Motor Drives:

A practical step pulse frequency resolution for full-step and half-step motor drives is the minimum 1 Hz range (2^0). This gives a maximum step pulse frequency of 32,767 steps per second yielding 9,830 RPM and 4,915 RPM respectively. It is highly unlikely either can drive the motors to that speed because of step motor limitations.

10 microstep drives can use the 2 Hz to 4Hz resolution range. This gives a maximum step pulse frequency of 65,535 to 131,072 step pulses per second yielding 1,966 RPM to 3,932 RPM at the motor.

125 or 250 microstep drives can use the 32 Hz to 128 Hz resolution range. In theory, a 250-microstep drive would reach 5,033 RPM at a step pulse frequency of 4.194 MHz.

Servo Motor Step and Direction Drives:

These can run the gamut since servomotors generally run at much higher speeds than step motors and their positional resolution is entirely determined by the encoder line-count. For instance a 6,000-RPM servomotor mounted with a 2,000-line encoder utilizing a drive with a X4 quadrature encoder would need an 800 kHz step pulse frequency. The 32 Hz resolution range (1.048 MHz max step pulse frequency) would be a nice fit.

Example Motion Control Algorithm:

The following is an example of a simple point-to-point motion control algorithm using the G-REX. Rate of acceleration / deceleration and maximum velocity are considered constants and are set to ‘2’ and ‘10’ respectively here. The distance to move is a variable and is set to ‘120’ for this example. The remaining variables are temporary values that change with the execution of the algorithm:

ST (state), variable, set to ‘1’ to begin (‘1’ = accelerate, ‘2’ = run, ‘3’ = decelerate, ‘0’ = done/stop)
ACC (rate of acceleration), constant, set to ‘2’
VEL (commanded velocity), variable, initialized to ‘0’
MAX (maximum velocity), constant, set to ‘10’
DN (distance of the move), variable, initialized to ’120’
UP (distance moved), variable, initialized to ‘0’
On every INT period, perform the following algorithm:

```
ST = '0'
        Exit

ST = ‘1’
        VEL becomes VEL + ACC
        is MAX > VEL ?
        if yes, then
                DN becomes DN – VEL
                UP becomes UP + VEL
                is UP > DN?
                if yes, then
                        ST becomes ‘3’
                        output VEL
                        exit
                otherwise
                        ST becomes ‘2’
                        VEL becomes MAX

ST = ‘2’
        DN becomes DN – VEL
        output VEL
        is UP > DN ?
        if yes, then
                ST becomes ‘3’
                exit
        otherwise
                exit

ST = ‘3’
        VEL becomes VEL – ACC
        DN becomes DN – VEL
        DN = ‘0’ ?
        if yes, then
                VEL becomes ‘0’
                output VEL
                ST becomes ‘0’
                exit
```

Here is an example of how the variables and state progress as the algorithm is executed:

Iter	VEL	DN	UP	Comments
0	0	120	0	Ready to move
1	2	118	2	VEL = VEL+ACC, accelerating
2	4	114	6	VEL = VEL+ACC, accelerating
3	6	108	12	VEL = VEL+ACC, accelerating
4	8	100	20	VEL = VEL+ACC, accelerating
5	10	90	30	VEL=MAX, end acceleration
6	10	80	30	Cruise
7	10	70	30	Cruise
8	10	60	30	Cruise
9	10	50	30	Cruise
10	10	40	30	Cruise
11	10	30	30	Cruise
12	10	20	30	UP > DN, begin decelerating
13	8	12	30	VEL = VEL-ACC, decelerating
14	6	6	30	VEL = VEL-ACC, decelerating
15	4	2	30	VEL = VEL-ACC, decelerating
16	2	0	30	VEL = VEL-ACC, decelerating
17	0	0	30	Move finished, axis stopped

The move in this example took 17 INT "ticks" to move the required 120 steps. It accelerated to the programmed velocity of 10, decelerated and came to a stop at the 120th step pulse.

The above algorithm is for illustrative purposes only and would quickly run into trouble with different initial conditions. The algorithm ‘language’ is an entirely made-up one for the purpose of clarity.

The important point is nothing more mathematically complex than 16-bit addition or subtraction is needed to perform a practical motion control interrupt service routine. Its simplicity is independent of the actual step pulse frequency required.

The White Heat Sequencer:

White Heat is a simple processor embedded in the FPGA. Its role is to perform functions which do not need the speed of dedicated logic circuits, but are faster than could reasonably expected of the MCU.

The functions which White Heat is required to perform are the analog and digital I/Os. These functions consume a small fraction of White Heat's capabilities; the remainder of its time and memory space can be devoted to other tasks as specified by the programmer.

Examples of other tasks that could be programmed are:

1) co-ordinate transformations such as would be required for conversion of X,Y,Z,roll,pitch,yaw co-ordinates into hexapod strut lengths, plus the inverse of this function.

2) implementation of feedback control algorithms which require a faster loop time than the 2048Hz available to the MCU. This would include control of step motor dynamics such as mid-band resonance compensation.

3) arithmetic operations such as fast sine, cosine or square-root.

White Heat is defined as a Harvard architecture. This means that it has a separate program and data memory space. As an additional specialization, the data space is divided into two parts: the "working set" and the "shared memory". The shared memory is for communication with the MCU. The working set is used for intermediate calculation results and constants.

A third address space is also available, called the "peripheral space". This is analogous to the "I/O space" of some processors, including the Rabbit. This space is used for accessing the other hardware within the FPGA.

Program Memory Space:

From White Heat's perspective, program memory is an array of 1024 x 36-bit words. There is one instruction encoded per word. White Heat cannot modify its program memory, and the MCU cannot read from it. The MCU can only write to it, and White heat can only execute from it.

Since the MCU to FPGA interface is only 8-bits wide, the MCU's view of the program memory is a remapping into 4096 words of 9 bits each. The 9th (most significant) bit is registered separately before issuing the store operation for the 8 other bits. See the description of the programming register set on page 11. When the MCU writes locations 0, 1, 2 and 3 with 9-bit words, these words are mapped to White Heat's instruction 0, in "little-endian" format. In other words, the MCU's word 0 corresponds to the 9 LSBs of White Heat's program word 0. MCU word 3 corresponds to the 9 MSBs of White Heat word 0.

Data Memory Space: Working Set

The working set is an array of 1024 x 18-bit words. This storage is analogous to the register set of more conventional processors. It is neither readable nor writable by the MCU, however it may be indirectly written (for the purpose of initialization) by having the MCU write a temporary White Heat program whose sole purpose is to store values in the working set. The only other means of initializing this memory space is to specify its initial value in the FPGA configuration file.

In general, White Heat uses all signed 18-bit or 36-bit arithmetic. The additional 2 bits (compared with typical processors' 16-bit word size) allows some additional insurance against arithmetic overflow, which can simplify some control algorithms.

There is no restriction on any working set location. All 1024 locations are equally valid for all instructions. Every arithmetic instruction reads the contents of two independent locations, and can write back 0, 1 or 2 locations, with the restriction that the write must be to the same location(s) as originally accessed in that instruction. For example, you can add locations 3 and 4, writing the result back to location 3 or 4, but not any other location.

Data Memory Space: Shared Memory

The purpose of the shared memory is to allow the MCU and White Heat to communicate. It allows parameters and data to be written by the MCU, and the results of operations to be read back by the MCU.

This is a true dual-ported memory. White Heat views this is an array of 1024 16-bit words, and the MCU views it as 2048 8-bit words. As for the program memory, the mapping is "little-endian". Because of the 8-bit data path between the MCU and the FPGA, it is impossible for the MCU to write a 16-bit value atomically. The LSB must be written separately from the MSB. This raises the possibility that White Heat could read the word when only half of it has been written. In order to overcome this problem, the shared memory is specially organized.

Bit 6 of the global control register (address 0x0) enables the so-called "shared memory double buffer". When enabled, this affects the lower half (512 x 16-bit words) of the shared memory. The lower half is divided again into two quarters which we call "quadrants". Each interrupt (INT) effectively swaps the first and second shared

memory quadrants. If White Heat and the MCU both access word 0, in the same INT period, they are really accessing separate memory locations: one in the first quadrant, the other in the second.

The intention of this is to prevent simultaneous acess to the same location. The upper quadrants (3 and 4) are not double buffered. These quadrants are intended for communications which cannot tolerate the additional interrupt cycle of latency caused by the double-buffering, or account for the possibility of a mis-read.

It is expected that the double buffered quadrants are used where the boundary between interrupt cycles is important. The other quadrants are used for "constant" data or data which must affect White Heat immediately.

The shared memory is 16 bits wide. When White Heat reads this memory, it always sign extends it to 18 bits (by duplicating bit 15 into bits 16 and 17). When White Heat writes to this memory space, it truncates the 2 most significant bits, 16 and 17, of its internal 18 bit representation.

Peripheral Space

This space is the interface between White Heat and the other, dedicated, logic within the FPGA. It is a 16-bit wide interface, with just a handful of locations. For reads by White Heat, the mapping is:

Name	Address	Data
gflags	0	Current tick counter (high byte), and contents of global control register (low byte).
sdata_in	1	Shift-in data from digital and analog inputs

For writes, the mapping is:

Name	Address	Data
sctl	0	Shifter control
sdata_out	1	Shift-out data to digital and analog outputs

These peripheral space registers are used by the reference program to access the peripherals. The MCU view of peripheral access is detailed on page 33.

White Heat Architecture:

Instructions are divided into two categories: arithmetic and control. Arithmetic instructions are used to move data between memory locations, optionally performing arithmetic on the data. Control instructions are for conditional branching, subroutine calls and returns.

There are no general purpose registers. All data is obtained from one or two data memory locations, and written back to memory after optionally passing through the arithmetic/logical unit (ALU). The only registers are a 36-bit accumulator, two 9-bit index registers, and a 36-bit product register.

The White Heat clock runs at 50.331648MHz (which we round to 50MHz for this discussion). All instructions execute in 1 or 2 clocks. The rule is simple for this: if a result needs to be written to memory (working set or shared) or an external "peripheral", then the instruction takes two clocks. Conditional or computed branches and indirect index loads take two clocks. All other instructions take one clock. The typical program will average about 1.5 clocks per instruction, resulting in a 33MIPS rate of execution. Notably, arithmetic where the result is left in the accumulator will execute at 50MIPS. This includes multiply-accumulate operations, since the multiplier is pipelined.

White Heat has an unconventional interrupt handling strategy. There are two sources of interrupts: one is the White Heat enable bit (bit 7 in the global control register). When the MCU sets this bit from 0 to 1, White Heat starts executing from program location 1 (like a reset). Otherwise, if bit 5 of the global control register is set,

White Heat is forced to execute from location 0 at a 32768Hz interrupt rate. Unlike other processors, White Heat is not expected to try to resume from where it was interrupted. Instead, it is intended to run as "one big ISR". The 32kHz interrupt is intended to be used as the loop or sampling time constant for control algorithms. If an operation needs to take longer than 1/32768 second, then the operation must be divided up into sections which take no more than this interval.

At the end of processing each 32kHz section, the final instruction should just jump to itself endlessly, since there is no dedicated "halt" instruction.

In each 1/32768 second interval (which we call a "tick"), White Heat has 1536 clocks available until the next tick. This allows it to execute the entire 1024 instructions in its program space (assuming the average of 1.5 clocks per instruction). If there are any loops in the code, then this becomes limited by the available time, rather than by memory. Typically, the total program will be divided up so that parts are executed over the number of ticks available in each INT cycle. Depending on the global control register settings, there will be 16, 32, 64 or 128 ticks per INT.

The current "tick number" within the overall cycle may be obtained by reading peripheral gflags (location 0). The high byte (actually, 7 bits, since the MSB is always 0) of this value increments from 0 to 127, then wraps around back to 0. Thus, White Heat can interrogate this value and branch to the code section which is appropriate for that tick of the cycle. Tick number 0 is guaranteed to occur on the main INT cycle. Tick 64 is also a starting point if the INT cycle is set to 512Hz or higher. Ticks 0, 32, 64 and 92 are starting points for a 1024Hz cycle, and any multiple of 16 denotes the start of a 2048Hz cycle.

Since each tick forces White Heat to execute from location 0, then the instruction at that location must be a jump to the handler which determines the tick number in the cycle (if this is important). This jump is necessary because the following instruction, at location 1, is the initial vector for reset.

When White Heat is reset then re-started by the MCU, it starts executing at location 1. Since this may occur asynchronously with respect to the tick, the handler for the reset event would normally perform whatever initialization was required, then simply wait for the next tick (or a particular tick number if necessary).

From the MCU point of view, starting White Heat should proceed as follows:

1) Set global bit 7 low - this holds White heat in reset.
2) Set global bit 6 as required for double buffering
3) Set global bit 5 low - this disables the 32kHz tick
4) Set global bit 7 high - this enables White Heat, which will execute from location 1.
5) Wait for White Heat to finish its initialization phase, according to the program - usually of the order of a few microseconds.
6) Set global bit 5 high if required, for the normal tick interrupt.
The first 3 of the above steps may be performed together, since it is a single write to the global control register.

Arithmetic Instructions:

The distinction between an arithmetic instruction and a control instruction (see next section) is determined by a single bit field (bit 26) in the instruction.

Within the arithmetic instructions (which includes simple data movement and logical operations) the other fields in the instruction have fixed meaning, and may be used in any combination. In this respect, White Heat is very orthogonal and RISC-like. The instruction fields are organized as follows:

Field Bits	Function	Description
9 - 0	Work Address 1	This specifies the first address in the work space. It is either an immediate (direct) address, or is added to the contents of index register 1 (AX) to form an indirect address. The resulting address is called MA1. The data at that location is called *MA1.
19 -10	Work Address 2	This specifies the second address in the work space. Indexed mode uses index register 2 (BX). The resulting address is called MA2, and the data is *MA2.
20	Index Mode 1	If 1, specifies index address mode for the first address. If 0, specifies immediate mode.
21	Index Mode 2	As above, for the second address.
22	Work Store Enable 1	If 1, enables store of a result to MA1. If 0, the data at *MA1 is not modified. The result is selected by bits 31 and 32.
23	Work Store Enable 1	As above, for MA2.
24	Shared Data Write Enable	If 1, enables a result to be written to the shared memory. The shared memory address is specified by MA2. If 0, there is no modification to shared memory.
25	Peripheral Write Enable	If 1, enables a result to be written to peripheral space. The peripheral address is specified by MA2. If 0, there is no modification to shared memory.
26	Arithmetic/Control	This is always 0 for arithmetic instructions
28,27	ALU A Input Selector	Selects the operand applied to the first ALU input: 00 - the 18-bit data from *MA1, sign extended to 36 bits. 01 - the 36-bit signed product from the previous instruction 10 - the above product, shifted left 1 bit (zero fill on right) 11 - the above product, shifted right 1 bit (sign extended on left)
30,29	ALU B Input Select	Selects the operand applied to the second ALU input: 00 - the 18-bit data from *MA2, sign extended to 36 bits. 01 - peripheral read data (16 bits, addressed by MA2, sign extended to 36 bits) 10 - the 18-bit data from *MA2, shifted left by 18 bits, padding the lower bits with zeros. 11 - the current accumulator result (36 bits).

Field Bits	Function	Description
32,31	Store Select	Selects the data to store. This is a 36-bit wide multiplexer, divided into two halves. Both halves use the same selection. The low 18 bits supply data to be stored to (a) working set addressed by MA1 (if bit 22 set) and/or (b) Shared memory addressed by MA2 (if bit 24) and/or (c) Peripheral space also selected by MA2 (if bit 25). The high 18 bits supply data only to the working set addressed by MA1 (if bit 23 set). The low 18 bits are selected as follows: 00 - ALU output low 18 bits 01 - ALU B input low 18 bits 10 - ALU A input low 18 bits 11 - The shared memory contents addressed by MA2 (16 bits, sign extended to 18 bits). The high 18 bits are selected as follows: 00 - ALU output high 18 bits 01 - ALU B input high 18 bits 10 - ALU A input high 18 bits 11 - MA1, as an immediate 10-bit constant, sign extended to 18 bits.
33	Accumulator Enable	If 1, then the current instruction's ALU result is latched in the 36-bit accumulator. If 0, the accumulator's previous value is retained.
36,35	ALU Function Select	Selects the ALU function to apply. Inputs and outputs are 36-bit quantities: 00 - ALU B + ALU A 01 - ALU B - ALU A 10 - The low 18 bits of the result are ALU A(low) & ALU B(low) The high 18 bits are ALU A(low) | ALU B(high) 11 - The low 18 bits of the result are ALU A(low) & ~ALU B(low) The high 18 bits are ALU A(low) ^ ALU B(high)

Table 17. Arithmetic Instruction Format

The above table makes no mention of the product. Every instruction (including control instructions) generate the product of the words addressed by MA1 and MA2. The product is latched in a register, for use by the next instruction.

The product is the signed, 36 bit, product of two signed 18-bit values. If it is desired to use the product, it must be accessed on the following instruction. In a sense, it costs nothing to obtain a product, however it must be used in a timely manner (for example, adding into the accumulator). The multiplier can be used as an arithmetic barrel shifter (left or right) by multiplying by a power of 2.

The ALU always generates a condition code. Like the product, the condition code must be used by the following instruction (i.e. a conditional branch), since all instructions, including control, modify the condition code.

The condition code is one of three mutually exclusive states. It indicates the result of the last ALU operation. The result is either zero, positive, or negative.

In the case of ALU functions '00' (add) and '01' (subtract), the condition code is computed for the entire 36-bit result. For functions '10' (logical AND and OR) and '11' (ANDNOT and XOR), the result only takes into account the 16 LSBs of the result, and the result will either be zero or "positive" (i.e. non-zero).

The organization of the logical operations may appear strange. They are designed to produce two possible 18-bit results, split into the low and high ALU outputs. The condition code for these only looks at the low 16 bits of the result, and thus is relevant only for the AND and ANDNOT operations. If desired to test the OR or XOR results, then the result must be stored and tested with an additional instruction.

The logical operations always use the low half of the ALU A input, but split the ALU B input into low and high parts. A common scenario is to provide the ALU A input from *MA1, and the ALU B input from either *MA2, or *MA2 shifted to the high half. The unshifted version is used for AND and ANDNOT, and the shifted version for OR or XOR.

There are almost no "immediate data" instructions. The only exception is the ability to store a constant (between -512 and +511 inclusive) to a working set location. All other program constants must be obtained from working set storage, having been previously set using initial values in the FPGA config, or some form of computation on the small constants available.

Indexed Addressing Modes:

The index mode bits (20 and 21) allow the option of indirect memory addressing. If the index mode bit is set to 0, then its corresponding address field is used as-is, as a direct working set address. Otherwise, the contents of one of the index registers (AX or BX) is added to the address, resulting in an indexed working set address. The final address is known as MA1 or MA2 for convenience.

The addition of the index register is not quite direct. The addition is only performed in the low 9 bits of the address. The MSB of the address is always passed through directly. The index registers AX and BX are 9 bits wide, and the addition "wraps around" in the low 9 bits, hence indexing cannot transparently cross the 512-word boundary. AX and BX are loaded by special control instructions, described next.

Control Instructions:

Control instructions include branches (conditional, unconditional and indirect) as well as index register manipulation. Some of the instruction bit fields are common between control and arithmetic instructions; others have different meanings.

Field Bits	Function	Description
9 - 0	Work Address 1	This specifies the first address in the work space. It is either an immediate (direct) address, or is added to the contents of index register 1 (AX) to form an indirect address. The resulting address is called MA1.
19 - 10	Work Address 2	This specifies the second address in the work space. indexed mode used index register 2 (BX). The resulting address is called MA2.
20	Index Mode 1	If 1, specifies index address mode for the first address. If 0, specifies immediate mode.
21	Index Mode 2	As above, for the second address.
Note that the above fields are the same as for arithmetic instructions		

Field Bits	Function	Description
25 - 22	Reserved	These should be set to zeros, for upward compatibility.
26	Arithmetic/Control	Always set to '1' for control instructions
27	Load AX Indirect	Loads the AX index register from working set addressed by MA1
28	Load BX Indirect	Loads the BX index register from working set addressed by MA2
29	Load AX Immediate	Loads the AX index register with MA1
30	Load BX Immediate	Loads the BX index register with MA2
31	Jump Indirect	If '1', the jump target address is the contents of the working set addressed by MA2. Otherwise, the target is MA2 directly.
34 - 32	Condition Select	This selects the conditional branch condition: 000 - never branch (used with pure index loads) 001 - branch if negative 010 - branch if zero 011 - branch if zero or negative 100 - branch if positive 101 - branch if non-zero 110 - branch if zero or positive 111 - unconditional branch
35	Reserved	This should be set to zero, for upward compatibility.

Table 18. Control Instruction Format

When branching, the target address is either the MA2 address, or the word at that location. This value must be one less than the actual intended target address location.
There is no native stack in White Heat, so there is no native call/return mechanism. However, it is still possible to define subroutines, provided that the subroutines do not directly or indirectly call themselves (recurse).
A subroutine call starts with explicitly storing the return address in a prearranged location (unique for each subroutine). Then, the program jumps to the start of that subroutine. When the subroutine wishes to return, it performs an indirect jump via the pre-arranged location (which contains the return address). This is merely a manual way of performing the call/return sequence of other processors with a stack. (This is not unheard of. For example, the IBM S/390 architecture has always used a register for the return address).
An important consideration is that branches always start executing at one instruction past the numeric value of the target address. For subroutine calls, this implies that the return address should numerically equal the address of the branch-to-subroutine instruction itself (not the instruction which would be the first to execute on return).

Instruction Execution Time

All instructions execute in 1 or 2 clocks (about 20 or 40ns).
Instructions take 2 clocks in the following cases:

1) An arithmetic instruction stored a result anywhere i.e. any one or more of bits 22-25 are set in the instruction word, and bit 26 is zero.

2) Indirect jumps (bits 26 and 31 set)

3) Indirect load of AX or BX (bit 26, and one or both bits 27 or 28 set)

4) Any branch which was not unconditional (bit 26 set, and bits 32-34 are not '111') AND the previous instruction was a 1-clock instruction.

In all other cases, the instruction takes 1 clock.

The Reference White Heat Program

Since White Heat programming is an arcane subject, most G-REX programmers will want to avoid delving into it. Fortunately, the reference FPGA configuration includes predefined White Heat code, which provides the missing link between the MCU and the non-axis-related facilities: namely, the general purpose digital and analog I/O.
The MCU indirectly accesses these facilities (which we call "peripherals") by writing and reading the White Heat shared memory space.
In spite of appearances, this is actually very easy to do. For the sake of concise explanation, the shared memory space is mapped as a C structure as follows. Note that the 'word' type means an unsigned 16-bit integer, and 'lword' means an unsigned 32-bit integer (both stored with LSB first, in the lowest addressed location).

```
struct {
        lwordoutputs;
        word dac_out[4];
        lword inputs;
        word adc_in[4];
} shared_mem;
```

To make use of this, note that the outputs come first (a total of 12 bytes), followed by the inputs (also 12 bytes).
Assuming the MCU implements an interrupt handler tied to the INT signal, then that ISR should perform the following steps to write the next set of outputs, and read the most recent inputs.

1) Write 0x0000 to the White Heat address register (0x4,0x5) - this resets the shared space address to zero, since the above structure is mapped to that location.

2) Write the first 12 bytes of the structure sequentially to the White Heat write data register (0x7).

3) Read the next 12 bytes of the structure from the White Heat read data register (0x3).

As you can see, this is a very simple procedure, and extremely efficient from the MCU's point of view. An example of Rabbit code to implement the above algorithm is on page 12.

The only missing piece of information is how to interpret the results as defined by the above structure. This is outlined below, for each field.

lword outputs

In this 32-bit field, the MCU packs the value for each of the 16 general-purpose digital outputs into the low half. The high half should be set to zeros for upward compatibility. The bit value for each output is set to '1' to turn the output ON (that is, pull it to ground). Bit 0 (the LSB) operates the output numbered 1, bit 1 operates output 2 and so on.

word dac_out[4]

This array of 4 16-bit values sets the Digital to Analog Converter outputs. A value of 0x0000 gives a zero-volt output, and a value of 0xFFFF gives full-scale output. The DACs on the G-REX are really 8-bit, so the low byte of each value is a "don't care".

lword inputs

In this 32-bit field will be placed the most recent general-purpose digitial input readings. In this case, the bit mapping is a bit more complex. Bits 31 down to 16 (the MSBs) contain the reading for the inputs numbered 16 down to 1 respectively. Bits 13 down to 8 contain the reading for the limit switch inputs for axes C, B, A, Z, Y, X respectively. Other bits should be ignored.

word adc_in[4]

This array of 4 16-bit values will contain the latest Analog to Digital Converter readings. As for the DAC outputs, these values are scaled up so that, in principle, 0x0000 is minimum and 0xFFFF is maximum. Since the ADCs are 8 bits, the actual values will have 0x00 for the low byte (implying that the maximum value will be 0xFF00).

G101 rev3 Block Diag.