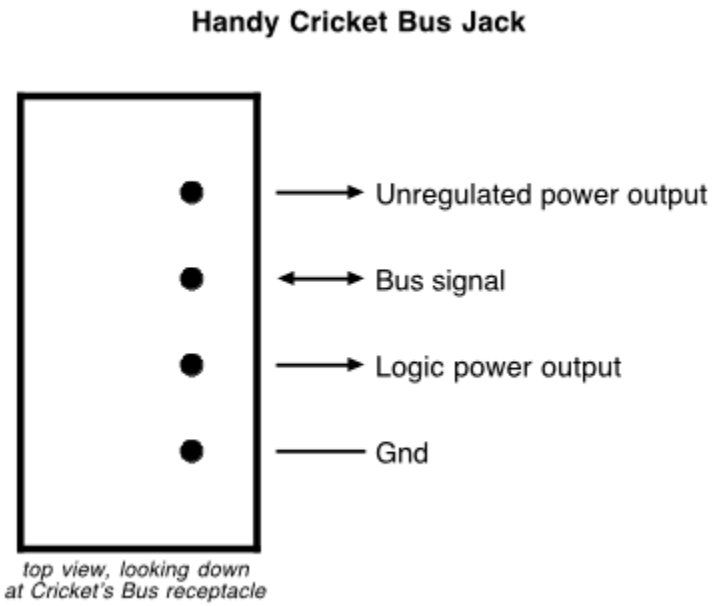
The Handy Cricket Bus

Bus Jack

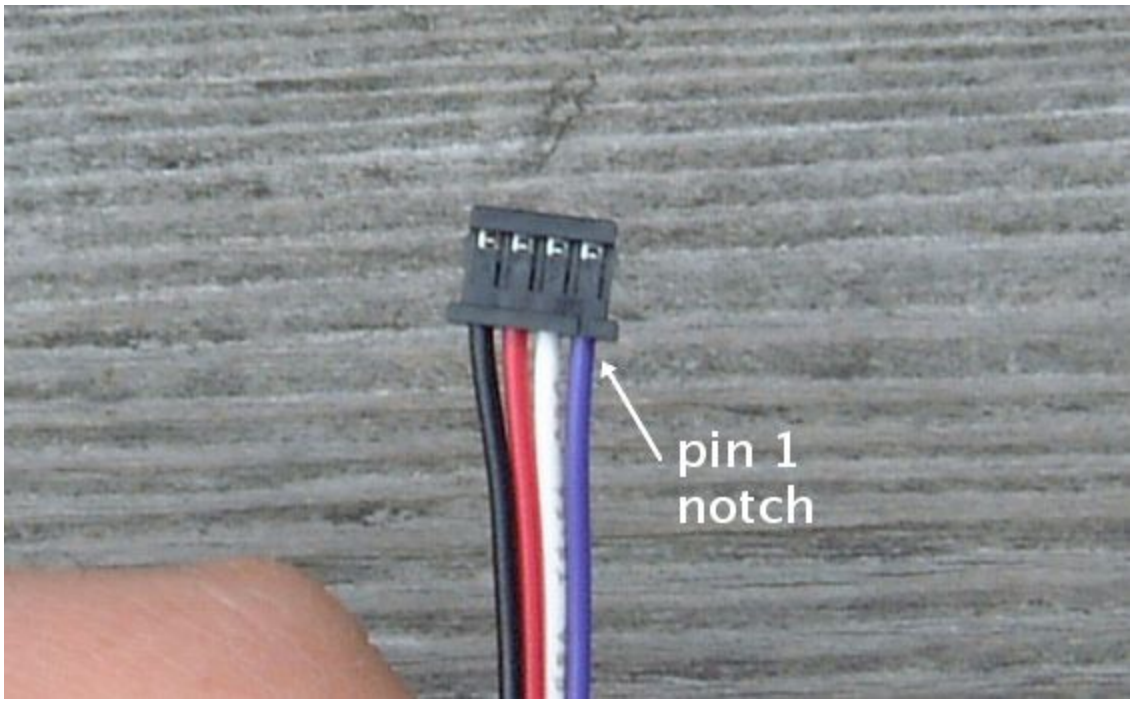
The Handy Cricket bus jack has four pins: the bus signal line, ground, an unregulated battery line, and a logic voltage line:



The plug that mates with this jack is Digi-Key part number H2085-ND. Typically, the plug is stuffed with four wires:

- black/ground H2BXT-10110-B4-ND
- red/+5v logic power output H2BXT-10110-R4-ND
- white/bus signal H2BXT-10110-W4-ND
- violet/unregulated power output H2BXT-10110-V4-ND

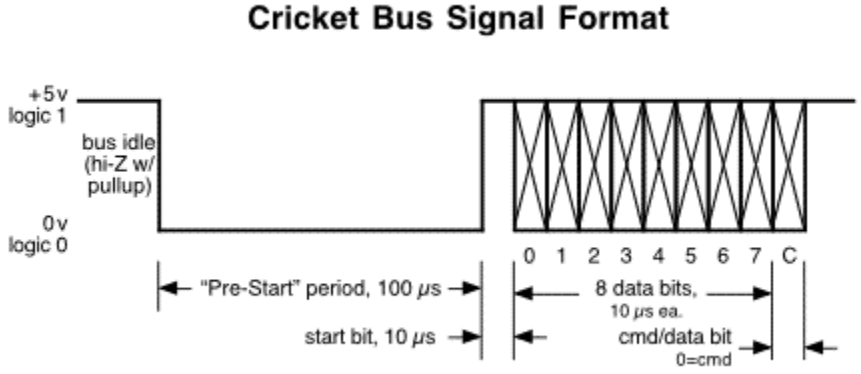
The wires are inserted into the plug as shown below. Note the pin 1 notch, which indicates placement of the violet wire:



Signal Format

The signal line for bus communications follows a custom asynchronous signal format developed especially for the Cricket. The signal was designed to be easily transmitted and received by any small 8-bit microprocessor (like the PIC chip which powers the Cricket), without any special hardware support from the micro.

This signal has a 10 microsecond bit length for the data bits, but includes a 100 microsecond long “pre-start” synchronization pulse. There is also a ninth bit in each word, which is the command/data bit (which will be described in a moment). The following diagram illustrates the format of the whole bus communications word:



The relatively long pre-start period allows receiving devices on the bus to get

ready for the relatively fast transmission of the actual data. Including the pre-start period, the 9 data bits take 200 us to transmit, for an effective data rate of nearly 50k baud.

In the Cricket bus implementation, the receiving device may interrupt on the falling edge of the bus line, and receive the bus communications in the interrupt routine. The 100 us pre-start period gives a generous latency for the interrupt routine to take control.

Alternately, the receiver may process bus communications in a main loop. In this case, other interrupt processing must never take more than 100 us at once.

Compare the Cricket bus format to other two popular micro-to-micro communications methods:

RS-232 serial.

> The ubiquituous RS-232 format may be used to link micros. But if the receiving micro does not have a hardware UART built in, the receiver must dedicate its attention to the incoming serial signal, or be able to capture the point at which the start bit begins, so that it can properly parse the data stream.

I2C synchronous comms.

> Philip's I2C protocol is a powerful, general-purpose method for device communications, and it's relatively easy to write the I2C master code. (Indeed, the Cricket uses an I2C memory to hold the user's program).
>
> Without hardware support, however, I2C receiver code also must be dedicated to waiting for a communication from the I2C master.

So, given the assumption that all devices on the bus will have a micro attached to them, the Cricket bus allows each of these micros to both do useful work and be able to fully attend to bus communications when necessary.

Command/Data Bit

The bus protocol follows a strict master-slave design, where the Cricket is the master device and all other devices on the bus are slaves. The Cricket sends commands to individual devices on the bus, and may retrieve answers from them, but the slave devices are not allowed to initiate communications or interact with each other.

As mentioned, each bus transaction sends nine bits—eight data bits plus a command/data flag bit. The command/data flag is used to indicate whether the bus word is meant to be a command to a device on the bus, or whether it's simply data—that is, additional arguments as part of the command, or the reply

from a bus device.

For instance, suppose a sensor device responds to its command ID with a sensor value. When the Cricket polls the sensor, it issues the command ID with the command/data flag set to command, but when the device responds, the flag is set to data. This way, if the data value matches the command ID for another device on the bus, that device knows to reject the communication.

In the low-level signal format, command is indicated by a zero value in the 9th bit position, and data by a one value.

Communications Protocol

Here, we'll discuss the protocol level which is built on top of the basic 9–bit transfer method. Remember that the Cricket bus is a single-master system in which the Cricket is the master and all other connected devices are slaves.

A given “meaningful communication” from the Cricket to a device may have an arbitrary number of individual words, but the Cricket and the device should know a priori how many words will be coming. The first word is marked as a command word; any subsequent words are sent as data words.

If the communication necessitates a response from a slave device, these are sent as data words.

Here is an example. Suppose the slave device is a numeric display which accepts a 16-bit number and displays it. The command ID for this device is arbitrarily determined to be 17. This is the device's "class ID" -- I've just proclaimed that devices that respond to a command ID of 17 are numeric displays.

The Cricket Logo command for issuing a word on the bus is "bsend" (Bus Send). Then a Cricket program for sending a value to this device for display might be the following:

```
to display :n
   bsend $100 + 17      ; sends the device class ID 17 as a command word
   bsend high-byte :n   ; sends high byte of the num to be displayed (data word)
   bsend low-byte :n     ; sends low byte of the num to be displayed (data word)
end
```

The "meaningful communication" for this display device is then a 3-word sequence consisting of a command word and two data words.

Suppose we want to plug more than one numeric display device into a single Cricket. Assuming we're sticking with 17 as the class ID for these devices, we can design a four-word command sequence, adding a device ID after the top-level command word:

```
to display :n
   bsend $100 + 17 ; top level command
   bsend 1                 ; device instance ID
   bsend high-byte :n
   bsend low-byte :n
end
```

This four-byte approach should be fairly standard for slave devices that need two bytes of input. In general, the command byte (device class ID) followed by the device instance ID will also be standard.

As this example illustrates, the "command word space" is a precious resource consisting capable of addressing 256 devices. The strategy for allocating this resource is to award one command word to each -type- of Cricket bus device. At some point, before the 256-member cmd space is exhausted, the top-level command word will act as an escape code for the next word.

I'll take charge of allocating device class IDs for publicly distributed bus devices. Of course, if you're building your own closed system, you can do whatever you like! You can assign device class IDs as you like, and you don't need to follow the class/instance sequence.

I'll probably establish a convention that all devices in a given class will respond to an instance ID of 0. Thus, if you have a pile of these numeric display devices, with multiple instance IDs (all the same class ID), and you just want to use one of them, then you can send an instance ID of 0 and any will comply.

There is also a Cricket command for receiving a byte on the bus, e.g., when a bus slave device wants to report a value. This command is "bsr," which stands for Bus Send Receive. As per the name, the bsr command also sends a value, so it must be given the value to send when it's used.

For example, suppose we have a bus sensor of some variety that reports a one-byte value, like the Sharp GP2D02 optical ranging sensor. We'll arbitrarily give this device a class ID of 33. If we have an instance of the device with an instance ID of (say) 2, then we'd talk to it with the following:

```
to distance
   bsend $100 + 33 ; send device class 33 with cmd bit set
   output bsr 2          ; send instance ID and get answer
end
```

The bsr command waits up to 1/4 sec for the slave device to send a byte back. If it doesn't get anything, it reports -1.

If you need to get more than one byte back from a device, you need to use multiple bsr's -- there's no Cricket-side primitive for just waiting for a byte to come in. So the protocol might need to be padded with dummy values that are sent from the Cricket to the device so that the device can report more than one byte at a time.

Please note the only special cleverness in this whole protocol is the idea of the command/data marker. This makes it very easy for a device to notice the beginning of a communications sequence -- devices don't need to know how long command sequences are, they just need to watch for command words. If a command word comes in that matches their class ID, then it's the beginning of a comms sequence and they are "at bat." The given device and the Cricket already know how many more words are expected; the device then receives them and "does its thing."

Also, because replies from devices must be marked as data, not commands, there's no chance of a device's reply inadvertently being interpreted as a command by another device.

As should also be apparent, the particular sequence for any given device is quite malleable. The concept of the class ID / instance ID isn't prescribed by the lower level design; it's just a convenience that will make good use of the top-level command word space.

Please let me know if this all makes sense. As I start to release bus devices as part of the Cricket system, I'll keep track of what the class ID-to-device mappings are. This will be public information.

Sample Code

The following code, bus.asm, implements a sample bus device. The code is written to run on a PIC microprocessor running at 4 MHz. Please note that cycle-counting is done for this specific crystal speed; the core communications routines would need to be re-written to run on different processors or the PIC operated at a different speed.

The code implements a sample bus device that responds to the command code 17 decimal by toggling one of its pins (Port B bit 7) each time the command code is received.

The bus device also replies with a single byte when the matching command code 17 is received. This demonstrates how a bus device can reply to a Cricket. After toggling, if the output pin becomes low, a 100 decimal is returned. If the output pin became high, a 101 decimal is returned.

Communication with the bus device may be demonstrated by running the following Logo statement on the Cricket:

```
if (bsr 256 + 17) = 101 [beep]
```

Each time this is run, the Cricket sends the matching command code 17. The 256 is added to set the command bit. The bus device reply is then the output of

the parenthesized `bsr` command. If this is equal to 101, the Cricket will beep. Thus, each time this statement is run, the bus device's output pin toggles, and every other time it is run, the Cricket will beep

To build this bus device, use a PIC16F84 chip with a 4 MHz oscillator. The chip may be powered by its own +5v supply or it may be powered by the logic power output on the Cricket Bus connector. The Cricket Bus signal line should be connected to the pin designated for bus communications as commented in the code listing (i.e., Port B bit 0).

Following is the bus.asm code listing.

```
;;; busdemo.asm
;;;
;;; demonstrates use of Handy Cricket bus
;;; toggles a pin when addressed by Cricket,
;;; then gives one of two replies based on resulting state of the pin.
;;;
;;; pin assignments:
;;; Port B 0 is bus I/O, Port B 7 is toggle pin
;;;
;;; bus values:
;;; 17 (decimal) is bus address
;;; 100 or 101 are return values based on toggle pin's state.
;;;
;;; NOTE!
;;; this bus code requires 4 MHz crystal with 1 usec per instruction clock

;       device pic16f84
;       config  OSC=xt
        radix dec

;;; register destination constants
W       equ     0
F       equ     1

;;; register constants
INDF    equ 0
STATUS  equ 3                   ; status register
C       equ 0                   ; status C bit
Z       equ 2                   ; status Z bit
RP0     equ 5                   ; status register page 0
FSR     equ 4
PORTB   equ 6                   ; port B register

;;; bus port and pin
BUSPORT equ PORTB
BUSTRIS equ PORTB + 0x80        ; for setting bus as output
BUSPIN  equ 0                   ; bit 0

;;; memory variables
buscntr equ 0x20                ; bit counter
busdata equ 0x21                ; bus data register
counter equ 0x22                ; temp register

        org 0
```

```
start:
        bsf STATUS,RP0
        bcf PORTB,7             ; make port B7 output
        bcf STATUS,RP0

;;; the main loop
;;; block until receives word on bus
;;; check if it's cmd or data, if data, ignore.
;;; if cmd, check if it's our magic number, if not, ignore
;;; if it's for us, toggle output pin then generate reply.
loop:
        call bus_tyi            ; get word from bus
        btfss STATUS,C          ; if C set, it's a cmd word
        goto loop

        movf busdata,W
        xorlw 17                ; 17 is our magic number
        btfss STATUS,Z
        goto loop               ; if Z clear, wasn't our number

        ;; toggle PORTB 7 so we'll know cmd word was received!
        call toggle

        ;; now, return 100 if PORTB 7 is low, 101 if high!
        ;; from logo, "IF (BSR 256 + 17) = 101 [BEEP]"
        ;;   will beep on alternate tries as the pin is toggled.
        movlw 100
        movwf busdata
        btfsc PORTB,7           ; if PB7 set,
        incf busdata,F          ;   incr busdata to 101
        call bus_tyo            ; now return it.

        goto loop

;;; inverts state of PORTB 7
toggle:
        btfsc PORTB,7
        goto togclr
        bsf PORTB,7
        return
togclr: bcf PORTB,7
        return


;;; returns byte in busdata
;;; the inverse of the stop bit in the carry
;;; commands have a 0 stop bit -> carry set
;;; data has a 1 stop bit -> carry clear
bus_tyi:
        btfsc BUSPORT,BUSPIN
        goto bus_tyi            ; loop until bus line goes low

;       bcf INTCON,GIE          ; disable interrupts (if necessary)

btyi20  btfss BUSPORT,BUSPIN
        goto btyi20             ; wait for end of prestart

        movlw 8
        movwf buscntr
        call an_rts             ; use 4 cycles, into middle of start bit
```

```
btyi30  nop
        nop
        nop
        rrf busdata,F           ; pre-rotate destination byte
        bcf busdata,7           ; clear high bit (it'll rotate down)
        btfsc BUSPORT,BUSPIN
        bsf busdata,7           ; if BUSPIN was set, set busdata bit
        decfsz buscntr,F
        goto btyi30

        call an_rts
        nop
        bsf STATUS,C            ; pre-set carry (cmd word)
        btfsc BUSPORT,BUSPIN
        bcf STATUS,C            ; if no stop bit, clear carry

;       bsf INTCON,GIE          ; re-enable interrupts (if necessary)
an_rts
        return

;;; send a byte down the bus.
;;; this is the slave device code, so send '1' as stop bit,
;;;   indicating a data word.
;;; put the byte to send in busdata before calling.
bus_tyo:
;       bcf INTCON,GIE          ; disable interrupts (if necessary)

        movlw BUSTRIS
        movwf FSR               ; will be controlling bus to output
        bcf BUSPORT,BUSPIN      ; put 0 into pin register
        bcf INDF,BUSPIN         ; write it out & begin pre-start pulse
        movlw 33
        movwf counter
btyo20: decfsz counter,F
        goto btyo20             ; wait 100 usec
        bsf BUSPORT,BUSPIN      ; end pre-start, begin start bit

        bsf STATUS,C            ; will be stop bit, indicating data word
        movlw 9                 ; 8 data bits plus C bit last
        movwf counter
        call an_rts             ; use up 4 cycles
        nop                     ; plus 1 more making 10 for start bit

btyo50: rrf busdata,F           ; bit -> carry
        btfss STATUS,C          ; if C is clear,
        bcf BUSPORT,BUSPIN      ;   clear bus pin
        btfsc STATUS,C          ; if C was set,
        bsf BUSPORT,BUSPIN      ;   set bus pin
        nop
        nop                     ; tune the loop to take 10 us
        decfsz counter,F
        goto btyo50
        nop
        nop
        bsf INDF,BUSPIN         ; bus to input again, float high
;       bsf INTCON,GIE          ; re-enable interrupts (if necessary)
        return

        end
```

Last modified: Thursday, 04-Mar-2004 09:59:26 PST by fredm at-sign handyboard dot com