# StickOS™ BASIC User's Guide v1.82

1   Overview

StickOS BASIC is an *entirely MCU-resident* interactive programming environment, which includes an easy-to-use editor, transparent line-by-line compiler, interactive debugger, performance profiler, and flash filesystem, all running entirely within the MCU and controlled thru an interactive command-line user interface.
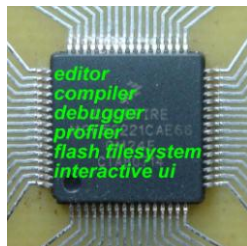
In StickOS, external MCU pins may be bound to BASIC "pin variables" for manipulation or examination, and internal MCU peripherals may be managed by BASIC control statements and interrupt handlers.

A StickOS-capable MCU may be connected to a host computer via a variety of transports and may then be controlled by any terminal emulator program, *with no additional software or hardware required on the host computer*.

Additionally, when coupled with an MC13201 ZigFlea Wireless Transceiver, the MCU may be remotely controlled by another MCU, via a telnet/rlogin-like interface, eliminating the need for a direct connection to the host computer altogether. Also, BASIC programs may trivially remotely access variables on other MCUs, enabling the use of "remote pin variables" or other forms of inter-MCU communication.

On selected MCUs, the USB interface can optionally be configured into USB Host Mode, creating a trivial data logger to an external USB flash drive.

Once program development is complete, the MCU may be disconnected from the host computer and configured to autorun its resident BASIC program autonomously.

1

By its very nature, StickOS supports in-circuit emulation when it is running --
all you need is a transport connecting the MCU to a host computer, and you
have full control over the target embedded system, just as if you were using an
in-circuit emulator! Alternatively, you can use the 2.4GHz zigflea wireless
transport and have full control over the target embedded system with no
connected transport at all!!!

The StickOS BASIC programming environment includes the following
features:

- o BASIC line editor
    - o ansi or vt100'ish terminal support
- o BASIC compiler
    - o compiles to a fast and safe intermediate bytecode
    - o transparent line-by-line compilation is invisible to the user
    - o integer variable/array support
    - o string variable support
    - o block structured programming and subroutine support
- o interactive BASIC debugger, supporting:
    - o breakpoints, assertions, and watchpoints
    - o live variable (and pin) manipulation and examination
    - o execution tracing and single-stepping
    - o edit-and-continue!
- o BASIC performance profiler
    - o trivially see where your program spends its time!
- o BASIC file system
    - o load and store multiple BASIC programs in flash
- o 2.4GHz zigflea wireless transport
    - o remote control via a telnet/rlogin-like interface
    - o remote variable access in BASIC
    - o wireless BASIC program update
    - o wireless StickOS firmware upgrade
- o USB Host Mode (on selected MCUs)
    - o log StickOS "print" statements to external USB flash drive
- o external control of MCU I/O pins, implicit thru "pin variables"
    - o digital input or output
    - o analog input or output (PWM actually)
    - o servo output
    - o frequency output
    - o uart input or output
    - o i2c master input and output
    - o qspi master input and output

- o 4-bit HD44780-compatible LCD output
- o 4x4 scanned keypad input
- o internal peripheral control
  - o interrupts delivered to BASIC handlers!
  - o interval timers, dma timers, ADC, PWM, uarts, i2c, qspi, etc.
  - o direct MCU register access from BASIC for low-level control, thru MCU register variables
- o internal flash memory control
  - o save programs and parameters to flash for standalone operation
  - o prolong flash lifetime by storing incremental updates in RAM
  - o clone one MCU's flash directly to another
  - o upgrade StickOS firmware via terminal emulator!
  - o no external flash programmers needed!

Note that for the purposes of examples in this User's Guide, we'll be running StickOS primarily on an MCF52221 and MCF51JM128; other MCUs are similar.

Table of Contents

2 Examples

2.1 Embedded Systems Made Easy

A simple embedded system, like a toaster oven temperature profile controller, can be brought online in record time!
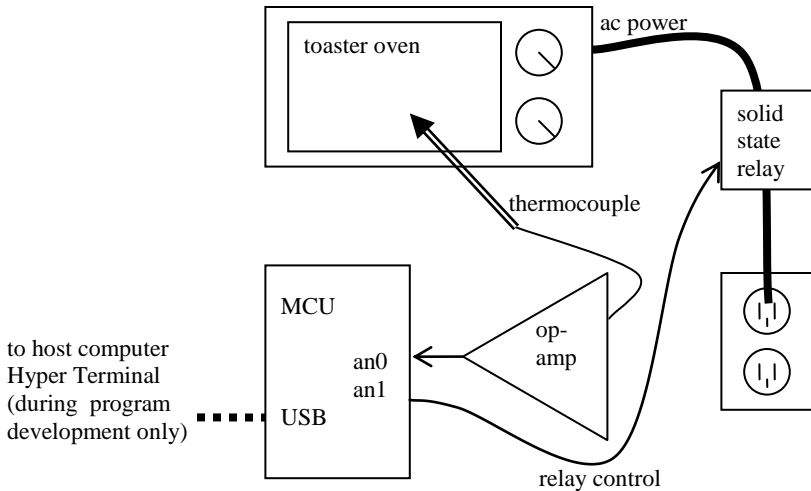
It's as easy as...

1. wire the MCU I/O pins to the embedded circuit
   a. wire MCU pin an0 to thermocouple op-amp output
      (I use an LM358)
   b. wire MCU pin an1 to solid state relay control input
      (I use a Teledyne STH24D25)
2. install the cpustick.inf file by saving it to a file, right-clicking on the file, and selecting "Install"; you can ignore warnings about an unsigned driver package -- the driver is straight from Microsoft, and only the INF file is unsigned; the INF file allows Windows to bind a human readable name, "CPUStick", to the USB VID/PID presented to the host by StickOS
3. connect a host computer to the USB interface on the MCU
4. let the host computer automatically install the new hardware
5. open a Hyper Terminal console window and connect to the MCU; press **<Enter>** for a command prompt
6. configure the MCU I/O pins as appropriate

>
> a. configure pin an0 as an analog input
> b. configure pin an1 as a digital output
7. write and debug your BASIC control program, live on the MCU (see below)
8. type "save"
9. type "autorun on"
10. turn the toaster oven full on (so that the relay can control it)
11. type "reset"
12. disconnect the host computer from the USB interface on the MCU

The entire toaster oven temperature profile controller BASIC control program is shown below:

```
cpustick - HyperTerminal
File  Edit  View  Call  Transfer  Help

Welcome to StickOS for Freescale MCF52221 v1.40a!
Copyright (c) 2008; all rights reserved.
http://www.cpustick.com
(checksum 0xb9ae)
> 10 dim target, secs
> 20 dim thermocouple as pin an0 for analog input
> 30 dim relay as pin an1 for digital output
> 40 data 512, 90, 746, 105, 894, 20, -1, -1
> 50 configure timer 0 for 1000 ms
> 60 on timer 0 do gosub adjust
> 70 while target!=-1 do
> 80    sleep secs s
> 90    read target, secs
> 100 endwhile
> 110 let relay = 0
> 120 end
> 130 sub adjust
> 140    if thermocouple>=target then
> 150       let relay = 0
> 160    else
> 170       let relay = 1
> 180    endif
> 190 endsub
> save_

Connected 0:00:11        ANSIW       115200 8-N-1     SCROLL   CAPS   NUM   Capture
```

- Line 10 declares two simple RAM variables named "target" and "secs" for use in the program, and initializes them to 0.

- Line 20 declares an analog input "pin variable" named "`thermocouple`" that is bound to pin *an0*, to read the thermocouple voltage, in millivolts
- Line 30 declares a digital output "pin variable" named "`relay`" that is bound to pin *an1*, to control the solid state relay.
- Line 40 declares the temperature target and delay time pairs for our temperature profile ramp.
- Lines 50 and 60 configure a timer interrupt to call the "`adjust`" subroutine asynchronously, every second, while the program runs.
- Lines 70 thru 100 set the target temperature profile while the program runs.
- Lines 110 and 120 end the program with the solid state relay control turned off.
- Lines 130 thru 190 use the declared pin variables to simply turn the solid state relay control off if the target temperature has been achieved, or on otherwise.

Then:

- "save" saves the program to non-volatile flash memory.
- "autorun on" sets the program to run automatically when the MCU is powered up.
- Finally, "reset" resets the MCU as if it was just powered up.

Note that if terse code were our goal, lines 60 and 130 thru 190 could have all been replaced with the single statement:

```
> 60 on timer 0 do let relay = thermocouple<target
```

2.2 Embedded Systems Made Functional!

With the advent of advanced serial peripherals based on the I2C or QSPI serial interfaces, embedded systems can take on a whole new level of real-world functionality!

An LCD digital thermometer, displaying both Celsius and Fahrenheit, can be brought online in minutes, with just a quick study of the I2C peripheral protocol definitions! The peripherals are:

- Texas Instruments TMP102 temperature sensor, at I2C address 0x48
- NewHaven Display NHD-C0220BiZ-FS(RGB)-FBW-3VM LCD display based on the ST7036 controller, at I2C address 0x3c
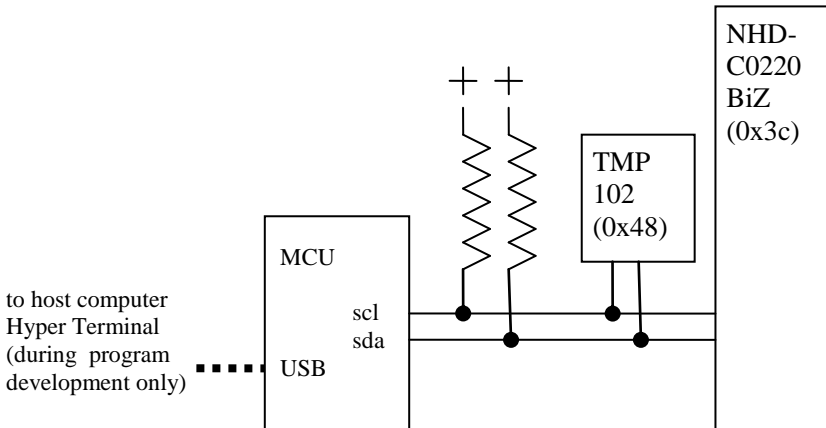
It's as easy as...

1. wire MCU to its embedded circuit
   a. wire MCU pin scl to the temperature sensor, LCD display, and pull-up resistor
   b. wire MCU pin sda to the temperature sensor, LCD display, and pull-up resistor
2. connect a host computer to the USB interface on the MCU (see above)
3. write and debug your BASIC control program, live on the MCU (see below)
4. type "save"
5. type "autorun on"

6. type "run"

The entire LCD digital thermometer BASIC control program is shown below:

```
COM4:9600baud - Tera Term VT
File  Edit  Setup  Control  Window  Help
Welcome to StickOS for Freescale MCF52252 v1.80!
Copyright (c) 2008-2010; all rights reserved.
http://www.cpustick.com
support@cpustick.com
(checksum 0xcd57)
>    10 dim temp, line1$[32], line2$[32], blink$[2]
>    20 let blink$ = " *"
>    30 gosub initdisplay
>    40 while 1 do
>    50   gosub gettemp temp
>    60   vprint line1$ = temp, "degrees C"
>    70   vprint line2$ = temp*9/5+32, "degrees F", blink$[seconds%2:1]
>    80   gosub display line1, line2
>    90   sleep 500 ms
>   100 endwhile
>   110 end
>   120 rem --- gettemp ---
>   130 sub gettemp temp
>   140   dim cmd as byte, rsp[2] as byte
>   150   let cmd = 0
>   160   i2c start 0x48
>   170   i2c write cmd
>   180   i2c read rsp
>   190   i2c stop
>   200   let temp = rsp[0]
>   210 endsub
>   220 rem --- display ---
>   230 sub display line1, line2
>   240   dim cmd1 as byte, data as byte, cmd2 as byte
>   250   let cmd1 = 0x80, data = 0x2, cmd2 = 0x40
>   260   i2c start 0x3c
>   270   i2c write cmd1, data, cmd2, line1
>   280   i2c stop
>   290   let cmd1 = 0x80, data = 0xc0, cmd2 = 0x40
>   300   i2c start 0x3c
>   310   i2c write cmd1, data, cmd2, line2
>   320   i2c stop
>   330 endsub
>   340 rem --- initdisplay ---
>   350 sub initdisplay
>   360   dim i, init[10] as byte
>   370   for i = 1 to init#
>   380     read init[i-1]
>   390   next
>   400   i2c start 0x3c
>   410   i2c write init
>   420   i2c stop
>   430   sleep 100 ms
>   440 endsub
>   450 data 0, 0x38, 0x39, 0x14, 0x78, 0x5e, 0x6d, 0xc, 0x1, 0x6
> ▉
```

- Line 10 declares four RAM variables: an integer to hold the current temperature in degrees C, two strings to represent the two lines of the display, and a third string used to blink an "activity indicator" on the display every two seconds.

- Line 20 initializes the activity indicator string to contain a space and an asterisk; these characters will be alternated on the right hand side of the second display line every other second.
- Line 30 initializes the LCD display by calling the "initdisplay" subroutine.
- Lines 40-100 are the main program loop:
  - first, we get the current temperature by calling the "gettemp" subroutine,
  - then, we format a string for the first line of the display in degrees Celsius,
  - then, we format a string for the second line of the display in degrees Fahrenheit, and include the activity indicator, and
  - finally, we display both lines by calling the "display" subroutine.
- Lines 130-210 are the "gettemp" subroutine, which use the I2C protocol on the temperature sensor to extract degrees Celsius
- Lines 230-330 are the "display" subroutine, which use the I2C protocol on the LCD display to display two lines of text
- Lines 350-440 are the "initdisplay" subroutine, which use the I2C protocol to initialize the LCD display
- Line 450 is read-only data used by the "initdisplay" subroutine to initialize the LCD display.

Then:

- "save" saves the program to non-volatile flash memory.
- "autorun on" sets the program to run automatically when the MCU is powered up.
- Finally, "run" runs the program.

Here is the LCD digital thermometer in action:
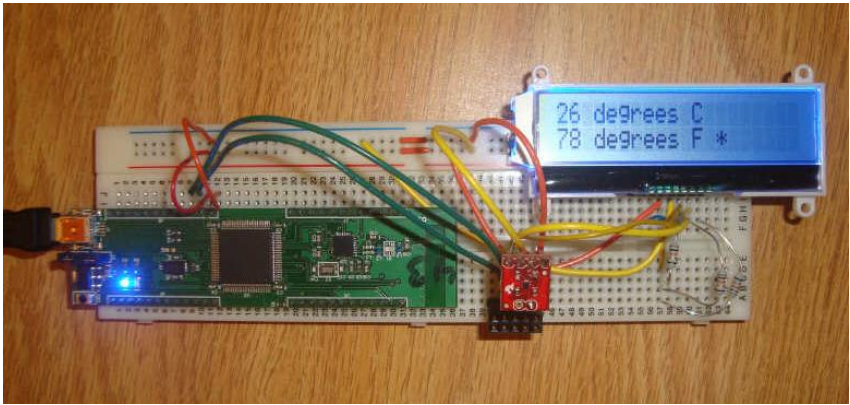
2.3 Wireless Embedded Systems Made Just as Easy!

With the aid of an MC13201 ZigFlea Wireless Transceiver, a simple wireless embedded system, like a remote LED dimmer, can be brought online just as easily as a local embedded system!
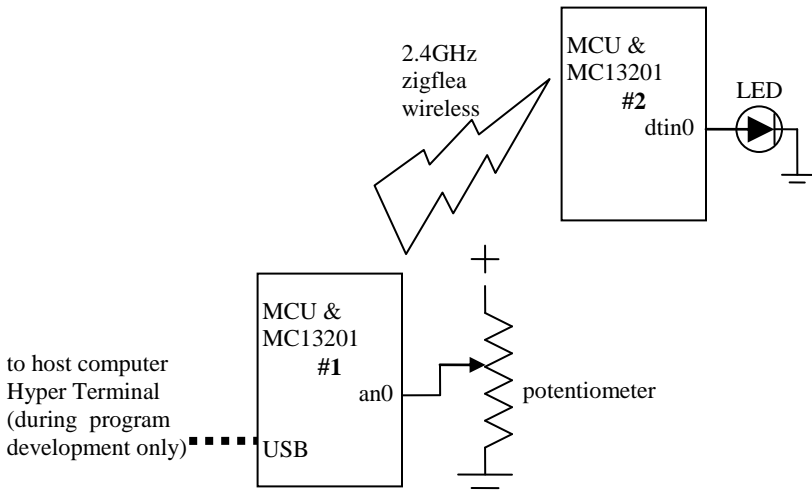
It's as easy as...

7. set the 2.4GHz zigflea wireless nodeid on each MCU
8. wire MCU #1 to its embedded circuit
      c. wire MCU #1 pin an0 to the potentiometer
9. wire MCU #2 to its embedded circuit
      a. wire MCU #2 pin dtin0 to the LED
10. connect a host computer to the USB interface on MCU #1 (see above)
11. write and debug your BASIC control program, live on MCU #1 (see below)
12. use the 2.4GHz zigflea wireless transport to connect to MCU #2
13. write and debug your BASIC control program, live on MCU #2 (see below)
14. run the program on MCU #2
15. disconnect from MCU #2
16. run the program on MCU #1

The entire debugging session, including the writing and running of both MCU's BASIC control programs, is shown below:


```
COM3:9600baud - Tera Term VT

File  Edit  Setup  Control  Window  Help

Welcome to StickOS for Freescale MCF52252 v1.80!
Copyright (c) 2008-2010; all rights reserved.
http://www.cpustick.com
support@cpustick.com
(checksum 0xcd57)
> nodeid
1
> 10 dim potentiometer as pin an0 for analog input
> 20 dim led as remote on nodeid 2
> 30 while 1 do
> 40    let led = potentiometer
> 50    sleep 100 ms
> 60 endwhile
> save
> autorun on
> connect 2
press Ctrl-D to disconnect...

Welcome to StickOS for Freescale MCF52252 v1.80!
Copyright (c) 2008-2010; all rights reserved.
http://www.cpustick.com
support@cpustick.com
(checksum 0xcd57)
> nodeid
2
> 10 dim led as pin dtin0 for analog output
> 20 while 1 do
> 30 endwhile
> save
> autorun on
> run
...disconnected
> nodeid
1
> run
█
```

Note that all of this debugging session is occurring on the Hyper Terminal connected to the USB interface on MCU #1!

First we write the program on MCU #1.

- Notice in line 10 that we declare a local pin variable named "potentiometer" to read the value of the potentiometer, through analog input pin *an0*, in millivolts.
- Then, in line 20, we declare a *remote pin variable* to control the LED on MCU #2 (through MCU #2's local pin variable!); the "as remote on nodeid 2" indicates that the real variable declaration is found on MCU #2.

- Then we simply enter an infinite loop reading the value of the potentiometer (again, in millivolts) every 100ms, and writing it to the LED on MCU #2.

We then save the program to flash memory on MCU #1 and configure it to run automatically when the MCU powers up.

Then we remotely connect to MCU #2 and write its program.

- Notice in line 10 that we declare a local pin variable named "led" to control the LED, through analog output pin *dtin0*, in millivolts.
- Then we simply enter an infinite loop, waiting for our local pin variable to be written remotely from MCU #1 every 100ms!

We then save the program to flash memory on MCU #2 and configure it to run automatically when the MCU powers up.

Finally, we run the program on MCU #2, disconnect from MCU #2 by pressing **<Ctrl-D>**, and run the program on MCU #1.

At this point, adjusting the potentiometer on MCU #1 causes the LED brightness on MCU #2 to be correspondingly adjusted, after a 100ms delay!!!

2.4 More Examples

2.4.1 Digital I/O Example

As a simple example, the following BASIC program generates a 1 Hz square wave on the "dtin0" pin:

```
> 10 dim square as pin dtin0 for digital output
> 20 while 1 do
> 30   let square = !square
> 40   sleep 500 ms
> 50 endwhile
> run
<Ctrl-C>
STOP at line 40!
> _
```

Press **<Ctrl-C>** to stop the program.

Line 10 configures the "dtin0" pin for digital output, and creates a variable named "square" whose updates are reflected at that pin. Line 20 starts an infinite loop (typically MCU programs run forever). Line 30 inverts the state of the dtin0 pin from its previous state -- note that you can examine as well as manipulate the (digital or analog or servo or frequency) output pins. Line 40 just delays the program execution for one half second. And finally line 50 ends the infinite loop.

If we want to run the program in a slightly more demonstrative way, we can use the "trace on" command to show every executed line and variable modification as it occurs:

```
> trace on
> run
  10 dim square as pin dtin0 for digital output
  20 while 1 do
  30   let square = !square
   let square = 0
  40   sleep 500 ms
  50 endwhile
  20 while 1 do
  30   let square = !square
   let square = 1
  40   sleep 500 ms
  50 endwhile
```

```
  20 while 1 do
  30   let square = !square
    let square = 0
  40   sleep 500 ms
```
<Ctrl-C>
```
STOP at line 40!
```
> **trace off**
```
> _
```

Again, press **<Ctrl-C>** to stop the program.

Note that almost all statements that can be run in a program can also be run in "immediate" mode, at the command prompt. For example, after having run the above program, the "square" variable (and dtin0 pin) remain configured, so you can type:

> **print "square is now", square**
```
square is now 0
```
> **let square = !square**
> **print "square is now", square**
```
square is now 1
> _
```

This also demonstrates how you can examine or manipulate variables (or pins!) at the command prompt during program debug.

2.4.2 Analog I/O Example

The MCU can perform analog I/O as simply as digital I/O.

The following BASIC program takes a single measurement of an analog input at pin "an0" and displays it:

```
> new
> 10 dim potentiometer as pin an0 for analog input
> 20 print "potentiometer is", potentiometer
> run
potentiometer is 2026
> _
```

Note that analog inputs and outputs are represented by integers in units of millivolts (mV).

Note that almost all statements that can be run in a program can also be run in "immediate" mode, at the command prompt. For example, after having run the above program, the "potentiometer" variable (and an0 pin) remain configured, so you can type:

```
> print "potentiometer is now", potentiometer
potentiometer is now 2027
> _
```

This also demonstrates how you can examine variables (or pins!) at the command prompt during program debug.

2.4.3 Servo I/O Example

The MCU can perform servo I/O as simply as digital or analog I/O.

The following program moves a servo on pin "dtin1" from one extreme (assumed calibrated to a 0.5ms pulse) to the other (assumed calibrated to a 2.5ms pulse) over the period of a second, using the default servo frequency of 45Hz:

```
> new
> servo
45
> 10 dim servo as pin dtin1 for servo output
> 20 for servo = 50 to 250 step 10
> 30   sleep 50 ms
> 40 next
> run
> _
```

Note that servo outputs are represented by integers in units of centi-milliseconds (1/100th of a millisecond), so we're generating pulses at 45Hz, and they start at 0.5ms and increase to 2.5ms.

Note that almost all statements that can be run in a program can also be run in "immediate" mode, at the command prompt. For example, after having run the above program, the "servo" variable (and dtin1 pin) remain configured, so you can type the following to return the servo to the other extreme position:

```
> let servo = 50
> _
```

2.4.4 Frequency I/O Example

The MCU can perform frequency I/O as simply as digital or analog I/O.

The following BASIC program generates a 1kHz square wave on a frequency output pin "dtin0" for 1 second:

```
> new
> 10 dim audio as pin dtin0 for frequency output
> 20 let audio = 1000
> 30 sleep 1 s
> 40 let audio = 0
> run
> _
```

Note that frequency outputs are represented by integers in units of hertz (Hz).

Note that almost all statements that can be run in a program can also be run in "immediate" mode, at the command prompt. For example, after having run the above program, the "audio" variable (and dtin0 pin) remain configured, so you can type:

```
> print "audio is now", audio
audio is now 0
> let audio = 2000
> print "audio is now", audio
audio is now 2000
> _
```

This also demonstrates how you can examine or manipulate variables (or pins!) at the command prompt during program debug.

2.4.5 UART I/O Example

The MCU can perform serial uart I/O as simply as digital or analog I/O.

The following BASIC program configures a uart for loopback mode, transmits two characters and then asserts it receives them correctly:

```
> new
> 10 configure uart 0 for 9600 baud 7 data even parity loopback
> 20 dim tx as pin utxd0 for uart output
> 30 dim rx as pin urxd0 for uart input
> 40 let tx = 48
> 50 let tx = 49
> 60 while tx do
> 70 endwhile
> 80 assert rx==48
> 90 assert rx==49
> 100 assert rx==0
> 110 print "ok!"
> run
ok!
> _
```

Line 10 configures uart 0 for 9600 baud loopback operation. Lines 20 and 30 configure the "utxd0" and "urxd0" pins for uart output and input, and creates two variable named "tx" and "rx" bound to those pins. Line 40 sends a character ('0', ASCII 48) out the uart and line 50 sends another ('1', ASCII 49). Line 60 waits until all characters are sent (when "tx" reads back 0). Line 80 and 90 then receive two characters from the uart and assert they are what we sent. Line 100 then asserts there are no more characters received ("rx" reads back 0).

The uart can also be controlled using interrupts rather than polling. The following program shows this:

```
> 10 configure uart 0 for 9600 baud 7 data even parity loopback
> 20 dim tx as pin utxd0 for uart output
> 30 dim rx as pin urxd0 for uart input
> 40 on uart 0 input do print "received", rx
> 50 let tx = 48, tx=49
> 60 sleep 1 s
> run
received 48
received 49
> _
```

Please note the pin variable method of accessing UART I/O should not be used on PIC32; see [below](#).

2.4.6 I2C Master I/O Example

The MCU can perform serial I2C master I/O as simply as digital or analog I/O.

The following BASIC program configures I2C to talk to a TI TMP102 temperature sensor at address 0x48. It displays the current temperature, in degrees Celsius:

```
> list
  10 dim r as byte, t[2] as byte
  20 let r = 0
  30 i2c start 0x48
  40 i2c write r
  50 i2c read t
  60 i2c stop
  70 print t[0]
end
> run
25
> _
```

Line 10 just dimensions a byte sized variable for the i2c command and a 2-byte sized array for the response; note that I2C transfers are sized by the variables specified in the i2c statement. Line 20 sets the command byte to 0. Line 30 starts the i2c transaction to the temperature sensor at address 0x48. Line 40 sends the command and line 50 reads the response. Line 60 completes the i2c transaction. Finally, line 70 prints the temperature, in degrees Celsius.

Note that i2c statements can also be run in immediate mode, allowing you to interactively discover the way your i2c peripherals work!!!

2.4.7 QSPI Master I/O Example

The MCU can perform serial QSPI master I/O as simply as digital or analog I/O.

The following BASIC program configures QSPI to talk to the EzPort of another MCU via QSPI, assuming a clone cable is attached. It enables flash memory writes and then queries the status register:

```
> list
  10 dim nrsti as pin scl for digital output
  20 dim ncs as pin qspi_cs0 for digital output
  30 dim cmd as byte, status as byte
  40 rem pulse rsti* low with cs*
  50 let ncs = 0, nrsti = 0, nrsti = 1
  60 sleep 100 ms
  70 let ncs = 1
  80 rem send write enable command
  90 let cmd = 0x6
 100 let ncs = 0
 110 qspi cmd
 120 let ncs = 1
 130 rem send read status register command
 140 let cmd = 0x5
 150 let ncs = 0
 160 qspi cmd, status
 170 let ncs = 1
 180 print hex status
end
> run
0x2
> _
```

Line 10 configures a digital output pin to reset the target MCU. Line 20 configures a digital output pin to drive the MCU chip select, for use with EzPort. Line 30 just dimensions two byte sized variables for use below; note that QSPI transfers are sized by the variables specified in the qspi statement. Lines 40 thru 70 reset the target MCU. Lines 80 thru 120 send a one byte "write enable" command, with chip select. Lines 130 thru 170 send a one byte "read status register" command and receive a one byte status, with chip select. Line 180 prints the status, which shows writes are enabled but the configuration register is not yet loaded.

Note that qspi statements can also be run in immediate mode, allowing you to interactively discover the way your qspi peripherals work!!!

3 MCU Connections

3.1 Interface

When the StickOS is running the "heartbeat" LED will blink slowly; when the BASIC program in the MCU is running, the "heartbeat" LED will blink quickly.

Holding the "autorun disable" switch depressed during power-on prevents autorun of the BASIC program. It also disables "usbhost" mode (enabling CDC/ACM device mode), resets the serial console baud rate to 9600 baud, and overrides static IP address assignment in favor of DHCP, in an effort to allow you to regain control of the MCU.

Use the **help pins** command to see the list of MCU pin names, and the **pins** command to see their LED and switch assignments.

3.2 External Pins

All MCU external pins support general purpose digital input or output.

In addition, certain external pins can support analog input, analog output (PWM actually), frequency output, UART input, UART output, I2C master input/output, and/or QSPI master input/output.

Use the **help pins** command to see the list of MCU pin names and their capabilities.

3.3 Command-Line Transports

StickOS is controlled via a terminal emulator program, such as Windows Hyper Terminal (typically found under Start -> All Programs -> Accessories -> Communications -> Hyper Terminal), thru one of the following command-line transports:

- USB, via a CDC/ACM Virtual COM port
- Ethernet, via a raw socket on port 1234
- UART, via a physical COM port

When using Hyper Terminal, if the USB or Ethernet connection is lost (such as when you unplug and re-plug in the MCU), press the "Disconnect" button followed by the "Call" button, to reconnect Hyper Terminal.

Note that if you do not have Hyper Terminal (the XP version runs fine on Windows 7, BTW), my favorite terminal emulator program is "Tera Term", available free from http://logmett.com/.

On Mac you can also just use the "screen" command under Terminal.

On Linux "minicom" works from http://alioth.debian.org/projects/minicom/.

Note that as of Windows 7, I experience USB hangs with putty when typing or pasting text into the terminal emulator. This appears to be a putty issue, as I experience the exact same hangs when talking to an FTDI chip, a physical serial port, or multiple CDC/ACM serial ports, and I don't experience the hangs with Hyper Terminal, SecureCRT, or Tera Term. If you experience these hangs, I suggest you try Tera Term, available free from http://logmett.com/.

3.3.1 USB Transport

Windows

Before connecting the MCU to a Windows USB host computer, install the cpustick.inf file by saving it to a file, right-clicking on the file, and selecting "Install"; you can ignore warnings about an unsigned driver package -- the driver is straight from Microsoft, and only the INF file itself is unsigned. The INF file allows Windows to bind a human readable name, "CPUStick", to the USB VID/PID presented to the host by StickOS. The latest version of the cpustick.inf file can always be found at: http://www.cpustick.com/cpustick.inf

cpustick.inf file

```
[Version]
Signature="$Windows NT$"
Class=Ports
ClassGuid={4D36E978-E325-11CE-BFC1-08002BE10318}
Provider=%ProviderName%
DriverVer=10/15/2009,1.0.0.0

[MANUFACTURER]
%ProviderName%=DeviceList, NTx86, NTamd64

[DeviceList.NTx86]
%CPUStick%=DriverInstall,USB\VID_0403&PID_A660

[DeviceList.NTamd64]
%CPUStick%=DriverInstall,USB\VID_0403&PID_A660

[DefaultInstall]
CopyInf=cpustick.inf

[DriverInstall]
include=mdmcpq.inf
CopyFiles=FakeModemCopyFileSection
AddReg=LowerFilterAddReg,SerialPropPageAddReg

[DriverInstall.Services]
include = mdmcpq.inf
AddService = usbser, 0x00000002, LowerFilter_Service_Inst

; This adds the serial port property tab to the device properties dialog
[SerialPropPageAddReg]
HKR,,EnumPropPages32,,"MsPorts.dll,SerialPortPropPageProvider"

[Strings]
ProviderName = "www.cpustick.com"
CPUStick = "CPUStick"
```

When the MCU is then connected to the USB host computer, it will present a CDC/ACM Serial Port function to the host computer. An appropriate driver (usbser.sys) will be loaded automatically from microsoft.com, if needed.

Please note one Windows peculiarity with usbser.sys and CDC/ACM Serial Port functions... If you have the virtual COM port held open by an application (such as a terminal emulator) and then disconnect and reconnect the MCU from the USB, when Windows re-creates the virtual COM port, it will not re-create the \DosDevices symbolic link, leaving the new (i.e., working!) virtual COM port inaccessible. To avoid this, close all applications using the virtual COM port before disconnecting and reconnecting the MCU from the USB (or close them after and then disconnect and reconnect the MCU from the USB again).

Once the driver is loaded, a new virtual COM port (VCP) will be present on your system. This virtual COM port will be visible in Device Manager with the "CPUStick" human readable name:
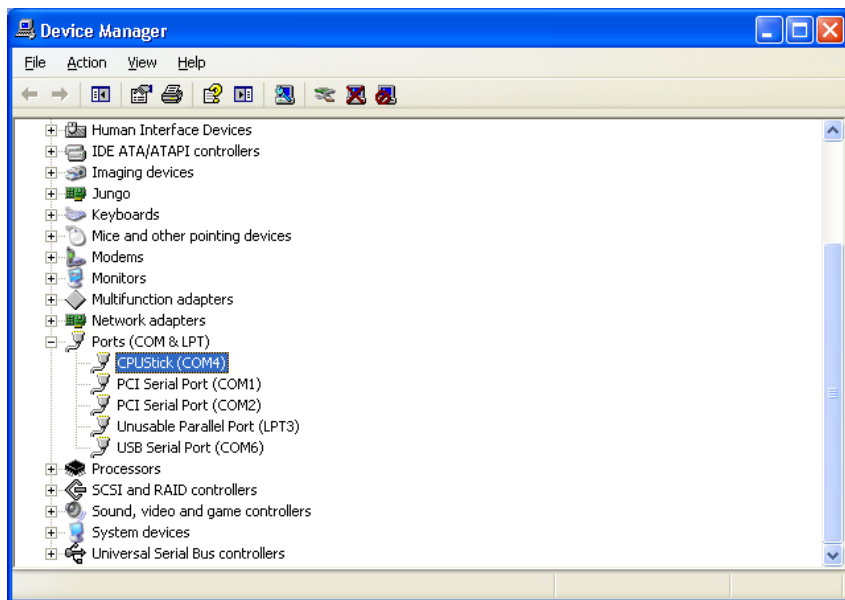
At this point you can use Hyper Terminal to connect to the new virtual COM port. Specify a new connection name, such as "CPUStick CDC", and then select the new virtual COM port under Connect Using; the baud rate and data characteristics in Port Settings are ignored.

Continue reading here.

Mac

When the MCU is connected to the USB host computer, it will present a CDC/ACM Serial Port function to the host computer. An appropriate driver will be loaded.

Once the driver is loaded, a new virtual COM port (VCP) will be present on your system. This virtual COM port will be visible in About This Mac -> More Info... -> Hardware -> USB -> USB Bus with the "CPUStick" human readable name:
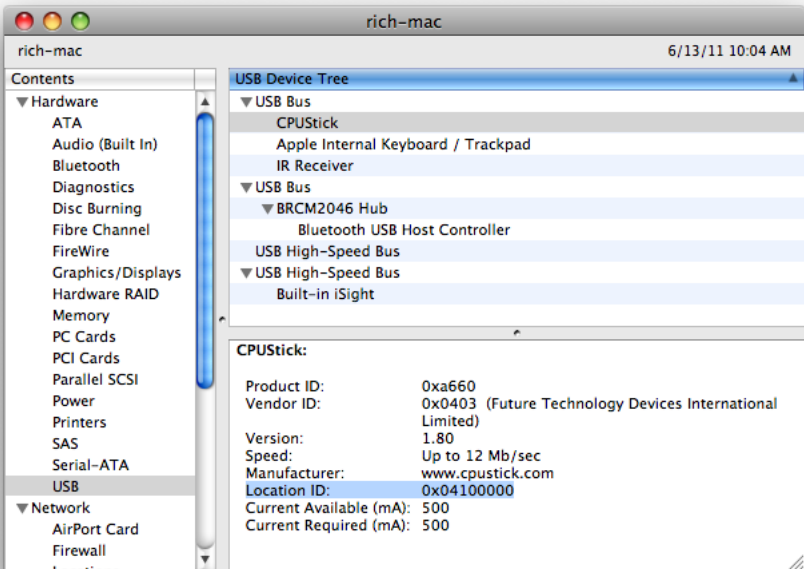
Note the "Location ID" above.

At this point you can use the Terminal program with the "screen" command to connect to the new virtual COM port. Specify a new connection name, such as "CPUStick CDC", and then enter the "screen" command and the new virtual COM port under "Run command"; the baud rate and data characteristics are ignored.
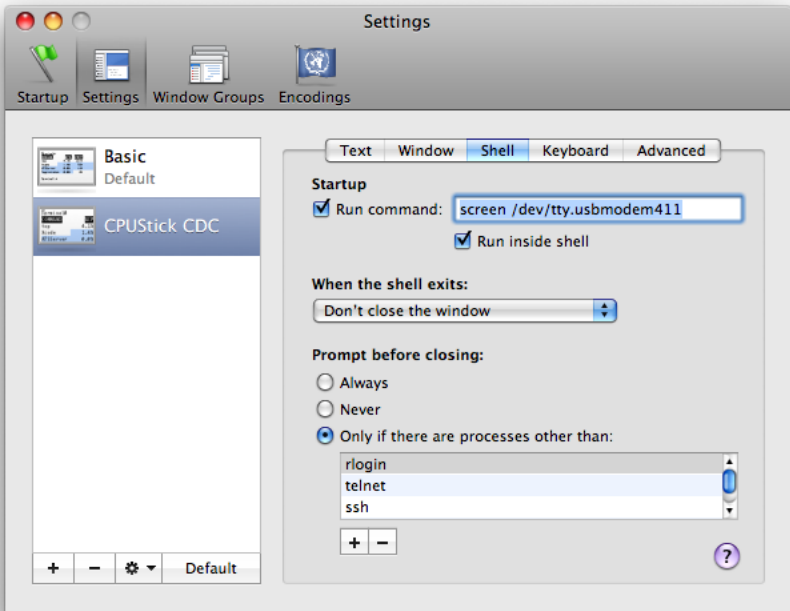
Continue reading here.

Linux

Unfortunately, it seems distributions are different naming their device files. On mine, the dev file is very easy to find -- it has the name ACM in it, like /dev/ttyACM0 -- that is because StickOS presents a CDC/ACM function. The actual number will likely depend on the exact physical USB port you use.
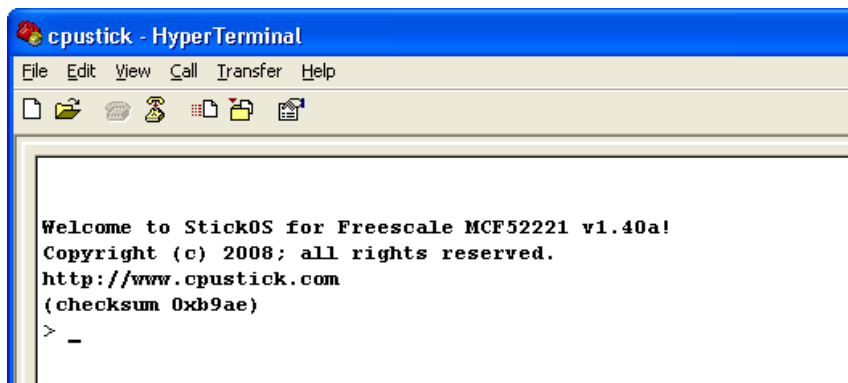
At this point you can use "minicom" to connect to the new virtual COM port. "<Ctrl-A>Z" gets you to a help screen, and "OA" allows you to specify the serial device file; the baud rate and data characteristics are ignored.
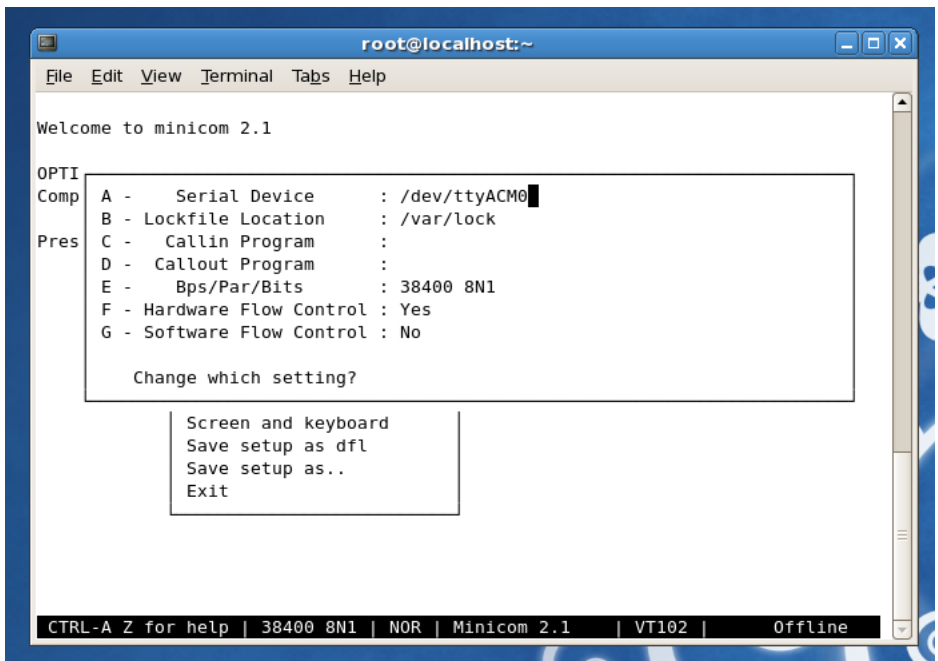
Continue reading here.

All

Press **\<Enter\>** when you are connected and you should see the command prompt:

```
cpustick - HyperTerminal
File  Edit  View  Call  Transfer  Help



Welcome to StickOS for Freescale MCF52221 v1.40a!
Copyright (c) 2008; all rights reserved.
http://www.cpustick.com
(checksum 0xb9ae)
>
```

You are now ready to enter StickOS commands and/or BASIC program statements!

3.3.2 USB Host Mode

If a UART Transport is used, the USB interface on selected MCUs can be configured into host mode to create a trivial USB data logging mechanism to an external USB flash drive.

The state of USB Host mode can be displayed (along with whether a USB flash drive is attached or not), turned on, or turned off with the commands:

```
usbhost
usbhost on
usbhost off
```

This takes effect after the next MCU reset.

You can override the "usbhost" mode and revert to device mode by holding the "autorun disable" switch depressed during power-on.

When USB host mode is turned on and an external USB flash drive is attached and supplied with appropriate VBUS power, all StickOS "print" statement output will be appended to the file `x:\stickos.log` in the USB flash drive, where `x:` is your drive letter.

The write-back cache is flushed every second, so you must wait one second after the last "print" statement before disconnecting the external USB flash drive.

### 3.3.3 UART Transport

Find the physical COM port in Device Manager:
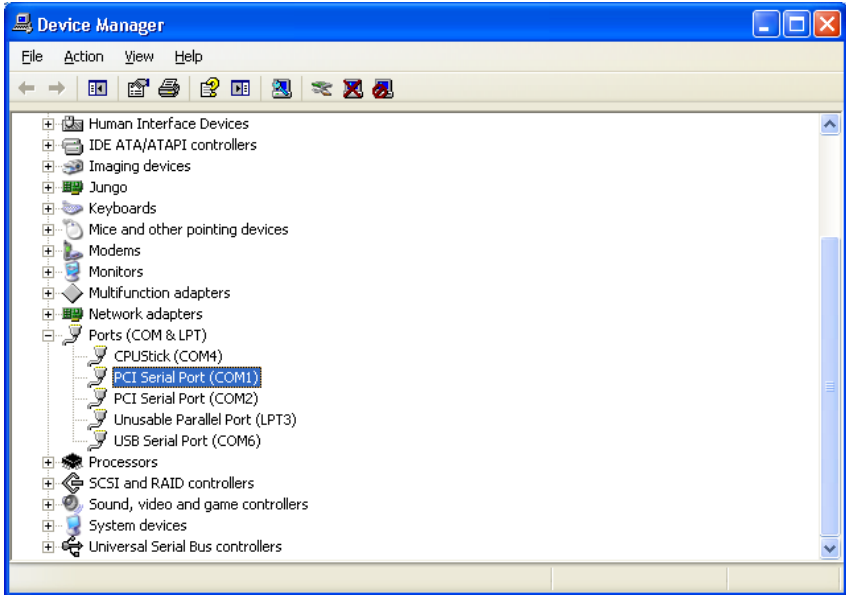
At this point you can use Hyper Terminal to connect to the physical COM port. Specify a new connection name, such as "cpustick", and then select the physical COM port under Connect Using; set the baud rate and data characteristics in Port Settings to:

Bits per second: 9600
Data bits: 8
Parity: None
Stop bits: 2
Flow control: Xon/Xoff

Press **<Enter>** when you are connected and you should see the command prompt:

```
cpustick - HyperTerminal
File  Edit  View  Call  Transfer  Help

Welcome to StickOS for Freescale MCF52221 v1.40a!
Copyright (c) 2008; all rights reserved.
http://www.cpustick.com
(checksum 0xb9ae)
> _
```
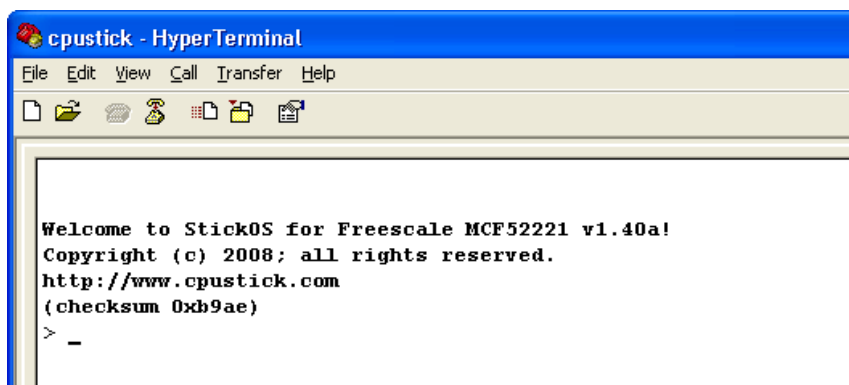
You are now ready to enter StickOS commands and/or BASIC program
statements!

The UART baud rate can be displayed or changed persistent with the
commands:

> **baud**
> **baud** *rate*

This takes effect after the next MCU reset.
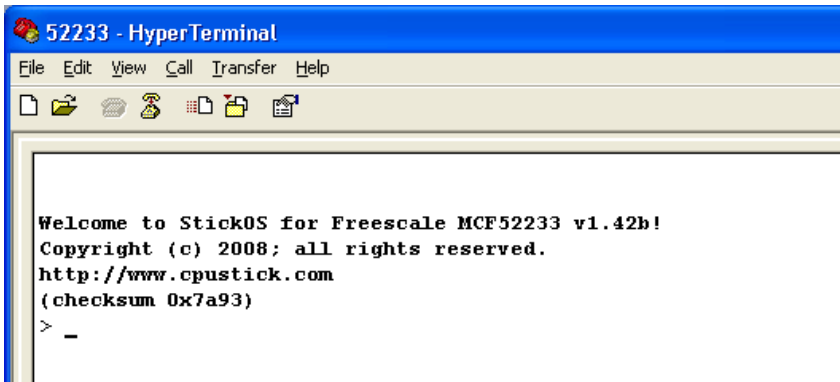
You can override the changed baud rate and revert to 9600 baud by holding
the "autorun disable" switch depressed during power-on.

3.3.4 Ethernet Transport

The MCU will acquire an IP address from DHCP (query your DHCP server to figure out which IP address it got).

At this point you can use Hyper Terminal to connect to the new IP address on TCP port 1234. Specify a new connection name, such as "52233", and then specify "TCP/IP" under Connect Using; then specify the new IP address as Host address and 1234 as Port number.

Press **<Enter>** when you are connected and you should see the command prompt:

You are now ready to enter StickOS commands and/or BASIC program statements!

You can subsequently use the "ipaddress" command to set a static IP address persistently. This takes effect after the next MCU reset.

You can override the static IP address and revert to DHCP by holding the "autorun disable" switch depressed during power-on.

4 StickOS

StickOS supports a BASIC programming environment with integer
variable/array and string support and block structured programming and
subroutine support, where external pins are bound to special "pin variables"
for manipulation or examination.

External pins can be dynamically configured as one of:

        o   digital input or output,
        o   analog input or output (PWM actually),
        o   servo output,
        o   frequency output,
        o   uart input or output,
        o   i2c master input/output, or
        o   qspi master input/output
        o   4-bit HD44780-compatible LCD output
        o   4x4 scanned keypad input

BASIC programs as well as "persistent parameters" can be stored in non-
volatile flash memory; volatile variables as well as recent code edits (up to the
next "save" command) are stored in RAM.

4.1 First Boot & Pin Assignments

When StickOS first boots, certain pin assignments default to "standard" board layouts. Since StickOS runs on any MCU, independent of its board layout, you may need to customize these pin assignments when you first log in if your board is different.

The following pin assignments are supported:

assign	function
heartbeat	indicates the position of the pin attached to the "heartbeat" LED (digital output).
safemode*	indicates the position of the safemode pin attached to the "autorun disable" switch (digital input).
qspi_cs*	indicates the position of the cs* pin used for QSPI transfers for clone and zigflea operations.
clone_rst*	indicates the position of the rst* pin used when cloning firmware to another MCU via EzPort (digital output)
zigflea_rst*	indicates the position of the rst* pin used to reset the MC1320x ZigFlea Transceiver (digital output)
zigflea_attn*	indicates the position of the attn* pin used to wake the MC1320x ZigFlea Transceiver (digital output) Note that this signal is only needed if the MC1320x circuit uses it; StickOS does not need it
zigflea_rxtxen	indicates the position of the rxtxen pin used to activate the MC1320x ZigFlea Transceiver (digital output)

lcd_d4 (v1.82+)	HD44780-compatible LCD data bit (lsb)
lcd_d5 (v1.82+)	HD44780-compatible LCD data bit
lcd_d6 (v1.82+)	HD44780-compatible LCD data bit
lcd_d7 (v1.82+)	HD44780-compatible LCD data bit (msb)
lcd_en (v1.82+)	HD44780-compatible LCD enable
lcd_rs (v1.82+)	HD44780-compatible LCD register select
kbd_s0 (v1.82+)	4x4 scanned keypad scan line (lsb)
kbd_s1 (v1.82+)	4x4 scanned keypad scan line
kbd_s2 (v1.82+)	4x4 scanned keypad scan line
kbd_s3 (v1.82+)	4x4 scanned keypad scan line (msb)
kbd_r0 (v1.82+)	4x4 scanned keypad return line (lsb)
kbd_r1 (v1.82+)	4x4 scanned keypad return line
kbd_r2 (v1.82+)	4x4 scanned keypad return line
kbd_r3 (v1.82+)	4x4 scanned keypad return line (msb)

The default pin assignments may be displayed with the command:

```
pins
```

An individual pin assignment may be displayed with the command:

```
pins assign
```

An individual pin may be reassigned persistently in flash with the command:

```
pins assign none
pins assign pinname
```

Use the following command to see a list of pin names for the MCU:

```
help pins
```

Examples

```
> help pins
pin names:
    0         1         2         3       4     5        6        7
  --------  --------- --------- -------- ----- -------- -------- ------+
  an0       an1       an2       an3      an4   an5      an6      an7  | AN
  scl       sda                                                      | AS
            irq1*                        irq4*                 irq7* | NQ
  qspi_dout qspi_din  qspi_clk  qspi_cs0       qspi_cs2 qspi_cs3     | QS
  dtin0     dtin1     dtin2     dtin3                                | TC
  utxd0     urxd0     urts0*    ucts0*                               | UA
  utxd1     urxd1     urts1*    ucts1*                               | UB

all pins support general purpose digital input/output
an? = potential analog input pins (mV)
dtin? = potential analog output (PWM) pins (mV)
dtin? = potential servo output (PWM) pins (cms)
dtin? = potential frequency output pins (Hz)
urxd? = potential uart input pins (received byte)
utxd? = potential uart output pins (transmit byte)
> pins
heartbeat dtin3
safemode* irq1*
qspi_cs* qspi_cs0
clone_rst* scl
zigflea_rst* an2
zigflea_attn* an3
zigflea_rxtxen an5
> pins heartbeat dtin2
> pins heartbeat
dtin2
> _
```

4.2 Command-Line

In the command and statement specifications that follow, the following
nomenclatures are used:

bold	literal text; enter exactly as shown
italics	parameterized text; enter actual parameter value
(**alternate1**| **alternate2**| **...**)	alternated text; enter exactly one alternate value
regular	displayed by StickOS
<key>	press this key

To avoid confusion with array indices (specified by **[**...**]**), optional text will
always be called out explicitly, either by example or by text, rather than
nomenclated with the traditional [...].

When StickOS is controlled with an ansi or vt100'ish terminal emulator,
command-line editing is enabled via the terminal keys, as follows:

key	function
←	move cursor left
→	move cursor right
↑	recall previous history line
↓	recall next history line
<Home>	move cursor to start of line
<End>	move cursor to end of line
<Backspace>	delete character before cursor
<Delete>	delete character at cursor
<Ctrl-C>	clear line (also stops running program)
<Ctrl-D>	disconnect from remote node (zigflea)
<Enter>	enter line to StickOS

If you enter a command or statement in error, StickOS will indicate the
position of the error, such as:

```
> print i forgot to use quotes
error -    ^
> _
```

4.2.1 StickOS Commands

StickOS commands are used to control the StickOS BASIC program. Unlike BASIC program statements, StickOS commands cannot be entered into the StickOS BASIC program with a line number.

4.2.2 Getting Help

The help command displays the top level list of help topics:

help

To get help on a subtopic, use the command:

help *subtopic*

Examples

```
> help
for more information:
  help about
  help commands
  help modes
  help statements
  help blocks
  help devices
  help expressions
  help strings
  help variables
  help pins
  help clone
  help zigflea

see also:
  http://www.cpustick.com
> help commands
<Ctrl-C>                   -- stop program
auto <line>                -- automatically number program lines
clear [flash]              -- clear ram [and flash] variables
cls                        -- clear terminal screen
cont [<line>]              -- continue program from stop
delete ([<line>][-][<line>]|<subname>) -- delete program lines
download <slave Hz>        -- download flash to slave MCU
dir                        -- list saved programs
edit <line>                -- edit program line
help [<topic>]             -- online help
list ([<line>][-][<line>]|<subname>) -- list program lines
load <name>                -- load saved program
memory                     -- print memory usage
new                        -- erase code ram and flash memories
profile ([<line>][-][<line>]|<subname>) -- display profile info
purge <name>               -- purge saved program
renumber [<line>]          -- renumber program lines (and save)
reset                      -- reset the MCU!
run [<line>]               -- run program
save [<name>]              -- save code ram to flash memory
undo                       -- undo code changes since last save
upgrade                    -- upgrade StickOS firmware!
uptime                     -- print time since last reset

for more information:
  help modes
> _
```

4.2.3 Entering Programs

To enter a statement into the BASIC program, precede it with a line number identifying its position in the program:

```
line statement
```

If the specified line already exists in the BASIC program, it is overwritten.

To delete a statement from the BASIC program, enter just its line number:

```
line
```

To edit an existing line of the BASIC program via command-line editing, use the command:

```
edit line
```

A copy of the unchanged line is also stored in the history buffer.

Note that statements are initially entered into a RAM buffer to avoid excessive writes to flash memory, and therefore can be lost if the MCU is reset or loses power before the program has been saved. When a program is run, the (newly edited) statements in RAM are seamlessly merged with the (previously saved) statements in flash memory, to give the appearance of a single "current program", at a slight performance penalty. When the newly edited program is subsequently saved again, the merged program is re-written to flash and the RAM buffer is cleared, resulting in maximum program performance. If the RAM buffer fills during program entry, an "auto save" is performed to accelerate the merging process.

To automatically number program lines as you enter them, use the command:

```
auto
auto line
```

Enter two blank lines to terminate automatic line numbering.

Note that you can edit a BASIC program in a text editor, without line numbers, and then paste it into the terminal emulator window with automatic line numbering, and then enter two blank lines to terminate automatic line numbering.

To list the BASIC program, or a range of lines from the BASIC program, use the command:

```
list
list line
list -line
list line-
list line-line
```

Alternately, you can list an entire subroutine by name with the command:

```
list subname
```

To set the listing indent mode, use the command:

```
indent (on|off)
```

To display the listing indent mode, use the command:

```
indent
```

If the listing indent mode is on, nested statements within a block will be indented by two characters, to improve program readability.

To set the line numbering mode, use the command:

```
numbers (on|off)
```

To display the line numbering mode, use the command:

```
numbers
```

Note that unnumbered listings are useful to paste back in to the "auto" command which automatically supplies line numbers to program statements.

To delete a range of lines from the BASIC program, use the command:

```
delete line
delete -line
delete line-
delete line-line
```

Alternately, you can delete an entire subroutine by name with the command:

```
delete subname
```

To undo changes to the BASIC program since it was last saved (or renumbered, or new'd, or loaded), use the command:

```
undo
```

To save the BASIC program permanently to flash memory, use the command:

```
save
```

Note that any unsaved changes to the BASIC program will be lost if the MCU is reset or loses power.

To renumber the BASIC program by 10's and save the BASIC program permanently to flash memory, use the command:

```
renumber
```

To delete all lines from the BASIC program, use the command:

```
new
```

Examples

```
> 10 dim a
> 20 for a = 1 to 10
> auto 30
> 30 print a
> 40 next
> 50
> 60
> save
> list 20-40
  20 for a = 1 to 10
  30   print a
  40 next
end
> delete 20-40
> list
  10 dim a
end
> undo
> list
  10 dim a
  20 for a = 1 to 10
  30   print a
  40 next
end
> 1 rem this is a comment
> list
   1 rem this is a comment
  10 dim a
  20 for a = 1 to 10
  30   print a
  40 next
end
> renumber
> list
  10 rem this is a comment
  20 dim a
  30 for a = 1 to 10
  40   print a
  50 next
end
> new
> list
end
> _
```

4.2.4 Running Programs

To run the BASIC program, use the command:

```
run
```

Alternately, to run the program starting at a specific line number, use the command:

```
run line
```

To stop a running BASIC program, press:

```
<Ctrl-C>
```

To continue a stopped BASIC program, use the command:

```
cont
```

Alternately, to continue a stopped BASIC program from a specific line number, use the command:

```
cont line
```

To set the autorun mode for the saved BASIC program, use the command:

```
autorun (on|off)
```

This takes effect after the next MCU reset.

To display the autorun mode for the saved BASIC program, use the command:

```
autorun
```

If the autorun mode is on, when the MCU is reset, it will start running the saved BASIC program automatically.

Note that any unsaved changes to the BASIC program will be lost if the MCU is reset or loses power.

Examples

```
> 10 dim a
> 20 while 1 do
> 30 let a = a+1
> 40 endwhile
> save
> run
<Ctrl-C>
STOP at line 40!
> print a
5272
> cont
<Ctrl-C>
STOP at line 30!
> print a
11546
> autorun
off
> autorun on
> _
```

4.2.5 Loading and Storing Programs

The "current program" has no name and is saved and run by default. In addition to the current program, StickOS can load and store a number of named BASIC programs in a flash filesystem. Named programs are simply copies of the current program that can be retrieved at a later time, but are otherwise unaffected by all other StickOS commands than these.

To display the list of currently stored named programs, use the command:

        dir

To store the current program under the specified name, use the command:

        save *name*

To load a named stored program to become the current program, use the command:

        load *name*

To purge (erase) a stored program, use the command:

        purge *name*

Examples
```
> 10 dim a
> 20 while 1 do
> 30 let a = a+1
> 40 endwhile
> dir
> save spinme
> dir
spinme
> new
> list
end
> load spinme
> list
  10 dim a
  20 while 1 do
  30   let a = a+1
  40 endwhile
end
> purge spinme
> dir
> _
```

4.2.6 Debugging Programs

There are a number of techniques you can use for debugging StickOS BASIC programs.

The simplest debugging technique is simply to insert print statements in the program at strategic locations, and display the values of variables.

A more powerful debugging technique is to insert one or more breakpoints in the program, with the following statement:

>   *line* **stop**

When program execution reaches line, the program will stop and then you can use immediate mode to display or modify the values of any and all variables.

To continue a stopped BASIC program, use the command:

>   **cont**
>   **cont** *line*

An even more powerful debugging technique is to insert one or more conditional breakpoints in the program, with the following statement:

>   *line* **assert** *expression*

When the program execution reaches line, expression is evaluated, and if it is false (i.e., 0), the program will stop and you can use immediate mode to display or modify the values of any and all variables.

Again, to continue a stopped BASIC program, use the command:

>   **cont**
>   **cont** *line*

The most powerful debugging technique, though also the most expensive in terms of program performance, is to insert a watchpoint expression in the program, with the following statement

>   *line* **on** *expression* **do** *statement*

The watchpoint expression is re-evaluated before every line of the program is executed; if the expression transitions from false to true, the watchpoint statement handler runs.

When debugging, the statement handler is typically a "stop" statement, such as:

```
line on expression do stop
```

This will cause the program to stop as soon as the specified expression becomes true, such as when a variable or pin takes on an incorrect value.

To set the smart watchpoint mode, which dramatically reduces watchpoint overhead at a slight delay of input pin sensitivity, use the command:

```
watchsmart (on|off)
```

To display the smart watchpoint mode, use the command:

```
watchsmart
```

At any time when a program is stopped, you can enter BASIC program statements at the command-line with no line number and they will be executed immediately; this is called "immediate mode". This allows you to display the values of variables, with an immediate mode statement like:

```
print expression
```

It also allow you to modify the value of variables, with an immediate mode statement like:

```
let variable = expression
```

Note that if an immediate mode statement references a pin variable, the live MCU pin is examined or manipulated, providing a very powerful debugging technique for the embedded system itself!

Thanks to StickOS's transparent line-by-line compilation, you can also edit a stopped BASIC program and then continue it, either from where you left off or from another program location!

When the techniques discussed above are insufficient for debugging, two additional techniques exist -- single-stepping and tracing.

To set the single-step mode for the BASIC program, use the command:

```
step (on|off)
```

To display the single-step mode for the BASIC program, use the command:

```
step
```

While single-step mode is on, the program will stop execution after every statement, as if a stop statement was inserted after every line.

Additionally, while single-step mode is on, pressing **\<Enter\>** (essentially entering what would otherwise be a blank command) is the same as the **cont** command.

To set the trace mode for the BASIC program, use the command:

```
trace (on|off)
```

To display the trace mode for the BASIC program, use the command:

```
trace
```

While trace mode is on, the program will display all executed lines and variable modifications while running.

Examples

```
> 10 dim a, sum
> 20 for a = 1 to 10000
> 30 let sum = sum+a
> 40 next
> 50 print sum
> run
50005000
> 25 stop
> run
STOP at line 25!
> print a, sum
1 0
> cont
STOP at line 25!
```

```
> print a, sum
2 1
> 25 assert a != 5000
> cont
assertion failed
STOP at line 25!
> print a, sum
5000 12497500
> cont
50005000
> delete 25
> trace
off
> step
off
> trace on
> step on
> list
  10 dim a, sum
  20 for a = 1 to 10000
  30    let sum = sum+a
  40 next
  50 print sum
end
> run
  10 dim a, sum
STOP at line 10!
> cont
  20 for a = 1 to 10000
    let a = 1
STOP at line 20!
> <Enter>
  30    let sum = sum+a
    let sum = 1
STOP at line 30!
> <Enter>
  40 next
    let a = 2
STOP at line 40!
> <Enter>
  30    let sum = sum+a
    let sum = 3
STOP at line 30!
> _
```

4.2.7 Other Commands

To clear BASIC program variables, and reset all pins to digital input mode, use the command:

```
clear
```

This command is also used after a stopped program has defined program variables and before redefining program variables in "immediate" mode, to avoid duplicate definition errors without having to erase the program with a "new" command.

To clear BASIC program variables, including flash parameters, use the command:

```
clear flash
```

To display the StickOS memory usage, use the command:

```
memory
```

To reset the MCU as if it was just powered up, use the command:

```
reset
```

Note that the reset command inherently breaks the USB or Ethernet connection between the MCU and host computer; press the "Disconnect" button followed by the "Call" button, to reconnect Hyper Terminal.

To clear the terminal screen, use the command:

```
cls
```

To display the time since the MCU was last reset, use the command:

```
uptime
```

Examples

```
> memory
  0% ram code bytes used
  0% flash code bytes used
  0% ram variable bytes used
  0% flash parameter bytes used
  0% variables used
> 10 dim a[100]
> 20 rem this is a loooooooooooooooooooooooooooooooong line
> run
> memory
  4% ram code bytes used (unsaved changes!)
  0% flash code bytes used
 19% ram variable bytes used
  0% flash parameter bytes used
  1% variables used
> save
> memory
  0% ram code bytes used
  1% flash code bytes used
 19% ram variable bytes used
  0% flash parameter bytes used
  1% variables used
> clear
> memory
  0% ram code bytes used
  1% flash code bytes used
  0% ram variable bytes used
  0% flash parameter bytes used
  0% variables used
> list
  10 dim a[100]
  20 rem this is a loooooooooooooooooooooooooooooooong line
end
> uptime
1d 15h 38m
> reset

_
```

4.3 BASIC Program Statements

BASIC Program statements are typically entered into the StickOS BASIC program with an associated line number, and then are executed when the program runs.

Most BASIC program statements can also be executed in immediate mode at the command prompt, without a line number, just as if the program had encountered the statement at the current point of execution.

4.3.1 Variable Declarations

All variables must be dimensioned prior to use. Accessing undimensioned variables results in an error and a value of 0.

Simple RAM variables

Simple RAM variables can be dimensioned as either integer (32 bits, signed, by default), short (16 bits, unsigned), or byte (8 bits, unsigned) with the following statements:

```
dim var
dim var as (short|byte)
```

Multiple variables can be dimensioned in the same statement, by separating them with commas:

```
dim var [as ...], var [as ...], ...
```

If no variable size (**short** or **byte**) is specified in a dimension statement, integer is assumed; if no **as ...** is specified, a RAM variable is assumed.

Array RAM variables

Array RAM variables can be dimensioned with the following statements:

```
dim var[n]
dim var[n] as (short|byte)
```

Note that simple variables are really just array variables with only a single array element in them, so the array element *var***[0]** is the same as *var*, and the dimension **dim** *var***[1]** is the same as **dim** *var*.

String RAM variables

String RAM variables can be dimensioned with the following statements:

```
dim var$[n]
```

Where *n* is the length of the array. Array indices start at 0 and end at the length of the array minus one.

Note also that string variables are really just a nul-terminated view into a byte array variable.

MCU register variables

Variables can also be dimensioned as MCU register variables at absolute addresses with the following statements:

```
dim varabs at address addr

dim varabs as (short|byte) at address addr

dim varabs[n] at address addr

dim varabs[n] as (short|byte) at address addr
```

Note that you can trivially crash your MCU by accessing registers incorrectly.

Persistent integer (32 bits) flash variables

Variables can also be dimensioned as persistent integer (32 bits) flash variables with the following statements:

```
dim varflash as flash

dim varflash[n] as flash
```

Persistent flash variables retain their values from one run of a program to another (even if power is lost between runs), unlike RAM variables which are cleared to 0 at the start of every run.

Note that since flash memory has a finite life (100,000 writes, typically), rewriting a flash variable should be a rare operation reserved for program configuration changes, etc. To attempt to enforce this, StickOS delays all flash variable modifications by 0.5 seconds (the same as all other flash memory updates).

Pin variables

Finally, variables can be dimensioned as pin variables, used to manipulate or examine the state of MCU I/O pins with the following statements:

```
dim varpin as pin pinname for (digital|analog|frequency|uart)\
        (input|output) [debounced] [inverted][open_drain]

dim varpin[n] as pin pinname for (digital|analog|frequency|uart)\
        (input|output) [debounced] [inverted][open_drain]
```

These are discussed in detail below, in the sections on Digital I/O, Analog I/O, Servo I/O, Frequency I/O, and UART I/O.

See also: ZigFlea Remote Variables

Examples

```
> new
> 10 dim array[4], b, volatile
> 20 dim led as pin dtin0 for digital output
> 30 dim potentiometer as pin an0 for analog input
> 40 dim persistent as flash
> 50 for b = 0 to 3
> 60   let array[b] = b*b
> 70 next
> 80 for b = 0 to 3
> 90   print array[b]
> 100   let led = !led
> 110 next
> 120 print "potentiometer is", potentiometer
> 130 print "volatile is", volatile
> 140 print "persistent is", persistent
> 150 let persistent = persistent+1
> run
0
1
4
9
potentiometer is 1745
volatile is 0
persistent is 0
> run
0
1
4
9
potentiometer is 1745
volatile is 0
persistent is 1
> dim pcntr0 as short at address 0x40150004
> print pcntr0
5338
> print pcntr0
2983
> _
```

4.3.2 System Variables

The following system variables may be used in expressions or simply with "print" statements to examine internal system state. These variables are all read-only.

analog (v1.82+)	analog supply millivolts
getchar	most recent console character
keychar (v1.82+)	most recent keypad character
nodeid	zigflea nodeid
msecs	number of milliseconds since boot
seconds	number of seconds since boot
ticks	number of ticks since boot
ticks_per_msec	number of ticks per millisecond

Examples

```
> print seconds, ticks, ticks/1000
2640 10562152 10562
>
```

4.3.3 Variable Assignments

Simple variables are assigned with the following statement:

```
let variable = expression
```

If the variable represents an output "pin variable", the corresponding MCU output pin is immediately updated.

Similarly, array variable elements are assigned with the following statement:

```
let variable[expression] = expression
```

Where the first *expression* evaluates to an array index between 0 and the length of the array minus one, and the second *expression* is assigned to the specified array element.

String variables are assigned with the following statement:

```
let variable$ = string
```

Multiple variables may be assigned in a single statement by separating them with commas:

```
let var1 = expr1, var2 = expr2, ...
```

Examples

```
> 10 dim simple, array[4]
> 20 while simple<4 do
> 30   let array[simple] = simple*simple
> 40   let simple = simple+1
> 50 endwhile
> 60 for simple = 0 to 3
> 70   print array[simple]
> 80 next
> run
0
1
4
9
> new
> 10 dim a$[20]
> 20 let a$="hello"+" "+"world!"
> 30 print a$
> run
hello world!
> _
```

4.3.4 Expressions

StickOS BASIC expressions are very similar to C expressions, and follow similar precedence and evaluation order rules.

The following operators are supported, in order of decreasing precedence:

n	decimal constant
0x*n*	hexadecimal constant
'*c*'	character constant
variable	simple variable
*variable***[***expression***]**	array variable element
*variable***#**	length of array or string
( )	grouping
! ~	logical not, bitwise not
* / %	times, divide, mod
+ −	plus, minus
>> <<	shift right, left
<= < >= >	inequalities
== !=	equal, not equal
| ^ &	bitwise or, xor, and
|| ^^ &&	logical or, xor, and

The plus and minus operators can be either binary (taking two arguments, one on the left and one on the right) or unary (taking one argument on the right); the logical and bitwise not operators are unary. All binary operators evaluate from left to right; all unary operators evaluate from right to left.

Note that the **#** operator evaluates to the length of the array or string variable whose name precedes it.

Logical and equality/inequality operators, above, evaluate to 1 if *true*, and 0 if *false*. For conditional expressions, any non-0 value is considered to be *true*, and 0 is considered to be *false*.

If the expression references an input "pin variable", the corresponding MCU input pin is sampled to evaluate the expression.

Note that when StickOS parses an expression and later displays it (such as when you enter a program line and then list it), what you are seeing is a de-compiled representation of the compiled code, since only the compiled code is stored, to conserve RAM and flash memory. So superfluous parenthesis (not to mention spaces) will be removed from the expression, based on the precedence rules above.

Examples

```
> 10 print 2*(3+4)
> 20 print 2+(3*4)
> list
  10 print 2*(3+4)
  20 print 2+3*4
end
> run
14
14
> print 3+4
7
> print -3+2
-1
> print !0
1
> print 5&6
4
> print 5&&6
1
> print 3<5
1
> print 5<3
0
> print 3<<1
6
> dim a[7]
> print a#
7
> _
```

4.3.5 Strings

StickOS supports string variables as a nul-terminated views into byte arrays.

A string variable may be declared, with a maximum length *n*, with:

> **dim** *var***$[***n***]**

A string may then be assigned with:

> **let** *variable***$ =** *string*

Where *string* is an expression composed of one or more of:

"*literal*"	literal string
variable$	variable string
variable$[*start:length*]	variable substring
+	string concatenation operator

A string may be tested in a conditional statement with a condition of the form:

> **if** *string relation string* **then**
> **while** *string relation string* **do**
> **until** *string relation string*

Where *relation* is one of:

<= < >= >	inequalities
== !=	equal, not equal
~ !~	contains, does not contain

The current length of a string can be represented in an integer expression by:

> *variable***#**

Strings may also be explicitly specified in **dim**, **input**, **let**, **print**, and **vprint** statements.

Examples

```
> new
> 10 dim i, a$[10]
> 20 input a$
> 30 for i = 0 to a#-1
> 40 print a$[i:1]
> 50 next
> run
? hello
h
e
l
l
o
> new
> 10 dim a$[10]
> 20 input a$
> 30 if a$ ~ "y" then
> 40 print "yes"
> 50 else
> 60 print "no"
> 70 endif
> run
? aya
yes
> run
? aaa
no
>
```

4.3.6 Print Statements

While the MCU is connected to the host computer, print statements can be observed on the Hyper Terminal console window.

Print statements can be used to print integer expressions, using either a decimal or hexadecimal output radix, or printing raw ASCII bytes:

> **print** [**dec**|**hex**|**raw**] *expression* [;]

Or strings:

> **print** *string*

Or various combinations of both:

> **print** *string*, [**dec**|**hex**|**raw**] *expression*, ... [;]

If the *expression* specifies an array, its entire array contents are printed. If the *expression* references an input "pin variable", the corresponding MCU input pin is sampled to evaluate the expression.

A trailing semi-colon (;) suppresses the carriage-return/linefeed that usually follows each printed line.

Note that when the MCU is disconnected from the host computer, print statement output is simply discarded.

Examples

```
> print "hello world"
hello world
> print 57*84
4788
> print hex 57*84
0x12b4
> print 9, "squared is", hex 9*9
9 squared is 0x51
> dim a[2]
> print a
0 0
> print 1;
1> _
```

4.3.7 Variable Print Statements

Variable print statements can be used to convert strings to integers and vice versa, as well as integers from decimal to hexadecimal radix, etc. Basically, variable print statements are identical to print statements, except rather than printing the result to the console, the result is "printed" to a variable.

Variable print statements can be used to print integer expressions, using either a decimal or hexadecimal output radix, or printing raw ASCII bytes:

>   **vprint** *variable*[**$**] **=** [**dec**|**hex**|**raw**] *expression*

Or strings:

>   **vprint** *variable*[**$**] **=** *string*

Or various combinations of both:

>   **vprint** *variable*[**$**] **=** *string*, ****
>   [**dec**|**hex**|**raw**] *expression*, **...**

In all cases, the resulting output is assigned to the specified integer or string variable. If a type conversion error occurs (such as assigning a non-integer string to an integer variable), program execution stops.

Examples

```
> clear
> dim a, b$[10]
> let b$="123"
> vprint a = b$[0:2]+"4"
> print a
124
> vprint b$ = "hello", a
> print b$
hello 124
> _
```

4.3.8 Input Statements

While the MCU is connected to the host computer, input statements can be serviced from the Hyper Terminal console window.

Input statements can be used to input integer expressions, using either a decimal or hexadecimal output radix, or input raw ASCII bytes:

> **input** [**dec**|**hex**|**raw**] *variable*[**$**], ...

If the *variable* specifies an array (or a string), the entire array (or string) contents are input. If the *expression* references an output "pin variable", the corresponding MCU output pin is immediately updated.

When the input statement is serviced, StickOS prints a prompt to the console:

> **?** _

And the user enters integer or string values, as appropriate, followed by the <Enter> key.

Note that while waiting for input, BASIC interrupt handlers continue to run.

Also, the most recent console input character is available in the system variable "getchar", which you will typically use as "getchar$".

Note that when the MCU is disconnected from the host computer, input statements hang the program.

Examples

```
> new
> 10 dim a, b$[20]
> 20 input a, b$
> 30 print a*2, b$
> run
? 12 hello world!
24 hello world!
> _
```

4.3.9 Read/Data Statements

A program can declare read-only data in its code statements, and then consume the data at run-time.

To declare the read-only data, use the **data** statement as many times as needed:

```
data n
data n, n, ...
```

To consume data values and assign them to variables at runtime, use the **read** statement:

```
read variable
read variable, variable, ...
```

If a read is attempted when no more data exists, the program stops with an "out of data" error.

A line may be labeled and the current data consumer pointer may be moved to a specific (labeled) line with the statements:

```
label label
restore label
```

Examples

```
> 10 dim a, b
> 20 data 1, 2, 3
> 30 data 4
> 40 data 5, 6
> 50 data 7
> 60 while 1 do
> 70   read a, b
> 80   print a, b
> 90 endwhile
> 100 data 8
> run
1 2
3 4
5 6
7 8
out of data
STOP at line 70!
> _
```

4.3.10   Conditional Statements

Non-looping conditional statements are of the form:

```
if expression then
      statements
elseif expression then
      statements
else
      statements
endif
```

Where *statements* is one or more program statements and the **elseif** and **else** clauses (and their corresponding *statements*) are optional.

Alternately, the string form of this statement is:

```
if string relation string then
      statements
elseif string relation string then
      statements
else
      statements
endif
```

Examples

```
> 10 dim a
> 20 for a = -4 to 4
> 30   if !a then
> 40     print a, "is zero"
> 50   elseif a%2 then
> 60     print a, "is odd"
> 70   else
> 80     print a, "is even"
> 90   endif
> 100 next
> run
-4 is even
-3 is odd
-2 is even
-1 is odd
0 is zero
1 is odd
2 is even
3 is odd
4 is even
> _
```

4.3.11　　　Looping Conditional Statements

Looping conditional statements include the traditional BASIC for-next loop and the more structured while-endwhile and do-until loops.

The for-next loop statements are of the form:

```
for variable = expression to expression step expression
      statements
next
```

Where *statements* is one or more program statements and the **step** *expression* clause is optional and defaults to 1.

The for-next loop expressions are evaluated only once, on initial entry to the loop. The loop variable is initially set to the value of the first expression. Each time the loop variable is within the range (inclusive) of the first and second expression, the statements within the loop execute. At the end of the loop, if the incremented loop variable would still be within the range (inclusive) of the first and second expression, the loop variable is incremented by the step value, and the loop repeats again. On exit from the loop, the loop variable is equal to the value it had during the last iteration of the loop.

The while-endwhile loop statements are of the form:

```
while expression do
      statements
endwhile
```

Where *statements* is one or more program statements .

Alternately, the string form of this statement is:

```
while string relation string do
      statements
endwhile
```

The while-endwhile loop conditional expression is evaluated on each entry to the loop. If it is true (non-0), the statements within the loop execute, and the loop repeats again. On exit from the loop, the conditional expression is false.

The do-until loop statements are of the form:

```
do
     statements
until expression
```

Where *statements* is one or more program statements .

Alternately, the string form of this statement is:

```
do
     statements
until string relation string
```

The do-until loop conditional expression is evaluated on each exit from the loop. If it is false (0), the loop repeats again. On exit from the loop, the conditional expression is true.

In all three kinds of loops, the loop can be exited prematurely using the statement:

```
break
```

This causes program execution to immediately jump to the statements following the terminal statement (i.e., the **next**, **endwhile**, or **until**) of the innermost loop.

Additionally, multiple nested loops can be exited prematurely together using the statement:

```
break n
```

Which causes program execution to immediately jump to the statements following the terminal statement (i.e., the **next**, **endwhile**, or **until**) of the innermost *n* loops.

Similarly, a loop can be continued, causing execution to resume immediately with the conditional expression evaluation, using the statement:

```
continue
```

This causes program execution to immediately jump to the conditional expression evaluation, at which point the loop may conditionally execute again.

Multiple nested loops can be continued together using the statement:

continue *n*

Which causes program execution to immediately jump to the conditional expression evaluation of the innermost *n* loops.

Examples

```
> 10 dim a, b, sum
> 20 while 1 do
> 30    if a==10 then
> 40      break
> 50    endif
> 60    let sum = 0
> 70    for b = 0 to a
> 80      let sum = sum+b
> 90    next
> 100   print "sum of integers 0 thru", a, "is", sum
> 110   let a = a+1
> 120 endwhile
> run
sum of integers 0 thru 0 is 0
sum of integers 0 thru 1 is 1
sum of integers 0 thru 2 is 3
sum of integers 0 thru 3 is 6
sum of integers 0 thru 4 is 10
sum of integers 0 thru 5 is 15
sum of integers 0 thru 6 is 21
sum of integers 0 thru 7 is 28
sum of integers 0 thru 8 is 36
sum of integers 0 thru 9 is 45
> _
```

4.3.12   Subroutines

A subroutine is called with the following statement:

```
gosub subname [expression, ...]
```

A subroutine is declared with the following statements:

```
sub subname [param, ...]
    statements
endsub
```

The sub can be exited prematurely using the statement:

```
return
```

This causes program execution to immediately return to the statements following the **gosub** statement that called the subroutine.

In general, subroutines should be declared out of the normal execution path of the code, and typically are defined at the end of the program.

Subroutine parameters are essentially variables local to the subroutine which are initialized to the values of the caller's gosub expressions. Simple variable caller's gosub *expression*'s, however, are passed to sub *param*'s by reference, allowing the sub to modify the caller's variables; all other caller's gosub expressions are passed by value.

Note that to force a variable to be passed by value to a subroutine, simply use a trivial expression like "var+0" in the gosub statement expression.

Note also that to return a value from a subroutine, pass in a simple variable (by reference) and have the subroutine modify the corresponding param before it returns.

Any variables dimensioned in a subroutine are local to that subroutine. Local variables hide variables of the same name dimensioned in outer-more scopes. Local variables are automatically un-dimensioned when the subroutine returns.

Examples

```
> 10 dim a
> 20 for a = 0 to 9
> 30   gosub sumit a
> 40 next
> 50 end
> 60 sub sumit numbers
> 70   dim a, sum
> 80   for a = 1 to numbers
> 90     let sum = sum+a
> 100   next
> 110   print "sum of integers 0 thru", numbers, "is", sum
> 120 endsub
> run
sum of integers 0 thru 0 is 0
sum of integers 0 thru 1 is 1
sum of integers 0 thru 2 is 3
sum of integers 0 thru 3 is 6
sum of integers 0 thru 4 is 10
sum of integers 0 thru 5 is 15
sum of integers 0 thru 6 is 21
sum of integers 0 thru 7 is 28
sum of integers 0 thru 8 is 36
sum of integers 0 thru 9 is 45
> new
> 10 dim a
> 20 print a
> 30 gosub increment a
> 40 gosub increment a
> 50 print a
> 60 end
> 70 sub increment value
> 80   let value = value+1
> 90 endsub
> run
0
2
> _
```

4.3.13 Timers

StickOS supports up to four internal interval timers (0 thru 3) for use by the program. Timer interrupts are delivered when the specified time interval has elapsed since the previous interrupt was delivered.

Timer interrupt intervals are configured with the statement:

configure timer *n* **for** *m* (**s**|**ms**|**us**)

This configures timer *n* to interrupt every *m* seconds, milliseconds, or microseconds.

Note that the minimum timer resolution is the clock tick, which is 0.25 milliseconds.

The timer interrupt can then be enabled, and the statement(s) to execute when it is delivered specified, with the statement:

on timer *n* *statement*

If *statement* is a "**gosub** *subname* ...", then all of the statements in the corresponding sub are executed when the timer interrupt is delivered; otherwise, just the single *statement* is executed.

The timer interrupt can later be completely ignored (i.e., discarded) with the statement:

off timer *n*

The timer interrupt can be temporarily masked (i.e., held off but not discarded) with the statement:

mask timer *n*

And can later be unmasked (i.e., any pending interrupts delivered) with the statement:

unmask timer *n*

Examples

```
> 10 dim ticks
> 20 configure timer 0 for 1000 ms
> 30 on timer 0 do print "slow"
> 40 configure timer 1 for 200 ms
> 50 on timer 1 do gosub fast
> 60 sleep 3 s
> 70 print "ticks is", ticks
> 80 end
> 90 sub fast
> 100   let ticks = ticks+1
> 110 endsub
> run
slow
slow
slow
ticks is 14
> _
```

4.3.14　　　Digital I/O

StickOS supports digital I/O on all pins.

A pin is configured for digital I/O, and a variable bound to that pin, with the following statement:

```
dim varpin as pin pinname for digital (input|output) \
    [debounced] [inverted] [open_drain]
```

If a pin is configured for digital input, then subsequently reading the variable *varpin* will return the value 0 if the digital input pin is currently at a low level, or 1 if the digital input pin is currently at a high level. It is illegal to attempt write the variable *varpin* (i.e., it is read-only).

If a pin is configured for digital output, then writing *varpin* with a 0 value will set the digital output pin to a low level, and writing it with a non-0 value will set the digital output pin to a high level. Reading the variable *varpin* will return the value 0 if the digital output pin is currently at a low level, or 1 if the digital output pin is currently at a high level.

If the **debounced** pin qualifier is used, input values are passed thru a 12ms glitch-elimination filter.

If the **inverted** pin qualifier is used, all input and output values are logically inverted (i.e., 0->1, 1->0) at the pin.

If the **open_drain** pin qualifier is used, an output pin is tri-stated for a logic 1 output.

Examples
See Digital I/O Example

4.3.15   Analog I/O

Use the **help pins** command to see the list of MCU pin names and their analog capabilities.

A pin is configured for analog I/O, and a variable bound to that pin, with the following statement:

```
dim varpin as pin pinname for analog (input|output) \
        [debounced] [inverted]
```

If a pin is configured for analog input, then subsequently reading the variable *varpin* will return the analog voltage level, in millivolts (mV). It is illegal to attempt write the variable *varpin* (i.e., it is read-only).

If a pin is configured for analog output, then writing *varpin* with a millivolt value will set the analog output (PWM actually) pin to a corresponding analog voltage level. Reading the variable *varpin* will return the analog voltage level, in millivolts (mV).

If the **debounced** pin qualifier is used, input values are passed thru a 12ms glitch-elimination filter.

If the **inverted** pin qualifier is used, all input and output values are logically inverted at the pin (i.e., the pin mV is replaced with the maximum analog supply voltage millivolts minus the pin mV).

The maximum analog supply voltage millivolts may be displayed with the command:

```
analog
```

Configure the maximum analog supply voltage millivolts with the following command:

```
analog millivolts
```

This value defaults to 3300 mV and is stored in flash and affects all analog I/O pins.

Examples

See [Analog I/O Example](#)

4.3.16   Servo I/O

Use the **help pins** command to see the list of MCU pin names and their servo capabilities.

A pin is configured for servo I/O, and a variable bound to that pin, with the following statement:

> **dim** *varpin* **as pin** *pinname* **for servo output**

If a pin is configured for servo output, then writing *varpin* with a centi-millisecond (cms) value will set the servo output pin to the specified pulse duration. Reading the variable *varpin* will return the output pulse duration, in centi-milliseconds (cms).

The servo frequency may be displayed with the command:

> **servo**

Configure the servo frequency (in Hz) with the following command:

> **servo** *Hz*

This takes effect after the next MCU reset.

This value defaults to 45 Hz and is stored in flash and affects all servo I/O pins.

Examples
See Servo I/O Example

4.3.17 Frequency I/O

Use the **help pins** command to see the list of MCU pin names and their frequency capabilities.

A pin is configured for frequency I/O, and a variable bound to that pin, with the following statement:

```
dim varpin as pin pinname for frequency output
```

If a pin is configured for frequency output, then writing *varpin* with a hertz (Hz) value will set the frequency output pin to the specified frequency. Reading the variable *varpin* will return the output frequency, in hertz (Hz).

Examples

See [Frequency I/O Example](#)

4.3.18   UART I/O

Use the **help pins** command to see the list of MCU pin names and their UART capabilities.

UARTs can be configured for a specific serial communication protocol and then used to transmit or receive serial data. UARTs can also be configured to generate interrupts when they receive or transmit a character (or more specifically, when the uart receive buffers are not empty, or when the uart transmit buffers are empty).

UART serial communication protocols are configured with the statement:

```
configure uart n for b baud d data (even|odd|no) parity
configure uart n for b baud d data (even|odd|no) parity loopback
```

This configures uart *n* for *b* baud operation, with *d* data bits and the specified parity; 2 stop bits are always transmitted and 1 stop bit is received. If the optional "**loopback**" parameter is specified, the UART is configured to loop all transmit data back into its own receiver, for testing purposes.

Once the UART is configured, pin variables should be bound to the specified UART's transmit and receive pins with one or more of the following statements:

```
        dim varrx as pin pinname for uart input
        dim vartx as pin pinname for uart output
```

This binds the *varrx* variable to the specified UART's receive data pin, and the *vartx* variable to the specified UART's transmit data pin. From then on, receive data can be examined by reading the *varrx* variable, and transmit data can be generated by writing the *vartx* variable.

Please note the pin variable method of accessing UART I/O should not be used on PIC32, because the binding of PIC32 UARTs to port pins changes magically as you move from part to part; instead, the statements below should be used which will usurp the correct (unspecified) port pins for the part.

Alternately, UART I/O can be performed explicitly with statements which specifies the data variables (and implicitly, the data sizes) to be transferred:

```
        uart n write variable, ...
```

```
       uart n read variable, ...
```

During the UART read/write statements, the current value of all variables, for
their corresponding sizes (8 bits, 16 bits, or 32 bits) will be shifted out (write)
or shifted in (read). If an array variable is specified, its entire array contents
are used. *Note that a read of data that is not available yet will block until the
data is available, holding off BASIC interrupt handlers; it is therefore best to
read only one byte at a time, after you have determined it is available.*

At this point, if desired, interrupt handlers can be set up to handle UART
receive and/or transmit interrupts. UART receive interrupts are delivered
when the uart receive buffers are not empty; UART transmit interrupts are
delivered when the uart transmit buffers are empty.

The UART receive or transmit interrupt can be enabled, and the statement(s)
to execute when it is delivered specified, with the statement:

```
       on uart n (input|output) statement
```

 If *statement* is a "**gosub** *subname ...*", then all of the statements in
the corresponding sub are executed when the timer interrupt is delivered;
otherwise, just the single *statement* is executed.

Note that an initial UART transmit interrupt is generated when the transmit
interrupt is enabled, since the uart transmit buffers are empty!

The UART receive or transmit interrupt can later be completely ignored (i.e.,
discarded) with the statement:

```
       off uart n (input|output)
```

The UART receive or transmit interrupt can be temporarily masked (i.e., held
off but not discarded) with the statement:

```
       mask uart n (input|output)
```

And can later be unmasked (i.e., any pending interrupts delivered) with the
statement:

```
       unmask uart n (input|output)
```

Examples

See [UART I/O Example](#)

4.3.19   I2C Master I/O

Use the **help pins** command to see the list of MCU pin names and their I2C capabilities.

Unlike UART I/O, pin variables are not used for I2C I/O. Instead, there is an **i2c** statement which specifies the data variables (and implicitly, the data sizes) to be transferred via I2C. I2C transfers may be unidirectional (write or read) or bidirectional (mixed in any combination).

An I2C transaction is started and the I2C device is addressed with the statement:

```
i2c start address
```

Where *address* is the 7-bit I2C address of the device. Note that an i2c address is in the range 0 to 127 (or 0x7f). If you have an address that is greater than 128 (or 0x80), that is actually the address shifted left by one bit, and so needs to be shifted right by one bit (divided by 2) to obtain the real address.

Once the transaction is started, data can be written or read with the statements:

```
i2c write variable, ...
i2c read variable, ...
```

During the i2c read/write statements, the current value of all variables, for their corresponding sizes (8 bits, 16 bits, or 32 bits) will be shifted out (write) or shifted in (read). If an array variable is specified, its entire array contents are used.

Finally, the I2C transaction is completed with the statement:

```
i2c stop
```

Note that i2c statements can also be run in immediate mode, allowing you to interactively discover the way your i2c peripherals work!!!

Examples

See I2C Master I/O Example

4.3.20 QSPI Master I/O

Use the **help pins** command to see the list of MCU pin names and their QSPI capabilities.

Unlike UART I/O, pin variables are not used for QSPI I/O. Instead, there is a **qspi** statement which specifies the data variables (and implicitly, the data sizes) to be transferred via QSPI. QSPI transfers are always bidirectional -- the current values of the variables are shifted out and new values are shifted in.

Data is transferred (again, bidirectionally) with a single statement:

```
qspi variable, ...
```

The BASIC program is responsible for defining a chip select pin as a digital output and asserting it prior to the qspi statement, and deasserting it afterwards. During the qspi statement, the current value of all variables, for their corresponding sizes (8 bits, 16 bits, or 32 bits) will be shifted out, and the new values for the same variables will be shifted in. If an array variable is specified, its entire array contents are used.

Note that qspi statements can also be run in immediate mode, allowing you to interactively discover the way your qspi peripherals work!!!

Examples

See QSPI Master I/O Example

4.3.21   Pin Interrupts

StickOS can also support pin interrupts on any input (or output, for that matter) pin thru the use of pin variables in the watchpoint expression:

> *line* **on** *expression* **do** *statement*

The watchpoint expression is re-evaluated before every line of the program is executed; if the expression transitions from false to true, the watchpoint statement handler runs.

Since watchpoints have to transition from false to true, you can think of them as an edge-sensitive interrupt on a digital input pin. On an analog input pin, you can think of then as detecting an edge at a specific voltage level.

To set the smart watchpoint mode, which dramatically reduces watchpoint overhead at a slight delay of input pin sensitivity, use the command:

> **watchsmart** (**on**|**off**)

To display the smart watchpoint mode, use the command:

> **watchsmart**

For the example below, the MCU sw1 should be pressed one or more times after running the program.

Examples
```
> 10 dim switch as pin irq1* for digital input
> 20 on ! switch do print "switch pressed!"
> 30 sleep 1000 s
> run
switch pressed!
switch pressed!
switch pressed!
> _
```

4.3.22   4x4 Scanned Keypad Support

(v1.82+) StickOS makes it easy to interface to a scanned 4x4 keypad. Using the "pins" command, you can select 4 digital output pins for the keypad scan lines (kbd_s0-kbd_s3) and 4 digital input pins as the keypad return lines (kbd_r0-kbd_r3).

You can then set the keypad scan character codes for the 16 keypad buttons with the command:

> **keychars** *16-ascii-characters*

You can display the keypad scan character codes with the command:

> **keychars**

From then on, the most recent keypad character is available to BASIC programs in the system variable "keychar", which you will typically use as "keychar$".

Examples

```
> pins
heartbeat pte6
safemode* ptg0
qspi_cs* pte7
clone_rst* none
zigflea_rst* none
zigflea_attn* none
zigflea_rxtxen none
lcd_d4 pta2
lcd_d5 pta3
lcd_d6 pta4
lcd_d7 pta5
lcd_en pta1
lcd_rs pta0
kbd_s0 ptd4
kbd_s1 ptd5
kbd_s2 ptd6
kbd_s3 ptd7
kbd_r0 ptd0
kbd_r1 ptd1
kbd_r2 ptd2
kbd_r3 ptd3
> keychars
123a456b789c*0#d
> 10 on keychar do print keychar$
> 20 halt
> run
11
22
33
44
STOP at line 20!
> _
```

4.3.23   HD44780-compatible LCD Support

(v1.82+) StickOS makes it easy to interface to a 4-bit LCD that is HD44780-compatible. Using the "pins" command, you can select 4 digital output pins for data bits (lcd_d4-lcd_d7), and 2 more digital outputs for the enable and register select control lines (lcd_en, lcd_rs).

You can then interface to the LCD using the lcd command:

    lcd *pos,* [**dec**|**hex**|**raw**] *expression*

Or strings:

    lcd *pos, string*

Or various combinations of both.

Where *pos* is an LCD line number (0-3) or a LCD ram buffer position (0x80-0xff).

See <u>Print Statements</u> for more details.

Examples

```
> pins
heartbeat pte6
safemode* ptg0
qspi_cs* pte7
clone_rst* none
zigflea_rst* none
zigflea_attn* none
zigflea_rxtxen none
lcd_d4 pta2
lcd_d5 pta3
lcd_d6 pta4
lcd_d7 pta5
lcd_en pta1
lcd_rs pta0
kbd_s0 ptd4
kbd_s1 ptd5
kbd_s2 ptd6
kbd_s3 ptd7
kbd_r0 ptd0
kbd_r1 ptd1
kbd_r2 ptd2
kbd_r3 ptd3
> lcd 0, "hello world!"
> _
```

4.3.24   Other Statements

You can delay program execution for a number of seconds, milliseconds, or microseconds using the statement:

sleep *expression* (**s**|**ms**|**us**)

Note that the minimum sleep resolution is the clock tick, which is 0.25 milliseconds. Note also that in general it would be a bad idea to use a **sleep** statement in the **on** handler for a timer or uart interrupt.

You can add remarks to the program, which have no impact on program execution, with the statement:

rem *remark*

Examples
```
> 10 rem this program takes 5 seconds to run
> 20 sleep 5 s
> run
> _
```

4.4 Performance

StickOS typically runs about 1000 BASIC statements per second *per processor MHz* (i.e., 50,000 lines/second on a 50MHz processor). Many issues affect performance, most notably the specific statement mix.

StickOS runs fastest when *not* merging program lines from RAM and flash, so you should always **save** your program (causing RAM and flash program lines to be re-merged back to flash) before running it. It is also a good idea to **renumber** your program to ensure profile buckets are evenly distributed to the lines of your program.

Once you have run a saved/renumbered program, you can use the following command to list the time spent in each line of the program:

```
profile
profile line
profile -line
profile line-
profile line-line
```

Alternately, you can list the time spent in an entire subroutine by name with the command:

```
profile subname
```

Examples

```
> new
>   10 dim a, sum
>   20 for a = 1 to 10000
>   30   let sum = sum+a
>   40 next
>   50 print sum
> save
> run
50005000
> profile
     0ms    10 dim a, sum
    22ms    20 for a = 1 to 10000
   315ms    30   let sum = sum+a
   141ms    40 next
     2ms    50 print sum
end
>
```

5 2.4GHz ZigFlea Wireless Operation

5.1 ZigFlea Configuration

Prior to any wireless operation, each MCU/IOStick pair needs to have a
unique zigflea nodeid set. ZigFlea nodeid's are integers from 0 to 65534. The
zigflea nodeid is set with the following command:

```
nodeid nodeid
```

5.2 ZigFlea Remote Control

To connect to another MCU from the current one, use the command:

```
> connect new-nodeid
press Ctrl-D to disconnect
```

At that point you should press **<Enter>** to get a prompt from the remote
MCU (or press **<Ctrl-C>** to stop it if it is running a program), and verify its
nodeid:

```
<Enter>
> nodeid
new-nodeid
> _
```

When you are done using the other MCU, press **<Ctrl-D>** and the original
MCU will print the following message, followed by a prompt:

```
...disconnected
```

It is always a good idea to re-verify its nodeid:

```
> nodeid
old-nodeid
> _
```

5.3 ZigFlea Remote Variables

A MCU can modify variables on a remote MCU using zigflea remote variables. A remote variable is declared with the statement:

> **dim** *varremote* **as remote on nodeid** *nodeid*

This tells StickOS that the variable *varremote* is actually dimensioned on another node, *nodeid*, and any updates of that variable on this node should be forwarded to that other node for processing.

When *varremote* is modified, the request will be forwarded to the other nodeid; if the other nodeid does not accept the request, *varremote* will be reset to -1 instead.

The following example shows a remote LED dimmer, where the potentiometer on nodeid 1 is used to control the LED on nodeid 2.

Examples

```
> nodeid
1
> 10 dim potentiometer as pin an0 for analog input
> 20 dim led as remote on nodeid 2
> 30 while 1 do
> 40   let led = potentiometer
> 50   sleep 100 ms
> 60 endwhile
> save
> autorun on
> connect 2
press Ctrl-D to disconnect
<Enter>
> nodeid
2
> 10 dim led as pin dtin0 for analog output
> 20 while 1 do
> 30 endwhile
> save
> autorun on
> run
<Ctrl-D> ...disconnected
> nodeid
1
> run
now adjust the potentiometer on nodeid 1
  and watch the LED change on nodeid 2!!!
```

6 Standalone Operation

Once the MCU is disconnected from the host computer, it may be run standalone.

Based on the "autorun" mode, when the MCU is powered up, it will typically start running the (saved) BASIC program automatically.

Again, when the StickOS is running the "heartbeat" LED will blink slowly; when the BASIC program in the MCU is running, the "heartbeat" LED will blink quickly.

Note that any unsaved changes to the BASIC program will be lost if the MCU is reset or loses power.

7 Slave Operation

Though this probably goes without saying, the MCU can also be *permanently* connected to the host computer and used as a slave data acquisition/control device, all under host computer software control!

To do this, the host computer software program would simply open the MCU virtual COM port or TCP/IP port and then write StickOS commands and/or statements to it, and then read the results back from it.

Often it is useful to disable terminal echo and prompts when running in slave mode.

To set the terminal echo and prompt modes, use the commands:

```
echo (on|off)
prompt (on|off)
```

To display the terminal echo and prompt modes, use the commands:

```
echo
prompt
```

8 MCU Cloning

A master MCU can clone its flash to a slave MCU, including any BASIC programs and flash parameter values, by simply connecting the master MCU to the slave MCU with the following cable:

master	slave
qspi_clk	qspi_clk (ezpck)
qspi_din	qspi_dout (ezpq)
qspi_dout	qspi_din (ezpd)
pins qspi_cs*	rcon* (ezpcs*)
pins qspi_rst*	rsti*
vss	vss
vdd	vdd

And then using the following command on the master MCU:

```
> clone
Welcome to StickOS for Freescale MCF52221 v1.2!
Copyright (c) 2008; all rights reserved.
cloning...
done!
> _
```

Or if you want the slave MCU to start running immediately following the clone procedure, use the following command instead:

```
> clone run
Welcome to StickOS for Freescale MCF52221 v1.2!
Copyright (c) 2008; all rights reserved.
cloning...
done!
> _
```

9 MCU Downloading

A master MCU can download a S19 or HEX file to a slave MCU using the same connections as above, but replacing the "clone" command with the "download" command followed by the slave operating frequency, in Hz:

```
> download 8000000
paste S19 upgrade file now...
```

At that point you should paste the entire S19 upgrade file into your terminal emulator.

10 MCU Upgrading

Note that the upgrade procedure wipes out all BASIC programs and parameters from flash memory.

A MCU's StickOS firmware (i.e., the BASIC development environment itself) can be upgraded with the following command:

```
> upgrade
paste S19 upgrade file now...
```

At that point you should paste the entire S19 upgrade file into your terminal emulator.

When upgrade is nearly complete (about two minutes), you will see:

```
paste done!
programming flash...
wait for MCU heartbeat LED to blink!
```

Then wait for the MCU "heartbeat" LED to blink, indicating flash programming is complete; press the "Disconnect" button followed by the "Call" button, to reconnect Hyper Terminal.

After the upgrade, you must then update the StickOS pin assignments; see First Boot & Pin Assignments in both this User's Guide and the CPUStick User's Guide, as appropriate. *Until you update the StickOS pin assignments, the heartbeat LED will not blink and zigflea will be unusable, but StickOS will still be running, so you can connect the terminal emulator and make the necessary pin assignment updates.*

Note that once flash programming begins, a failed (or interrupted) upgrade procedure may only be able to be recovered via a re-clone from a working MCU.

11 Appendix

11.1 StickOS Command Reference

11.1.1 Commands

```
<Ctrl-C>                                  -- stop program
auto <line>                               -- automatically number program lines
clear [flash]                             -- clear ram [and flash] variables
cls                                       -- clear terminal screen
cont [<line>]                             -- continue program from stop
delete ([<line>][-][<line>]|<subname>)    -- delete program lines
download <slave Hz>                       -- download flash to slave MCU
dir                                       -- list saved programs
edit <line>                               -- edit program line
help [<topic>]                            -- online help
list ([<line>][-][<line>]|<subname>)      -- list program lines
load <name>                               -- load saved program
memory                                    -- print memory usage
new                                       -- erase code ram and flash memories
profile ([<line>][-][<line>]|<subname>)   -- display profile info
purge <name>                              -- purge saved program
renumber [<line>]                         -- renumber program lines (and save)
reset                                     -- reset the MCU!
run [<line>]                              -- run program
save [<name>]                             -- save code ram to flash memory
undo                                      -- undo code changes since last save
upgrade                                   -- upgrade StickOS firmware!
uptime                                    -- print time since last reset
```

11.1.2 Modes

```
analog [<millivolts>]             -- set/display analog voltage scale
baud [<rate>]                     -- set/display uart console baud rate
autorun [on|off]                  -- autorun mode (on reset)
echo [on|off]                     -- terminal echo mode
indent [on|off]                   -- listing indent mode
keychars [<keychars>]             -- set/display keypad scan chars
nodeid [<nodeid>|none]            -- set/display zigflea nodeid
numbers [on|off]                  -- listing line numbers mode
pins [<assign> [<pinname>|none]]  -- set/display StickOS pin assignments
prompt [on|off]                   -- terminal prompt mode
servo [<Hz>]                      -- set/display servo Hz (on reset)
step [on|off]                     -- debugger single-step mode
trace [on|off]                    -- debugger trace mode
watchsmart [on|off]               -- low-overhead watchpoint mode

pin assignments:
  heartbeat   safemode*
  qspi_cs*  zigflea_rst*  zigflea_attn*  zigflea_rxtxen
```

11.2 BASIC Program Statement Reference

11.2.1　　　Statements

```
<line>                                -- delete program line from code ram
<line> <statement>                    -- enter program line into code ram

assert <expression>                   -- break if expression is false
data <n> [, ...]                      -- read-only data
dim <variable>[$][[n]] [as ...] [, ...] -- dimension variables
end                                   -- end program
halt                                  -- loop forever
input [dec|hex|raw] <variable>[$] [, ...] -- input data
label <label>                         -- read/data label
lcd <pos>, [dec|hex|raw] <expression> [, ...] [;] -- display results on lcd
let <variable>[$] = <expression> [, ...] -- assign variable
print [dec|hex|raw] <expression> [, ...] [;] -- print results
read <variable> [, ...]               -- read read-only data into variables
rem <remark>                          -- remark
restore [<label>]                     -- restore read-only data pointer
sleep <expression> (s|ms|us)          -- delay program execution
stop                                  -- insert breakpoint in code
vprint <variable>[$] = [dec|hex|raw] <expression> [, ...] -- print to variable
```

11.2.2　　　Block Statements

```
if <expression> then
[elseif <expression> then]
[else]
endif

for <variable> = <expression> to <expression> [step <expression>]
  [(break|continue) [n]]
next

while <expression> do
  [(break|continue) [n]]
endwhile

do
  [(break|continue) [n]]
until <expression>

gosub <subname> [<expression>, ...]

sub <subname> [<param>, ...]
  [return]
endsub
```

11.2.3   Device Statements

```
timers:
  configure timer <n> for <n> (s|ms|us)
  on timer <n> do <statement>                -- on timer execute statement
  off timer <n>                              -- disable timer interrupt
  mask timer <n>                             -- mask/hold timer interrupt
  unmask timer <n>                           -- unmask timer interrupt

uarts:
  configure uart <n> for <n> baud <n> data (even|odd|no) parity [loopback]
  on uart <n> (input|output) do <statement>  -- on uart execute statement
  off uart <n> (input|output)                -- disable uart interrupt
  mask uart <n> (input|output)               -- mask/hold uart interrupt
  unmask uart <n> (input|output)             -- unmask uart interrupt
  uart <n> (read|write) <variable> [, ...]   -- perform uart I/O

i2c:
  i2c (start <addr>|(read|write) <variable> [, ...]|stop) -- master i2c I/O

qspi:
  qspi <variable> [, ...]                    -- master qspi I/O

watchpoints:
  on <expression> do <statement>             -- on expr execute statement
  off <expression>                           -- disable expr watchpoint
  mask <expression>                          -- mask/hold expr watchpoint
  unmask <expression>                        -- unmask expr watchpoint
```

11.2.4   Expressions

```
the following operators are supported as in C,
in order of decreasing precedence:
  <n>                      -- decimal constant
  0x<n>                    -- hexadecimal constant
  'c'                      -- character constant
  <variable>               -- simple variable
  <variable>[<expression>] -- array variable element
  <variable>#              -- length of array or string
  (   )                    -- grouping
  !   ~                    -- logical not, bitwise not
  *   /   %                -- times, divide, mod
  +   -                    -- plus, minus
  >>  <<                   -- shift right, left
  <=  <  >=  >             -- inequalities
  ==  !=                   -- equal, not equal
  |   ^   &                -- bitwise or, xor, and
  ||  ^^  &&               -- logical or, xor, and
```

11.2.5   Strings

```
v$ is a nul-terminated view into a byte array v[]

string statements:
  dim, input, let, print, vprint
  if <expression> <relation> <expression> then
  while <expression> <relation> <expression> do
  until <expression> <relation> <expression> do

string expressions:
  "literal"                  -- literal string
  <variable>$                -- variable string
  <variable>$[<start>:<length>]  -- variable substring
  +                          -- concatenates strings

string relations:
  <=  <  >=  >               -- inequalities
  ==  !=                     -- equal, not equal
  ~   !~                     -- contains, does not contain
```

11.2.6   Variables

```
all variables must be dimensioned!
variables dimensioned in a sub are local to that sub
simple variables are passed to sub params by reference; otherwise, by value
array variable indices start at 0
v is the same as v[0], except for input/print/i2c/qspi/uart statements

ram variables:
  dim <var>[$][[n]]
  dim <var>[[n]] as (byte|short)

absolute variables:
  dim <var>[[n]] [as (byte|short)] at address <addr>

flash parameter variables:
  dim <varflash>[[n]] as flash

pin alias variables:
  dim <varpin> as pin <pinname> for (digital|analog|servo|frequency|uart) \
                                   (input|output) \
                                   [debounced] [inverted] [open_drain]

system variables (read-only):
  analog  getchar  keychar  nodeid  msecs  seconds  ticks  ticks_per_msec
```