**Parallel and Serial Communications**

Fundamentally, parallel I/O is quite easy to understand. A group of bits (typically a byte) is transferred from one device to another. A wire is used for each of the data bits. A common example of a parallel device is a printer connected to an LPT port of a PC. Traditionally parallel I/O has been regarded as being faster than serial I/O. This is easy to understand because several bits are transferred at a time. However, it is possible for the individual bits to arrive at the receiving end at slightly different times, the data is ‘skewed’, consequently very high speed parallel devices can be complex and special attention must be paid to the cabling etc. Recent high speed serial interfaces (USB and IEEE1394 or ‘Firewire’) have demonstrated that very high data transfers can be achieved quite simply and cheaply using serial I/O.

Usually when two devices communicate (either serial or parallel) a system of handshaking is involved. Apart from the actual data connections there may be additional lines to support the handshaking. Let’s call these extra lines DAV (Data AVailable) and DACK (Data ACKnowledge). Typically the device sending data will output its data and then assert DAV, telling the receiving device that new data is available, and then it waits for a response. The receiving device will recognise that data is available, receive it and then assert DACK. This tells the transmitting device that it can continue.

**Serial Communications**

Instead of latching eight bits of parallel data, we could pass each bit in the byte to a single line, one at a time. Known as bit-serial interfacing, there are serial standards that cover this kind of transmission. Such standards are briefly discussed later.

Since microcomputers are parallel systems, we need to convert an eight bit byte of data to serial form before output, and from serial form to input. There are two ways to perform this conversion: by software, or with a UART (universal asynchronous receiver-transmitter).

**Software Serial I/O:**

In software, a program can simply accomplish the serialisation-deserialisation. On input, the program will wait until it senses a start bit, then sample at the proper times to read the data bits. On output, the program will send the series of ones and zeroes to a single line, with a programmed delay between each bit.

An example of a serial output subroutine for the 6805 appears below. This program sends the byte stored at ‘out_byte’ through bit 0 of portb. The X register is used to perform the serialisation, and another memory location, counter is used to check that all eight bits are sent. The data rate is determined by the duration of the ‘delay’ subroutine which is not shown. The principles of a serialization routine is to assemble an 8 (or more)-bit word in the X register (we could have used the accumulator), and to shift it out, one bit at a time, at the proper frequency. The simplest way is to output the contents of the register to an output port which is connected only to line 0. The register is then shifted right, by one bit position, a delay is implemented, and the next bit is output. After 8 (or more) outputs initial parallel data has been serialised.

```
send   SEI                       ;interrupts would foul up the timing – disable them
       BCLR 0, portb            ;this is a 'start bit'
       JSR  delay               ;delay determines the 'bit' rate
       LDX  $8
       STX  counter
       LDX  out_byte            ;send the out_byte

obit  STX  portb
       JSR  delay               ;wait one bit time
       RORX                     ;rotate used to serialise
       DEC  counter
       BNE  obit
       BSET 0, portb            ;add a stop bit
       JSR  delay               ;enable interrupts
       RTS
```
Conversely, assembling serial data into parallel form by program is just as simple. Bit 0 is read into a register. The register is shifted left. After a specified delay, bit 0 is read again. After eight shifts, a byte has been assembled. It is left as an exercise for the reader to produce a subroutine which performs the serial to parallel conversion!

The advantage of a programmed implementation is simplicity and the elimination of external hardware. However, it is slow, and might impair the microprocessor’s performance. Also, it can be difficult to guarantee reliable delays in a system which uses interrupts. A hardware implementation is often required.

**UART**

One of the earliest standard LSI devices was the UART (Universal Asynchronous Receiver Transmitter). A UART is a serial to parallel and parallel to serial converter. The UART has two functions: to take parallel data and convert it into a serial bitstream with start, stop and parity bits, and take a serial bitstream and convert it into parallel data.

The underlying hardware in the UART is little more than a shift register in which the individual flip-flops may be set or cleared.

A useful example of a UART to study is Motorola's 6850. This is a very old device (we certainly wouldn’t use it in a new design) but it is very simple and (hopefully) easy to understand. Nowadays it is quite common for microcontrollers to have various types of serial I/O built in, consequently no additional hardware is required.

**Error Detection – Parity**

A byte of data contains either an even or an odd number of 1’s. By adding an extra bit we can ensure that the number of bits is always odd or always even. Hence, when the byte is received it can be checked for errors. If the number of 1’s received is odd but should have been even (or vice versa) then an error has occurred. Note however that this is a fairly rough and ready approach: if two bits have changed then the error will pass undetected.

**RS232 in brief**

Of the various serial communication standards, you are probably more likely to encounter ‘EIA RS232C’ than any other. This covers the electrical specification for bit serial transmission as well as the physical specifications and the handshaking signals used to control ‘standard’ equipment.

Electrically, the standard uses nominal +/- 12 volt pulses to effect information transfer. The RS232C standard specifies a 25 pin connector. You may often see 9 pin connectors used (COM ports on PCs). Strictly speaking, the 9 pin connector does not conform to the standard.

**Asynchronous and Synchronous Communication**

So far we have only considered asynchronous communication. In the previous illustration a character is formed from the start bit, data bits, and stop bit(s) which all occur with precise timing. However the time between successive characters is indeterminate. In essence, the receiver recognises the start bit, and because the data rate is predefined (usually called the Baud rate) it can synchronise to the subsequent data bits. If the start and stop bits were not present and the data was sent as a long stream of bits then the receiver would eventually lose synchronisation. This is because of slight differences between the timing of the transmitting and receiving devices – neither of which can be 100% accurate.

One way of overcoming this problem is to send synchronising characters every hundred bytes or so. The receiver contains additional logic to resynchronize the decoding circuitry. In this method (known as synchronous communication) there will typically be eight extra bits for every 800 data bits. This represents a redundancy of only 1% compared to at least 20% in the asynchronous case.

Various kinds of SDLC or Synchronous Data Link Control Schemes exist.

**Signalling techniques**

The problem of synchronisation is a common one, and various schemes have been devised so that synchronisation information is embedded with the data. We are accustomed to thinking of logic signals as abstract 1’s and 0’s. In fact these
logic levels are usually represented by voltages. Typically (in a TTL logic circuit) a logic 1 is represented by a voltage between 2.5 and 5 volts whereas a logic 0 is represented by a voltage between 0 and 1.5 volts.

If we examined a RS232 data transmission we might observe that 0 is usually around +12 volts and 1 is around −12 volts. As long as the designer understands the rules we can use whatever representation is most convenient. Our simple representation has two problems: it is only suitable for directly connected systems (ie using wires) and synchronisation can be a problem.

**NRZ – Is that a gun in your pocket?**

Although we have not given it a name, the signalling technique where bits are transmitted serially using a simple succession of different voltages to represent different logic levels is called Non Return to Zero (NRZ for short).

A nice, everyday example of a serial communication device is the remote control for your TV or VCR etc. You can imagine that when you press a button data is transmitted (I hope you can! whatever did you think was happening?). This data is transmitted serially, via Infra Red usually, but if we think for a moment or two it can’t be using our simple NRZ technique; the handset is a little like a torch – it is either transmitting or not transmitting. If you used a torch to signal to someone it wouldn’t make a lot of sense for ‘on’ to mean 1 and off to mean 0. Would it mean that when your torch was out of sight, in your pocket, you were actually transmitting an endless string of 0’s?

One of the earliest solutions to this problem was/is Morse code. Although Morse code isn’t sufficiently close to our digital dilemma to warrant further comparison it does reinforce the issue.

**NRZI**

A simple solution to the problems of NRZ is NRZI (Non Return to Zero Inverted). This scheme can be difficult to understand, largely because it doesn’t make a lot of sense! In NRZI, a logic zero is represented by a level which is the inverse of the previous bit, whereas a one is represented by a level the same as the previous bit. This works fine if zeroes are continuously transmitted but a succession of 1’s will cause problems. Essentially, it’s only half an answer – it is necessary to ensure that long sequences of 1’s are never sent.

**Manchester code**

Manchester code is nice! It’s easy to understand; If we think about changes in voltage representing logic values then we understand Manchester coding. A change from 0 to 1 means logic 1 and a change from 1 to 0 means logic 0. You have come across this, ethernet uses Manchester coding.

The attached appendix is an example which uses Manchester coding. If you read it you might spot some references to biphase coding. It turned out that Manchester was easier than biphase! Don’t worry if you don’t understand it, it is a tutorial example I use with my final year students.

**Biphase coding**

Biphase is rather similar to Manchester coding. A transition occurs in the middle of a logic 1 but not a logic 0, so a ‘zero’ is represented by two consecutive 1’s and a ‘one’ is represented by a 01 or 10 pair.
Appendix A

* This program illustrates several things:
  * 1) My bad programming!
  * 2) Interrupts and some basic concepts of embedded control
  * 3) Comms (eg why we use bi-phase or Manchester coding)
  * 4) Engineering (should we WAIT or SLEEP?) - what's the difference?
  * How can we make this thing run for a year on a single battery?!

* It is based on a student project which used hardware rather than s/w

* The program is part of the transmitter for a wireless (RF) security system
  * Typically the microcontroller would be attached to a (eg) PIR.
  * the program sleeps, waiting for an interrupt. A timer interrupt occurs
  * every 65.5 msecs and a heartbeat transmission is generated after
  * 256 timer interrupts (about 16 seconds) using a Real Time Interrupt
  * The heartbeat indicates to a central controller that the unit is alive.
  * This is required by BSI for wireless alarm systems.
  * An IRQ input (from an external sensor) indicates an alarm condition.
  * The 'battery low' input is polled every sixteen seconds in the Real
  * Time Interrupt routine, the battery condition is transmitted as part
  * of the status byte.
  * NB above times assume a 4MHz xtal

* Two hex switches are connected to porta. One provides the system (building) address
  * The other is the device number. These values are stored in sys_id and
  * dev_id respectively. These values are biphase encoded ready for transmission.
  * The switch on pa7-4 provides the system address. pa3-0 the device number.

* The heartbeat is only transmitted once - this is not sufficient to guarantee
  * reception. The receiver must allow for this before it complains.
  * A tolerance level should be programmed at the receiver.

* When you run the simulation very little happens, eventually b0 transmits
  * the heartbeat and then it goes back to sleep.
  * Forcing IRQ to 0 (irq 0) simulates an alarm condition.

* Work to do: devise changes required for 32kHz xtal used in 'L' versions
  * of 6805KJ1
* Examine coding technique used in (eg) Holtek remote devices
  * These are genuine biphase rather than Manchester (why?)
* Review heartbeat regularity - currently it is far more regular
  * than necessary (17 seconds) and wastes battery life - DONE
  *---------------------------------------------------------------

* Device programming information:

* MOR SHOULD BE $02: (MOR is the 'mask option register')
* SWPDI 0 - software pulldown enabled/disabled
* PIN3 0 - two pin oscillator configured
* RC 0 - oscillator configured for external xtal
* SWAIT 0 - STOP inst. not converted to WAIT inst.
* LVRE 0 - low voltage reset disabled
* PIRQ 0 - PA3-PA0 disabled as interrupt sources
* LEVEL 1 - IRQ level triggered
* COPEN 0 - COP watchdog disabled

********************************************************
* Equates
dval EQU $FF ; delay seed for bit timing
ramsize EQU $1F  ;1F for K1
mult EQU $0A  ; heartbeat delay multiplier (x 17 seconds)
pdra EQU $10  ; pull-down register A
* RAM - variables

ORG RAM
* Define storage areas in RAM

tempa ds 1  ; temporary storage for the accumulator
tempx ds 1  ; temporary storage for the X register
sys_id ds 1  ; this identifies the system (house) address
dev_id ds 1  ; this identifies the device address
status ds 1  ; the device status
source ds 1  ; source & dest are for the bi-phase coding
dest ds 1
rticnt ds 1  ; count real-time interrupts
rticnt2 ds 1  ; 16 bit counter
scratch ds 1  ; scratchpad memory
scratch2 ds 1  ;

* the status byte is a bi-phase/Manchester encoded 4 bit word. The bits have the
* following function:
  * bit 0  - alarm bit - call the cops!
  * bit 1  - battery low - replace it
  * bit 2  - heartbeat - device functioning ok
  * bit 3  - other/reserved

********************************************************

* PROGRAM

ORG ROM       ; ROM is a reserved word for assembler

* INIT - Initialization routine
* The program starts executing with 'INIT'
* It sets the direction of the porta
* and portb pins, initializes storage locations in ram, and
enables/disables the various interrupts used in the program.

* 

INIT:

LDA  #$1            ;port b...
STA  ddb            ;pb0 output, pb1 input
LDA  #0             ;initialise entire ram area to zero
STA  pdra           ;turn on pull down transistors
STA  ddr           ;and port a as inputs
LDX  #ramsize       ;this is primarily for the simulator which
zero  STA  ram,X        ;doesn't like references to uninitialised memory
DECY
BPL  zero
LDA  #$13           ;enable real time interrupts - every 65.5ms
STA  tcsr           ;clear timer ints, DISable timer overflow ints
LDA  #mult          ;initialise msb of heartbeat timer
STA  rticnt2

* Read the hex switches. Convert to biphase/manchester and store result in
* sys_id and dev_id

LDA  porta          ;read the switches
STA  source
LDX  #$4            ;convert low nibble to biphase
JSR  biph
LDX  dest
STX  dev_id         ;low nibble is device id
LDX  #$4            ;high nibble has become low nibble
JSR  biph
LDX  dest
STX  sys_id         ;high nibble is system id

*********************************************************************
* After initialisation the CPU sleeps. It is woken by an external interrupt
* which indicates an alarm condition, or a Real Time Interrupt to generate
* a heartbeat transmission.

slp1  RSP                 ;enable interrupts
CLI                 ;and reset the stack pointer...
SLEEP  WAIT                ;...back to sleep
BRA  SLEEP

*********************************************************************

* SubRoutine(s) using software delays to send the serial bitstream
* the bit time is about 1ms so total transmission time = 24ms
* the rate is dictated by the available bandwidth of MPT1340 (5kHz)

send  SEI                   ;ints would foul up the timing
BSET  0,portb          ;this is a 'start bit'
JSR  delay            ;delay determines the 'bit' rate
LDX  #$8
STX  tempx
LDX  sys_id           ;send the sys_id byte
obit  STX  portb
JSR  delay
RORX
DEC  tempx
BNE  obit

LDX  #$8              ;then the dev_id byte
STX  tempx
LDX  dev_id
obit2  STX  portb
JSR  delay
RORX
DEC  tempx
BNE  obit2

LDX  #$8              ;then the status byte
STX  tempx
LDX  status
obit3  STX  portb
JSR  delay
RORX
DEC  tempx
BNE  obit3
BCLR 0,portb         ;ensure transmitter is off
CLI
RTS

*********************************************************************
* Delay subroutine used for bit timing.  *
* No entry parameters.  A is corrupted.  *
******
delay  LDA  #dval     ;FF gives a delay of about 1ms with 4MHz xtal
       STA  tempa     ;ensure it is matched at the receiver!
tloop  DEC  tempa
       BNE  tloop
RTS

*********************************************************************
* This subroutine converts a binary nibble to a biphase byte  *
* In fact it is strictly manchester rather than biphase  *
* Entry: The source of the nibble is in 'source'  *
* Exit: The result is in 'dest'  *
* The value in X determines whether the low or high nibble from source  *
* is converted.  If X=4 it's the low nibble.  If X=8 it's the high nibble  *
*********************************************************************
biph    ROR  source        ; lsb into the carry flag
        BCS  one
        ROR  dest        ; carry flag (zero) into dest
        SEC
        ROR  dest        ; prepend a '1' binary 0 = biphase 10
        DECX
        BNE  biph        ; all done?
        RTS
one     ROR  dest
        CLC
        ROR  dest
        DECX
        BNE  biph
        RTS

*********************************************************************
* ISR - interrupt service routines
*       This program contains two interrupt service routines.
*       The first, TIMER_ISR, is executed when a timer overflow
*       occurs. The second, EXTERNAL_ISR, is executed when
*       the IRQ pin goes low.

**********************************************************************
* a real time interrupt occurs every 65.5ms with 4MHz xtal.
* to send a heartbeat transmission count eg 10*256 rtis, ie 170 seconds.

TIMER_ISR:                   ;this code executes if a RTI occurs
          BSET 2,tcsr        ;clear RTI flag
          DEC  rticnt        ;count 256 rtis before acting
          BNE  tim_done      ;comment out this loop for debugging
          DEC  rticnt2       ;otherwise it takes forever!
          BNE  tim_done
          LDA  #mult         ;eg 10*256
          STA  rticnt2
          LDA  #$1            ;don't touch alarm bit
          AND  source        ;make everything else 0
          STA  source
          BRSET 1,portb ,batok ;pb1 is the low battery input
          BSET 1,source       ;the battery is low - set the bit
          batok    BSET 2,source       ;set the heartbeat bit
          LDX  #$4            ;convert bits 0-3 of source
          JSR  biph           ;to biphase format
          LDX  dest           ;the biphase byte
          STX  status        ;ready to send
          JSR  send
          tim_done JMP  slp1           ;return to main program
* don't use an RTI here because it fouls up the stack

**********************************************************************
* An alarm condition has occurred. Set the alarm bit in status.
* Use the existing heartbeat routine to send the message. Because the
* interrupt is level triggered, we should be back here very quickly.
* A delay is inserted to prevent the device from transmitting continually
* and producing a conflict with other devices which might be transmitting.
**********************************************************************

EXTERNAL_ISR:                 ;this is the alarm condition
          BSET 0,status       ;set the alarm bit
          BCLR 1,status       ;remember it is biphase
          JSR  send
          BSET 1,status       ;otherwise alarm will be sent with
          BCLR 0,status       ;don't jam the airwaves with continual
          BE_quiet  DEC  scratch       ;transmission
          BNE  BE_quiet
          JMP  slp1          ;heartbeat transmission is used

**********************************************************************

ORG VECTORS

dw  timer_isr          ;timer overflow service routine
dw  external_isr       ;external interrupt service routine
dw init ; there is no swi interrupt service routine
dw init ; reset vector