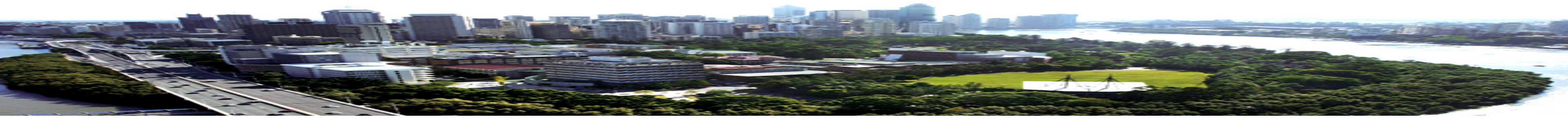
ENB 350 Real-time Computer based systems

Lecture 6 – Interrupt driven I/O

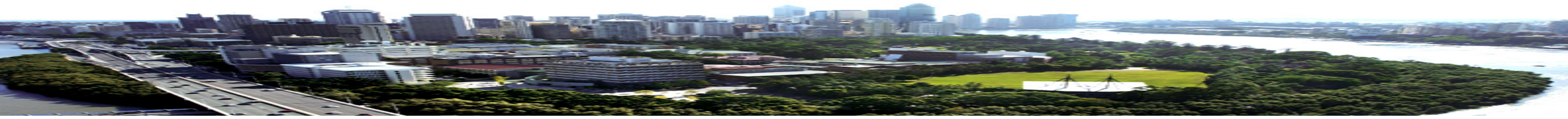
Analog I/O

V. Chandran



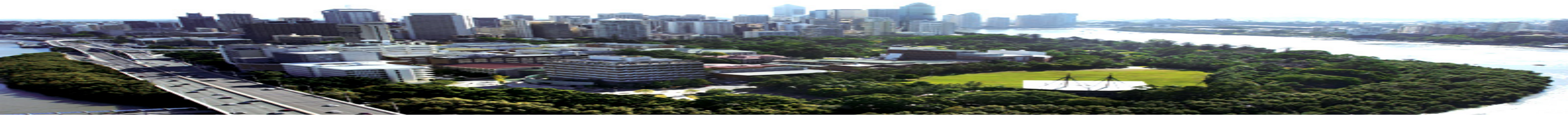
Contents

- Interrupt driven I/O
- Interrupt service routines
- Serial I/O – interrupt driven
- Demonstration
- Analog I/O
- Analog to Digital conversion
- Demonstration



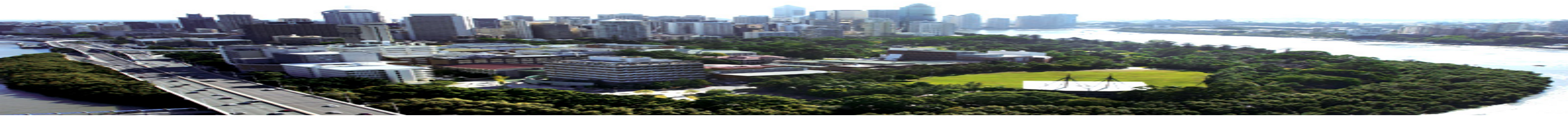
Interrupts = ?

- A voltage level held for a period of time or an edge, that is asserted at an interrupt request pin
- It causes the CPU to be 'interrupted' from its normal fetch and execute sequence
- CPU finishes the current instruction and executes an interrupt acknowledge cycle which may result in a jump to a routine
- Some Rabbit instructions are chained atomic and cannot be interrupted in between either
- The above description is for hardware interrupts. They are asynchronous.
- Software interrupts are different – arising from processor exceptions or a special instruction



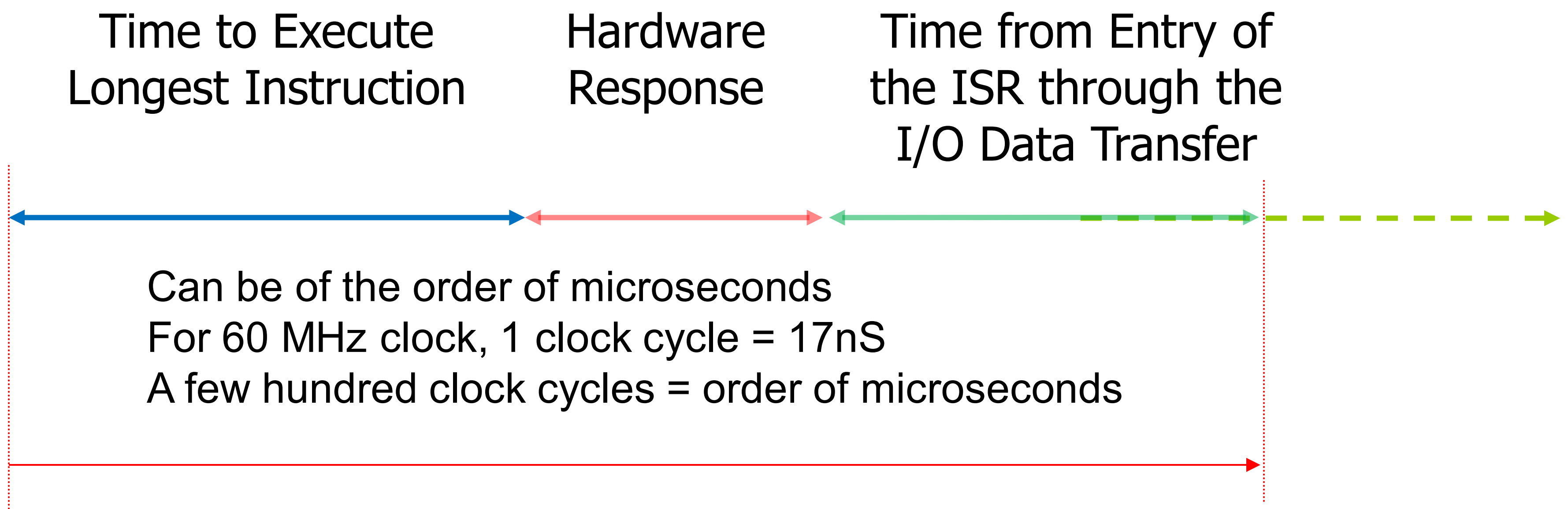
Interrupt-driven i/o

- When the device is ready to send/receive data it interrupts the CPU
- CPU responds after the present instruction is complete and jumps to an interrupt service routine (ISR)
- For real-time response this is good
- Data transfer is done (after some housekeeping) by the ISR



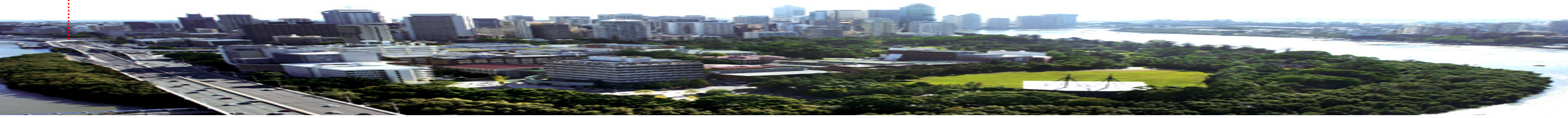
Interrupt latency

Time from occurrence of the interrupt to data transfer by the ISR.


Time to Execute Longest Instruction
Hardware Response
Time from Entry of the ISR through the I/O Data Transfer

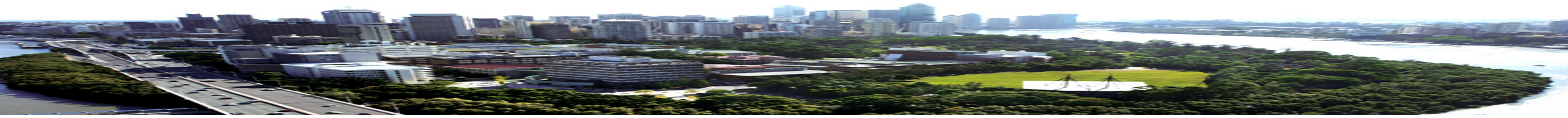
Can be of the order of microseconds
For 60 MHz clock, 1 clock cycle = 17nS
A few hundred clock cycles = order of microseconds

Note: data transfer rate is decided by time to execute the complete ISR



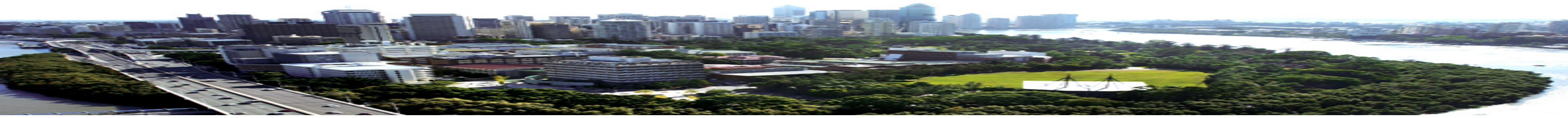
Interrupt latency

- Need to minimize this for hard real-time systems
- If the interrupt is not being blocked, the max latency for Rabbit is 29 clock cycles .
- Is non-deterministic and depends on
  - The length of the instruction currently executing
  - Other higher priority interrupts pending
- Can be reduced by
  - Prioritizing and assigning a high priority
  - Not inhibiting any interrupts



Interrupt processing

- When an interrupt is asserted the CPU finishes the current instruction (fetch-execute) and starts executing an interrupt cycle
- It may read/use an interrupt vector number and jumps to the address of an interrupt service routine at a given offset within a table of addresses of such routines
- The interrupt service routine clears the interrupt and handles the event and returns
- Multiple interrupts must be arbitrated using an interrupt controller. An interrupt mask is used to selectively allow and disallow particular types of interrupts. The interrupt controller allows prioritization.



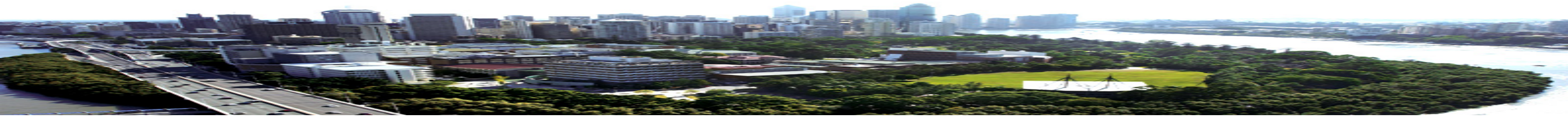
Rabbit interrupts

- Internal and external interrupts – corresponding interrupt registers are IIR and EIR. They store upper bytes of addresses of interrupt vector tables but are not used in current versions of Dynamic C.

- Macros for vector offsets are defined in sysio.lib. For serial port C, for example, SERC_OFS. The base address is XINTVEC_BASE.

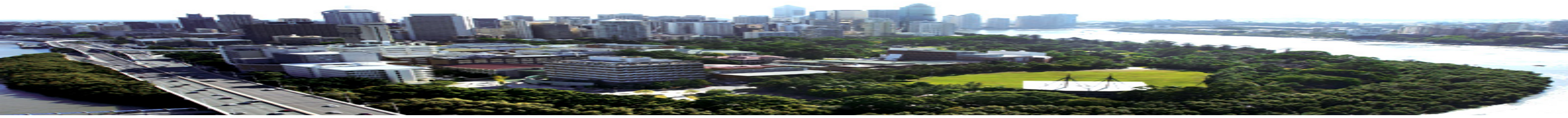
- At the address XINTVEC_BASE + SER_OFS there is the jump instruction code and address of the ISR

- Chapter 6,7 in R4000 hardware manual and Section 12.7 in the Dynamic C reference manual.
- This is not necessary for the lab exercise – see example code given later instead.



Rabbit interrupts

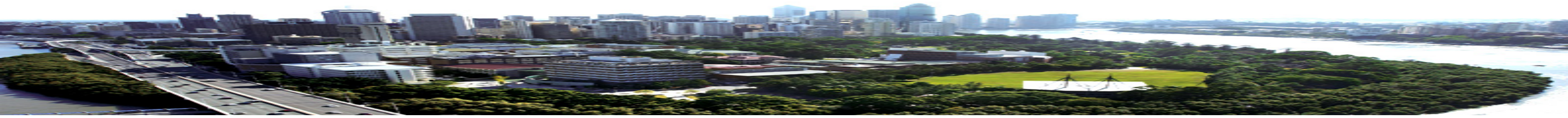
- User written ISRs – addresses are put in the table using SetVectIntern (or SetVectExtern for external interrupts).
- For example, SetVectIntern(0x0B, timerb_isr);
- Each vector begins on a 16 byte boundary – possible to hold a small routine but typical to place a call to an ISR
- Vector table structure is given in Table 6-1 of the R4000 manual. The offsets are given here. Doing CTRL-H after highlighting SetVectIntern will show vector numbers and offsets as well.



Rabbit interrupt priorities

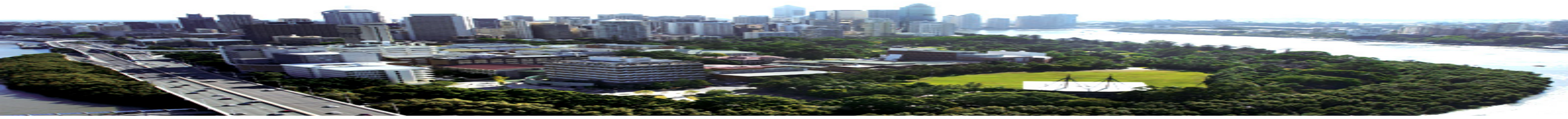
- There are 4 priority levels for the processor and for interrupts. A pending interrupt is handled only if its priority is greater than the current processor priority.
- The 8 bit IP register holds the current priority and three previous values (as a stack).
- IPSET n will place the value of n in the last 2 bits after shifting 2 bits left. (n=0,1,2, or 3)
- IPRES shifts 2 bits right
- Interrupt priorities for different types of interrupt are given in table 6-3 of the R4000 manual.

- Priority 0 is the standard operating level – where all interrupts are handled



Interrupt Service Routines

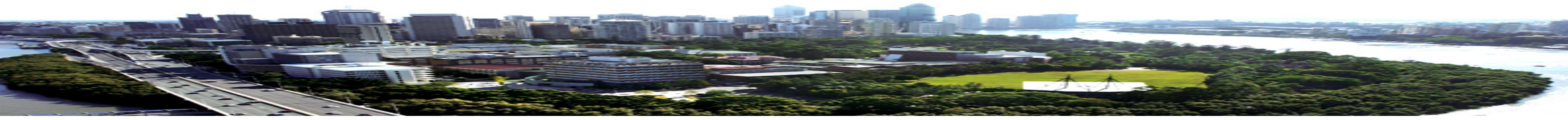
- Upon entry, save all registers that will be used by pushing to the stack. ISRs written in C save automatically. Assembly ISRs must do this explicitly.
- When entering an ISR, mask interrupts of the same or lower priority.
- Determine the cause of the interrupt by interrogating relevant chip registers. For example, serial ISRs must determine whether the interrupt was caused by read buffer full or transmit buffer empty.
- Remove the cause of the interrupt to prevent repeated triggering. This is done by clearing an interrupt flag in the appropriate register.
- On exit, restore the priority (IP) level.
- Re-enable interrupts that were disabled.
- After executing the body of the ISR, restore registers that were saved earlier.



Rabbit Serial I/O interrupt – port set up

```
nodebug void setupSerialC(long bitrate)
{
long divisor;
// set bit rate divisors.  (divisor+1) for 19200 baud is stored in BIOS variable "freq_divider“  (see POLLED)
// use parallel port C for serial port C I/O
#asm
// parallel port C needs to be set up for serial ports C, D
// this can be done elsewhere
ld     a,(PCFRShadow)
    set  CDRIVE_TXD, a
    ld     (PCFRShadow),a
ioi ld     (PCFR),a
    ld a, 0x01;                  // THIS IS DIFFERENT FROM  THE POLLED EXAMPLE IN LECTURE 5
ld     (SCCRShadow),a
ioi ld     (SCCR),a       ; enable interrupts, use parallel port C for serial port C
#endasm

// WrPortI (SCCR, &SCCRShadow, 0x01);          // asynch, 8N1, int enabled  ... Can do this instead.

// bind serial port C ISR to associated interrupt vector                        (AND THIS IS ALSO DIFFERENT)
      SetVectIntern(serial_C_int_num, serC_ISR);                                // assuming same INSTR and DATA space
} // setupSerialC
```

Serial_C_int_num is a macro defined to be the interrupt number for serial port C
serC_ISR is the address (label) of the ISR.



CRICOS No. 00213J

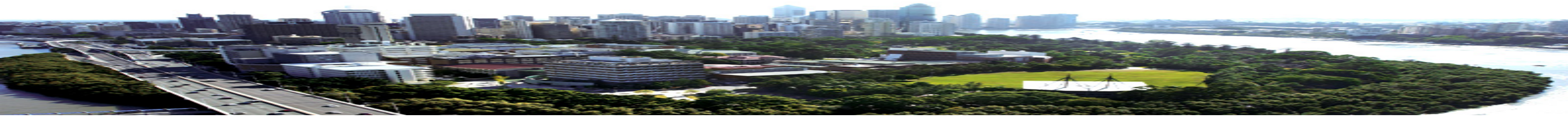
Rabbit serial I/O interrupt - ISR

```
#asm nodebug root
serC_ISR::
        push    af                      ; save used registers
        push    hl
        ioi     ld a, (SCSR)            ; get status register value. Bit 7 is 1 for Rx Buffer full
        rla                             ; byte received? bit 7 goes to carry which is checked
        jr      nc, SerC_TX             ; jump if no carry –  it is a TX interrupt then

// Serial ISR Receive Routine
SerC_RX::
        ioi     ld a, (SCDR)            ; read the byte and clear the interrupt
        //In this example we are making a change to the character code (adding 10) and
        // sending it back.  In practice for seriall communication, the byte received will be
       // transferred to a circular buffer
                add a,10
        ioi     ld (SCDR), a            ; send the byte back to echo it
        jr      ISR_exit

// Serial ISR Transmit Routine
SerC_TX::
        ioi     ld (SCSR), a            ; clear the TX interrupt
ISR_exit:
        pop     hl                      ; restore used registers
        pop     af
        ipres                           ; restore interrupt priority
        ret
#endasm
```

If interrupt was not caused by RxB full, it must be TxB empty

Reading the byte also clears the interrupt

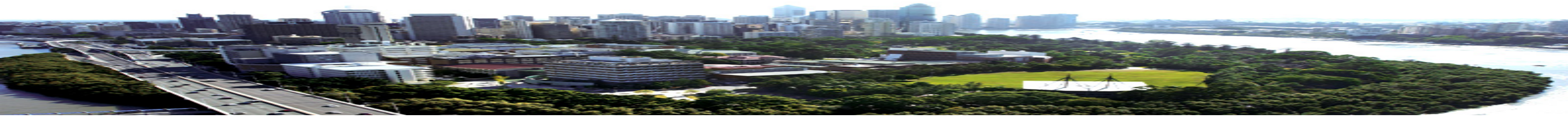
Tx interrupt is cleared by attempting to write to the status register (also possible by writing to SCDR, SCAR)



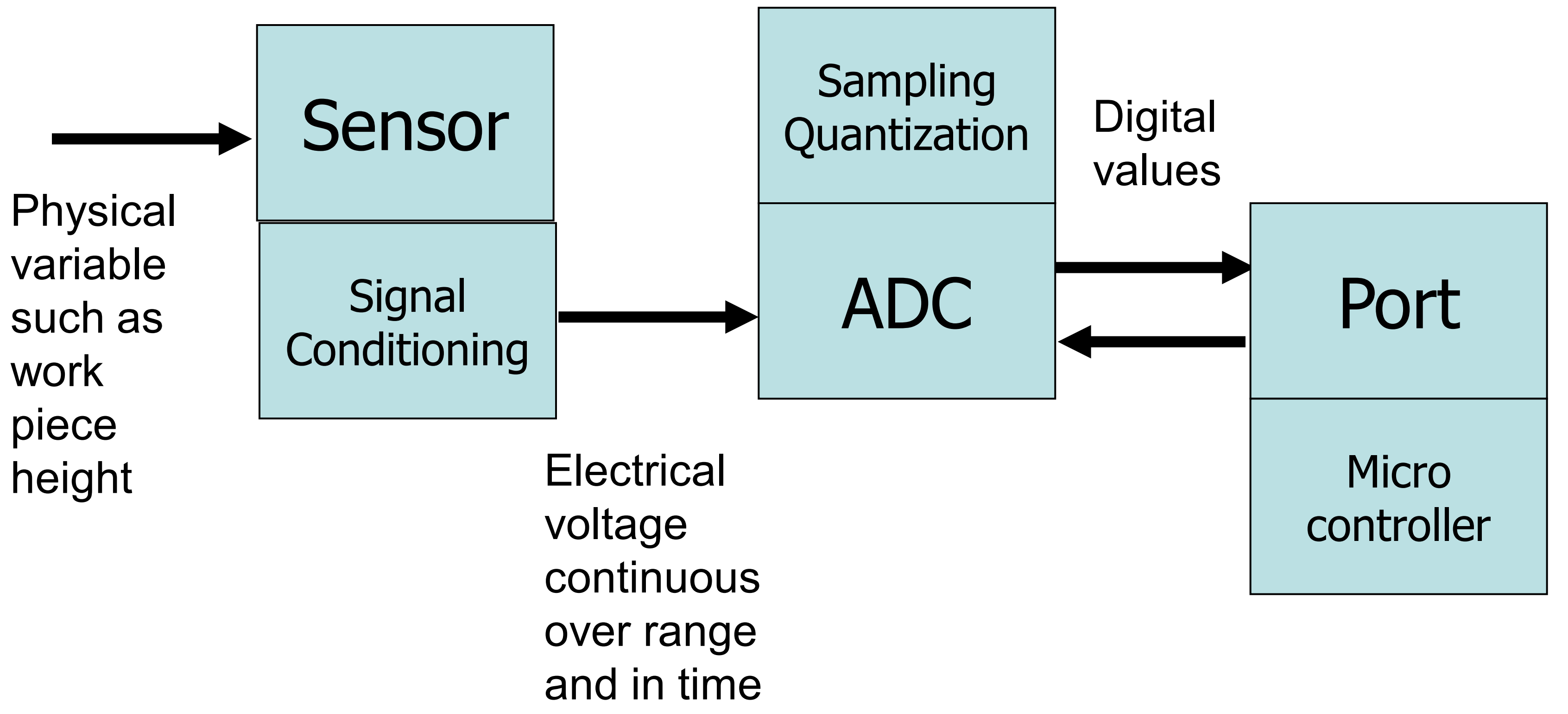
Demonstration

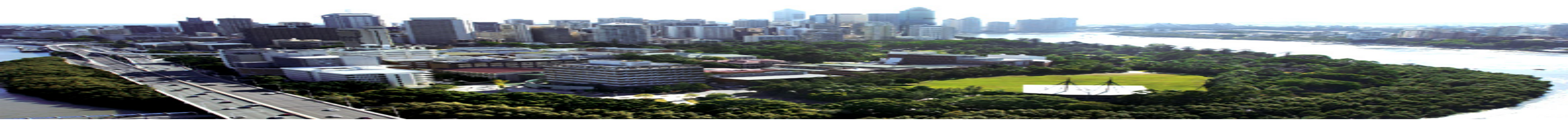
- Interrupt driven i/o

A program that receives characters through serial port C adds 10 to each character code and sends it back. Serial communication is interrupt driven. Serial C is connected to a PC USB to RS232 convertor and the PC is running HyperTerm. Further, each time the ISR is entered an LED on the board turns ON and OFF.



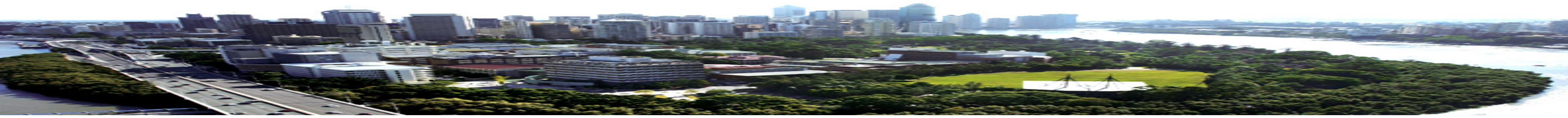
Analog Interfacing


Sensor
Signal Conditioning
Physical variable such as work piece height
Electrical voltage continuous over range and in time
Sampling Quantization
ADC
Digital values
Port
Micro controller

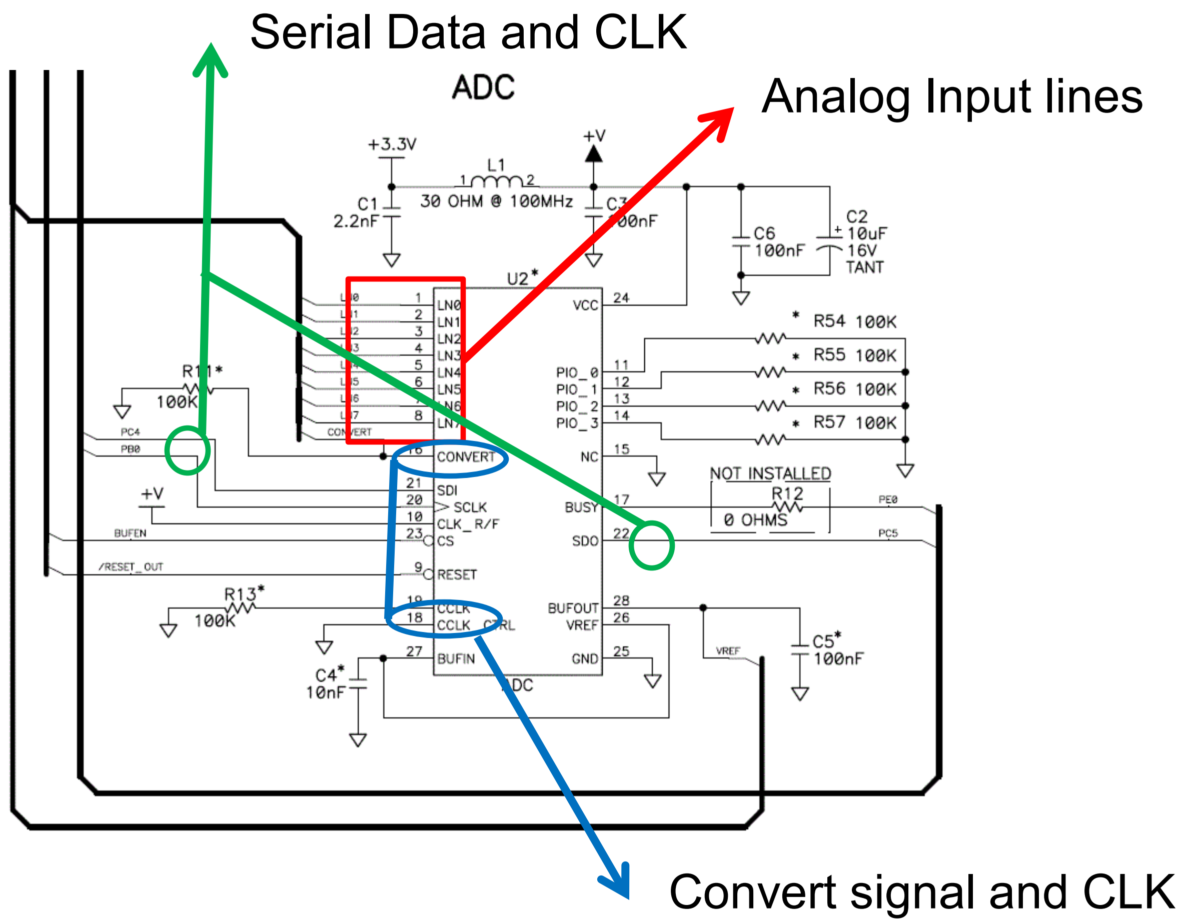


RCM4000 support for ADC

- On-board ADC with 8 input channels, 12 bit (ADS7870)
- 11 bit quantization - range 0 to 2047 (for single ended)
- single ended and differential input configurations
- LN0-LN6 available, LN7 is used as a thermistor input
- Internal reference voltage can be set to 1.15, 2.048 or 2.5 V. Can sample the reference voltage



ADC on the RCM4000



ADC conversion start

A conversion is started by an active (rising) edge on the CONVERT pin. The CONVERT pin must stay low for at least two CCLK periods before going high for at least two CCLK periods. Figure 12 shows the timing of a conversion start. The double falling arrow on CCLK indicates the actual start of the conversion cycle.

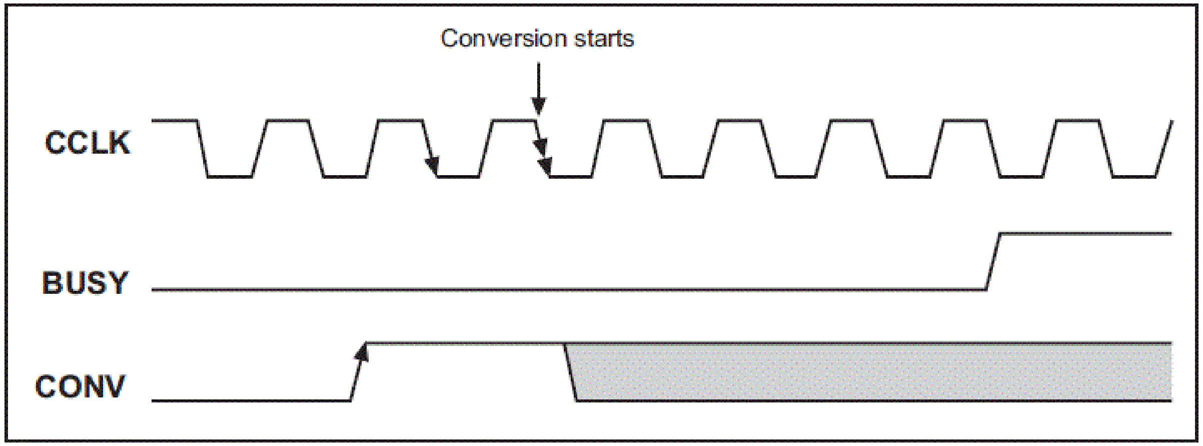
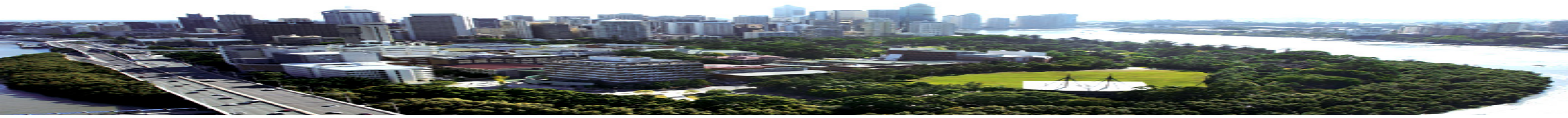
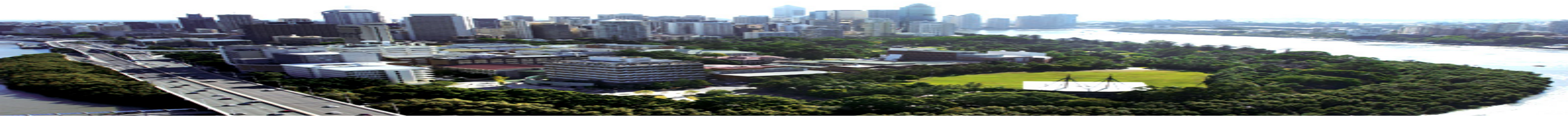


Figure 12. Timing Diagram for Conversion Start Using CONVERT Pin



RCM4000 support for ADC

- ADC uses serial port B and multiplexes the outputs.
- With the use of library functions such as anaIn() this is hidden from the user.
- Input voltage 0-2V or -2 to 2V (differential). Using resistors on the motherboard this is extended to 22.2V by the use of voltage division. The max input can be scaled to a number of values and the ADC gain is also correspondingly software programmable.
- Note: LN0 is connected to the Festo interface connector on the box as an analog input and the height sensor output is connected to a corresponding pin at the testing station end.



Input attenuation

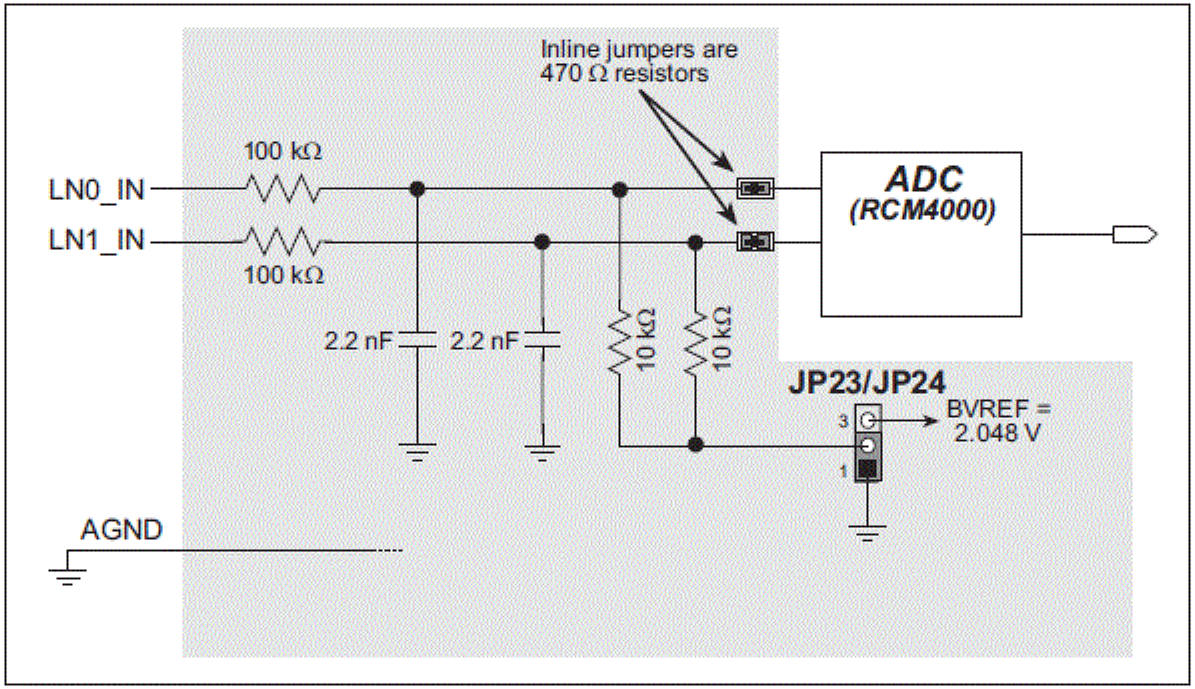


Figure B-6. A/D Converter Inputs

Max voltage and Gain

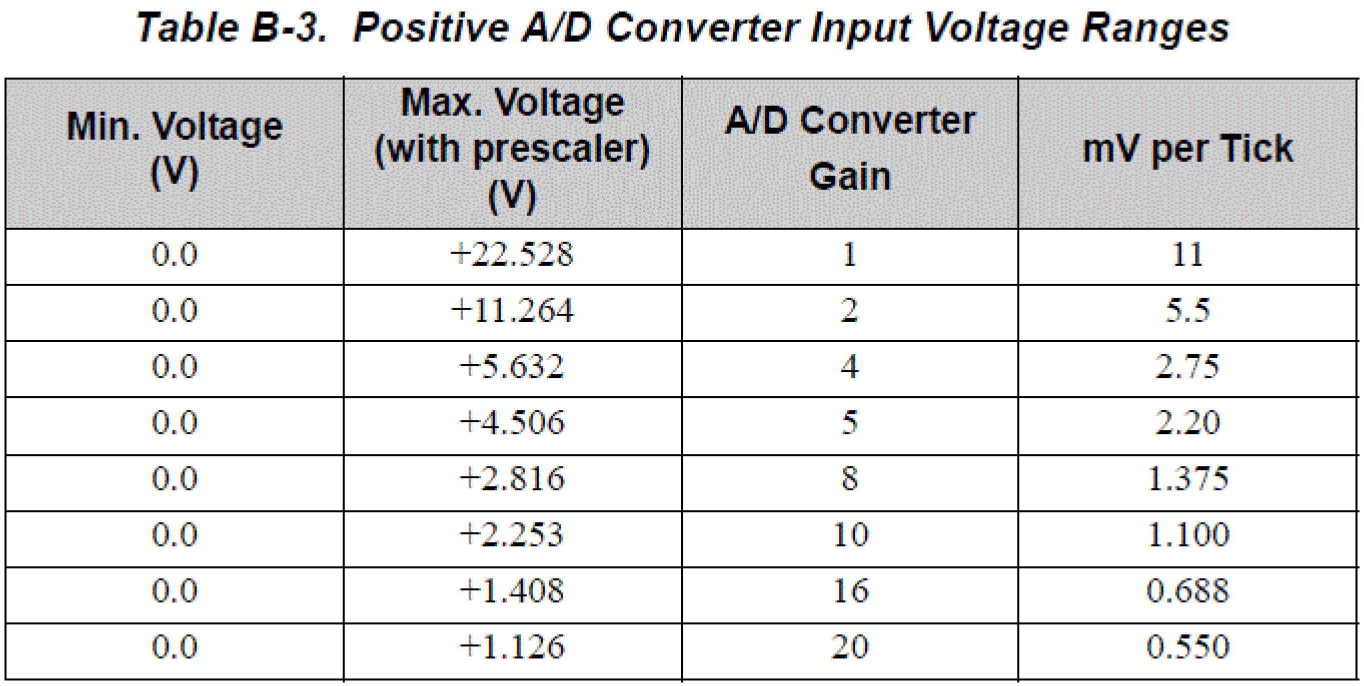
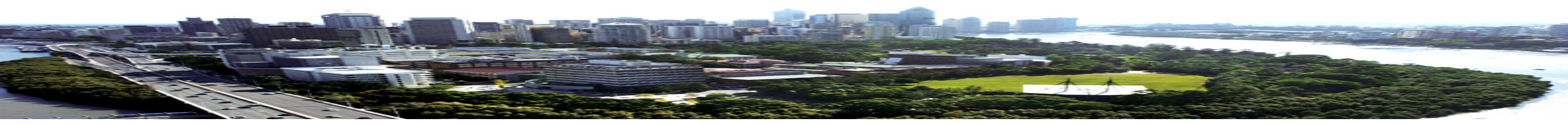


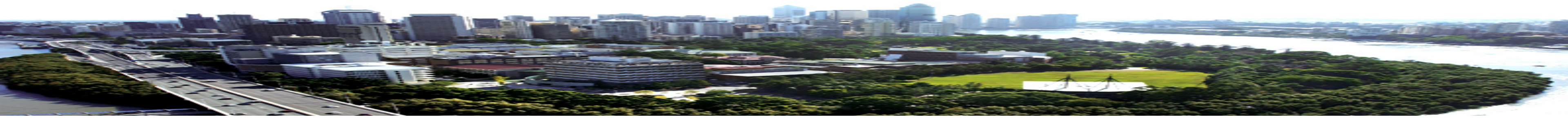
Table B-3. Positive A/D Converter Input Voltage Ranges

Min. Voltage (V)	Max. Voltage (with prescaler) (V)	A/D Converter Gain	mV per Tick
0.0	+22.528	1	11
0.0	+11.264	2	5.5
0.0	+5.632	4	2.75
0.0	+4.506	5	2.20
0.0	+2.816	8	1.375
0.0	+2.253	10	1.100
0.0	+1.408	16	0.688
0.0	+1.126	20	0.550



Example ADC calculations

- 11 bits -> 0 to 2047. Single ended inputs
- ADC accepts 0-22.528V.
- Each quantization step is 22.528/2047 = approx. 11mV
- Gain control of the ADC rescales the input.
- If the input is 0 to 5V, gain should be set to 4. From the table, the max voltage is 5.632 V. Each step or tick (or increment with the LSB) is then 2.75 mV.
- Need to calibrate using reference measurements.

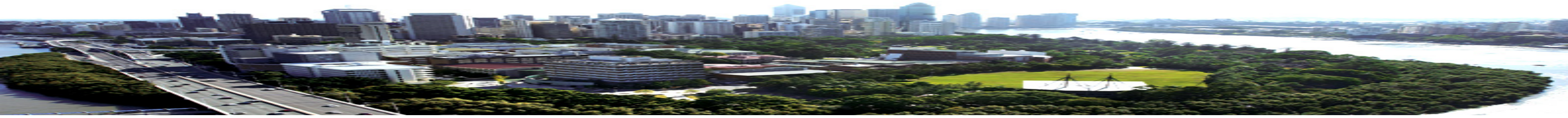


Example ADC calculations

- Assume a precision reference voltage is 3.3V.
- If this is measured with gain 4, it should read 3.3/0.00275 = 1200 on the scale from 0 to 2047. If not there is an offset. Say reading is 1192. Then the offset is -8 levels or -22mV. Any reading X in the range 0 to 2047 should be converted to a true voltage according to

  Y = X *2.75 mv + 22mV   OR   Y = (X+8)*2.75mV

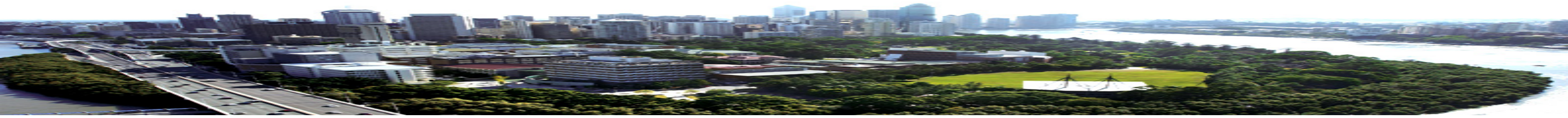
- With two reference voltages, we can measure offset and linear slope in that range. This is more accurate than assuming 2.75mV/step as true over the entire range. Calibration constants can be saved.
- The standard deviation of the precision voltage measurement provides a measure of the channel noise. A measurement will not be accurate to the full 11 bits of quantization when there is noise.



RCM4000 Analog I/O software

- Sample programs
  - ADC_SAMPLE.C (edited version used for demonstration)
    - Demonstrates reading an analog voltage, calculating the average over a number of samples, converting to a scaled value and displaying
  - AD_CAL_CHAN.C etc. (edited version used for demonstration)
    - Demonstrates how to recalibrate a single ended channel with one gain using two known voltages

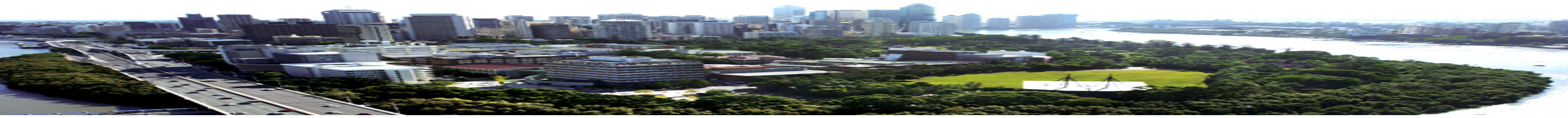
- There are others but these are not necessary for the assignment



RCM4000 analog I/O example usage

```
#use RCM40xx.LIB

#define ADC_SCLKBAUD 115200ul

#define GAINSET 2           // gain code set to 2
channel = 0;                // channel 0

//convert channel and gain to ADS7870 format
// in a direct mode
    cmd = 0x80|(GAINSET*16+(channel | 0x08));

// execute low level A/D driver routine to read data
    rawdata = (long) anaInDriver(cmd);

// remove offset and scale
    newval = (rawdata - _adcCalibS[channel][GAINSET].offset) *
             (_adcCalibS[channel][GAINSET].kconst);
```

This assumes that you have previously run the calibration program to calibrate the channel. Note that here the GAINSET is a code taking values 0 to 7



Demonstration

- Analog input and ADC
  - Reading values
  - Calibrating the channel