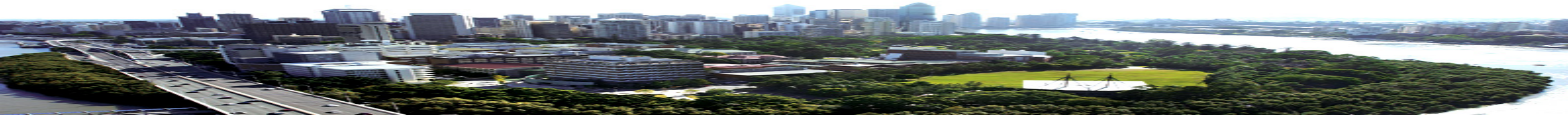
ENB 350 Real-time Computer based systems

Lecture 5 – SERIAL COMMUNICATION

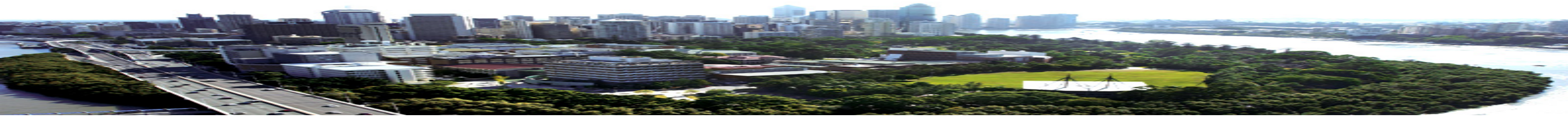
V. Chandran



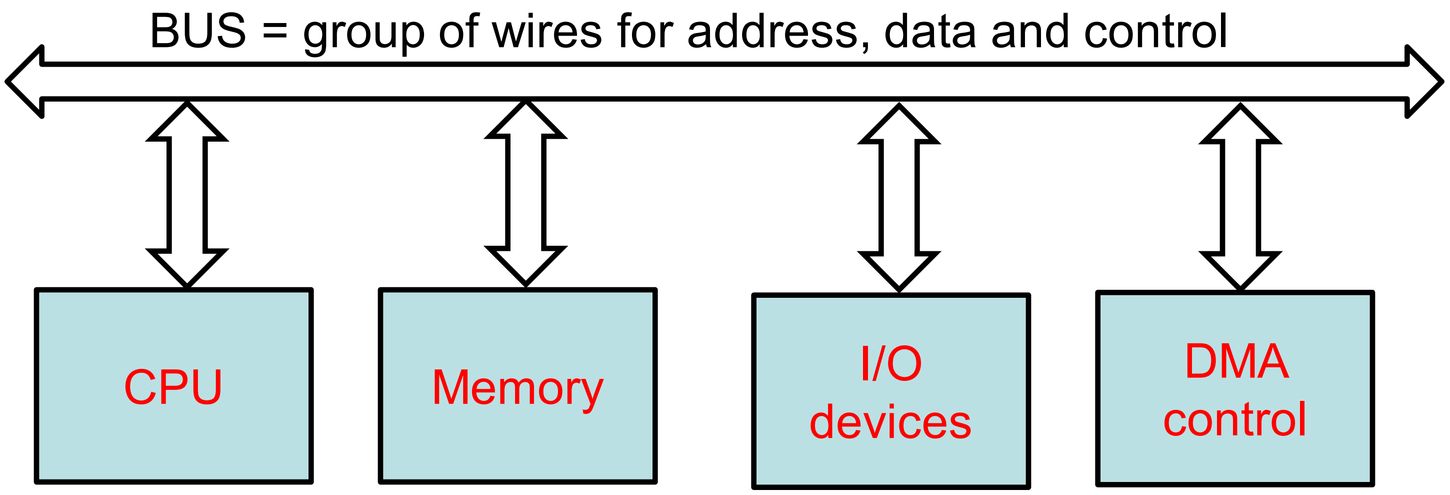
Contents

- Types of I/O
- Serial I/O
  - USART
  - Low level port set up details
  - Polled mode transfer
- RS232 library functions for serial I/O
- Demonstration

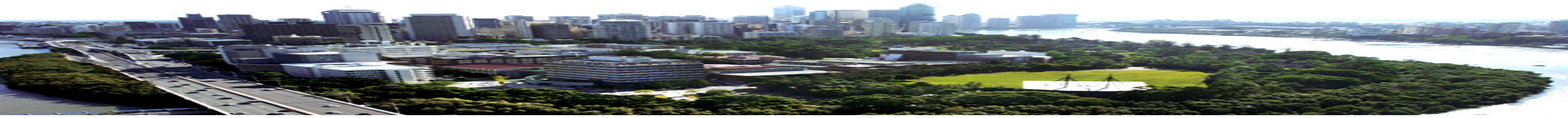
- Memory organization
  - Logical and physical addresses
  - Logical memory segments – root, data, stack and XPC
  - root and extended memory



Typical embedded system data transfer configuration



If I/O devices appear as memory locations and use the same bus, it is memory mapped I/O. (as shown above)
If they use a separate bus and there are separate instructions for I/O, it is I/O mapped I/O



Types of I/O

By bits transferred

Parallel

Serial

The time at which data from an i/o device is ready is not under CPU control. There is a need for synchronization.

(at bit level – local independent CLK)

asynchronous

Clocked

(at bit level – using a CLK line)

By synchronization method (at byte or data frame level)

Polled

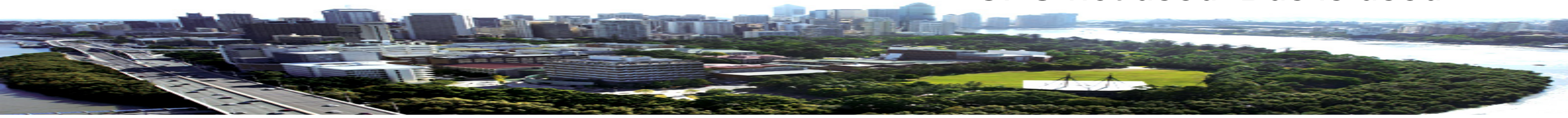
Interrupt driven

Direct Memory Access

CPU checks status and waits

Device interrupts the CPU. ISR transfers data

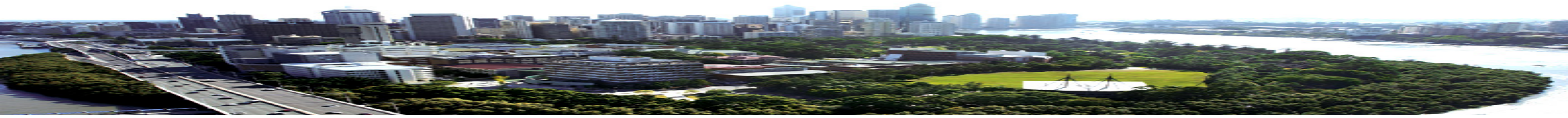
DMA controller transfers directly to/from memory, CPU not used. Bus is used.



CRICOS No. 00213J

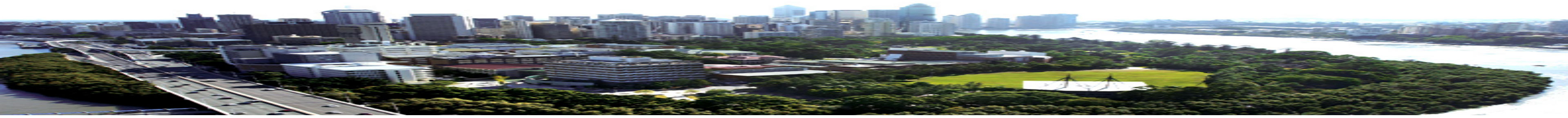
Parallel vs Serial

- Parallel
  - Fast
  - Requires 1 line for each bit
  - Expensive over large distances
- Serial
  - Requires fewer lines (minimum Tx, Rx, Gnd)
  - Economical over large distances
  - Requires parallel to serial conversion (shift registers and clocked transfer of bits)
  - uses a universal synchronous/asynchronous receiver transmitter (UART/USART)



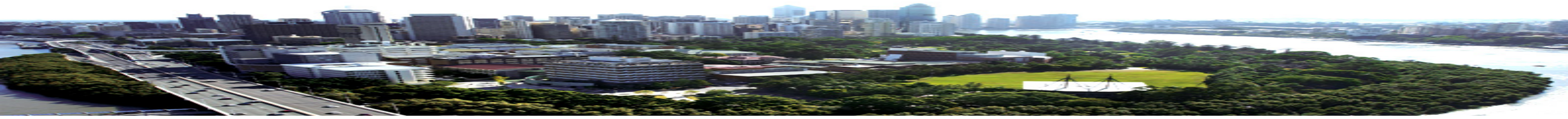
Transfer rate and Latency

- Transfer rate = Number of bytes of data transferred per unit of time
- Latency = time delay from the instant a device is ready to the time the first data byte is transferred.
- DMA is the fastest and has the lowest latency
- Polled waiting loops are easy to implement and have good transfer rate – but have unpredictable latency
- Interrupt driven I/O is slow if only one byte is transferred at a time – but has low latency



Polling

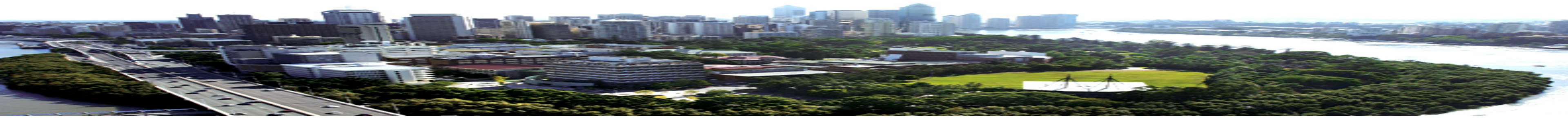
- Application code samples a status register and executes specific code to send or receive data based on the status indicating that the device is ready.
- The application has to constantly check the status of the device and can be wasteful of CPU
- Polling happens under program control
- If device speed and data transfer rate are closely matched for high speed, polling can be fast.



UART

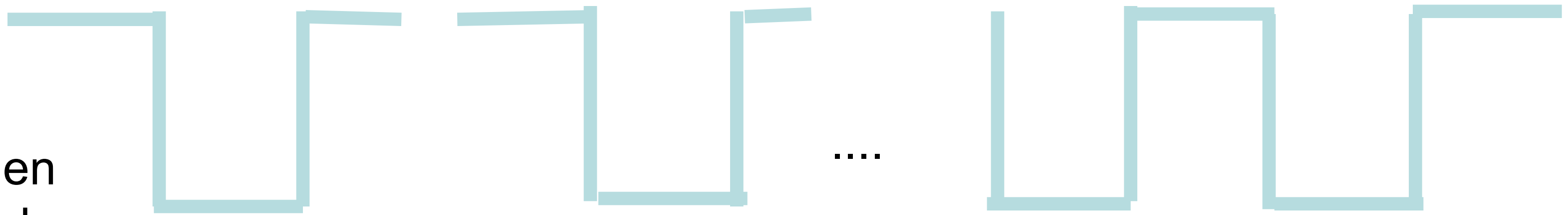
- Universal (Synchronous) Asynchronous Receiver Transmitter

- Has internal registers to shift input bits in and output bits out
- Data register, Status register and Control register
- Uses a local clock in asynchronous mode
- Can generate an interrupt after a receive or transmit operation is complete to alert the CPU
- Can use parity bits for error detection
- Frame format (start bit, data bits, parity bits, stop bit) can be set
- Clock (baud) rate can be set
- May optionally have buffers to increase transmission speed



Asynchronous serial data format

If idle high, then asserted low to mark.



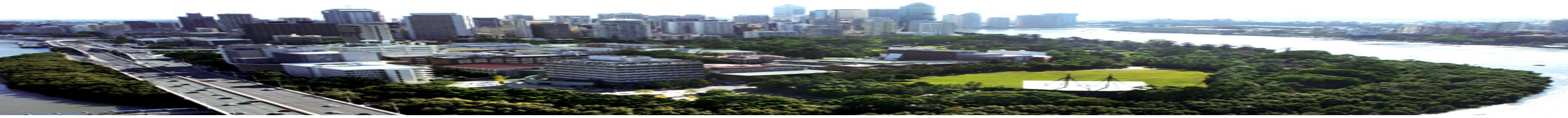
Start bit
Marks the beginning
Transition from idle

Data bits 7 or 8
1s and 0s.

Parity bit
optional

Stop bit(s)
optional

Bits following the start are latched and shifted in with a local clock. Set to a baud rate (bits/sec)

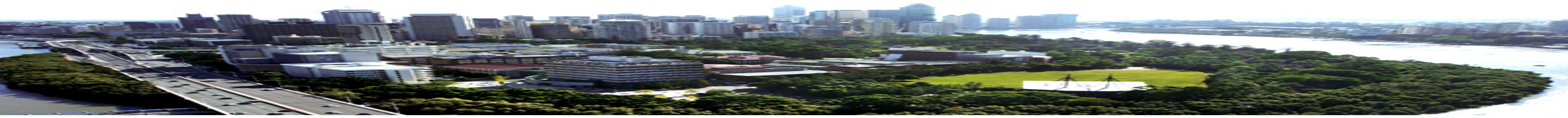


UART receive

- Sense input line (Rx, Gnd)
- Detect change from idle and start bit
- Latch input data bits into data register
- Set flag bit for receive buffer full in the status register

- A polling device will check this bit continuously reading the port

- An interrupt is generated and the interrupt service routine can check this bit to confirm that it was a receive interrupt and not a transmit one.

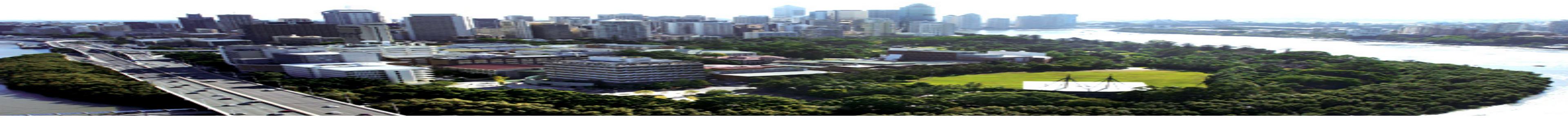


UART transmit

- Force output line to start bit
- Force output line to data bits
- Force output line to parity and stop bit states if necessary
- Set the transmit buffer empty flag in the status register
- Generate an interrupt if interrupt driven

- In polled transfer, the program can continuously check the flag to know when to put the next output byte.

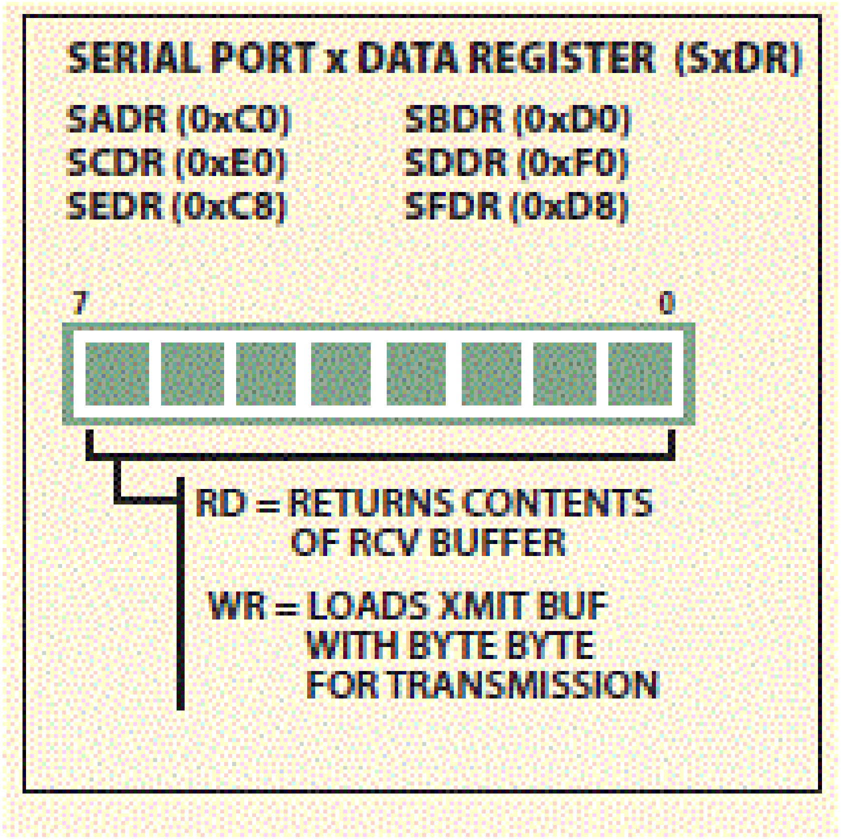
- In interrupt driven transfer, an interrupt service routine can fetch the next byte from the transmit (circular) buffer to the UART transmit register



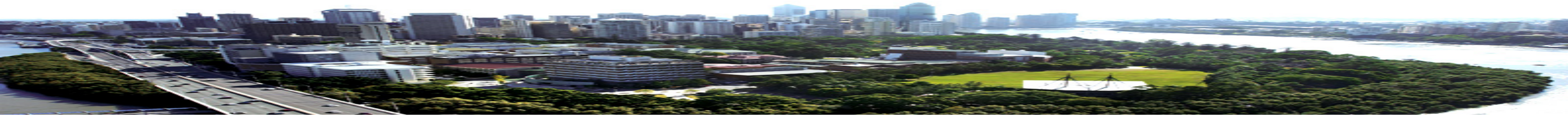
R4000 Serial – Data register

What is the C macro for serial port C data register?

What is the HEX address?



SERIAL PORT x DATA REGISTER (SxDR)
SADR (0xC0)
SBDR (0xD0)
SCDR (0xE0)
SDDR (0xF0)
SEDR (0xC8)
SFDR (0xD8)
7
0
RD = RETURNS CONTENTS OF RCV BUFFER
WR = LOADS XMIT BUF WITH BYTE BYTE FOR TRANSMISSION

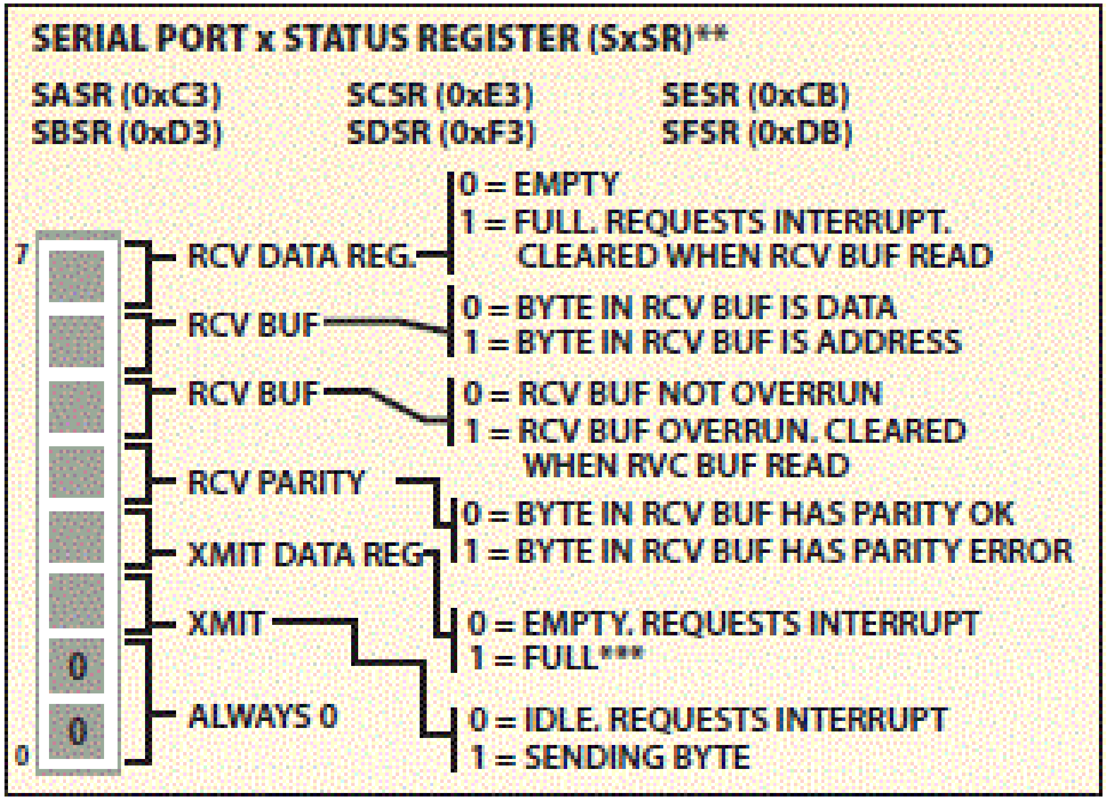


CRICOS No. 00213J

R4000 Serial – Status register

Which bit of SCSR shows that RCV BUF is full and data is ready to be read?

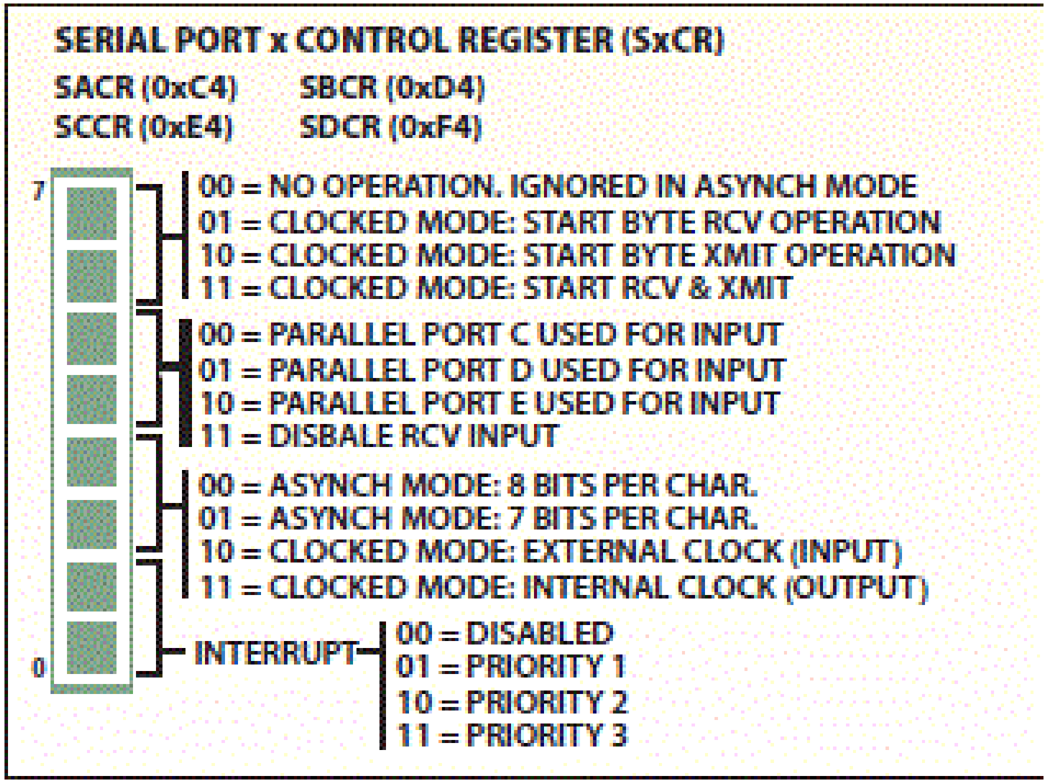
What happens when the data is read?

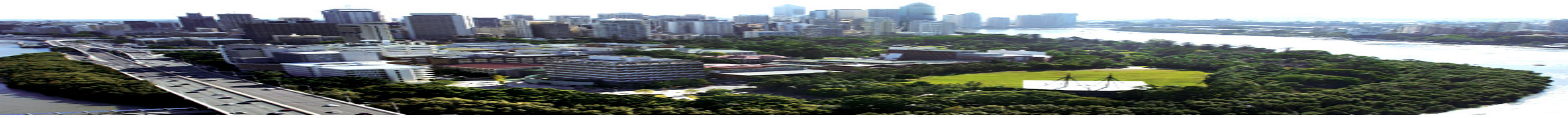

SERIAL PORT x STATUS REGISTER (SxSR)**
SASR (0xC3)
SCSR (0xE3)
SESR (0xCB)
SBSR (0xD3)
SDSR (0xF3)
SFSR (0xDB)
7
0
RCV DATA REG.
0 = EMPTY
1 = FULL. REQUESTS INTERRUPT. CLEARED WHEN RCV BUF READ
RCV BUF
0 = BYTE IN RCV BUF IS DATA
1 = BYTE IN RCV BUF IS ADDRESS
RCV BUF
0 = RCV BUF NOT OVERRUN
1 = RCV BUF OVERRUN. CLEARED WHEN RVC BUF READ
RCV PARITY
0 = BYTE IN RCV BUF HAS PARITY OK
1 = BYTE IN RCV BUF HAS PARITY ERROR
XMIT DATA REG
0 = EMPTY. REQUESTS INTERRUPT
1 = FULL***
XMIT
0 = IDLE. REQUESTS INTERRUPT
1 = SENDING BYTE
ALWAYS 0
0
0

Which bit of SCSR shows that XMIT BUF is empty and new data can be written?

R4000 Serial – Control Register

What should be written to SCCR to have asynchronous mode, with parallel port C used for input 8 bits per char and polled transfers?


SERIAL PORT x CONTROL REGISTER (SxCR)
SACR (0xC4)
SBCR (0xD4)
SCCR (0xE4)
SDCR (0xF4)
7
0
00 = NO OPERATION. IGNORED IN ASYNCH MODE
01 = CLOCKED MODE: START BYTE RCV OPERATION
10 = CLOCKED MODE: START BYTE XMIT OPERATION
11 = CLOCKED MODE: START RCV & XMIT
00 = PARALLEL PORT C USED FOR INPUT
01 = PARALLEL PORT D USED FOR INPUT
10 = PARALLEL PORT E USED FOR INPUT
11 = DISBALE RCV INPUT
00 = ASYNCH MODE: 8 BITS PER CHAR.
01 = ASYNCH MODE: 7 BITS PER CHAR.
10 = CLOCKED MODE: EXTERNAL CLOCK (INPUT)
11 = CLOCKED MODE: INTERNAL CLOCK (OUTPUT)
INTERRUPT
00 = DISABLED
01 = PRIORITY 1
10 = PRIORITY 2
11 = PRIORITY 3

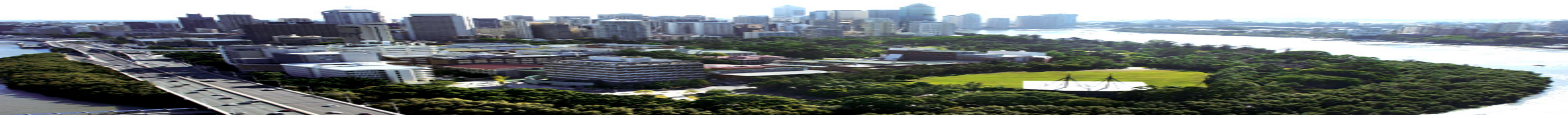


Rabbit serial ports - setting up

- UART needs to be locally clocked. The baud rate is set using Timer A registers and Timer A uses the peripheral clock. Timer A consists of 10 separate 8 bit down counters.

- Set the parallel port to be used, asynchronous or clocked, interrupts enabled or not, etc. using the serial port control register (SXCR)

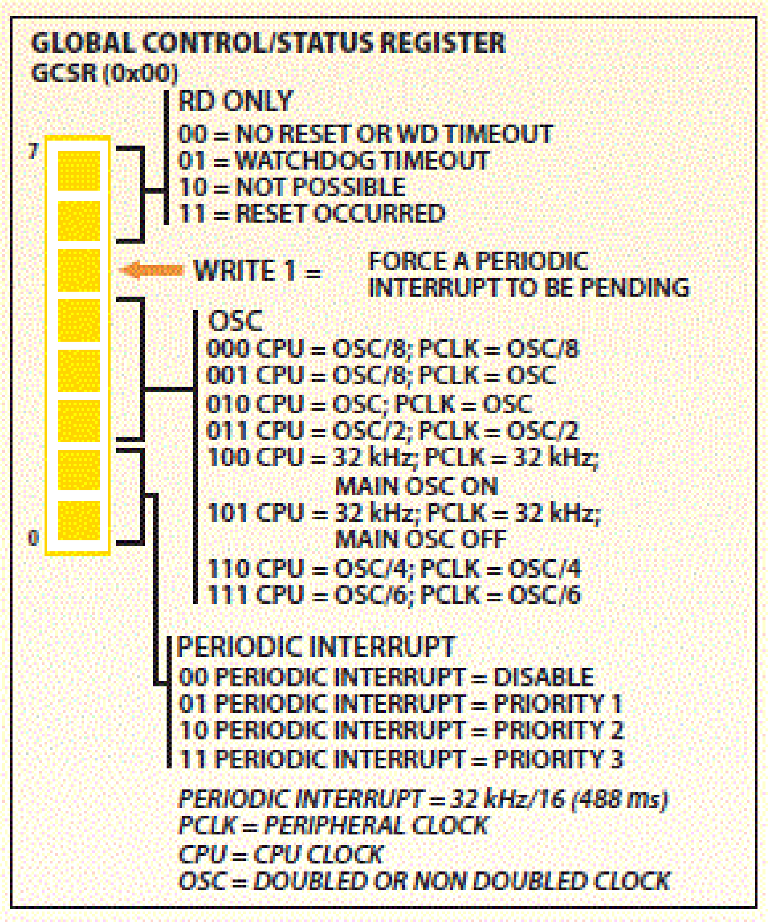
If we use a library such as RS232.LIB with Dynamic C, these are done using higher level functions. If we write custom assembly and C code for serial communication, the details are necessary.



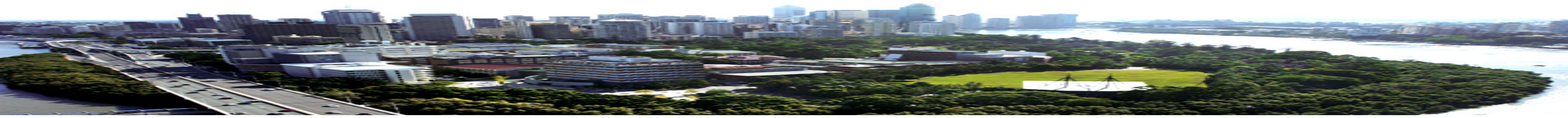
Global Control Status register

This register decides how the CPU clock and the peripheral clock are related to the main clock.

No need to go down to this level for lab exercises. Default settings will suffice.

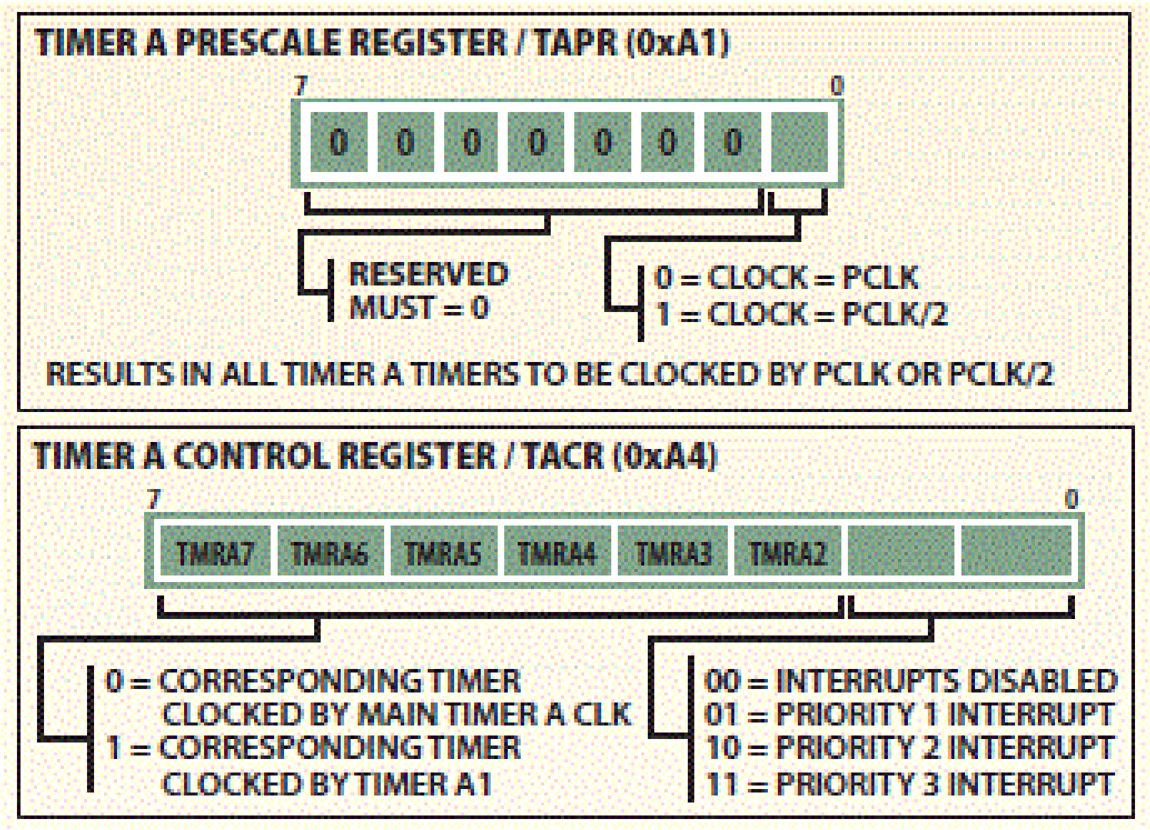


GLOBAL CONTROL/STATUS REGISTER
GCSR (0x00)
7
0
RD ONLY
00 = NO RESET OR WD TIMEOUT
01 = WATCHDOG TIMEOUT
10 = NOT POSSIBLE
11 = RESET OCCURRED
WRITE 1 = FORCE A PERIODIC INTERRUPT TO BE PENDING
OSC
000 CPU = OSC/8; PCLK = OSC/8
001 CPU = OSC/8; PCLK = OSC
010 CPU = OSC; PCLK = OSC
011 CPU = OSC/2; PCLK = OSC/2
100 CPU = 32 kHz; PCLK = 32 kHz; MAIN OSC ON
101 CPU = 32 kHz; PCLK = 32 kHz; MAIN OSC OFF
110 CPU = OSC/4; PCLK = OSC/4
111 CPU = OSC/6; PCLK = OSC/6
PERIODIC INTERRUPT
00 PERIODIC INTERRUPT = DISABLE
01 PERIODIC INTERRUPT = PRIORITY 1
10 PERIODIC INTERRUPT = PRIORITY 2
11 PERIODIC INTERRUPT = PRIORITY 3
PERIODIC INTERRUPT = 32 kHz/16 (488 ms)
PCLK = PERIPHERAL CLOCK
CPU = CPU CLOCK
OSC = DOUBLED OR NON DOUBLED CLOCK



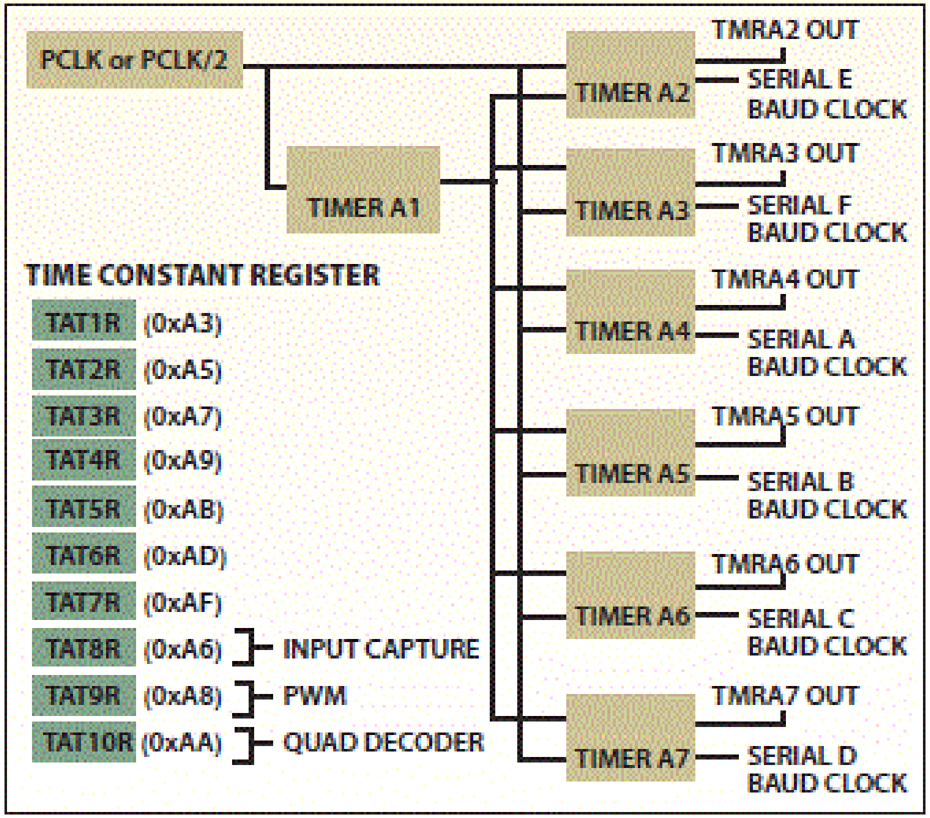
Timer A registers

Default is PCLK/2


TIMER A PRESCALE REGISTER / TAPR (0xA1)
7
0
0 0 0 0 0 0 0
RESERVED MUST = 0
0 = CLOCK = PCLK
1 = CLOCK = PCLK/2
RESULTS IN ALL TIMER A TIMERS TO BE CLOCKED BY PCLK OR PCLK/2
TIMER A CONTROL REGISTER / TACR (0xA4)
7
0
TMRA7 TMRA6 TMRA5 TMRA4 TMRA3 TMRA2
0 = CORRESPONDING TIMER CLOCKED BY MAIN TIMER A CLK
1 = CORRESPONDING TIMER CLOCKED BY TIMER A1
00 = INTERRUPTS DISABLED
01 = PRIORITY 1 INTERRUPT
10 = PRIORITY 2 INTERRUPT
11 = PRIORITY 3 INTERRUPT

Time Constant registers

Need to select the appropriate clock source. Serial C would be timer A6.



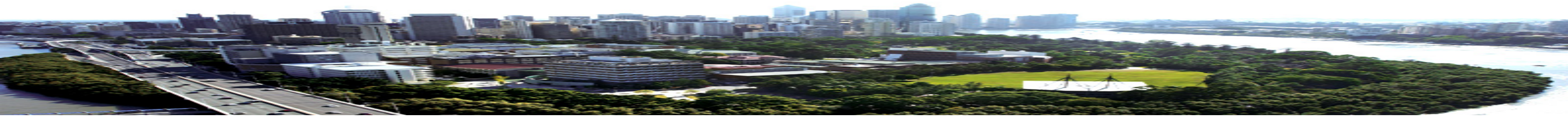
Time constant register TAT6R will then be set to the correct divisor value for desired bit rate

Bit rate - divisor

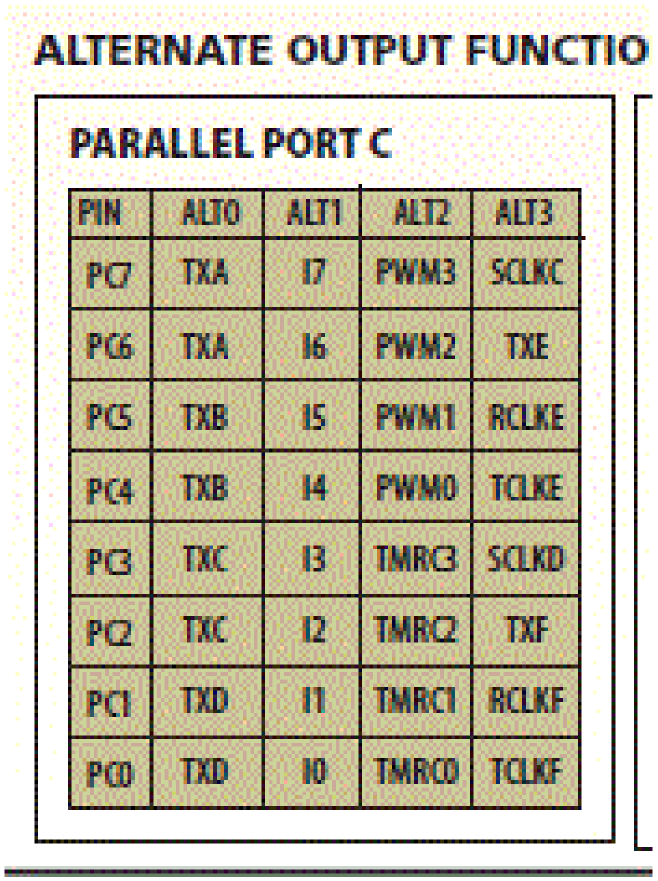
$$divisor = (long)(freq\_divider * 19200/(float)required\_bit\_rate + 0.5) - 1L$$

Variable freq_divider is a Dynamic C system variable.

Note: When using the RS232 library all these details are not necessary – they are hidden from the user who calls a function to set up the desired port with a desired bit rate. Example serCopen(19200L);



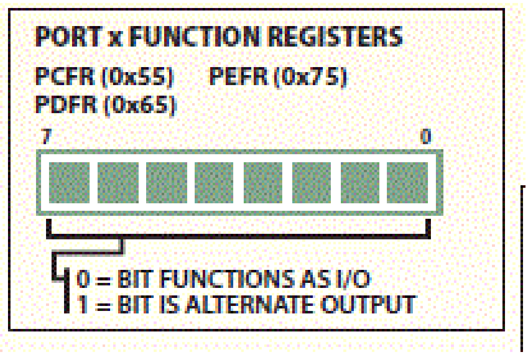
Port C Alternate functions



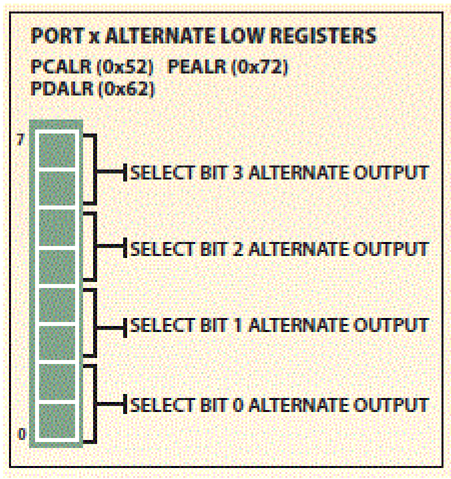
ALTERNATE OUTPUT FUNCTIO

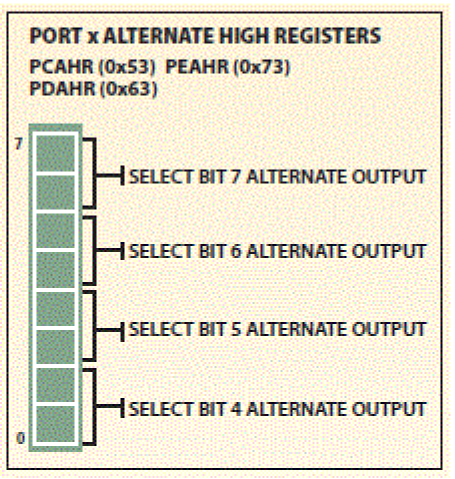
PARALLEL PORT C

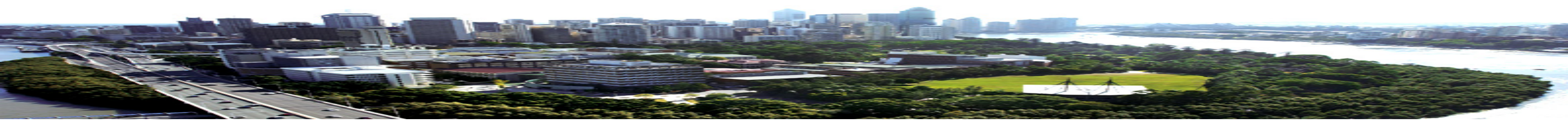
PIN	ALT0	ALT1	ALT2	ALT3
PC7	TXA	I7	PWM3	SCLKC
PC6	TXA	I6	PWM2	TXE
PC5	TXB	I5	PWM1	RCLKE
PC4	TXB	I4	PWM0	TCLKE
PC3	TXC	I3	TMRC3	SCLKD
PC2	TXC	I2	TMRC2	TXF
PC1	TXD	I1	TMRC1	RCLKF
PC0	TXD	I0	TMRC0	TCLKF



ALT0 mode configures pins 2 and 3 for serial port C, Set bit 2 for output (TX). Using PCFR.

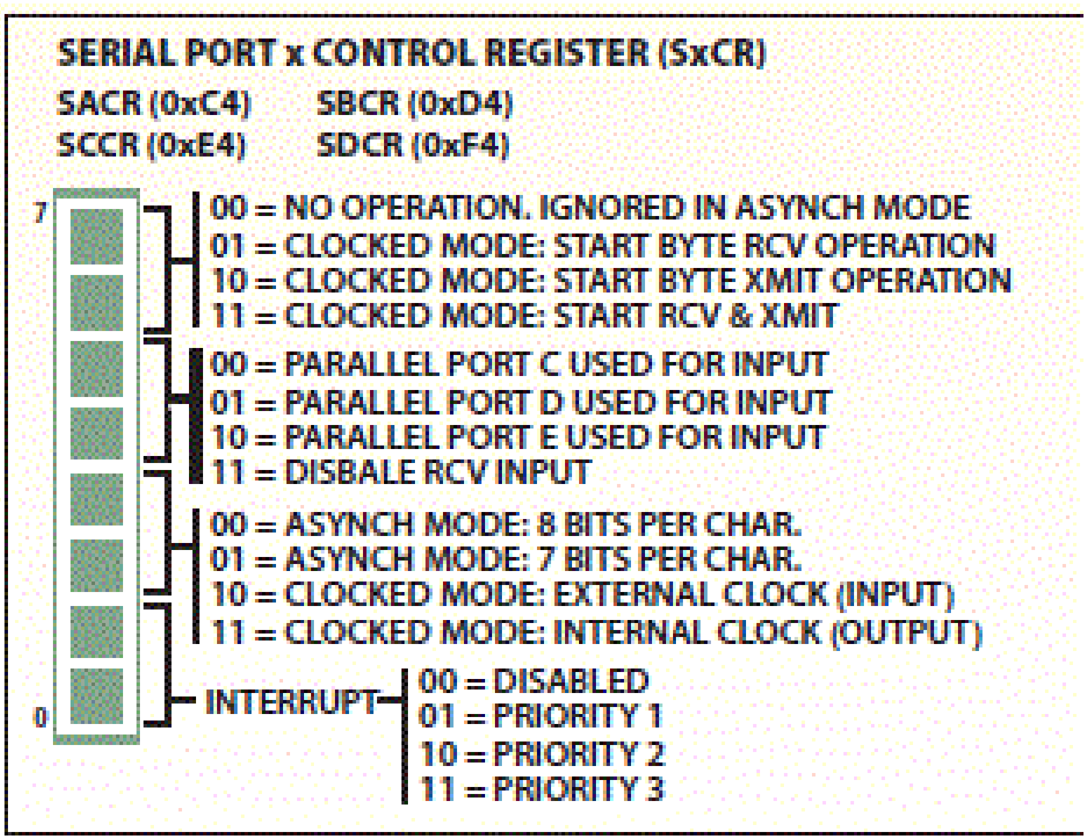

PORT x ALTERNATE LOW REGISTERS
PCALR (0x52) PEALR (0x72)
PDALR (0x62)
7
0
SELECT BIT 3 ALTERNATE OUTPUT
SELECT BIT 2 ALTERNATE OUTPUT
SELECT BIT 1 ALTERNATE OUTPUT
SELECT BIT 0 ALTERNATE OUTPUT


PORT x ALTERNATE HIGH REGISTERS
PCAHR (0x53) PEAHR (0x73)
PDAHR (0x63)
7
0
SELECT BIT 7 ALTERNATE OUTPUT
SELECT BIT 6 ALTERNATE OUTPUT
SELECT BIT 5 ALTERNATE OUTPUT
SELECT BIT 4 ALTERNATE OUTPUT

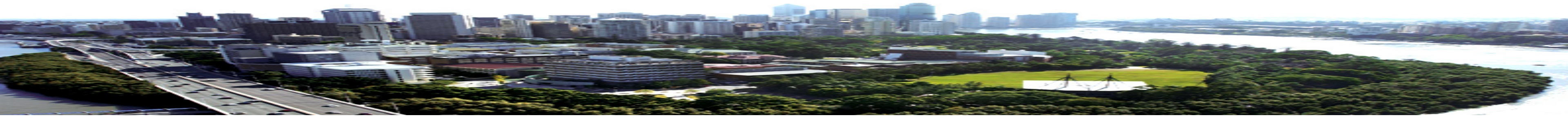


Serial C control register

What should be written to SCCR to have asynchronous mode, with parallel port C used for input 8 bits per char and polled transfers?



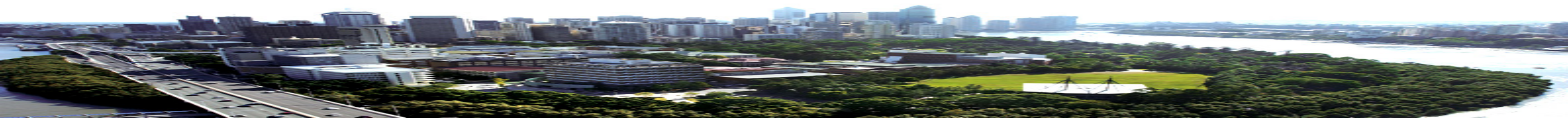
SERIAL PORT x CONTROL REGISTER (SxCR)
SACR (0xC4)
SBCR (0xD4)
SCCR (0xE4)
SDCR (0xF4)
7
0
00 = NO OPERATION. IGNORED IN ASYNCH MODE
01 = CLOCKED MODE: START BYTE RCV OPERATION
10 = CLOCKED MODE: START BYTE XMIT OPERATION
11 = CLOCKED MODE: START RCV & XMIT
00 = PARALLEL PORT C USED FOR INPUT
01 = PARALLEL PORT D USED FOR INPUT
10 = PARALLEL PORT E USED FOR INPUT
11 = DISBALE RCV INPUT
00 = ASYNCH MODE: 8 BITS PER CHAR.
01 = ASYNCH MODE: 7 BITS PER CHAR.
10 = CLOCKED MODE: EXTERNAL CLOCK (INPUT)
11 = CLOCKED MODE: INTERNAL CLOCK (OUTPUT)
INTERRUPT
00 = DISABLED
01 = PRIORITY 1
10 = PRIORITY 2
11 = PRIORITY 3



Rabbit Serial I/O polled – port definitions

```
// define macro for bit rate to use for serial communication
#define  BIT_RATE 19200L

// register macro definitions for Timer A Control Register
#define  TACR_TA6 0xAD      // Serial port C timer control

// status macro definitions for Serial Ports
#define  SER_RCV    7
#define  SER_XMIT   3
#define CDRIVE_TXC  2
#define CDRIVE_RXC  3
```

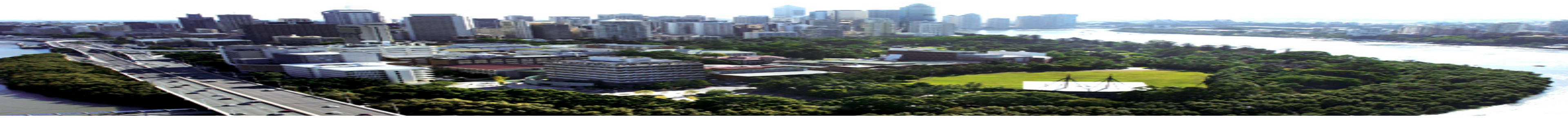
The receiver buffer full and transmit buffer empty status flag bits are bits 7 and 3 respectively on the status register. Data RX and TX bits are 2,3. Parallel port C can be used for serial port C by writing to its function register and to serial C control register.



QUT
CRICOS No. 00213J

Rabbit Serial I/O polled – port set up

```
nodebug void setupSerialC(long bitrate)
{
long divisor;
// set bit rate divisors
   divisor = (long)(freq_divider * 19200.0 / (float)bitrate + 0.5) - 1L;
   WrPortI (TACR, &TACRShadow, TACRShadow & ~TACR_TA6);  // TA6 to use Pclk/2
   WrPortI (TAT6R,&TAT6RShadow, (char)divisor);      // set bitrate generator

// use parallel port C for serial port C I/O
#asm
// set  up the parallel port C for serial C . Set bit 2 for alternate output.
// If we make a change from the default, it is best to check the shadow register and just change the
// necessary bits.
ld    a,(PCFRShadow)
    set  CDRIVE_TXC, a
    ld    (PCFRShadow),a
ioi ld    (PCFR),a
#endasm

// using Dynamic C port write function
 WrPortI (SCCR, &SCCRShadow, 0x00);          // asynch, 8N1, int disabled  (can do this instead of assembly)
} // setupSerialC
```



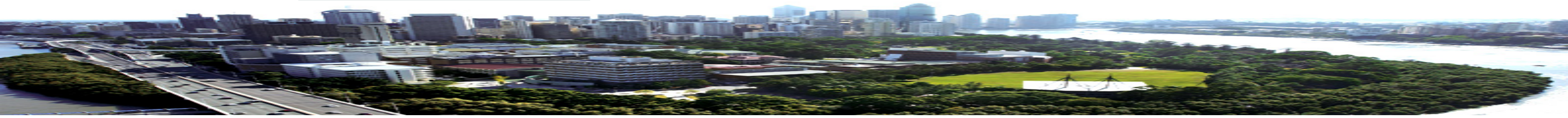
Rabbit Serial I/O polled – transfers

```
char received_data;

setupSerialC(BIT_RATE);

// bit 7 set means receive buffer full
   if (BitRdPortI(SCSR, SER_RCV))
  {
   received_data = RdPortI(SCDR);
   printf("%c", received_data);
}

   // bit 2 reset means transmit buffer empty  (should check these for Rabbit 4000)
   if (!BitRdPortI(SCSR, SER_XMIT))
   {
   WrPortI(SCDR, NULL, received_data);
   }
  }
```

These bits are polled within a loop

Note macros SCSR, SCDR for registers and port read/write functions

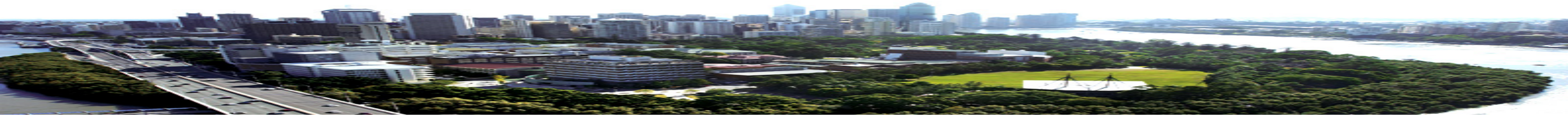
In this example just echoing



Serial communication protocols

- RS 232 (Recommended standard 232 by the IEEE, introduced 1962, uses voltage levels to GND, mostly been replaced by USB)
- RS-485 (enables local networks, multidrop protocol, uses voltage difference between lines A and B)
- HDLC/SDLC (high speed synchronous data link protocols) support frame checking, error checking etc

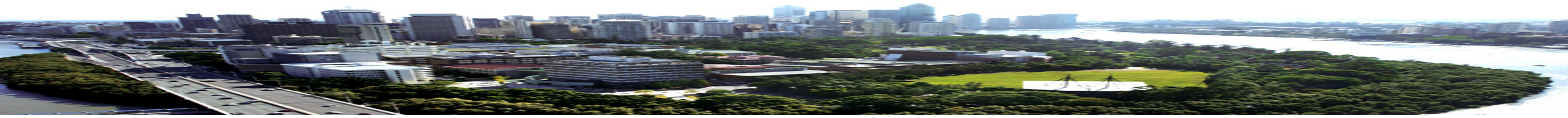
- (Some other communication protocols at hardware / data link layer are : Network communication protocols at bus level – I2C, CAN and packet serial – Ethernet)



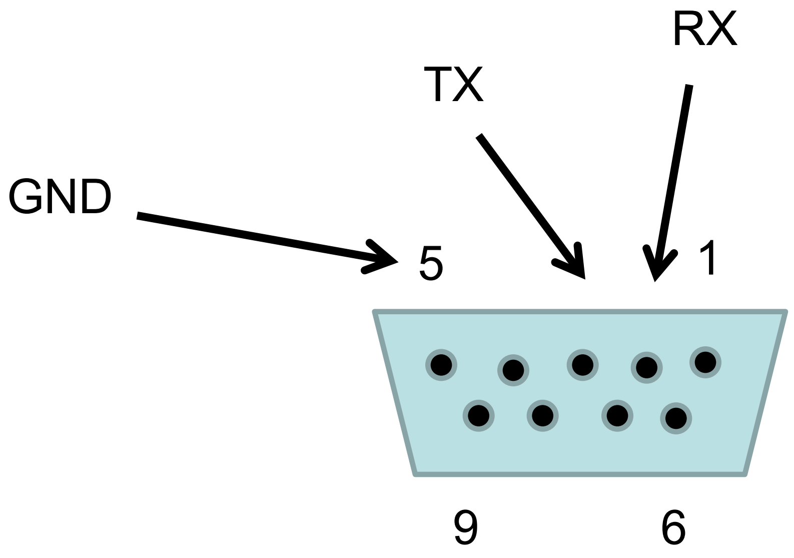
RS-232

- Recommended standard 232 –series of standards for serial binary communication by IEEE introduced in 1962
- COM ports in personal computers used these; now replaced by USB
- RS232 Standard specifies +/- 3 to 15V for binary signals. Rabbit i/o uses 3.3V. RCM4000 has an RS232 interface on the protoboard.
- Other than Rx, Tx and Gnd there are handshaking pins. Hardware handshaking is often not used and 3-wire connections are common.
- 9 pin and 25 pin standard connectors

The RCM4000 in the lab has two female DB-9 RS-232 connectors brought out for serial ports C and D.



RS232 DB9 female pins



Only 3 wire configuration shown here. For other pins consult sources on the web.

Male pins will be mirror imaged to correspond.

Cable may be straight through or cross over TX and RX

RS232.LIB

- Dynamic C library for managing serial ports. Written in assembly.
- Use circular buffers
- Easier to use these high level routines. Implementation in assembly hidden.
- Provide buffering of input and output (buffers are protected by semaphores)
- Interrupt driven
- Some functions are non re-entrant and will not work properly if interrupted in a multi-tasking environment

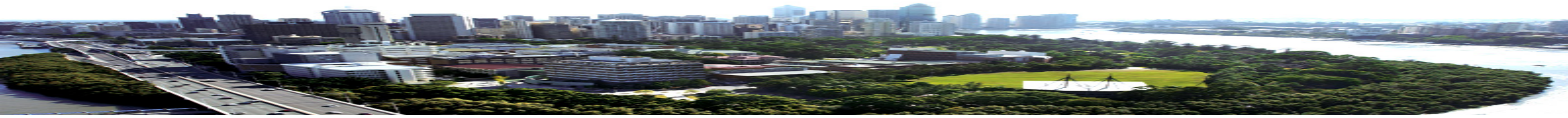
The naming convention is serXfn:

ser - serial

X - the port being used: A,B,C, or D

fn - the function being implemented

Example: serBgetc() is the serial port B function getc(), which returns a character. serBopen(bitrate) opens serial port B for communication.

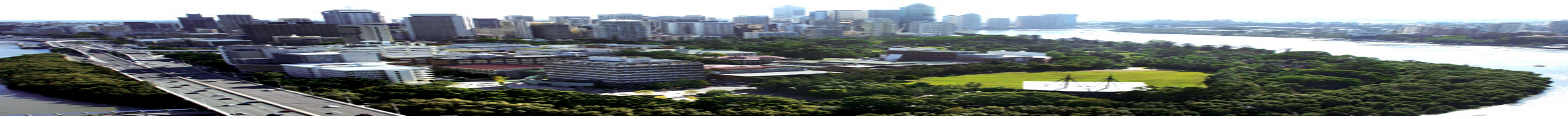


Blocking functions in RS232.LIB

a) Complete their entire serial tasks before returning.
b) Do not require the use of costatements or cofunctions.
c) Simple to use but can hog the processor.

```
int serXgetc();
int serXread(void *data, int length, unsigned long tmout);
int serXputc(int c);
int serXputs(char *s);
int serXwrite(void *data, int length);
```

SYNTAX:	int serAgetc();
DESCRIPTION:	Get next available character from serial port A read buffer. This function is non-reentrant.
PARAMETER1:	None
RETURN VALUE:	success: the next character in the low byte, 0 in the high byte failure: the integer -1



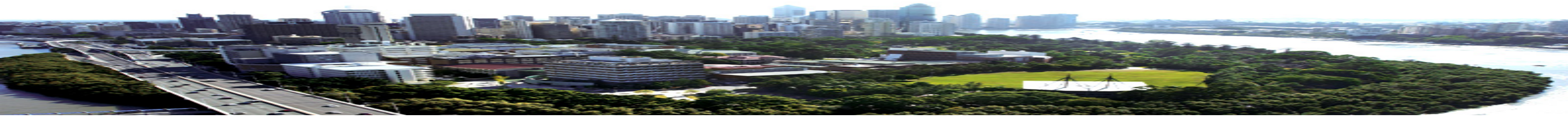
CRICOS No. 00213J

Single user co-functions in RS232.LIB

a) Yield to other tasks whenever circular buffer is full or empty.
b) Must be called within costatements.
c) Require more learning to use properly but share processing better.

```
scofunc int  cof_serXgetc();
scofunc int  cof_serXgets(char *s, int length, unsigned long tmout);
scofunc int  cof_serXread(void *data, int length, unsigned long tmout);
scofunc void cof_serXputc(int c);
scofunc void cof_serXputs(char *str);
scofunc void cof_serXwrite(void *data, int length);
```

```
SYNTAX:          int cof_serAgetc();
DESCRIPTION:     Yields to other tasks until a character is read from port A.
                 This function is non-reentrant.
PARAMETER1:      None
RETURN VALUE: The character read from serial port A
```



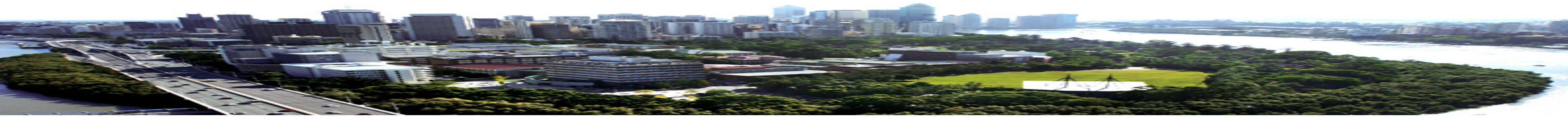
CRICOS No. 00213J

Buffer functions in RS232.LIB

These functions act upon or report status of the circular transmit/ receive buffers.

```
int  serXpeek();
void serXrdFlush();
void serXwrFlush();
int  serXrdFree();
int  serXwrFree();
int  serXrdUsed();
```

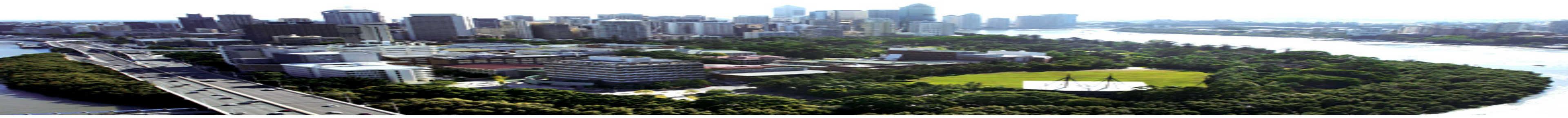
SYNTAX:	int serArdFree();
DESCRIPTION:	Returns the number of characters of unused data space in the port A input buffer. This function is non-reentrant.
PARAMETER1:	None
RETURN VALUE:	the number of chars it would take to fill the A input buffer



CRICOS No. 00213J

Blocking or non-blocking?

- If input <u>must</u> be obtained before proceeding with further execution of instructions we need to use blocking. The task can, however, yield to other tasks and return to that point if we use multitasking. For lab exercise 3, cooperative multitasking with costates would be useful.
- If the task <u>can proceed without the input</u> for the time being use non-blocking. Current task may have to attempt the read or write operation again later.

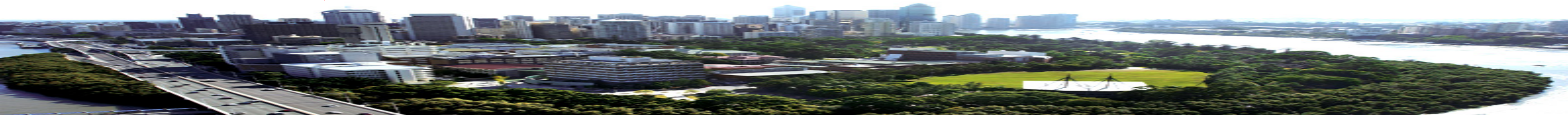


Serial I/O example with RS232.LIB functions

```
#define CINBUFSIZE      15
#define COUTBUFSIZE     15
#define CH_ESCAPE       27
#define BITRATE         19200L

void main()
{
  int c;
  c = 0;
  serCopen(BITRATE);

  while (c != CH_ESCAPE)                                    // Exits on Escape character
   {
    if (((c = serCgetc()) != -1) && (c != CH_ESCAPE))   {   // reads character and echoes back upper case
      serCputc(toupper(c));  }
   }
  serCputs("\nDone\n");

  while (serCwrFree() != COUTBUFSIZE);      // allow transmission to complete before closing
  serCclose(); // close serial port
}
```

What does serCgetc() return? Char or int? Why?

For Lab 3 Task A, use code similar to this example with costates for tasks that handle input from ports C and D respectively. Note that the functions are blocking – and code here needs modification.

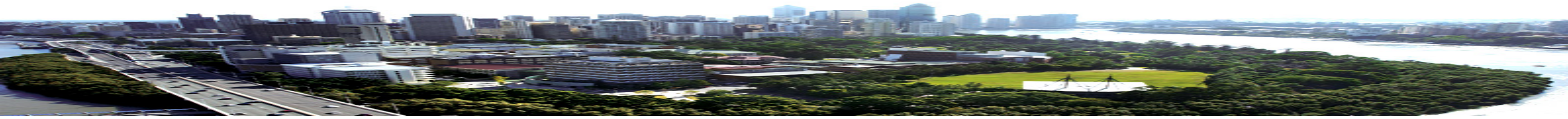


CRICOS No. 00213J

```
#define CINBUFSIZE  15
#define COUTBUFSIZE 15


// This prevents possible spurious line break interrupts.
#define RS232_NOCHARASSYINBRK


main()
{
        auto int nIn1, nIn2;
        auto char cOut;
        auto int icount;

        serCopen(19200);  // set bit rate
        serCwrFlush();    // flush buffers
        serCrdFlush();

  serCputs("Type a Character\n");  // prompt for input

// first, wait until the serial buffer is empty
        while (serCwrFree() != COUTBUFSIZE);

// then, wait until the Tx data register and the Tx shift
// register are both empty
        while (BitRdPortI(SCSR, 3) || BitRdPortI(SCSR, 2));

        while (1)
        {
        while ((nIn1=serCgetc()) == -1);  // Wait for input
        serCputc (toupper(nIn1));         //Send the upper case char back
        }
}
```

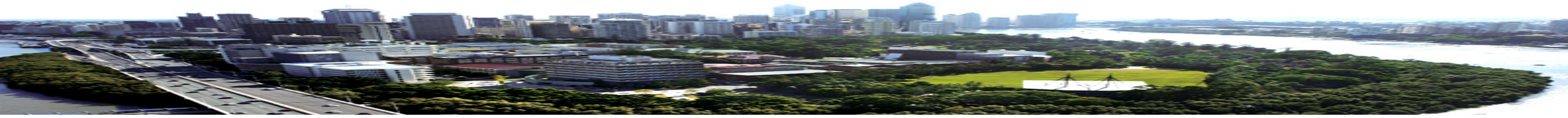
Another example. Port C should be connected to a PC and running HyperTerm. Baud rates should be matched. No hardware control. Preferably put the HyperTerm in VT100 emulation mode. Type characters and observe the output. Turn the echo mode (local) off and check.

Used for demonstration



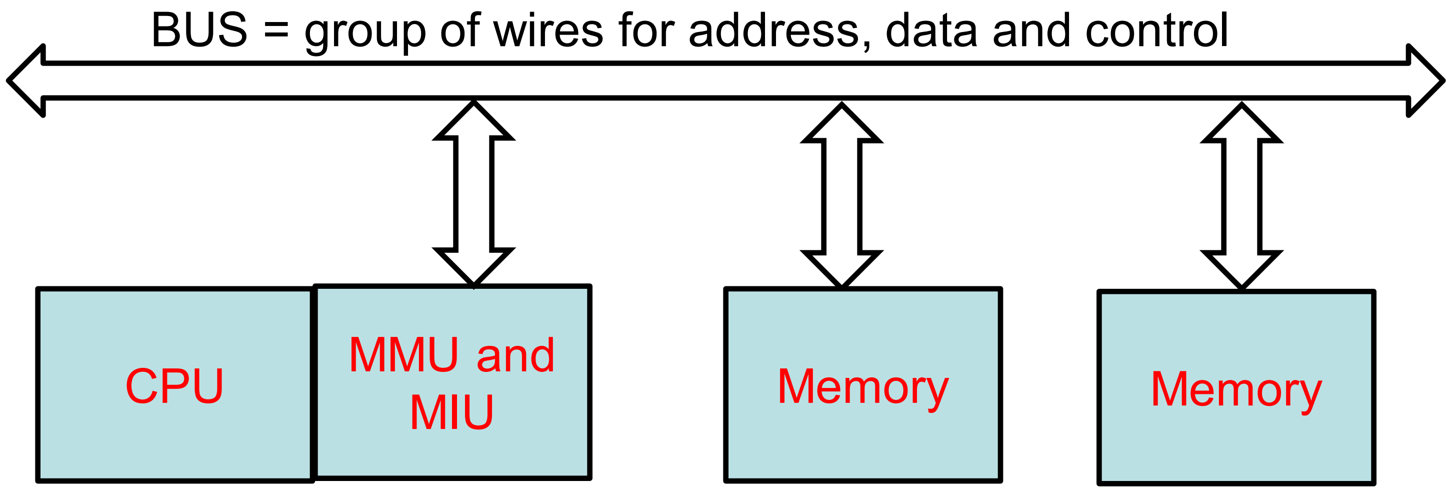
Demonstration

- Polled Serial Communication

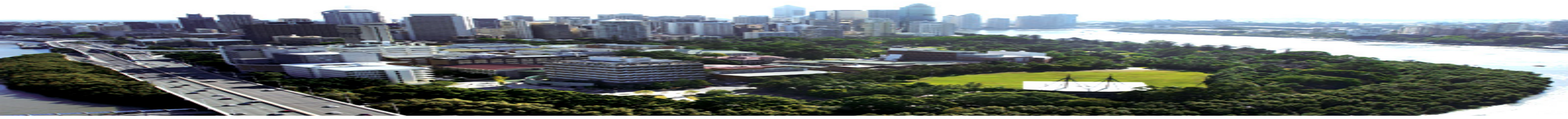


Memory organization

Memory management and interface units can be used to effectively get more address lines (bits) than that used in logical addresses generated and have access to 'extended' physical memory



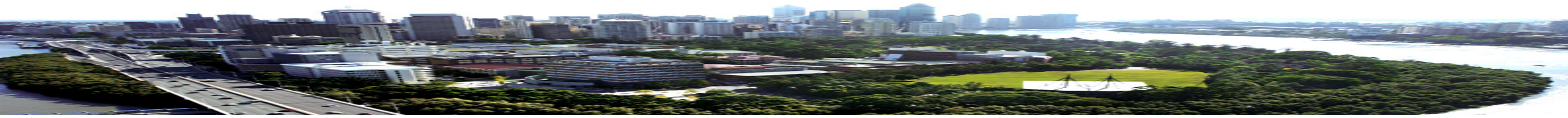
The R4000 architecture is modified to do this. PC is 16 bits. XPC is 8 bits. LXPC is 12 bits.



Physical memory – smaller or bigger?

- Desktop applications usually have a large logical address space (32 bit or 64 bit addresses) and limited actual physical memory. They use virtual memory and bring pages or segments from secondary storage to physical memory when required.

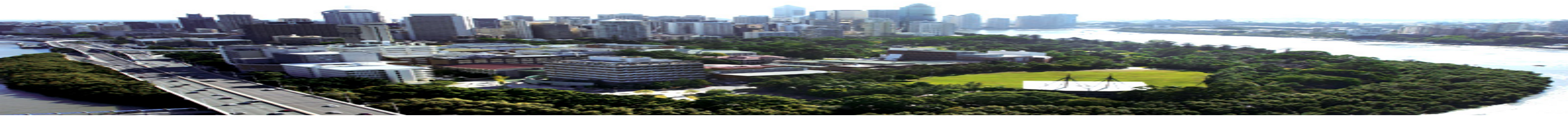
- Embedded processors (usually 8 or 16 bits) and their compilers on the other hand are limited in logical address space and need to map into 'extended' portions in physical memory. This is similar to the problems faced by desktop PCs when the processors were 8 and 16 bit. Segments are mapped. Special registers are used.



Rabbit 4000

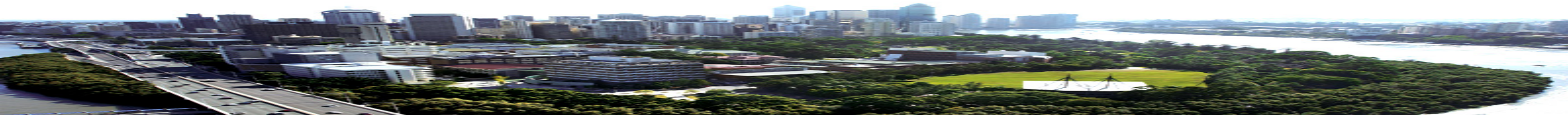
- 16 address lines -> 64K for addresses generated by ordinary instructions

- 20 bit or 24 bit address implemented using an extended program counter register and memory management hardware. Up to 16MB of physical memory is possible.



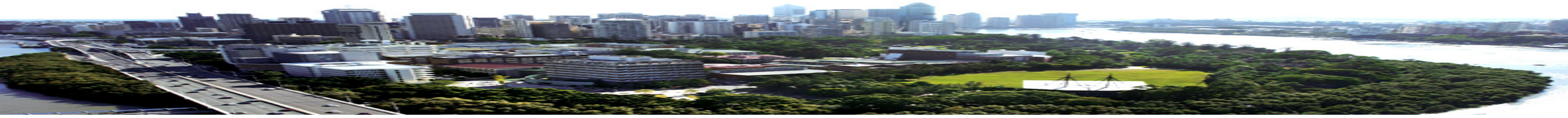
Logical address space division

- The MMU divides the 64K logical address space into 4 regions
- Root, Data, Stack and XPC
- Code in extended memory is accessed via the XPC segment which is 8K in size.
- Program design must ensure that the 56K is not exceeded by root code, data and stack (because of too many strings, too much near code or too much stack space).

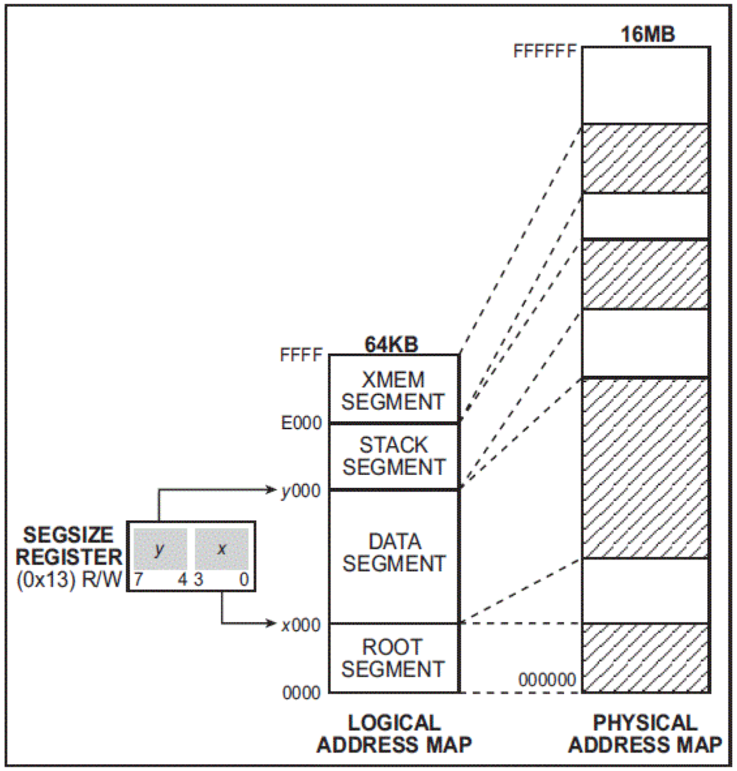


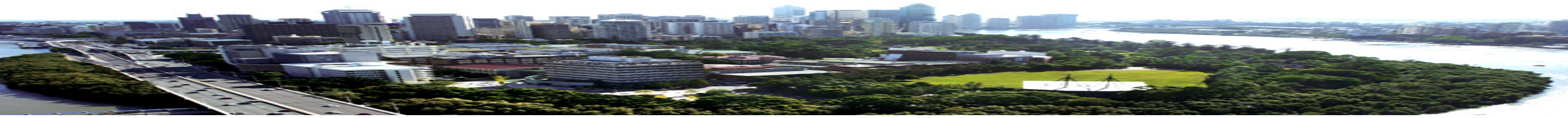
Memory segments

- The first segment, called **root** or base, starts at logical address 0x0000 and physical address 0x00000 (for 20 bit) or 0x000000 (for 24 bit)
- The size of the second (**data** segment) and third (**stack** segment) are decided by the SEGSIZE register. 4 LSB for start of data, 4 MSB for start of stack.
- The fourth segment called the **XPC** segment or extended memory segment always starts at logical address 0xE000.



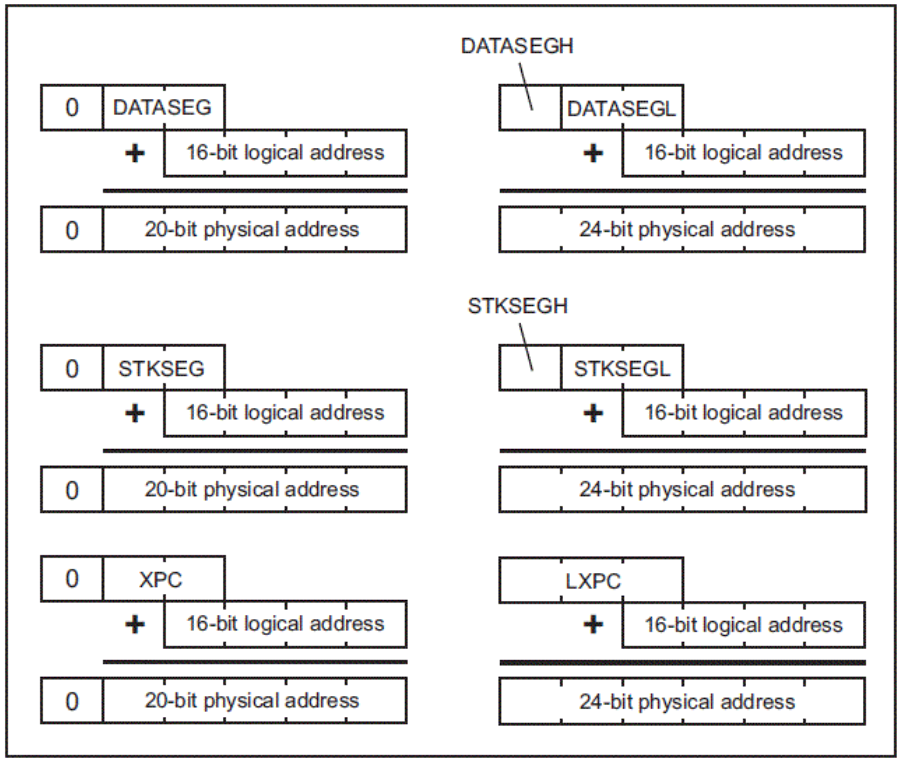
MMU address mapping





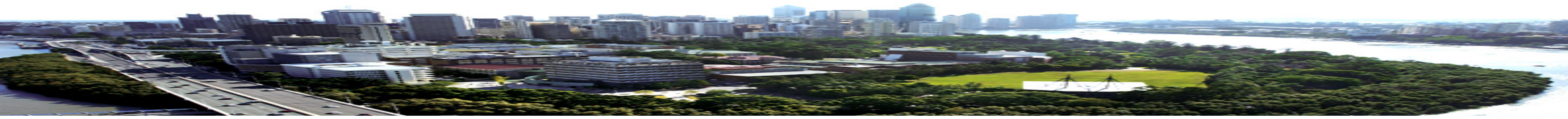
Logical to Physical address

Depending on the value in the DATASEG/STKSEG/XPC register, a region in physical memory is mapped.



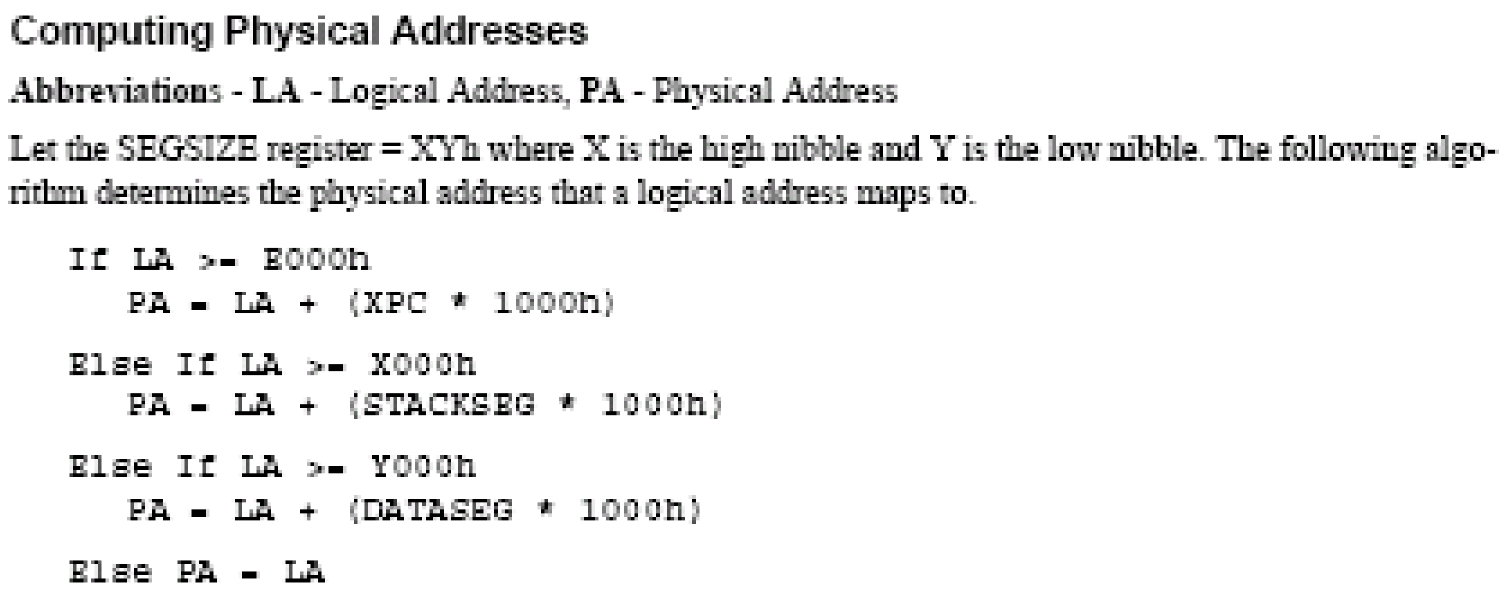
For consecutive values, the regions overlap. By how much?

What is the size of each region?



CRICOS No. 00213J

Physical address calculation

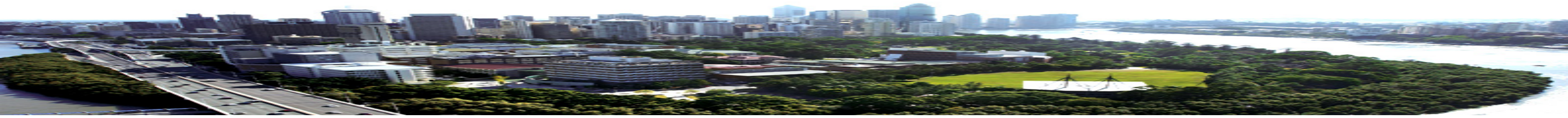


Computing Physical Addresses

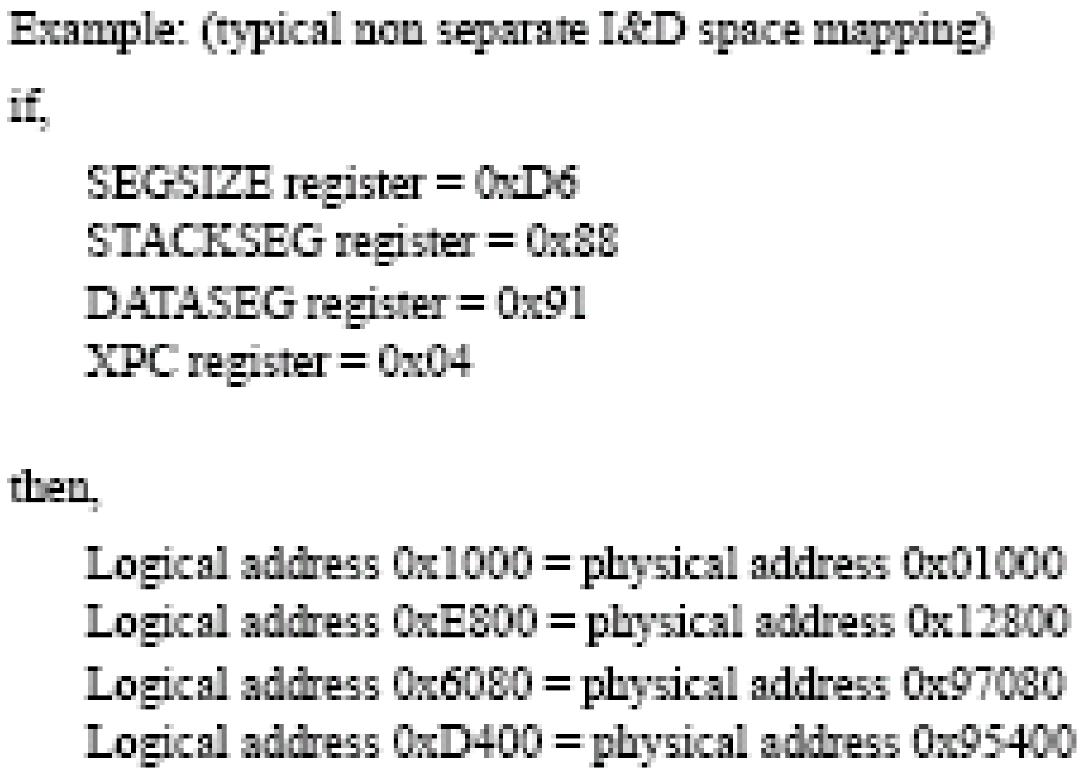
Abbreviations - LA - Logical Address, PA - Physical Address

Let the SEGSIZE register = XYh where X is the high nibble and Y is the low nibble. The following algorithm determines the physical address that a logical address maps to.

```
If LA >= E000h
   PA = LA + (XPC * 1000h)

Else If LA >= X000h
   PA = LA + (STACKSEG * 1000h)

Else If LA >= Y000h
   PA = LA + (DATASEG * 1000h)

Else PA = LA
```



20 bit physical address examples



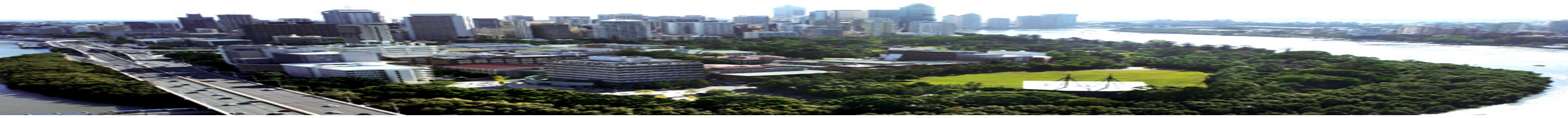
Example: (typical non separate I&D space mapping)

if,

SEGSIZE register = 0xD6
STACKSEG register = 0x88
DATASEG register = 0x91
XPC register = 0x04

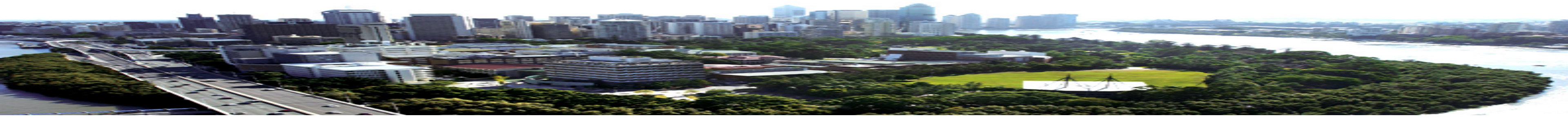
then,

Logical address 0x1000 = physical address 0x01000
Logical address 0xE800 = physical address 0x12800
Logical address 0x6080 = physical address 0x97080
Logical address 0xD400 = physical address 0x95400



Where to place code?

- Functions can be placed in root or XMEM interchangeably in general
- ISRs and functions that modify MMU mapping of the XPC register must be placed in root memory
- Moving functions to XMEM increases available root code space
- Dynamic C will automatically place C functions in xmem as root memory fills up
- Short, frequently used functions must be declared with the root keyword to force Dynamic C to place them in root memory.
- The keyword xmem forces a function to be placed in extended memory.



Separate I and D space

- Makes a distinction between instruction (code) and data.
- With separate I & D enabled,
  - the lower MMU segments (root and data) both map to flash for code fetches while
  - the root segment maps to flash for constants during data fetches and
  - the data segment maps to SRAM for variables during data fetches.
- Thus root code can be found in both segments with separate I & D enabled.
- It doubles root memory.
- Full 0xD000 available for root code and same amount available for root constants and variables.
- Must be enabled using compiler options. It is not the default.
- Stack and XPC segments are unaffected