WG006 Fingertip Programming Guide

Sept 14, 2011

Table of Contents

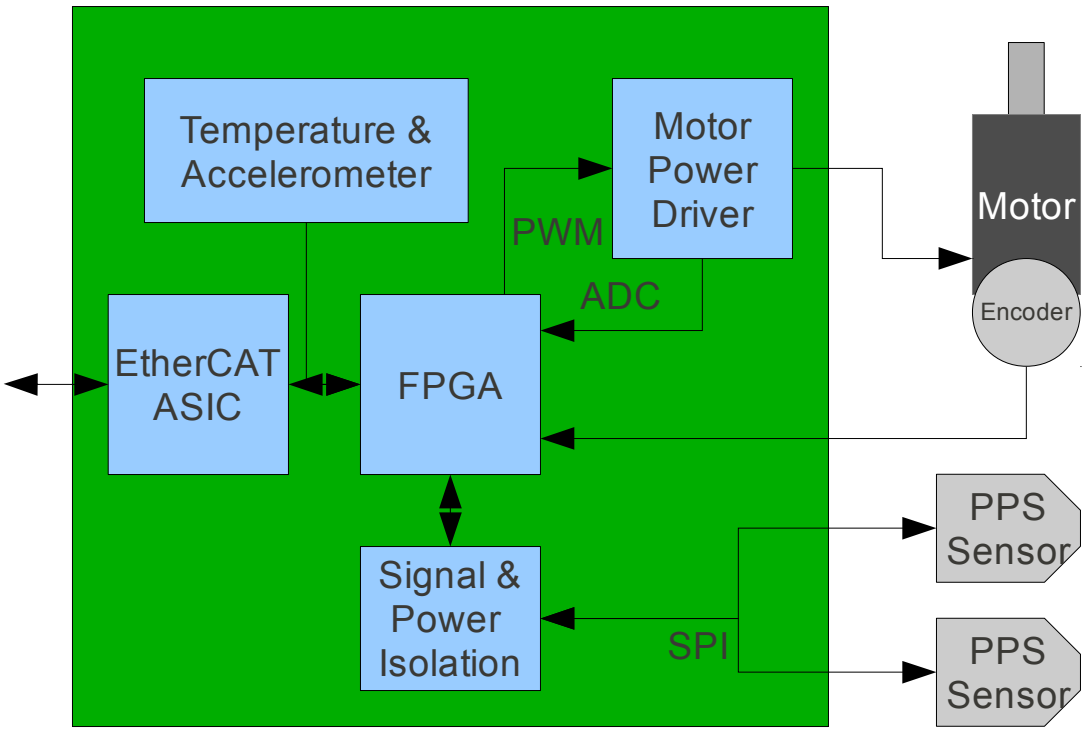
WG006 Motor Controller Board....4
PicoBlaze Microcontroller....5
PicoBlaze....5
Assembler....5
Hardware peripherals....5
Advantages of using a soft-processor....5
Getting Started....6
Unpack files....6
Setting $PATH environment variable....6
Using eclist....8
Using pressure utility....9
Saving binary pressure data....10
Update gripper firmware (fwprog)....11
Build picoasm....13
Compiling firmware with picoasm : Hello World....14
Loading new PicoBlaze firmware....14
Compiling pressure sensor firmware....15
PicoBlaze Peripherals....16
Pressure Data Buffer....16
Double-buffering and data transfer....16
Sequential writes using index pointer....17
Transfer atomicity....18
Transfer sequencing and flow control....18
Short Delays....19
SPI Master....21
SPI clock speed....21
SPI data transfer....21
SPI chip selects....21
SPI code example....22
High throughput SPI reads....23
Timers....25
Timer usage example....25
Timer versus Delay....26
ROS Integration....27
pr2_etherCAT....27
ethercat_hardware....27
EthercatDevice....27
WG0X....27
WG06....27
Getting pr2_ethercat_drivers overlay....27
Building and running pr2_etherCAT....29
Adding Hello World to wg06 pressure diagnostics....30
Realtime data in ROS....30
Publishing raw pressure sensor data....31

Next steps....33
Gripper Development Kit....34
Kit contents....34
Board Safety Issues....34
Electrostatic discharge (ESD)....34
Hot plugging....34
Power and communication wiring harness....34
Power wiring hookup....34
Power supply requirements....34
Communication wiring....35
Install ROS....35
Networking....35
pr2-grant....36
Glossary....37

WG006 Motor Controller Board

The WG006 motor controller board (MCB) is only used in the PR2 gripper. The name "WG006" or "WG06" comes from the Willow Garage product ID for the device : 68-05006. Only the version "E" of the WG06 is used in production PR2s. The rest of this document is specific to that version.

Below is a simplified diagram of the WG006 system architecture. The EtherCAT ASIC is used for communication to the computer. All the rest of the peripherals are controlled by an FPGA. The FPGA is the "brains" of the device, reading sensor data and executing control commands.



The WG006 uses a SPI for communicating with two Pressure Profile Systems (PPS) pressure/tactile sensor at the tips of the PR2 gripper. The best information on the electrical and connector information is the PR2 Tactile Sensor Guide on the PR2 Modulaty Webpage.

For purposes of fingertips sensor modularity a PicoBlaze soft-processor is implement in the FPGA and controls SPI bus. The PicoBlaze's is built as a sandbox, bad firmware should not effect the operation of the rest of the FPGA.

PicoBlaze Microcontroller

WG006 firmware revision 2.xx uses a (version 3) Xilinx PicoBlaze soft processor to drive communication with device. Version 1.xx uses a state machine to communication with the pressure sensors on the figure tips.

PicoBlaze

The PicoBlaze can described as an 8bit microcontroller with a minimal instruction set. The PicoBlaze uses exactly 2-clock cycles per instruction. On the WG006 the PicoBlaze clock speed is 25Mhz, which provides 12.5 MIPS of processing power.
A good guide on the PicoBlaze processor is provided by Xilinx in the PicoBlaze 8-bit Embedded Microcontroller User Guide.

Assembler

There are a couple free/open source assemblers for the PicoBlaze. Most of the current PicoBlaze firmware was written with the open-source kPicosim IDE. kPicosim provides an editor, assembler, and simple simulator.

To compile PicoBlaze assembly into a format that can be loaded on the WG006, a special version of a Linux command line tool called **picoasm** is provided by Willow Garage.

Hardware peripherals

The PicoBlaze does not provide any standard peripherals. Instead it provides an I/O for communicating with hardware peripherals. The I/O bus uses an 8bit address and 8bit data. I/O bus reads can be performed with the "**input"** instruction. I/O bus writes can be performed with the "**output"** instruction.
perform reads on the I/O bus The hardware peripherals for the WG006 PicoBlaze were designed by Willow Garage for the specific purpose of communicating with general SPI slave devices. The peripherals include:

- 8bit SPI master peripheral for transferring data to/from fingertip sensors
- 2x 8bit timers to allow easy timing of communication cycles
- 512 byte double-buffer for sending sensor data to computer

Advantages of using a soft-processor

The PicoBlaze firmware can be recompiled and reloaded in seconds. Recompiling the FPGA firmware takes minutes. As mentioned before, the PicoBlaze is implemented in a sandbox, even the worst PicoBlaze firmware should not adversely effect other WG006 functionality. Bad WG006 FPGA firmware can essentially brick the device (or worse). Finally, the PicoBlaze has a simple assembly instruction set, making it easy to learn. While the FPGA is programmed in a hardware description language (HDL) which many programmers find awkward to use.

Getting Started

This getting started guide is meant to be run on computer C1 of the PR2.

Unpack files

First get file 'wg006_fingertip_dev.tbz', then unpack it in home directory.

```
cd ~/.
tar xvjf wg006_fingertip_dev.tbz
```

It should create the following directory tree:

```
~/wg006_fingertip_dev/
          WG006ProgrammersGuide.pdf
          picoasm/
                    ...
          tools/
                    eclist
                    fwprog
                    picoblaze_prog
                    pressure
                    delay_cycles
          firmware/
                    hello.s
                    count.s
                    pressure_fw.s
```

Setting $PATH environment variable

There are a couple of program in the “tools” and “picoasm” sub directories. Create a file called “setup.bash” that will add these directories to your path.

```
cd ~/wg006_fingertip_dev
echo export "PATH=$PWD/tools:$PWD/picoasm:\$PATH" > setup.bash
```

You can now source the “setup.bash” script to added these tools to your path.

```
source setup.bash
```

Run a quick check to make sure the “eclist” program is found on your path:

```
which eclist
```

It should output something like:

```
/u/<username>/wg006_fingertip_dev/tools/eclist
```

Using eclist

The tools directory provides some binary tools that could be useful when developing EtherCAT firmware.

One tool call "**eclist**" will list all EtherCAT devices it finds. Try running it:

```
cd ~/wg006_fingertip_dev/tools
pr2-grant eclist -iecat0
```

You should see output like :

```
 Position  Serial  Product ID   Board Rev   FW Rev  Name      Motor Name
        1  01026     6805014         B.2           WG014
        2  01026     6805014         B.2           WG014
        3  03116     6805005         F.3    1.22   WG05      br_caster_r_wheel_motor
        4  01420     6805005         F.2    1.22   WG05      br_caster_l_wheel_motor
          ... MORE OUTPUT ...
```

If you want to see only the gripper (WG06) devices,

```
pr2-grant eclist -iecat0 | grep WG06
```

You'll see something like this:

```
 Position           Serial  Product ID   Board Rev   FW Rev   Name      Motor Name
       25            01100     6805006         E.0    1.01    WG06      l_gripper_motor
       34            01071     6805006         E.0    1.01    WG06      r_gripper_motor
```

In the abouve example, the right gripper is at position 25 and the left gripper is at position 34. Remember the position of WG06 (gripper MCB) that you will be developing on. You will be using this position for the rest of the tutorial. This position will be used as a command line argument for **pressure**, **fwprog**, and **picoblaze_prog**.

Using pressure utility

One tool called "**pressure**" will read an output pressure data from a single gripper MCB. Try using running it on the PR2. Replace the **XX** in the command below with the position of on the gripper gripper MCBs.

```
cd ~/wg006_fingertip_dev/tools
pr2-grant pressure -iecat0 -pXX
```

Use should see streaming output that might look something like:

```
timestamp  = 176958001
pressure sensor 1 data:
0x0000 : 0758 12ec 0733 086b 094a 088f 148f 089e
0x0008 : 0893 08c7 0914 0939 089f 0962 09e2 074c
0x0010 : 08fb 09e1 0744 0986 0bac 07ac
pressure sensor 2 data:
0x0000 : 0670 1124 06b1 07dd 07c0 0769 134d 07eb
0x0008 : 07a7 0890 0799 081e 07a9 0843 0879 076a
0x0010 : 085a 09a2 0782 07d0 0a25 05fa
```

If you don't have the PPS fingertips installed, the output will be zeros and you will see warning messages:

```
timestamp  = 2437619000
pressure sensor 1 data:
0x0000 : 0000 0000 0000 0000 0000 0000 0000 0000
0x0008 : 0000 0000 0000 0000 0000 0000 0000 0000
0x0010 : 0000 0000 0000 0000 0000 0000
pressure sensor 2 data:
0x0000 : 0000 0000 0000 0000 0000 0000 0000 0000
0x0008 : 0000 0000 0000 0000 0000 0000 0000 0000
0x0010 : 0000 0000 0000 0000 0000 0000
WARNING ! - pressure sensor 1 data is possibly bad
WARNING ! - pressure sensor 2 data is possibly bad
WARNING ! - possibly bad pressure data
```

Use Ctrl-C to kill program.

Saving binary pressure data

It's also possible to use pressure utility to save raw binary data from pressure data buffer info file. This will be useful later since you will be changing PicoBlaze FW to write pressure data in a format that is different than what pressure utility normally expects:

```
cd ~/wg006_fingertip_dev/tools
pr2-grant pressure -iecat0 -R /tmp/pressure_data.bin -pXX
```

You can use **hexdump** to view the pressure data in a human-readable form:

```
hexdump -Cv /tmp/pressure_data.bin
```

You might see something like:

```
00000000  31 25 41 28 3f 07 ed 12  1a 07 46 08 45 09 86 08  |1%A(?.....F.E...|
00000010  8f 14 90 08 95 08 d1 08  08 09 14 09 95 08 54 09  |..............T.|
00000020  c8 09 40 07 e1 08 c9 09  3e 07 7d 09 ab 0b a9 07  |..@.....>.}.....|
00000030  65 06 1b 11 a5 06 ce 07  b8 07 75 07 49 13 ea 07  |e.........u.I...|
00000040  b4 07 9d 08 8f 07 12 08  b3 07 50 08 86 08 73 07  |..........P...s.|
00000050  64 08 a4 09 8d 07 dc 07  32 0a ee 05              |d.......2...|
0000005c
```

hexdump is a standard Linux utility. Use the man pages to learn more about command line options that are available.

Update gripper firmware (fwprog)

To use the PicoBlaze soft-processor, you will need to update the gripper FW to version 2.XX . This operation is dangerous. If done improperly, it can "brick" the MCB. Please read all the instructions in entire section before running any commands on the PR2

First, make sure no ROS programs are running on the robot. Both pr2_etherCAT (ROS) and fwprog act as EtherCAT masters. There can only be one EtherCAT master, running more than one EtherCAT master at the same time will cause both programs to encounter odd errors as they compete for control of the EtherCAT bus.

Next, make sure the PR2 is plugged into a wall. Having the PR2 run out of power while updating the firmware will brick the MCB.

Run **fwprog** with firmware file "wg006_gripper_E_2.208_rev2005.bit". The -a option will instruction firmware to use the FW with all devices that it is applicable. It takes a while for the program to complete, so be patient.

```
cd ~/wg006_fingertip_dev/tools
pr2-grant fwprog -iecat0 ../firmware/wg006_gripper_E_2.208_rev2005.bit -pXX
```

fwprog should output something like:

```
../firmware/wg006_gripper_E_2.208_rev2005.bit contains firmware for WG006 (gripper) PCB revE
Using master 0 'ecat0'
Found WG006 : Product Code = 6805006, Serial = 1100, Firmware Revision 1.01, PCB Revision E.00
Programming device 25 with ../firmware/wg006_gripper_E_2.208_rev2005.bit :
Success writing firmware - needed to modify 576 pages
```

To be extra sure, try update the firmware again. This should takes less time, since the tool will detect that no changes need to be made to the firmware. You should see output like:

```
../firmware/wg006_gripper_E_2.208_rev2005.bit contains firmware for WG006 (gripper) PCB revE
Using master 0 'ecat0'
Found WG006 : Product Code = 6805006, Serial = 1100, Firmware Revision 1.01, PCB Revision E.00
Programming device 25 with ../firmware/wg006_gripper_E_2.208_rev2005.bit :
Success writing firmware - needed to modify 0 pages
```

Now the new firmware is programmed into the flash. However the FPGA will only reload its firmware on power-up. To force the FPGA to reload the gripper firmware from flash, you will need to to power-cycle the gripper MCB. The easiest way to power cycle the MCBs is to launch ROS on the PR2, then use the pr2_dashboard to first disable all breakers, and then re-enable all breakers.

Stop ROS once you have power-cycled the MCBs.

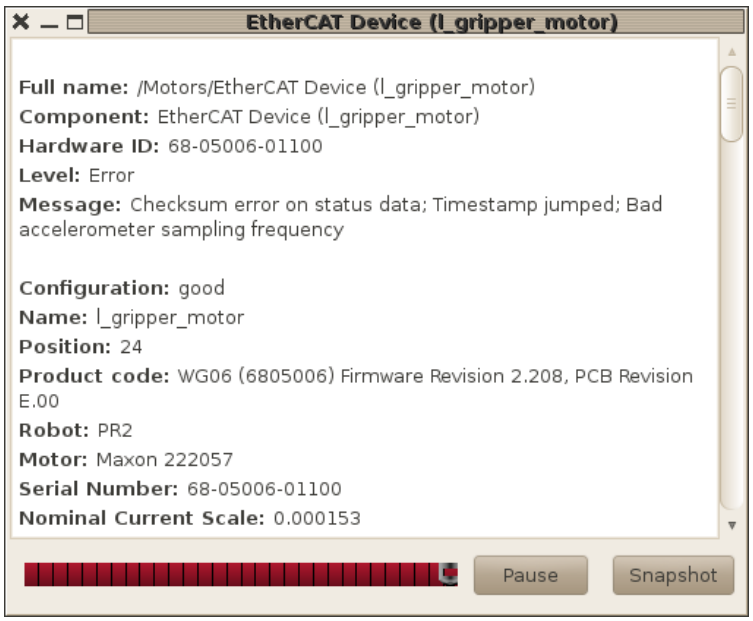
Run **eclist** utility to list the gripper MCBs. You should see one of the gripper MCBs is now running firmware version 2.xx

```
pr2-grant eclist | grep WG06
```

You'll now see that one of the grippers has firmware version 2.208:

```
  Position           Serial   Product ID    Board Rev    FW Rev    Name       Motor Name
        25            01100      6805006          E.0     2.208    WG06       l_gripper_motor
        34            01071      6805006          E.0     1.01     WG06       r_gripper_motor
```

Now the gripper firmware has been updated. Unfortunately, the problem with the new gripper FW is that older versions of **pr2_etherCAT** in ROS do not understand how to communicate with the new gripper FW. If you run an old version **pr2_etherCAT**, it will halt on startup complaining that there is a checksum error with on status data.



Build picoasm

A special version of **picoasm** (availble in the picoasm subdirectory) is needed to build a firmware image that you can load into the gripper.

First compile **picoasm**

```
cd ~/wg006_fingertip_dev/picoasm
make
```

Now try running **picoasm** without any input files:

```
picoasm
```

You should get output like:

```
ERR: Input source file missing.
picoasm Version 0.2 - PicoBlaze Assembler based on kpicosim
USAGE:
 -i <input file>      PicoBlaze source file
 [-t <template file>] Verilog/VHDL template file.
 [-v <output file>]   write generated Verilog output to file.
 [-v <output file>]   write generated VHDL output to file.
 [-m <module name>]   Verilog module or VHDL entity name.
                      Default = input file base name.
 [-H <filename>]      Output assembled code in special ASCII hex format.
 [-l <filename>]      name file to put debug listing.
```

Compiling firmware with picoasm : Hello World

There is a very simple firmware file for the pressure sensor PicoBlaze processor in the firmware directory called “hello_world.asm”. This firmware puts the ASCII characters for “Hello” into pressure data buffer.

```
cd ~/wg006_fingertip_dev/firmware
picoasm -i hello.s -H hello.hex
```

picoasm will produce the output message:

```
Generated hex file 'hello.hex'
```

The file “hello.hex” contains the compiled firmware in a format that can be loaded onto the PicoBlaze processor on the gripper MCB.

Loading new PicoBlaze firmware

The **picoblaze_prog** tool can be used to new firmware onto a PicoBlaze processor . Try loading the “hello.hex” :

```
cd ~/wg006_fingertip_dev/tools
pr2-grant picoblaze_prog -r ../firmware/hello.hex -iecat0 -pXX -W
```

You should see output like:

```
Reading FW from file hello.hex
Found WG006 : Product Code = 6805006, Serial = 1101, Firmware Revision
2.208, PCB Revision E.00
Writing FW to device
```

The following command should have loaded the new firmware. Use **pressure** and **hexdump** to read the and display the message from the PicoBlaze output data buffer:

```
pr2-grant pressure -iecat0 -R /tmp/hello_data.bin -pXX
hexdump -Cv /tmp/hello_data.bin
```

The **hexdump** output should have the message “Hello WG006” in it:

```
00000000  48 65 6c 6c 6f 20 57 47  30 30 36 21 0c 0d 0e 0f  |Hello WG006!....|
00000010  10 11 12 13 14 15 16 17  18 19 1a 1b 1c 1d 1e 1f  |................|
```

Take a look at “hello.s” to get and understanding of what is going on. For fun, modify the code with a different message is put in the data buffer.

Compiling pressure sensor firmware

The file “pressure_fw.s” contains the firmware used to read pressure data from the PPS pressure sensors on the finger tips. It uses some PicoBlaze peripherals that weren't used in “hello.s”.

Compile pressure firmware and load onto WG006 :

```
cd ~/wg006_fingertip_dev/firmware
picoasm -i pressure_fw.s -H pressure_fw.hex
cd ~/wg006_fingertip_dev/tools
pr2-grant picoblaze_prog -r ../firmware/pressure_fw.hex -iecat0 -pXX -W
```

Then use pressure tool to verify new firmware is reading data from pressure sensor as expected.

```
pr2-grant pressure -iecat0 -pXX
```

The beginning of “pressure_fw.s” has constants for all the peripheral registers available to the PicoBlaze processor. Because of this, **“pressure_fw.s” is a good starting point for writing your own firmware for a custom fingertip device.**

PicoBlaze Peripherals

This section describes the hardware peripherals that can be accessed by the PicoBlaze processor.

Pressure Data Buffer

A 512 byte double-buffer is provided to allow data pressure data between PicoBlaze and EtherCAT master running on computer. Anything from 0-512 bytes of data can be written. When data in buffer is complete, a flag can be set to flip buffer and start transfer of data to EtherCAT master.

Double-buffering and data transfer

Because of the double-buffer, new data can be written to buffer immediately after transfer is requested. Before flipping buffer you will need to wait for data to be transferred to computer from old buffer. Most of the time, the old data will have already been transferred to master before buffer is ready to be flipped, however state of buffer should always be checked before requesting new transfer.
Below is psuedo-code that gives an example of how buffer flip should be performed:

```
; Put data in buffer
          call fill_buffer
; Now that buffer is full, we can request new transfer...
; However, wait for old transfer to complete first
wait_loop:
          input s0, PRESSURE_CTRL_REG
          test s0, PRESSURE_BUSY_FLAG
          jump NZ, wait_loop
; Now new buffer transfer can now be requested
          load s0, PRESSURE_READY_FLAG
          output s0, PRESSURE_CTRL_REG
; Because of double-buffer, new data can be put in
;   buffer immediately after transfer is requested
          call fill_buffer
```

Sequential writes using index pointer

Data buffer is setup to make writing sequential data very easy and fast. An index pointer is used to specify where new data is put in buffer. When **PRESSURE_DATA_REG** is written the value is redirected to the pressure buffer based on index pointer, then the index pointer is post-incremented. And example of writing the values 0x01, 0x02, 0x03 sequentially to the buffer is given below

```
; reset index pointer to beginning of buffer (0)
          load s0, 0
          output s0, PRESSURE_INDEX_LOW_REG
          output s0, PRESSURE_INDEX_HIGH_REG
; write 0x00, 0x01, 0x02 to pressure buffer
          load s0, 0
          output s0, PRESSURE_DATA_REG
          load s0, 0
          output s0, PRESSURE_DATA_REG
          load s0, 0
          output s0, PRESSURE_DATA_REG
```

Its possible to randomly access the buffer by changing `PRESSURE_INDEX_REG` value before writing `PRESSURE_DATA_REG`. Below is an example of this:

```
; write 0x42 to byte 300 ( 0x12C ) of pressure data buffer
          load s0, 1
          output s0, PRESSURE_INDEX_HIGH_REG
          load s0, 2C
          output s0, PRESSURE_INDEX_LOW_REG
          load s0, 42
          output s0, PRESSURE_DATA_REG
```

Transfer atomicity

Data transferred between pressure buffer and EtherCAT master is guaranteed be atomic by hardware, if hardware is used correctly. This means that EtherCAT master will never get data where half the data is old and half the data is new.

Transfer sequencing and flow control

There is no sequencing or flow control of pressure buffer data as there are with protocols like TCP/IP. The pressure data might get lost or received more than once. It best it try to keep data as stateless as possible (usefulness of current data does not depend on having previous data).
The realtime_loop on the PR2 reads data from every device at rate of 1kHz. If PicoBlaze updates pressure data buffer every 3 milliseconds, then the realtime loop will read the same data more than once. Also, if PicoBlaze flips buffer every $1/10^{th}$ millisecond, then the realtime loop will 9 of 10 updates.
The simple solution for preventing repeated data to put a sequence counter in the data buffer. Have the sequence counter increment every time the pressure data buffer is flipped. By using sequence counter, the computer software can determine if data is new, or a repeat.
The firmware for the PPS pressure sensor uses a 32bit timestamp that acts as sequence counter. Just before reading new sensor data, the PicoBlaze grabs the current timestamp and writes it to beginning of buffer.
The solution for lost data is not so clear. TCP/IP uses feedback from receiver to verify that previously sent data was received correctly. However, there is currently no way to get such feed back to the PicoBlaze microcontroller.
In the future, supported will be added to hardware to allow reliable sequential communication can be passed between PicoBlaze and realtime loop in computer.

Short Delays

Sometimes, short delays are needed between certain communication operations. Since PicoBlaze is programmed in assembly, and every instruction always takes a predictable amount of time, it is possible to create short delays with instructions that do nothing, such as :

```
load s0, s0
```

For delay from 1us and longer its possible to create a function that runs an idle loop for a certain number of cycles. One such example function is given below:

```
;***************************************************************
; Delays for a given number of milliseconds by running an idle loop
; Modifies
;   s0
; Inputs
;   s0 : Number of delay cycles - use DELAY_XXUS_COUNT constants
; Outputs:
;   NONE
delay_us:
          sub s0, 1
          jump NZ, delay_us
          return
```

This function just preforms an idle loop for a given number of microseconds. It it up to the programmer to provide the right number of loop cycles (in s0) to get the proper delay. For the An example of call to this function is given below to help us determine how many instruction cycles a given call takes.

```
; delay for 10 cycles
load s0, 10
call delay_us
```

Lets calculate how much time this code takes:

Instruction(s)	# Instruction Cycles	Note
load s0, 10	1	
call delay_us	1	
sub s0, 1	1 x 10 = 10	Loop is run 10 times
jump NZ, delay_us	1 x 10 = 10	Loop is run 10 times
return	1	
Total	3 + 2 x 10 = 23	

From the above example its easy to see, that is s0 is loaded with the value N, it will take (2*N + 3) instructions.
On the PicoBlaze, every instruction takes 2 clock cycles. With a 25Mhz clock each instruction takes 800 nanoseconds or (0.08 microseconds).
So if s0 was loaded with N and then delay_us was called, it would take (2*N + 3) * 0.08 microseconds.
It is also possible to determine the number of cycles that are need to delay a certain number of

microseconds:

num_cycles = ((delay_time_us * 12.5) - 3) / 2 = delay_time * 6.25 – 1.5

For example To delay 1 microsecond, you would load 5 into s0 before calling delay_us

(1us * 6.25) – 1.5 = 4.75 → 5

To make code readable, its best to make table of named constants for different delays, and to use those constants:

```
CONSTANT DELAY_5US_COUNT, 1E
CONSTANT DELAY_4US_COUNT, 18
CONSTANT DELAY_3US_COUNT, 12
CONSTANT DELAY_2US_COUNT, B
CONSTANT DELAY_1US_COUNT, 5
```

There is simple python script in the tools directory that will print determine delay cycles for given delay values. Try running:

```
delay_cycles 4.5  10  20
```

You should see output like:

```
delay (us)          cycles    actual delay (us)
4.500000            0x1B      4.560000
10.000000           0x3D      10.000000
20.000000           0x7C      20.080000
```

SPI Master

There is a peripheral that perform fast SPI transfers to a slave device. The peripheral will transfer 8bits of SPI data to/from slave device with selectable clock speed.

Currently, the SPI peripheral only runs in mode 0 (CPOL=0, CPHA=0). There Wikipedia Serial Peripheral Interface page that has a good explanation of the different SPI modes.

SPI clock speed

The system clock runs at 25Mhz. This clock can be divided down to produce the clock speed used for the SPI transfers. The formula for determining the SPI clock speed from the divisor value is:

SPI Clock Frequency = 25Mhz / 2 / (Divisor+1) = 12.5Mhz / (Divisor + 1)

The divisor is an 8-bit value so the slowest clock speed available is :

12.5Mhz / (255+1) = 48.8kHz

The fastest (theoretical) clock speed available is 12.5 Mhz. However, high speed SPI signaling has never been tested and may require proper signal termination on the figure tip devices.

SPI data transfer

The SPI peripheral transfers 1-byte data at a time. Starting transfer is as simple as writing SPI_DATA_REG with new output value. A busy status flag can be polled to wait for transfer to complete. To retrieve the received data, the SPI_DATA_REG can be read after a SPI transfer has completed.

Unlink other protocols (ie RS232 serial), SPI always receives 1 byte of input while sending 1 byte of output data that is sent. In some situations either received data does not matter. On these cases, DATA_REG does not need to be read after transfer is complete. In other situations, only the input data matters. For these cases, start a SPI transfer by writing “dummy” data to SPI_DATA_REG.

SPI chip selects

The SPI bus has two chip selects, one for each finger. The chip selects are active-low – they are assert with the signal is low. The digital isolator used for the chip selects is slower than the isolator used for the SPI bus. After asserting a chip-select, the program should wait for 1 or 2 microseconds before transferring SPI data.

SPI code example

Below is an example of performing communication with an Analog Devices LTC1867L 16-bit ADC. The LTC1867L is given a 7bit command telling it what input channel to convert. While sending the device a new command, the device provides the 16-bit result for the last conversion.
In the example a 16 bit transfer is performed as 2x 8-bit transfers. Only the first 7bits of command data matter, for the remain 9bits we send zeros. The command value used in the example is 1000000b, which ends up being 0x8000 when 9 bits of zeros appended to end.

```
; First set SPI clock speed to 500kHz.
;   Divisor = 12.5Mhz / 500kHz  + 1   =   26  = 0x1A
          load s0, 1A
          output s0, SPI_CLOCK_REG
; Assert Chip-select 1
          load s0, SPI_ASSERT_CHIPSEL_1_FLAG
          output s0, SPI_CTRL_REG
; Wait for 2us for chip-select to propagate to device
          load s0, 2
          call delay_us
; Send MSByte of command to device  (0x80)
          load s0, 80
          output s0, SPI_DATA_REG
; Now wait for transfer to complete
          wait_loop_1:
                    input s0, SPI_CTRL_REG
                    test s0, SPI_BUSY_FLAG
                    jump NZ, wait_loop_1
; Get save first byte read from LTC1C1967L into s2
          input s2, SPI_DATA_REG
; Send LSByte of command (0x00)
          load s0, 0
          output s0, SPI_DATA_REG
; Now wait for transfer of second byte to complete
          wait_loop_2:
                    input s0, SPI_CTRL_REG
                    test s0, SPI_BUSY_FLAG
                    jump NZ, wait_loop_2
; Save second byte read from LTC1C1967L into s1
          input s1, SPI_DATA_REG
; De-assert chip-select
          load s0, SPI_DEASSERT_CHIPSEL_FLAG
          output s0, SPI_CTRL_REG
```

High throughput SPI reads

Its possible achieve higher SPI throughput by doing certain processing at points were the PicoBlaze would normally be waiting for previous operation to complete. This extra throughput comes at cost of more code complexity.

For example, assume that you want to read 100 bytes from SPI slave device into the pressure buffer.

```
; s3 = 0 for rest of code
         load s3, 0
; to transfer 100 bytes, perform loop 99 times (100-1 => 99)
         load s1, 63  ; 99 = 0x63
; start first transfer before entering main loop
         output s3, SPI_DATA_REG
copy_loop:
         ; wait for transfer to complete
         wait_loop:
                  input s0, SPI_CTRL_REG
                  test s0, SPI_BUSY_FLAG
                  jump NZ, wait_loop
         ; start next SPI xfer immediately after prev xfer one completes
                  load output s3, SPI_DATA_REG
         ; data from previous transfer is stored in buffer that remains
         ; valid until new transfer complete, grab it and put in buffer
                  load s1, SPI_DATA_REG
                  output s1, PRESSURE_DATA_REG
         ; should we repeat loop?
                  sub s1, 1
                  jump NZ, copy_loop

; wait for last transfer to complete and store data
wait_loop_2:
         input s0, SPI_CTRL_REG
         test s0, SPI_BUSY_FLAG
         jump NZ, wait_loop
; put last byte into pressure buffer
         input s1, SPI_DATA_REG
         output s1, PRESSURE_DATA_REG
```

Timers

There are two 8-bit counters that can be used to produce accurate timing for certain tasks : TimerA and TimerB. Timer A increments every 0.1 milliseconds. Timer B increments every 1 microseconds. Each timer has an associated 8-bit limit register when timer matches limit register, the timer value is reset to 0, and an overflow flag is set. A task can poll this overflow flag to accurately time a cyclic task.

Timer usage example

For an example, assume we want to perform some task every 40 microseconds. You can setup Timer B is perfect for this:

```
; Setup timerB to expire every 40us
; 40 us = 1us * 40
          load s0, 28 ; 40 = 0x28
          output s0, TIMER_B_COMPARE_REG
; Reset Timer B counter to 0 and clear overflow flag
          load s0, 0
          output s0, TIMER_B_COUNTER_REG
          load s0, TIMER_OVERFLOW_FLAG
          output s0, TIMER_B_OVERFLOW_REG

main_loop:
          ; perform some type of task...
                    call my_task

          ; wait for timer B to overflow
          wait_loop:
                    input s0, TIMER_B_OVERFLOW_REG
                    test s0, TIMER_OVERFLOW_FLAG
                    jump Z, wait_loop

          ; repeat
          jump main_loop
```

Timer versus Delay

You might be wondering why use a timer over delaying in a busy-loop. To answer this, consider the following example that uses to delay function from earlier instead of a busy loop:

```
main_loop:
          ; perform some type of task...
                    call my_task

          ; wait 40 microseconds
          load s0, DELAY_40US_COUNT
          call delay_us

          ; repeat
          jump main_loop
```

The delay code is a lot shorter. However, the problem with the delay, is that is does not take into account the time taken by my_task. If `my_task` took 5 microseconds to complete, then the main loop would repeat every 45 microseconds instead of every 40 microseconds.

ROS Integration

pr2_etherCAT

pr2_etherCAT is a ROS package in the pr2_robot stack. pr2_etherCAT is also a ROS node in the pr2_etherCAT package. On the PR2, the pr2_etherCAT node is also know as the "realtime_loop". pr2_etherCAT communicates with EtherCAT devices (MCBs) and run controllers plug-ins to command those devices.
pr2_etherCAT will find and load a plug-in for each type of EtherCAT device it detects. The plug-ins are loaded based on their EtherCAT product ID. For the gripper MCB, the EtherCAT product ID is 6805006.

ethercat_hardware

The plug-in driver for gripper MCB is located in the ethercat_hardware package.
The ethercat_hardware driver code for the gripper MCB has been recently re-factored, so this document refers to special SVN branch that has been created for the purposed of this document.

EthercatDevice

For the EtherCAT drivers to be loaded as plug-ins they need to be a subclasses of the EthercatDevice. All sub-classes must implement virtual functions such as **construct**, **intialize**, **unpackState**, and **packCommand**. The EthercatDevice class also provides some generic functionality that is useful for most EtherCAT drivers.

WG0X

There is a lot of shared code for Willow Garage MCBs drivers (WG005/WG006/WG021) . Much of the this shared code is part of the WG0X class defined in *ethercat_hardware/src/wg0x.cpp*. The drivers the the WG005, WG006, and WG021 are all sub-classes of the WG0X class.

WG06

The gripper EtherCAT driver is defined in a class called WG06 which is subclass of WG0X. Most of the WG06 code located in *ethercat_hardware/src/wg06.cpp* .

Getting pr2_ethercat_drivers overlay

When changing the code for the WG006 you will need to "overlay" the installed ROS packages with and code from source repository. When running from ROS from overlay, the packages in overlay will be use installed of other stacks/packages that are already installed. For developing in ROS you will need an "overlay" of the following ROS stacks:

- pr2_robot
- pr2_mechanism
- pr2_ethercat_drivers

The utility to create overlay is called **rosinstall**. Unfortunately it may not be installed on your system.

Try running it to find out:

```
rosinstall
```

If rosinstall is not present, install it with the following command:

```
sudo apt-get install python-setuptools
sudo easy_install -U rosinstall
```

Use the *wg006.rosinstall* and rosinstall to create an overlay over electric:

```
cd ~/wg006_fingertip_dev
rosinstall ros_dev /opt/ros/electric/ wg006.rosinstall
```

This will create a new directory called ros_dev.

```
ls ~/wg006_fingertip_dev/ros_dev
```

You should see the following files/folders

```
pr2_ethercat_drivers
pr2_mechanism
pr2_robot
setup.bash
setup.sh
setup.zsh
```

The rosinstall will create a script called setup.bash that will setup your ROS path correctly to used the overlayed stacks instead of the default one. In every new terminal, you will need to source the setup script before using ROS. Alternatively, modify your ~/.bashrc file to do this automatically.

```
source ~/wg006_fingertip_dev/ros_dev/setup.bash
```

Now that you have sourced setup.bash, take a look at your ROS package path environmental variable.

```
echo $ROS_PACKAGE_PATH
```

You should see the directories with the new code mentioned before /opt/ros/electric . This is extremely important. When searching for the ethercat_hardware package, ROS should find the one in your home directory, not the one in /opt/ros/electric.

```
~/wg006_fingertip_dev/ros_dev/pr2_robot:
~/wg006_fingertip_dev/ros_dev/pr2_mechanism:
~/wg006_fingertip_dev/ros_dev/pr2_ethercat_drivers:
/opt/ros/electric/stacks
```

Try running following command :

```
rospack find ethercat_hardware
```

It should have found the ethercat_hardware package in your home directory:

```
~/wg006_fingertip_dev/ros_dev/pr2_ethercat_drivers/ethercat_hardware
```

It is important to remember to source the setup script. If you forget to do this, you will be running the wrong version of code.

Building and running pr2_etherCAT

To build pr2_etherCAT run :

```
rosmake pr2_etherCAT
```

rosmake will build both pr2_etherCAT and packages it depends on (such as pr2_ethercat_drivers).

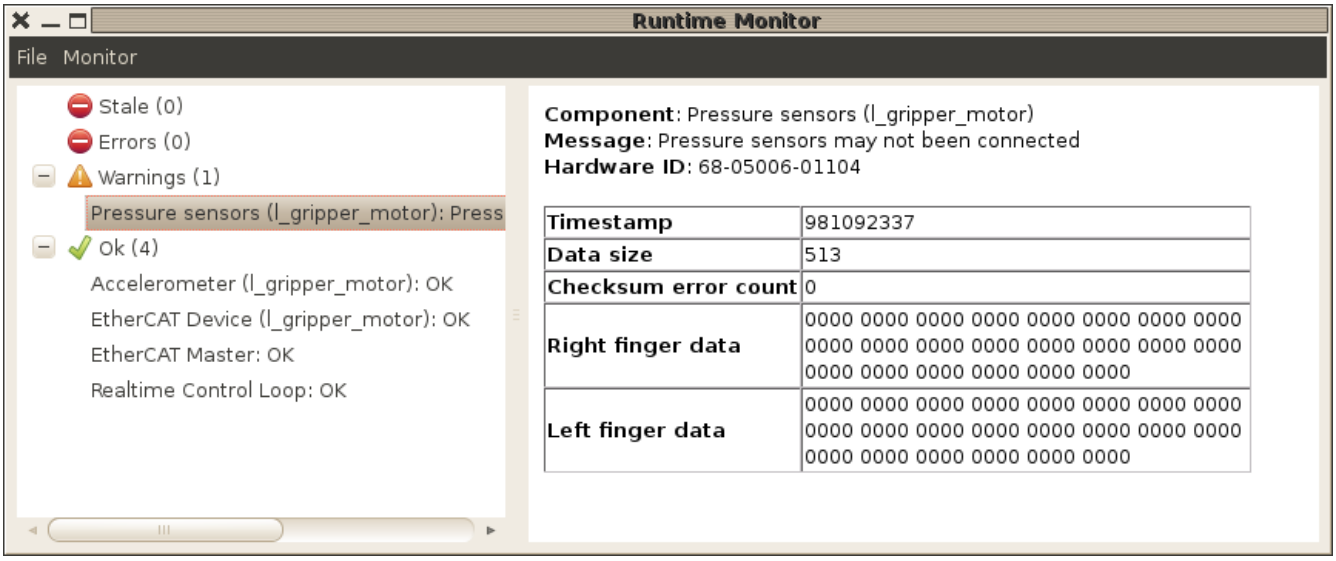
Once build is complete try running a roscore and pr2_etherCAT

```
roscore &
roscd pr2_etherCAT
pr2-grant pr2_etherCAT -iecat0 -x ~/wg006_fingertip_dev/robot.xml
```

On a desktop, use the runtime_monitor to look at the diagnostics data coming from pr2_etherCAT:

```
export ROS_MASTER_URI=http://pr10XX:11311
rosrun runtime_monitor monitor
```

The runtime monitor is similar version to the robot_monitor (which is used as part of the pr2_dashboard). The runtime monitor listens to messages from /diagnostics and organizes the messages by the name field in the message. You can see there is no sub-grouping to the different diagnostics, there is just one big flat list. Try looking at the diagnostics for the right gripper pressure sensor, you might see something like the following :



Usually pr2_etherCAT is started by roslaunching */etc/ros/robot.launch*. However, the previous just ran pr2_etherCAT running by itself. This makes is faster and easier to develop, compile, and test new code. Running and stopping everything in */etc/ros/robot.launch* just takes too long. There is tutorial about running pr2_etherCAT on the ROS wiki if you want to learn more. Below is a quick description of the arguments:

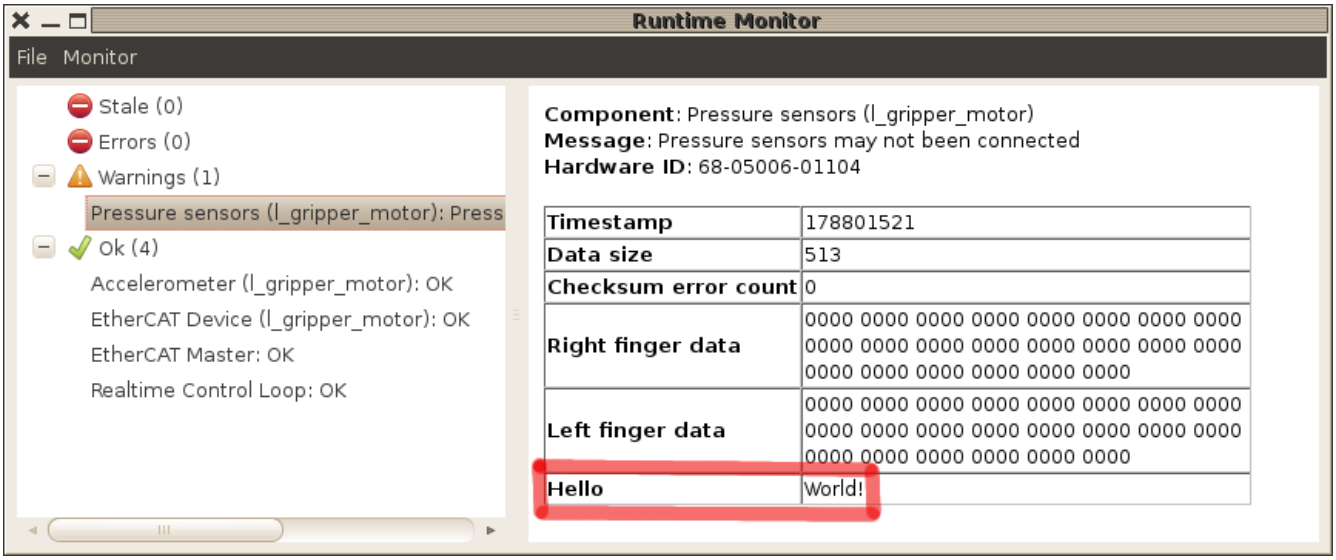
- -i ecat0 : Use the ethernet interface **ecat0** to communicate with the EtherCAT devices
- -x ~/wg006_fingertip_dev/robot.xml : Use a special version of robot description (URDF). This special version has just enough stuff to get pr2_etherCAT running and nothing more.

Adding Hello World to wg06 pressure diagnostics

To help get a accustomed to the code in ethercat_hardware, we are going to add a key-value pair to the WG006 pressure sensor diagnostics. Open up ethercat_hardware/src/wg06.cpp and search for the **diagnosticsPressure** function. Read through the function and try to a get a feel for what it is doing. At the end of the function add the following line:

```
d.addf("Hello",  "%s",  "World");
```

Now compile the ethercat_hardware package and restart pr2_etherCAT. Take a look at the WG006 pressure sensor diagnostics with the runtime monitor. You should see you new piece of data with the key : "*Hello*" and value "*World*".



Realtime data in ROS

While the diagnostics topic is a good place to put debugging information, for putting bulk data. The diagnostics data is only published once and second and all key value pairs are published as strings.

One way to get bulk data, it to publish it on a ROS topic. Unfortunately, the a ROS publisher is not realtime safe. A call to ROS publish *could* block for a long period of time. Blocking the realtime loop, would mean that it would no longer be able to make its 1000 Hz loop rate. This would have detrimental effects on any realtime controller or even other EtherCAT drivers. In fact, you will want to avoid using ROS_ERROR, ROS_WARN, printf, std::cout, std::cerr from the realtime loop. Using the previously mentioned functions wont' crash the program, but they might prevent the realtime loop from running consistently at 1kHz rate.

An alternate to a calling ROS publish directly is to use a "realtime_publisher". The realtime_publisher

is a wrapper around the ROS publisher that protects you from a block ROS publish call. Unfortunately one possible problems with the realtime_publisher, is can "drop" ROS messages.

Publishing raw pressure sensor data

The version of **pr2_ethercat_drivers** that you got with the overlay is from a special branch that is meant for this development kit. There is already a function called publishPressure512 that will take all 512 bytes of the pressure data and publish it as **std_msgs::ByteMultiArray**. Do enable this publishing functionality you will need to make a couple changes to the code.

First open up *ethercat_hardware/include/ethercat_hardware/wg06.h* . Find the following line of code and uncomment it:

```
realtime_tools::RealtimePublisher<std_msgs::ByteMultiArray> *raw_pressure_publisher_;
```

Now search for the **initializePressure** function in *wg06.cpp*. Uncomment the follow chunk of code :

```
// For development purposes publish a ROS message with raw values of pressure data
topic = "raw_pressure";
if (!actuator_.name_.empty())
  topic = topic + "/" + string(actuator_.name_);
raw_pressure_publisher_ =
  new realtime_tools::RealtimePublisher<std_msgs::ByteMultiArray>(ros::NodeHandle(), topic, 2);
// reserve room sure there is room for pressure data in message
raw_pressure_publisher_->msg_.data.reserve(pressure_size_);
```

Now search for the **unpackPressure** function in wg06.cpp. Find the following code and uncomment it.

```
  // Also publish raw pressure sensor data new message every realtime cycle
  // NOTE : this will product a lot of data to ROS, might not be a good idea
  // to do this for things other than development
  if (raw_pressure_publisher_ != NULL)
  {
    if (!raw_pressure_publisher_->trylock())
    {
      // If we could not obtain lock, it probably mean that
      // the ROS publisher just can't keep up.
      // In this case, we just silently drop message.
    }
    else
    {
      // Have lock on realtime publisher message.
      // Can change message while holding lock

      // First copy data into "data" element of ByteMultiArray.
      // For C++ the the data element ends up being a std::vector<uint8_t>
      raw_pressure_publisher_->msg_.data.resize(pressure_size_);
      for (unsigned ii=0; ii<pressure_size_; ++ii)
      {
        raw_pressure_publisher_->msg_.data[ii] = pressure_buf[ii];
      }

      // The std_msgs::ByteMultiArray has a complex "format" element
      // that describes how to convert a 1D data array into
      // a multi-dimensional array.
      // However, we are going to leave the header blank and assume
      // that the program receives the data won't care about the
      // format field.

      // Now that we are done with the data.
      // release the lock and allow data to get published
```

```
      raw_pressure_publisher_->unlockAndPublish();
    }
  }
```

Now compile the ethercat_hardware and restart pr2_etherCAT. Check for the new raw pressure topic:

```
rostopic list | grep raw_pressure
```

You might see something like:

```
/raw_pressure/l_gripper_motor
```

Try echo the pressure data:

```
rostopic echo raw_pressure/l_gripper_motor
```

You might see something like this:

```
---
layout:
          dim: []
          data_offset: 0
data: [illegible]
```

You might have noticed the odd format on the **data** field. This is caused by rostopic trying to print the data as ASCII text. Many binary values don't have ASCII representation so lost of the values are garbage. Another way to view the data is with the echo_raw_pressure in the **wg006_fingertip_dev_tools** package.

```
rosmake wg006_fingertip_dev_tools
rosrun wg006_fingertop_dev_tools  echo_raw_pressure  raw_pressure/l_gripper_motor
```

You might see output like:

```
71 51 bf 24 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00
```

If you want, take a look at the python source code for the tool. It is extremely simple and may be a good starting point for adding parsing functionality.

Next steps

There are a couple ways to proceed from here. On possibility, is to write ROS listener that takes data from raw pressure topic and parses it to useful format. The advantage of this approach, is that it is probably easier and has a faster development cycle. You can also easily record raw data to a bag file and replay data when developing your parse. The disadvantage of this approach is that you might not what to use this as your final result, and might not be very efficient in terms of processing power or bandwidth usage.

Another possibility is to define a new message type your data and do conversion inside of **unpackPressure** function. You can use code for publishing the raw pressure as a template for your own code.

A final possibility and a new data to pr2_hardware_interface. Adding a new data type could allow realtime controllers to receive and make control decisions in realtime (< 1milliseconds). There is no latency guarantee on the data from a ROS topic. Also the realtime controllers reside in the same memory space and the EtherCAT drivers. Passing data through pr2_hardware_interface is as simple as passing a pointer around. The disadvantage of this method, is that it takes much more development effort. You would also need to make changes to the pr2_hardware_interface which might not be accepted into the main-line release.

Gripper Development Kit

While using the gripper MCB on the PR2 is possible, it can be much more convenient to develop on a stand-alone WG006 and a desktop machine. However, more initial setup needs to be performed to use this option.

Kit contents

Board Safety Issues

Electrostatic discharge (ESD)

Electro-Static-Discharge or ESD can damage electrics in ways that are difficult to diagnose. Many of the WG006 I/O interfaces (Ethernet, encoder inputs, pressure sensor SPI bus) have higher level of ESD protections than other board ICs. When working with board, avoid coming in contact with ICs on PCB.

Hot plugging

The act of plugging or unplugging device into active power supply is called hot-plugging. The WG006 should **NOT** be hot-plugged. Before a power supply is connected to WG006, the power supply output should be disabled or turned off. Power supply can be turned on, after power cable is connected. Before unplugging WG006, the power supply should first be shut off.

Power and communication wiring harness

The development kit should have two provide 2x power and communication wiring harnesses. Each harness connects to the 6-position JST GH connector (J1) on the WG006. Connector J1 provides both power and EtherCAT communication to the PCB.

Power wiring hookup

The power for the harness is should be provided to twist pair of black and red wires. The black wire should be connected to the negative supply, the red wire should be connected to the power supply. Plugging in power backwards WILL destroy board.

Power supply requirements

The gripper can run from a any supply ranging from 18V to 70V. However, it is recommend that the lowest possible supply voltage is used. If possible use a power supply with a current limit, this reduces the probability of catastrophic damage to circuit if something is shorted, or connect improperly. For running off an 18V supply, use a current limit of 0.3Amps. The following table gives a set of supply currents to expect when board is working properly. This table assumes supply voltage is 18V.

Situation	Typical supply current from

	18V supply (in Amps)
Start up current	0.300
Board powered	0.040
Board powered + EtherCAT connection	0.053
Board powered + EtherCAT connection + 2x PPS Sensors	0.055
Board powered + EtherCAT connection + 2x PPS Sensors + H-bridge enabled (pr2_etherCAT enables H-bridge)	

During startup, the DC/DC converters on the WG006 need much higher current limit than when board is running. After board is powered, there is very little current draw. If the gripper uses much more current than specified above, then circuitry may have been damaged in some way.

Communication wiring

The wiring harness also has short Ethernet/EtherCAT cable with RJ45 jack. The cable may be too short to be attached to computer, so a RJ45 coupler is provided to connect to a long Ethernet patch cable. While EtherCAT is electrically compatible with Ethernet, it's protocol is not. EtherCAT devices cannot be shared with another Ethernet devices by plugging it into a switch (or hub). For the EtherCAT device to work properly, it must be connected driectly to NIC and computer. For computer s without an extra NIC, a USB-to-Ethernet is provided. The USB-to-Ethernet adapter works on many Linux machines without the need for installing extra drivers.

Install ROS

You will need to install the ros-electric-pr2-desktop package :

```
sudo apt-get install ros-electric-pr2
```

Networking

You will need a computer with a free network card. For non-realtime tools, a USB-to-Ethernet adapter can be used. However for realtime loop (pr2_etherCAT), a USB-to-Ethernet adapter will probably not work because of the the 1 millisecond bus cycle of USB.

If you don't already know the name of the network interface you've plugged the EtherCAT device into, you will need to determine this. On most systems, the first Ethernet interface is called "eth0". There are some tips on figuring out the name of a certain network interface to used on the following ROS wiki page : http://www.ros.org/wiki/pr2_etherCAT/Tutorials/Running

pr2-grant

All programs special Linux networking privileges to communicate with the EtherCAT devices on the PR2. It is possible to run some of these programs with sudo. However, sudo will not because of how ROS sets up the library search path. Also sudo provides more privileges than the programs really need.

Luckily the is a utility known as pr2_grant. Like sudo, pr2-grant will run another program with elevated privileges. Unlike sudo, pr2-grant will only provides certain privileges. pr2-grant also works with ROS programs.

pr2-grant installed on the PR2 by default. However, for desktop systems you may will probability need to download and compile it yourself. Check out the newest code for pr2-grant from the following location:

```
svn co https://code.ros.org/svn/pr2debs/trunk/packages/pr2/pr2-grant
```

Compile the code with make.

```
cd pr2-grant
make
```

pr2-grant is easiest to use when its put on the system path:

```
sudo cp pr2-grant /usr/local/bin
```

For pr2-grant to work, it needs to have the SUID bit set:

```
sudo chown root:root /usr/local/bin/pr2-grant
sudo chmod og+s /usr/local/bin/pr2-grant
```

Glossary

MCB : Motor Control Board.

WG006 / WG06 : Motor Control Board used in PR2 Gripper.

EtherCAT : A Real-time Communication Protocol use to communicate with PR2 actuators.

EtherCAT Master : Computer + Software the controls EtherCAT devices.

pr2_etherCAT/realtime_loop : Real-time control application for PR2. Consists of an EtherCAT Master + EtherCAT Device Plugins + Controller Plugins.

PicoBlaze : 8-bit soft-processor. Designed especially for light processing in an FPGA.

picoasm : Assembler for PicoBlaze processor

SPI : Serial Peripheral Interface. A simple synchronous serial interface used by many devices.

ESD : Electro-Static-Discharge.

Willow Garage Inc.
68 Willow Garage
Menlo Park, CA 94025

www.willowgarage.com