CPUStick™ and StickOS™ User's Guide v1.01

1 Overview

CPUStick is a 0.9”x1.8” very low cost standalone USB embedded computer based on the Freescale MCF52221 ColdFire MCU, similar in size and form to a USB “memory stick”.

(actual size)

Internal to the MCF52221 is an entire resident StickOS BASIC programming environment (including an easy-to-use editor, compiler, flasher, and debugger), where external pins are mapped to special “pin variables” for manipulation or examination, and internal peripherals are managed by BASIC control statements and interrupt handlers.

The CPUStick may be connected to any USB host computer that supports an FTDI Serial Port (including Windows and Linux) and may then be controlled directly by any terminal emulator program, *with no additional software or hardware required on the host computer*. The USB host computer may then be disconnected for standalone operation.

Together, CPUStick and StickOS revolutionize embedded system development!

The StickOS BASIC programming environment includes the following features:

- BASIC line editor
  - ansi or vt100'ish terminal support
- BASIC compiler
  - compiles to a fast and safe intermediate bytecode
  - transparent line-by-line compilation is invisible to the user
- BASIC debugger, supporting:
  - breakpoints and assertions
  - variable (and pin) manipulation and examination
  - execution tracing and single-stepping
  - edit-and-continue!
- BASIC file system
  - load and store up to three BASIC programs
- external control of 29 I/O pins, implicit thru "pin variables"
  - digital input or output
  - uart input or output
  - analog input or output (PWM actually)
- internal interval timer and uart control
  - interrupts delivered to BASIC handlers!
- internal flash memory control
  - save programs and parameters to flash for standalone operation
  - prolong flash lifetime by storing incremental updates in RAM
  - clone one CPUStick’s flash directly to another for easy production
  - upgrade a CPUStick’s StickOS firmware via USB!
  - no external flash programmers needed!

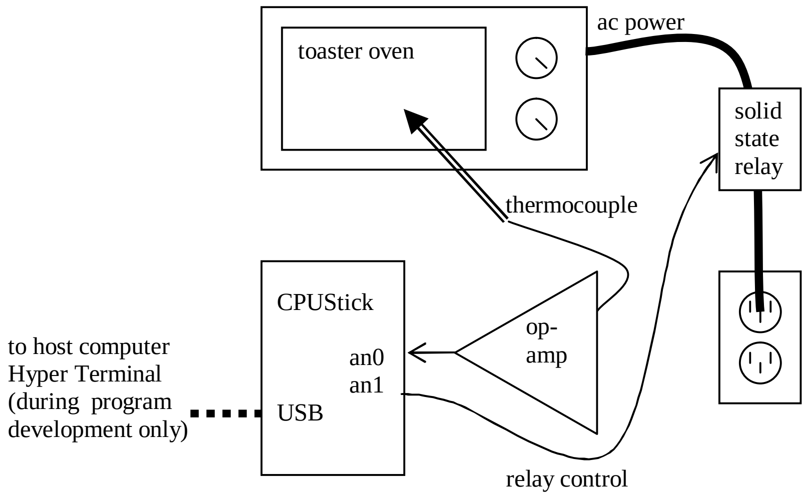
By its very nature, StickOS supports in-circuit emulation when it is running in any StickOS-capable MCU -- all you need is three (USB) wires connecting the MCU to a USB host computer, and you have full control over the target embedded system!

Once the CPUStick is disconnected from the USB host computer, it may be run standalone from external power or the optional bottom-side CR-2 battery holder.

Table of Contents

2 Embedded Systems Made Easy

A simple embedded system, like a toaster oven temperature profile controller, can be brought online in record time!

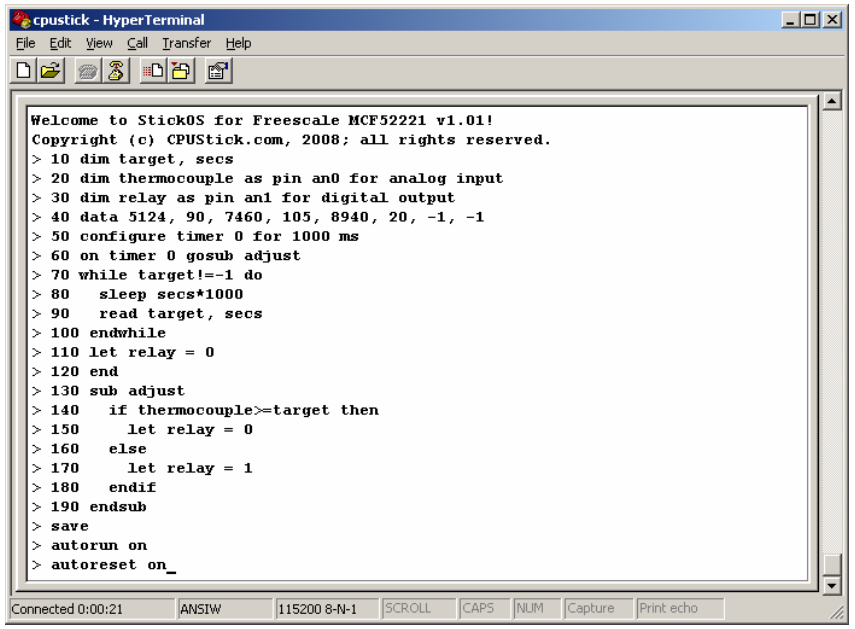


It's as easy as...

1. wire the CPUStick I/O pins to the embedded circuit
    a. wire CPUStick pin an0 to thermocouple op-amp output (I use an LM358)
    b. wire CPUStick pin an1 to solid state relay control input (I use a Teledyne STH24D25)
2. connect a host computer to the USB interface on the CPUStick
3. let the host computer automatically install the FTDI Serial Port transport drivers
4. open a Hyper Terminal console window and connect to the CPUStick; press <Enter> for a command prompt

5. configure the CPUStick I/O pins as appropriate
    a. configure pin an0 as an analog input
    b. configure pin an1 as a digital output
6. write and debug your BASIC control program, live on the CPUStick (see below)
7. type "save"
8. type "autorun on"
9. type "autoreset 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 CPUStick

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


cpustick - HyperTerminal
File Edit View Call Transfer Help
Connected 0:00:21 ANSIW 115200 8-N-1 SCROLL CAPS NUM Capture Print echo

```
Welcome to StickOS for Freescale MCF52221 v1.01!
Copyright (c) CPUStick.com, 2008; all rights reserved.
> 10 dim target, secs
> 20 dim thermocouple as pin an0 for analog input
> 30 dim relay as pin an1 for digital output
> 40 data 5124, 90, 7460, 105, 8940, 20, -1, -1
> 50 configure timer 0 for 1000 ms
> 60 on timer 0 gosub adjust
> 70 while target!=-1 do
> 80   sleep secs*1000
> 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
> autorun on
> autoreset on_
```

Line 10 declares two simple RAM variables for use in the program, and initializes them to 0.

Line 20 declares an analog input "pin variable" bound to pin an0, to read the thermocouple; line 30 declares a digital output "pin variable" bound to pin an1, to control the solid state relay.

Lines 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" sub 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.

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

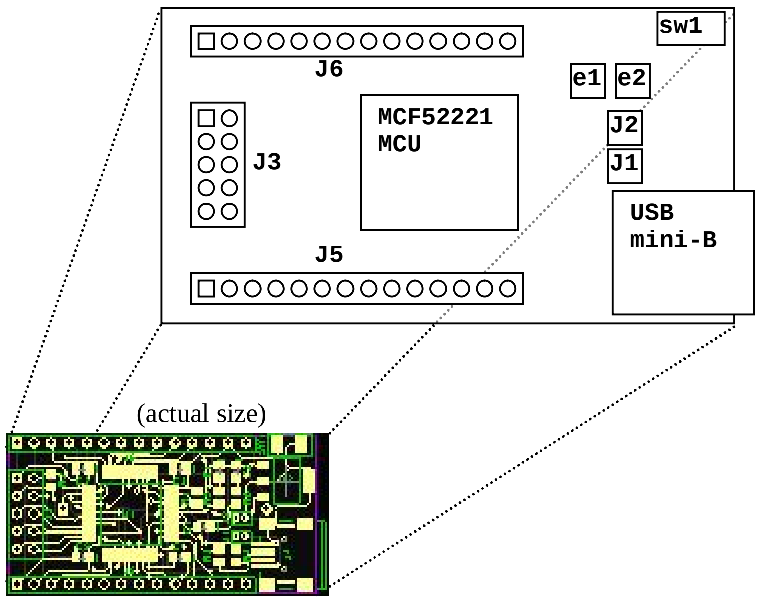
```
> 60 on timer 0 let relay = thermocouple<target
```

“Save” saves the program to non-volatile flash memory, “autorun on” sets the program to run automatically when the CPUStick is powered up, and “autoreset on” sets the CPUStick to automatically reset itself when it is awakened from a sleep (as opposed to continuing running the saved BASIC program from where it left off). Finally, “reset” resets the CPUStick as if it was just powered up.

3 CPUStick

3.1 Interface

The CPUStick contains a sleep switch, two LEDs, a USB mini-B connector, 38 external 0.1” header pins, and 2 power jumpers, depicted below.



When the StickOS in the CPUStick is running LED “e1” (on digital output pin “irq7*”) will blink slowly; when the BASIC program in the CPUStick is running, LED “e1” will blink quickly. LED “e2” is under BASIC program control on digital output pin “irq4*”.

When the sleep switch “sw1” (on pin “irq1*”) is depressed, the CPUStick enters a low-power mode (drawing ~70 uA) and tri-states all external pins until the sleep switch is depressed again. Holding switch “sw1” depressed during power-on prevents autorun of the BASIC program. If autoreset mode is set, the CPUStick automatically resets itself when it is awakened from a sleep (as opposed to continuing running the saved BASIC program from where it left off).

3.2 External Pins

CPUStick external pins must be configured prior to use.

All CPUStick external pins support general purpose digital input or output. In addition, eight an? pins can support analog input, four dtin? pins can support analog output (PWM actually), two urxd? pins can support UART input, and two utxd? pins can support UART output.

The CPUStick 0.1" header pinout is listed below. Pins indicated with ~~strikethrough~~ text are intended for debug and initial code load only.

J3

1	GND
2	+3.3V
3	~~rsti*~~
4	scl
5	sda
6	qspi_din
7	qspi_dout
8	qspi_clk
9	qspi_cs0*
10	~~rcon*~~

J5

1	GND
2	+3.3V
3	dtin3
4	dtin2
5	dtin1
6	dtin0
7	an0
8	an1
9	an2
10	an3
11	an4
12	an5
13	an6
14	an7

J6

1	GND
2	+3.3V
3	ucts0*
4	urts0*
5	urxd0
6	utxd0
7	ucts1*
8	urts1*
9	urxd1
10	utxd1
11	irq7*
12	irq4*
13	irq1*
14	+5V

3.3 Power

The power jumpers J1 and J2 and the optional CR-2 battery holder determine where the CPUStick will draw power from, as follows:

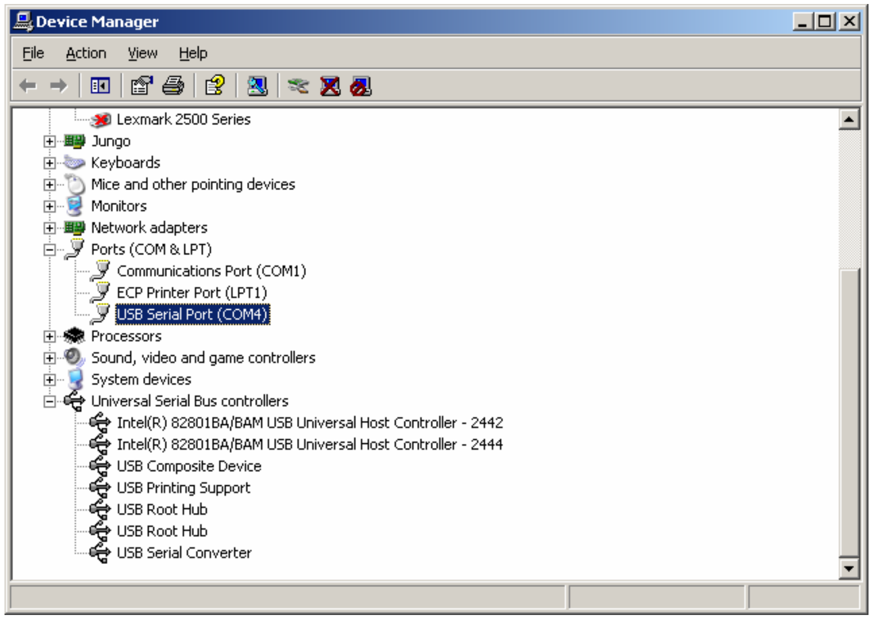
J1	J2	CR-2	Power
Installed	Installed	-	USB
-	Installed	-	+5V header
-	-	Installed	CR-2
-	-	-	+3V header

All digital and analog circuitry within the CPUStick runs at +3.3V. See the schematic in the appendix for details.

3.4 USB

When the CPUStick is connected to a USB host computer, it will present an FTDI Serial Port function to the host computer. An appropriate driver will be loaded automatically from microsoft.com, if needed, or you can manually install the VCP driver from http://www.ftdichip.com/FTDrivers.htm.

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:

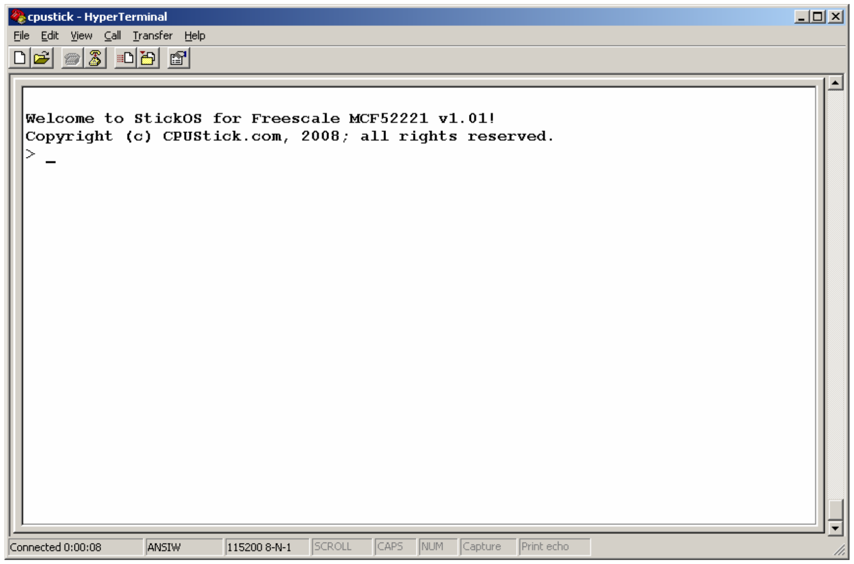


```
    Lexmark 2500 Series
+ Jungo
+ Keyboards
+ Mice and other pointing devices
+ Monitors
+ Network adapters
- Ports (COM & LPT)
    Communications Port (COM1)
    ECP Printer Port (LPT1)
    USB Serial Port (COM4)
+ Processors
+ Sound, video and game controllers
+ System devices
- Universal Serial Bus controllers
    Intel(R) 82801BA/BAM USB Universal Host Controller - 2442
    Intel(R) 82801BA/BAM USB Universal Host Controller - 2444
    USB Composite Device
    USB Printing Support
    USB Root Hub
    USB Root Hub
    USB Serial Converter
```

At this point you can use Hyper Terminal (typically found under Start -> All Programs -> Accessories -> Communications -> Hyper Terminal) to connect to the new virtual COM port.

Specify a new connection name, such as "cpustick", and then select the new virtual COM port under Connect To; the baud rate and data characteristics in Port Settings are ignored.

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



```
Welcome to StickOS for Freescale MCF52221 v1.01!
Copyright (c) CPUStick.com, 2008; all rights reserved.
> _
```

If the USB connection is lost (such as when you unplug and re-plug in the CPUStick), press the "Disconnect" button followed by the "Call" button, to reconnect Hyper Terminal.

Note that if you do not have Hyper Terminal (such as in Windows Vista or Linux), my favorite terminal emulator program is "putty", available free from http://www.putty.org/; the latest version supports serial port connections.

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

4 StickOS

StickOS supports a BASIC programming environment, where external pins are mapped to special "pin variables" for manipulation or examination.

External pins can be dynamically configured as one of:

- digital input or output,
- uart input or output, or
- analog input or output (PWM actually)

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 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
`<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 -   ^
>
```

4.1.1 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
> 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 CPUStick 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) 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 variable update as it occurs:

```
> trace on
> run
30 let square = 1
30 let square = 0
30 let square = 1
30 let square = 0
<Ctrl-C>
STOP at line 40!
>
```

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

Note that almost all commands 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.

4.1.2 UART I/O Example

The CPUStick can perform serial uart I/O as simply as digital 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
  40 let tx = 48
  50 let tx = 49
ok!
>
```

(Note that tracing is still enabled from the previous example.) 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:

```
> trace off
> 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 gosub receive
> 50 let tx = 48
> 60 let tx = 49
> 70 sleep 1000
> 80 end
> 90 sub receive
> 100   print "received", rx
> 110 endsub
> run
received 48
received 49
>
```

4.1.3 Analog I/O Example

As a final introductory example, 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 20264
>
```

Note that analog inputs and outputs are represented by integers in the range of 0 to 32767, where 0 represents a 0V input or output and 32767 represents a 3.3V input or output.

Note that almost all commands 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 20288
>
```

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

4.2 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.

4.2.1 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 variables
  help pins
  help board
  help clone
> help commands
clear [flash]                -- clear ram [and flash] variables
clone [run]                  -- clone flash to slave CPUStick
cont [<line>]                -- continue program from stop
delete [<line>][-][<line>]  -- delete program lines
dir                          -- list saved programs
edit <line>                  -- edit program line
help [<topic>]               -- online help
list [<line>][-][<line>]    -- list program lines
load <name>                  -- load saved program
memory                       -- print memory usage
new                          -- erase code ram and flash memories
purge <name>                 -- purge saved program
renumber [<line>]            -- renumber program lines (and save)
reset                        -- reset the CPUStick!
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
>
```

4.2.2 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 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
```

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

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

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 CPUStick 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
> 30 print a
> 40 next a
> 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
>
```

4.2.3 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`**`)`

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

`autorun`

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

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

```
        autoreset (on|off)
```

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

```
        autoreset
```

If the autoreset mode is on, the CPUStick will reset itself when it is awakened from a sleep, as if powering on; otherwise, it will continue running the saved BASIC program from where it left off.

Note that any unsaved changes to the BASIC program will be lost if the CPUStick 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
>
```

4.2.4 Loading and Storing Programs

StickOS can load and store up to three BASIC programs by name.

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

```
        dir
```

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

```
        save name
```

To load a 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
>
```

4.2.5 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
```

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 CPUStick 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 variable modifications while running.

Examples

```
> 10 dim a, sum
> 20 for a = 1 to 10000
> 30 let sum = sum+a
> 40 next a
> 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
STOP at line 10!
> cont
  20 let a = 1
STOP at line 20!
> <Enter>
  30 let sum = 1
STOP at line 30!
> <Enter>
  40 let a = 2
STOP at line 40!
> <Enter>
  30 let sum = 3
STOP at line 30!
>
```

4.2.6 Other Commands

To clear BASIC program variables, use the command:

```
clear
```

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 CPUStick as if it was just powered up, use the command:

```
reset
```

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

To display the time since the CPUStick 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 looooooooooooooooooooooooooooooooooooong 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 looooooooooooooooooooooooooooooooooooong 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 can be dimensioned as either integer (32 bits, signed) or byte (8 bits, unsigned) with the following statements:

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

Array RAM variables can be dimensioned with the following statements:

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

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

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`.

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

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

```
dim var, var, ...
```

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).

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

```
dim varpin as pin pinname for (digital|uart|analog) (input|output)
```

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

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 17456
volatile is 0
persistent is 0
> run
0
1
4
9
potentiometer is 17456
volatile is 0
persistent is 1
>
```

4.3.2 Variable Assignments

Simple variables are assigned with the following statement:

```
let variable = expression
```

If the variable represents an output "pin variable", the corresponding CPUStick 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.

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
>
```

4.3.3 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 increasing precedence:

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

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.

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 CPUStick 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
>
```

4.3.4 Print Statements

While the CPUStick is connected to the host computer's USB port, print statements can be observed on the Hyper Terminal console window.

Print statements can be used to print integer expressions:

```
      print expression
```

Or strings:

```
      print "string"
```

Or various combinations of both:

```
      print "string", expression, ...
```

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

Note that when the CPUStick is disconnected from the host computer's USB port, print statement output is simply discarded.

Examples

```
> print "hello world"
hello world
> print 57*84
4788
> print 9, "squared is", 9*9
9 squared is 81
>
```

4.3.5 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.

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!
>
```

4.3.6 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.

Examples

```
> 10 dim a
> 20 for a = -5 to 5
> 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
-5 is odd
-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
5 is odd
>
```

4.3.7 Looping Conditional Statements

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

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 .

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.

In both the for-next and while-endwhile 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`** or **`endwhile`**) 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`** or **`endwhile`**) 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
>
```

4.3.8 Subroutines

A subroutine is called with the following statement:

```
gosub subname
```

A subroutine is declared with the following statements:

```
sub subname
      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.

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.

In StickOS v1.01, there is no way to pass parameters to or return values from subroutines. This is the highest-priority roadmap item.

Examples

```
> 10 dim a
> 20 while 1 do
> 30   if a==10 then
> 40     break
> 50   endif
> 60   gosub sumit
> 70 endwhile
> 80 end
> 90 sub sumit
> 100   dim b, sum
> 110   for b = 0 to a
> 120     let sum = sum+b
> 130   next
> 140   print "sum of integers 0 thru", a, "is", sum
> 150   let a = a+1
> 160 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
>
```

4.3.9 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`* **`ms`**

This configures timer *`n`* to interrupt every *`m`* 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 print "slow"
> 40 configure timer 1 for 200 ms
> 50 on timer 1 gosub fast
> 60 sleep 3000
> 70 print "ticks is", ticks
> 80 end
> 90 sub fast
> 100   let ticks = ticks+1
> 110 endsub
> run
slow
slow
slow
ticks is 14
>
```

4.3.10 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)
```

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.

Examples

See Digital I/O Example

4.3.11 UART I/O

StickOS supports up to 2 uarts (0 and 1). 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 urxdn for uart input
        dim vartx as pin utxdn 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.

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

4.3.12 Analog I/O

StickOS supports analog input on the pins an0 thru an7, and analog output (PWM actually) on the pins dtin0 thru dtin3.

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`**`)`

If a pin is configured for analog input, then subsequently reading the variable *`varpin`* will return the analog voltage level, in the range 0..32767, corresponding to 0V..3.3V of the input pin. 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 value in the range 0..32767 will set the analog output (PWM actually) pin to a corresponding analog voltage level in the range of 0V..3.3V. reading the variable *`varpin`* will return the analog voltage level, in the range 0..32767, corresponding to 0V..3.3V of the output pin.

In StickOS v1.01, analog output (PWM actually) is not supported. This is a high-priority roadmap item.

Examples

See Analog I/O Example

4.3.13 Other Statements

You can delay program execution for a number of milliseconds using the statement:

> **`sleep`** *`expression`*

Note 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 5000
> run
>
```

5 Standalone Operation

Once the CPUStick is disconnected from the USB host computer, it may be run standalone from external power or the optional bottom-side CR-2 battery holder.

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

Again, when the StickOS in the CPUStick is running LED “e1” (on digital output pin “irq7*”) will blink slowly; when the BASIC program in the CPUStick is running, LED “e1” will blink quickly. LED “e2” is under BASIC program control on digital output pin “irq4*”.

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

6 Slave Operation

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

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

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
```

7 CPUStick Cloning

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

master	**slave**
qspi_clk	qspi_clk (ezpck)
qspi_din	qspi_dout (ezpq)
qspi_dout	qspi_din (ezpd)
qspi_cs0	rcon* (ezpcs*)
scl	rsti*
vss	vss
vdd	vdd

And then using the following command on the master CPUStick:

```
> clone
Welcome to StickOS for Freescale MCF52221 v1.01!
Copyright (c) CPUStick.com, 2008; all rights reserved.
cloning...
done!
>
```

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

```
> clone run
Welcome to StickOS for Freescale MCF52221 v1.01!
Copyright (c) CPUStick.com, 2008; all rights reserved.
cloning...
done!
>
```

8 CPUStick Upgrading

A CPUStick'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 CPUStick LED e1 to blink!
```

Then wait for the CPUStick LED e1 to blink, indicating flash programming is complete. Then reconnect your Hyper Terminal. Note that once flash programming begins, a failed (or interrupted) upgrade procedure can only be recovered via a re-clone from a working CPUStick.

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

9 Appendix

9.1 CPUStick Features

The CPUStick includes the following features:

- Freescale MCF52221 ColdFire MCU with 128k bytes flash and 16k bytes RAM
    - SPI serial flash programming interface for initial code load
    - 19 high-current GPIO pins and 10 low current GPIO pins
    - 8 analog input pins, 4 analog output pins (PWM actually)
    - 2 uarts
    - timers, SPI master serial port, i2c serial port, etc.
    - USB on-the-go host/device interface
    - low-current standby mode
    - internal 8MHz on-chip oscillator, PLL'd up to 48MHz
- 0.1” headers for all external connections
- soft power switch
- up to two surface mount LEDs
- optional 48MHz crystal
- optional CR-2 battery holder

9.2 StickOS Command Reference

9.2.1 Commands

```
clear [flash]               -- clear ram [and flash] variables
clone [run]                 -- clone flash to slave CPUStick [and run]
cont [<line>]               -- continue program from stop
delete [<line>][-][<line>]  -- delete program lines
dir                         -- list saved programs
edit <line>                 -- edit program line
help [<topic>]              -- online help
list [<line>][-][<line>]    -- list program lines
load <name>                 -- load saved program
memory                      -- print memory usage
new                         -- erase code ram and flash memories
purge <name>                -- purge saved program
renumber [<line>]           -- renumber program lines (and save)
reset                       -- reset the CPUStick!
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
```

9.2.2 Modes

```
autoreset [on|off]          -- autoreset (on wake) mode
autorun [on|off]            -- autorun (on reset) mode
echo [on|off]               -- terminal echo mode
indent [on|off]             -- listing indent mode
prompt [on|off]             -- terminal prompt mode
step [on|off]               -- debugger single-step mode
trace [on|off]              -- debugger trace mode
```

9.3 BASIC Program Statement Reference

9.3.1 Statements

```
<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
let <variable> = <expression>          -- assign variable
print ("string"|<expression>) [, ...]  -- print strings/expressions
read <variable> [, ...]                -- read read-only data into
variables
rem <remark>                           -- remark
restore [<line>]                       -- restore read-only data pointer
sleep <expression>                     -- delay program execution (ms)
stop                                   -- insert breakpoint in code
```

9.3.2 Block Statements

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

for <variable> = <expression> to <expression> [step <expression>]
  [break [n]]
next

while <expression> do
  [break [n]]
endwhile

gosub <subname>

sub <subname>
  [return]
endsub
```

9.3.3 Device Statements

```
timers:
  configure timer <n> for <n> ms
  on timer <n> <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) <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
```

9.3.4 Variables

```
all variables must be dimensioned!
variables dimensioned in a sub are local to that sub
array variable indices start at 0; v[0] is the same as v

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

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

pin alias variables
  dim <varpin> as pin <pinname> (analog|digital|uart) (input|output)
```

9.3.5 Expressions

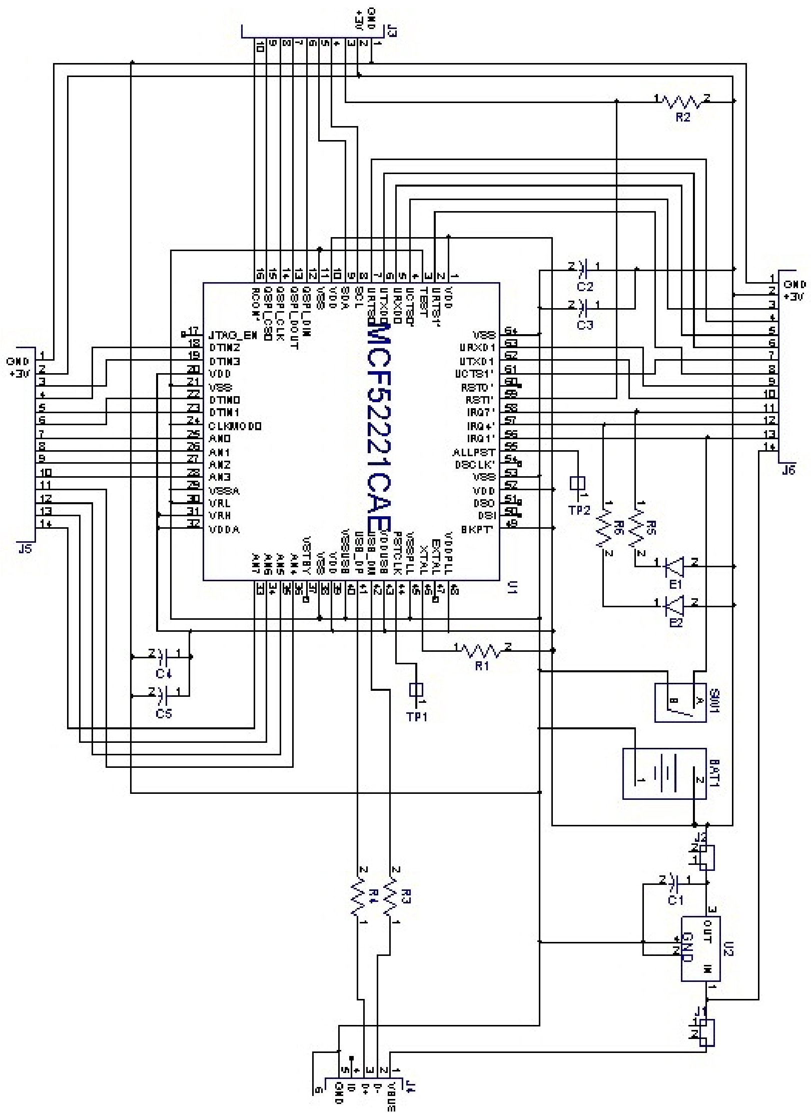
```
the following operators are supported as in C,
in order of increasing precedence:
  ||  ^^  &&                 -- logical or, xor, and
  |   ^   &                  -- bitwise or, xor, and
  ==  !=                     -- equal, not equal
  <=  <  >=  >               -- inequalities
  >>  <<      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

9.3.6 Pins

```
pin names:
  dtin3     dtin2     dtin1     dtin0
  an0       an1       an2       an3
  an4       an5       an6       an7
  ucts0*    urts0*    urxd0     utxd0
  ucts1*    urts1*    urxd1     utxd1
  irq7*     irq4*     irq1*
  qspi_cs0  qspi_clk  qspi_din  qspi_dout
  scl       sda

all pins support general purpose digital input/output
dtin? = potential analog output (PWM actually) pins (scale 0..32767)
an? = potential analog input pins (scale 0..32767)
urxd? = potential uart input pins (received byte)
utxd? = potential uart output pins (transmit byte)
```

9.4 CPUStick Schematic


MCF5221CAE
U1
J3
J4
J5
J6
GND
+3V
R1
R2
R3
R4
R5
R6
C1
C2
C3
C4
C5
E1
E2
TP1
TP2
SW1
BAT1
U2
J1
J2
VBUS
GND
JTAG_EN
DTIN2
DTIN3
VDD
VSS
DTIN0
DTIN1
CLKMOD0
AN0
AN1
AN2
AN3
VSSA
VRL
VRH
VDDA
URXD1
UTXD1
UCTS1*
RSTO*
RSTI*
IRQ7*
IRQ4*
IRQ1*
ALLPST
DSCLK*
DSO
DSI
BKPT*
VDDPLL
EXTAL
XTAL
VSSPLL
PSTCLK
VDDUSB
USB_DM
USB_DP
VSSUSB
VSTBY
AN4
AN5
AN6
AN7
QSPI_CS0
QSPI_CLK
QSPI_DIN
QSPI_DOUT
RCON*
SDA
SCL
URTS0*
UTXD0
URXD0
UCTS0*
URTS1*
TEST