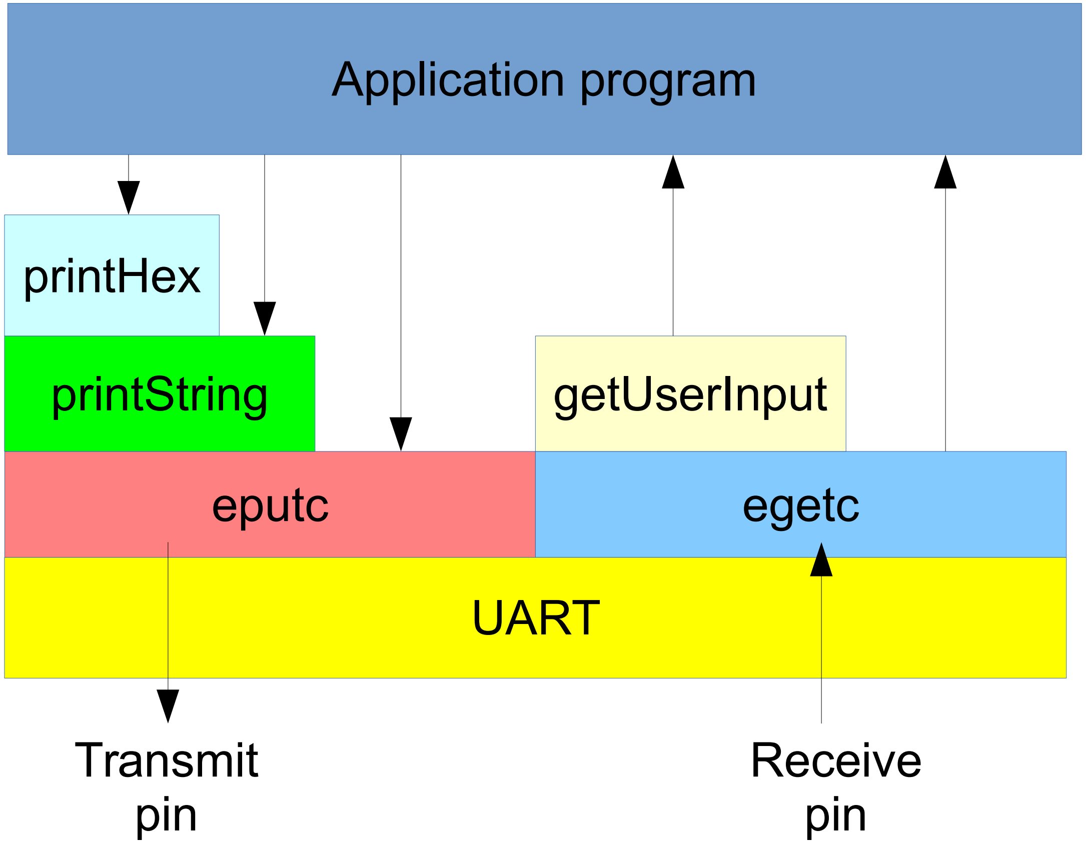
Serial communications (continued)

Serial communications (continued)



Serial communications (continued)

```
void eputc(char c)
{
   U0THR = c; // put char in UART0 Transmit Holding Register
   while((U0LSR & BIT5) == 0); // Wait for tx to finish
}
char egetc()
{
   return U0RBR;  // return contents of UART0 Receive Buffer
                  // register
}
void printString(char *String)
{
   while(*String)
   {
      eputc(*String);
      String++;
   }
}
```

Serial communications (continued)

```
void printHex(unsigned int Number)
{
   // Output the number over the serial port as
   // a decimal string.
   char TxString[9];
   int Index=8;
   TxString[Index]=0; // terminate the string
   Index--;
   while(Index >=0)
   {
      TxString[Index]=HexDigit(Number & 0x0f);
      Number = Number >> 4;
      Index--;
   }
   printString(TxString);
}
```

Serial communications (continued)

```
void printDecimal(unsigned int Number)
{
   // range of values: 0 to 4294967295
   // This is 10 digits long
   // Need to allocate enough buffer space for this
   // and a trailing null character
   char TxString[11];
   int Index=10;
   TxString[Index]=0; // terminate the string
   Index--;
   while(Index >=0)
   {
      TxString[Index]=(Number % 10)+48;
      Number = Number / 10;
      Index--;
   }
   printString(TxString);
}
```

Serial communications (continued)

```
void printDecimal(unsigned int Number)
{
   // range of values: 0 to 4294967295
   // This is 10 digits long
   // Need to allocate enough buffer space for this
   // and a trailing null character
   char TxString[11];
   int Index=10;
   TxString[Index]=0; // terminate the string
   Index--;
   while(Index >=0)
   {
      TxString[Index]=(Number % 10)+48;
      Number = Number / 10;
      Index--;
   }
   Index++;
   printString(&TxString[Index]);
}
```

Serial communications (continued)

```
void printDecimal(unsigned int Number)
{
   // range of values: 0 to 4294967295
   // This is 10 digits long
   // Need to allocate enough buffer space for this
   // and a trailing null character
   char TxString[11];
   int Index=10;
   int Done=0;
   TxString[Index]=0; // terminate the string
   Index--;
   while( (Index >=0) && (!Done) )
   {
      TxString[Index]=(Number % 10)+48;
      Number = Number / 10;
      Index--;
      if (Number == 0)
      {
         Done = 1;
      }
   }
   Index++;
   printString(&TxString[Index]);
}
```

Serial communications (continued)

```
void printDecimal(unsigned int Number)
{
   // range of values: 0 to 4294967295
   // This is 10 digits long
   // Need to allocate enough buffer space for this
   // and a trailing null character
   char TxString[11];
   int Index=10;
   int Done=0;
   TxString[Index]=0; // terminate the string
   Index--;
   while( !Done )
   {
      TxString[Index]=(Number % 10)+48;
      Number = Number / 10;
      if (Number == 0)
      {
         Done = 1;
      }
      else
         Index--;
   }
   printString(&TxString[Index]);
}
```

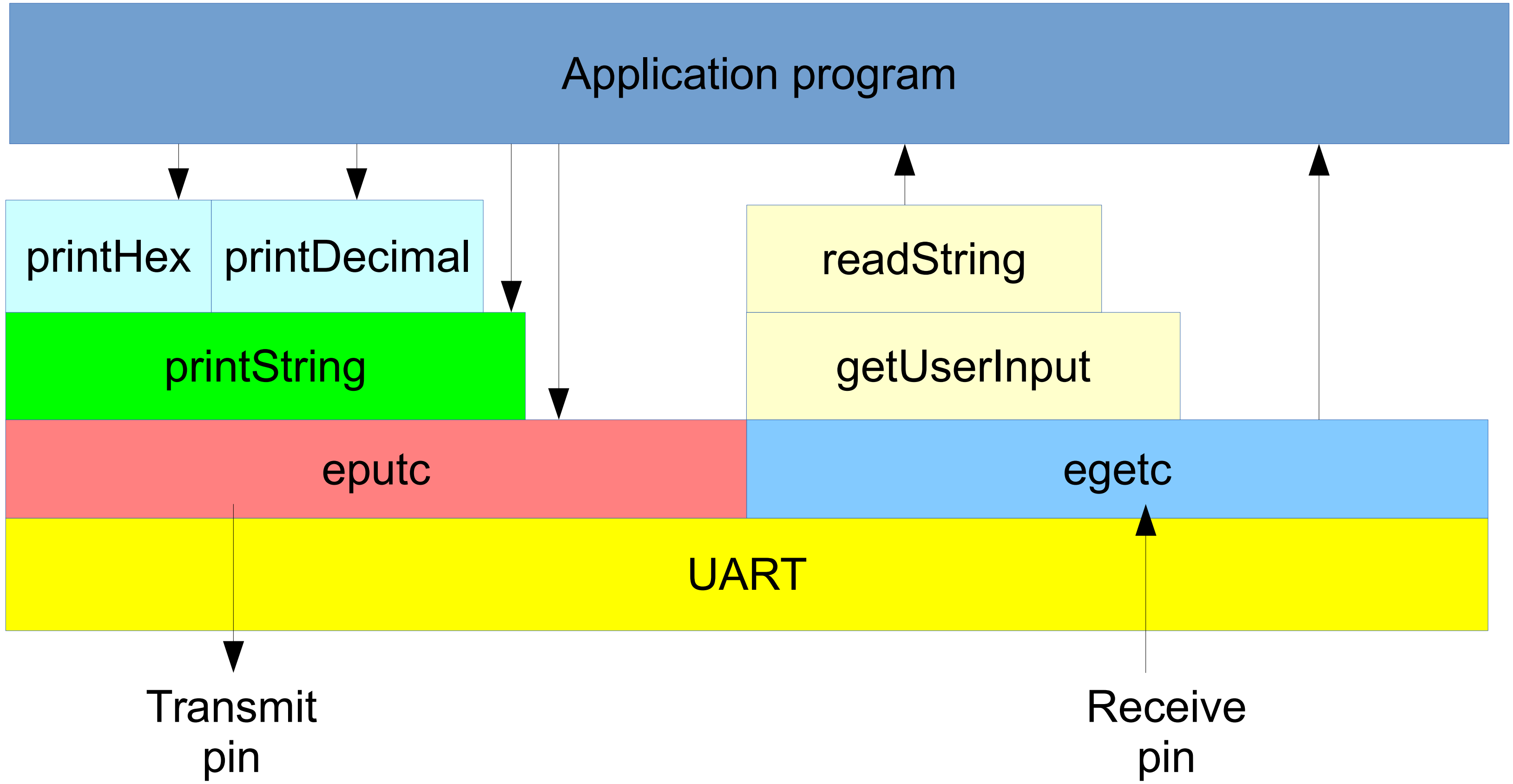
Serial communications (continued)

```
char GetUserInput()
{
    char ch = 0;
    while (ch == 0)
        ch = egetc();
    return ch;
}
```

Serial communications (continued)

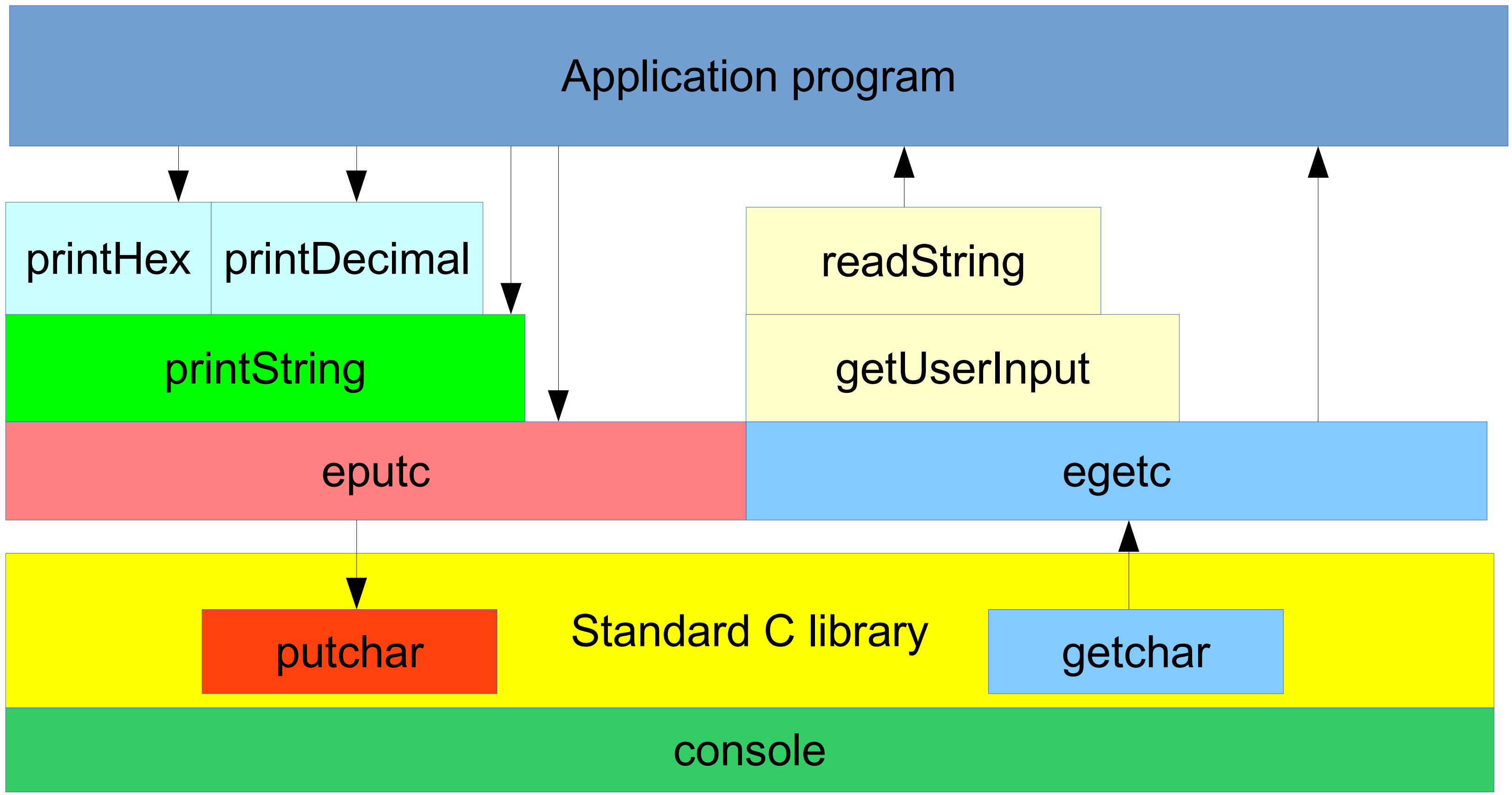
```
void readString(char *String, int Max)
{
   int Index;
   char ch;
   while ( (Index < Max-1) && (ch != '\n') )
   {
      ch = GetUserInput();
      if (ch != '\n')
      {
         eputc(ch);
         String[Index++]=ch;
      }
   }
   String[Index]=0;
}
```

Serial communications (continued)



New Extended API

Serial communications (continued)



Porting to another system

Serial communications (continued)

- Lab 6 discussion points

Serial communications (continued)

- What happens if the user enters text while the system is busy?

Serial communications (continued)

- What happens if the user enters text while the system is busy?
- In the absence of interrupt driven communications, these characters will be missed.
- System must wait until transmissions complete – wasteful of CPU time (and battery)

Serial communications (continued)

- Establishing interrupt driven communications:
  - Configure UART to generate interrupts
  - Configure Interrupt vector table
  - Write interrupt service routine

Serial communications (continued)

- Establishing interrupt driven communications:
  - Configure UART to generate interrupts
  - Configure Interrupt vector table
  - Write interrupt service routine
- Requires circular buffer

Serial communications: Initializing the UART

```
void initUART()
{
    RXBuffer.count = RXBuffer.head = RXBuffer.tail = 0;
    TXBuffer.count = TXBuffer.head = TXBuffer.tail = 0;

    SYSAHBCLKCTRL |= BIT6 + BIT16; // Turn on clock for GPIO and IOCON
    // Enable UART RX function on PIO1_6
    IOCON_PIO1_6 |= BIT0;
    IOCON_PIO1_6 &= ~(BIT1+BIT2);
    // Enable UART TX function on PIO1_7
    IOCON_PIO1_7 |= BIT0;
    IOCON_PIO1_7 &= ~(BIT1+BIT2);
    // Turn on clock for UART
    SYSAHBCLKCTRL |= BIT12;
    UARTCLKDIV = 1;
```

Serial communications: Initializing the UART

```
void initUART()
{

    // PCLK = 48Mhz. Desired Baud rate = 9600
    // See table 199
    // 9600=48MHz/(16* (256*U0DLM + U0DLL)*(1+DivAddVal/MulVal))
    // 312.5 = (256*U0DLM+U0DLL)*(1+DivAddVal/MulVal)
    // let U0DLM=1, DivAddVal=0,MulVal =1
    // 312.5=256+U0DLL
    // U0DLL=56.5.
    // Choose U0DLL=56.
    // Actual baud rate achieved = 9615 - close enough.
    U0LCR |= BIT7; // Enable divisor latch access
    U0FDR = (1<<4)+0; // Set DivAddVal = 0; MulVal = 1
    U0DLL = 56;
    U0DLM = 1;
    U0LCR &= ~BIT7; // Disable divisor latch access
    U0LCR |= (BIT1+BIT0); // set word length to 8 bits.
    U0IER = BIT0+BIT1+BIT2;   // Enable UART TX,RX Data  interrupts
    ISER |= BIT21;            // enable UART IRQ's in NVIC was 13
}
```

Serial communications: Setting up the interrupt vector table

```
const void * Vectors[]
__attribute__((section(".vectors"))) ={
    (void *)0x10002000,   /* Top of stack */
    init,             /* Reset Handler */
    Default_Handler,  /* NMI */
    Default_Handler,  /* Hard Fault */
    0,                  /* Reserved */
    0,                  /* Reserved */
    0,                  /* Reserved */
    0,                  /* Reserved */
    0,                  /* Reserved */
    0,                  /* Reserved */
    0,                  /* Reserved */
    Default_Handler,     /* SVC */
    0,                  /* Reserved */
    0,                  /* Reserved */
    Default_Handler,    /* PendSV */
    SysTick,             /* SysTick */
```

Serial communications: Setting up the interrupt vector table

```
/* External interrupt handlers follow */
    Default_Handler,   /* PIO0_0 */
    Default_Handler,   /* PIO0_1 */
    Default_Handler,   /* PIO0_2 */
    Default_Handler,   /* PIO0_3 */
    Default_Handler,   /* PIO0_4 */
    Default_Handler,   /* PIO0_5 */
    Default_Handler,   /* PIO0_6 */
    Default_Handler,   /* PIO0_7 */
    Default_Handler,   /* PIO0_8 */
    Default_Handler,   /* PIO0_9 */
    Default_Handler,   /* PIO0_10 */
    Default_Handler,   /* PIO0_11 */
    Default_Handler,   /* PIO1_0 */
    Default_Handler ,   /* C_CAN */
    Default_Handler,   /* SSP1 */
    Default_Handler,   /* I2C */
```

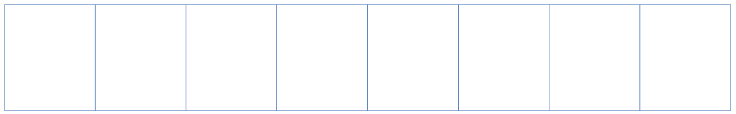
Serial communications: Setting up the interrupt vector table

```
    Default_Handler,   /* CT16B0 */
    Default_Handler,   /* CT16B1 */
    Default_Handler,   /* CT32B0 */
    Default_Handler,   /* CT32B1 */
    Default_Handler,   /* SSP0 */
    UART_isr,          /* UART */
    Default_Handler,   /* RESERVED */
    Default_Handler,   /* RESERVED */
    Default_Handler,   /* ADC */
    Default_Handler,   /* WDT */
    Default_Handler,   /* BOD */
    Default_Handler,   /* RESERVED */
    Default_Handler,   /* PIO3 */
    Default_Handler,   /* PIO2 */
    Default_Handler,   /* PIO1 */
    Default_Handler    /* PIO0 */
};
```

Serial communications: Setting up the interrupt vector table

```
void UART_isr(void)
{
    int Source=U0IIR; // Read the interrupt ID register in UART
    if (Source & BIT2) // RX Interrupt
    {
        putBuf(&RXBuffer,U0RBR);
    }
    if (Source & BIT1) // TX Interrupt
    {
        if (TXBuffer.count > 0)
            U0THR = getBuf(&TXBuffer);
    }
}
```

Circular buffer



Head = 0
Tail = 0
Count = 0

Circular buffer

Received : 'q'

q							

Head = 1
Tail = 0
Count = 1

Circular buffer

Received : 'w'

q	w						

Head = 2
Tail = 0
Count = 2

Circular buffer

Received : 'e'

q	w	e					

Head = 3
Tail = 0
Count = 3

Circular buffer

Received : 'r'

q	w	e	r				

Head = 4
Tail = 0
Count = 4

Circular buffer

Received : 't'

q	w	e	r	t			

Head = 5
Tail = 0
Count = 5

Circular buffer

Received : 'u'

q	w	e	r	t	u		

Head = 6
Tail = 0
Count = 6

Circular buffer

Received : 'i'

q	w	e	r	t	u	i	

Head = 7
Tail = 0
Count = 7

Circular buffer

Received : 'o'

q	w	e	r	t	u	i	o

Head = 0
Tail = 0
Count = 8

Buffer Full!

Circular buffer

Received : 'o'

q	w	e	r	t	u	i	o

Head = 0
Tail = 0
Count = 8

Circular buffer

Read out 'q'

q	w	e	r	t	u	i	o

Head = 0
Tail = 1
Count = 7

Circular buffer

Read out : 'w'

q	w	e	r	t	u	i	o

Head = 0
Tail = 2
Count = 6

Circular buffer

Received 'a'

a	w	e	r	t	u	i	o

Head = 1
Tail = 2
Count = 7

Circular buffer

Received 'a'

a	w	e	r	t	u	i	o

Head = 1
Tail = 2
Count = 7

Circular buffer

- Allows memory to be reused in streaming applications
- Requires maintenance of head and tail variables
- Optionally, requires use of Count variable
- (Demo)