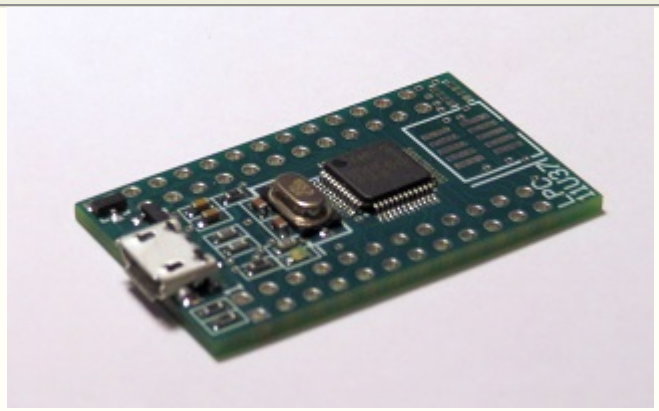
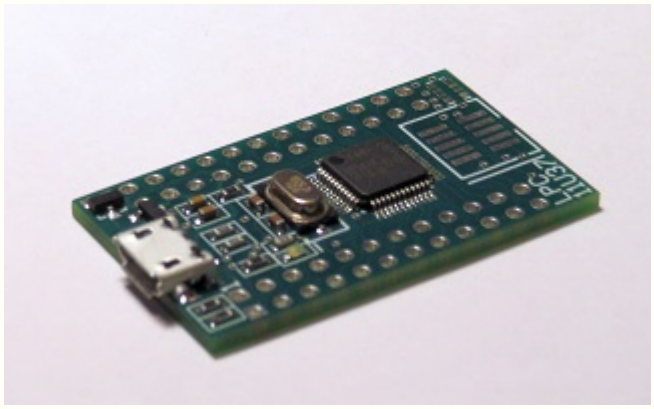
Table of Contents



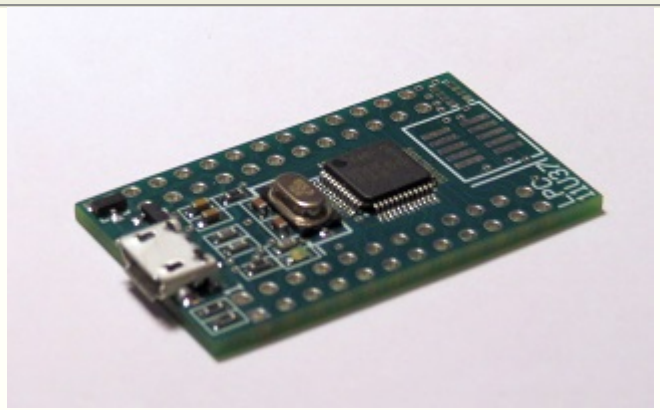
Getting Started



Getting Started

- ARM Stamp LPC11U37
- SuperPRO and Arduino variants
- Web enabled products
- Getting started with mBed
- BASIC on Teensy

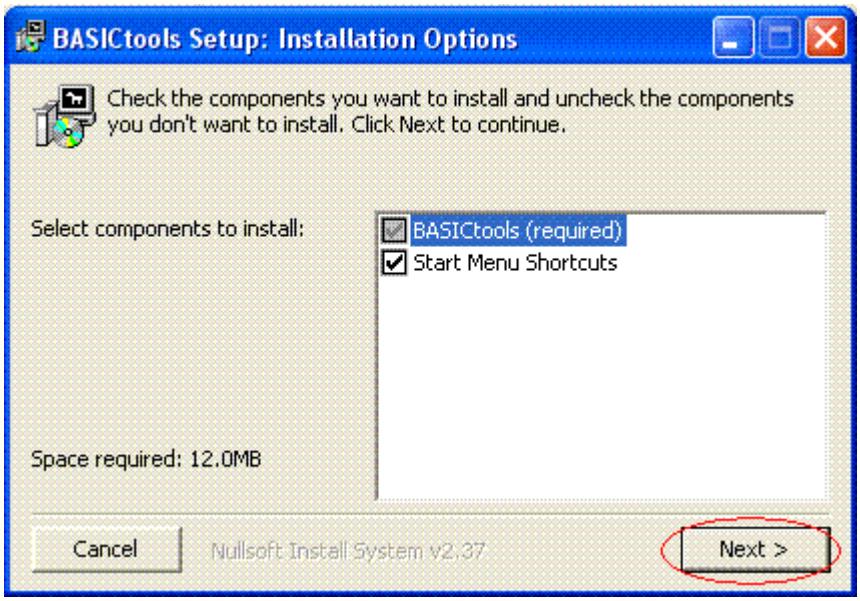
ARM Stamp LPC11U37 getting started



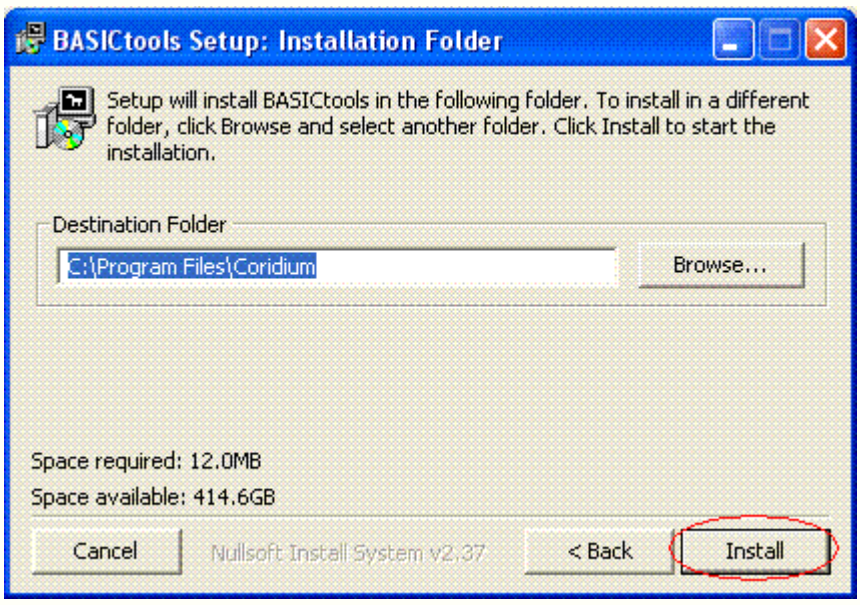
Step 1: Install Software

Coridium Products use a BASIC Compiler that runs on the PC. Coridium supplies BASICtools which includes a terminal emulator and IDE that is specifically designed for the PRO family, BASICchip, ARMmite and Ethernet boards. Also, a number of help files and documents about the ARMbasic will be installed on the machine at this time. This installer is meant for Windows 10, WIndows 8, Windows 7, or Windows XPboth 32 and 64 bit version.

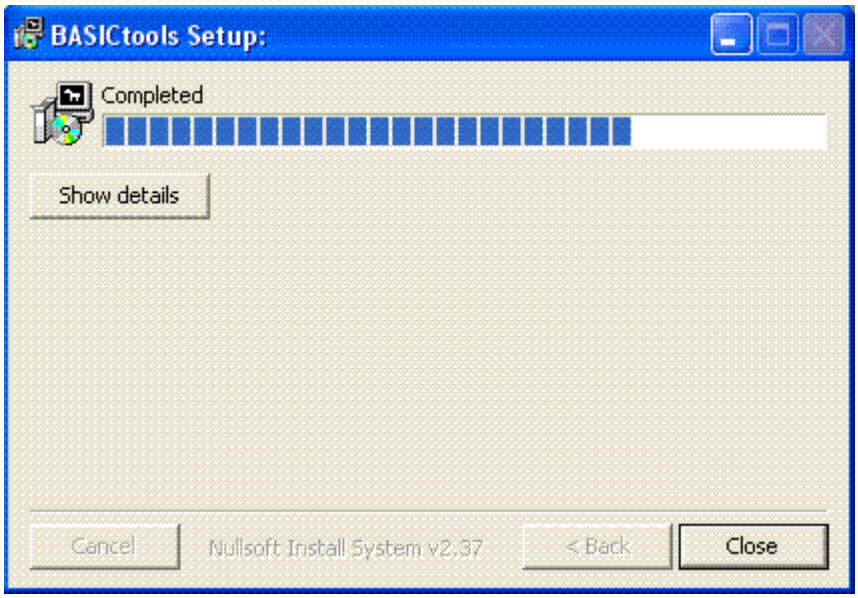
Download **www.coridium.us/files/setupBASIC.exe** from the web, and run the program to start the installation.



Click **Next** to get started.



Accept the defaults and **Install**. You may chose a different target directory.



The installation will now run, and when it finishes hit **Close** .

And its as easy as that.

On to Connecting to ARM Stamp

Step 2: Connect USB

Connect USB Cable to ARM Stamp

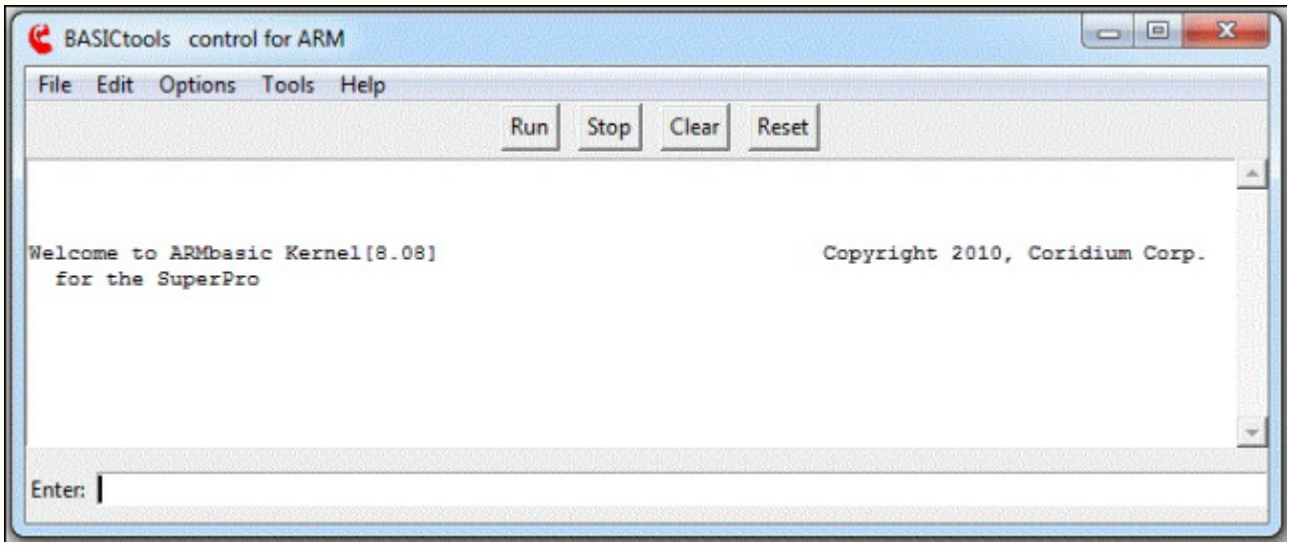
Use a standard USB A to micro-B cable. Just plug it into the PC. We identify as a Microssoft USB Serial device, so no drivers need to be installed.

That was pretty easy...

On to Step 3

Step 3: Writing your first Program with BASICtools

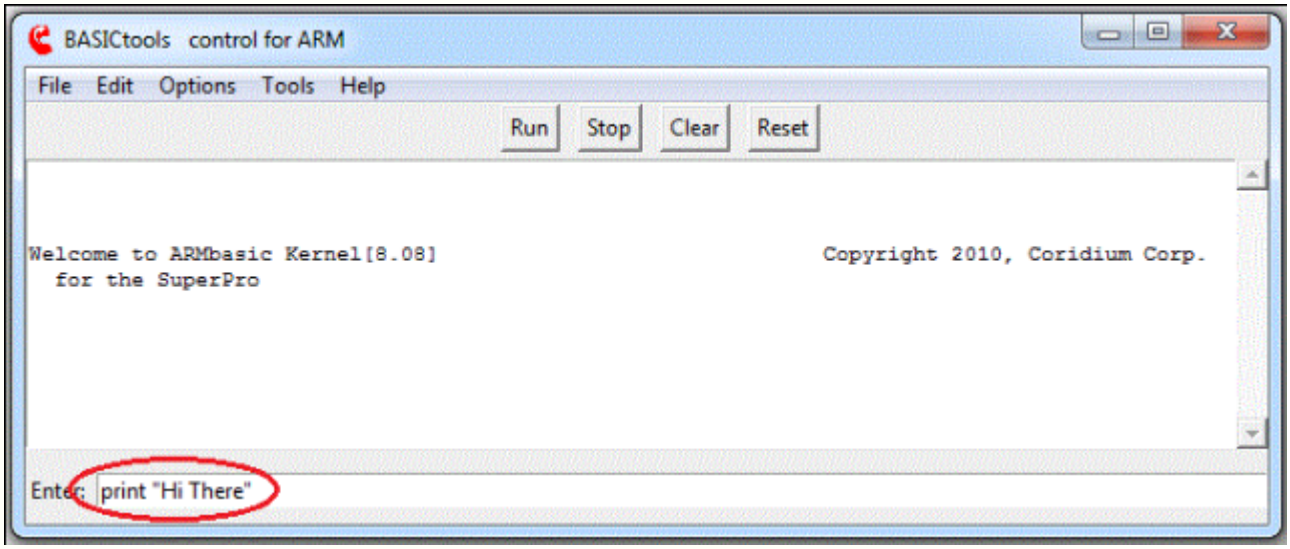
Start the BASICtools from the Start-Menu or from the Desktop Icon. You should see a welcome message which has been sent from the ARM processor-



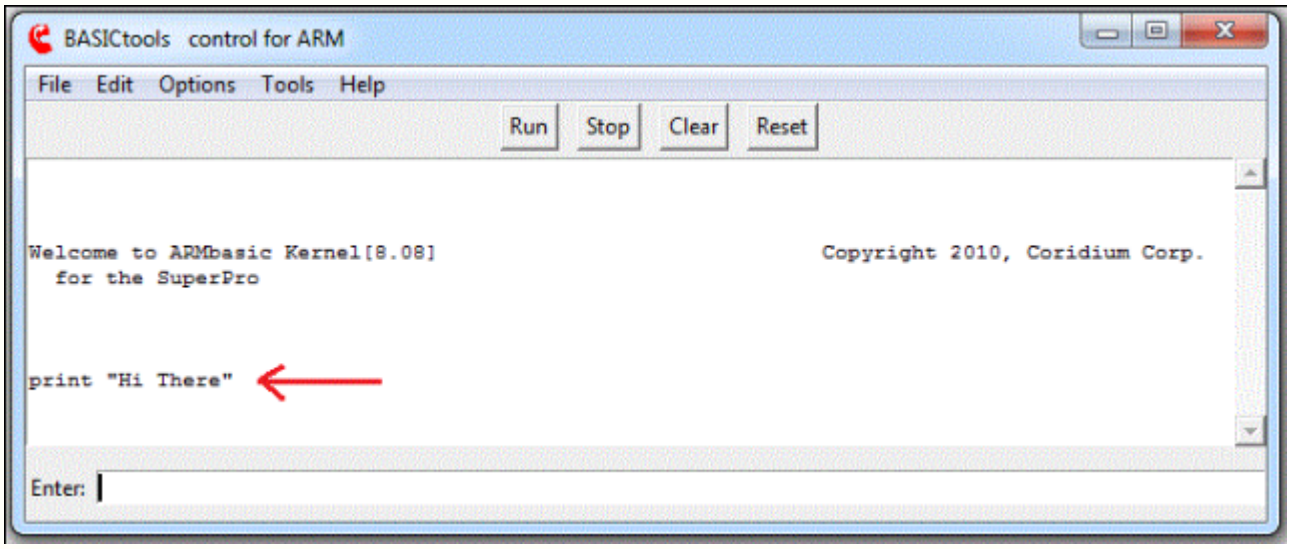
If you do not see this welcome, even after pushing the RESET button, then communication has not been established.

- check cables
- check power supply
- check COM port choice in BASICtools -> Options
- check baud rate in BASICtools -> Options
- on non-Coridium Boards, remove any BOOT select jumpers, press RESET again
- if still not working, check the**Trouble Shooting Section**

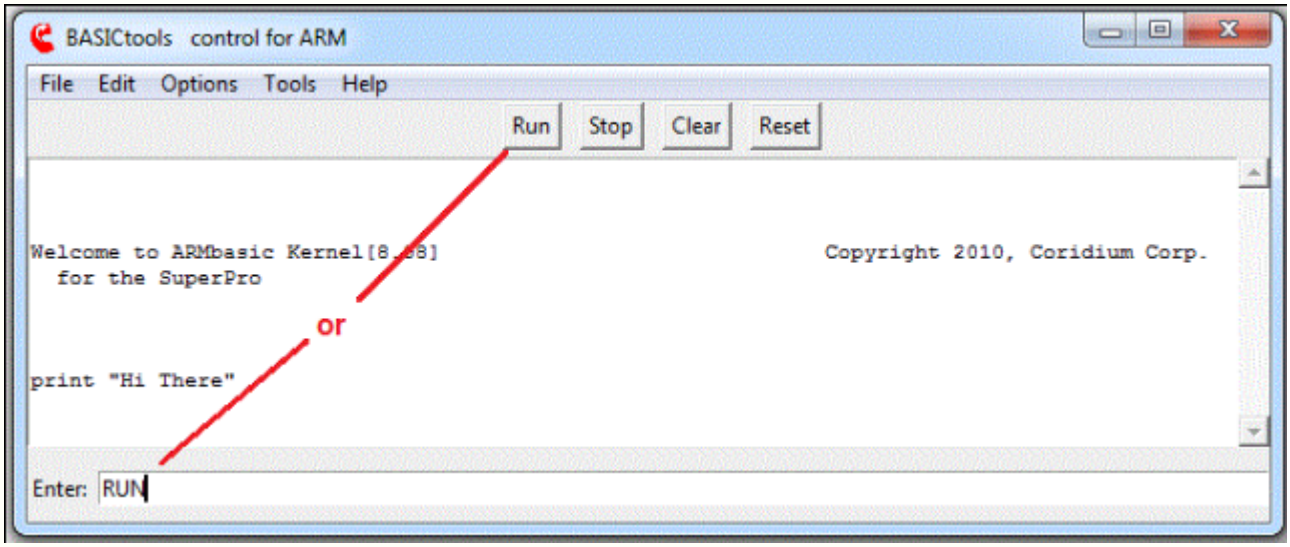
The traditional "Hi World" program



So type something like the traditional PRINT "Hi There"
When you hit the ENTER key it will be sent to the compiler on the PC and be echoed back in the console window. (below)

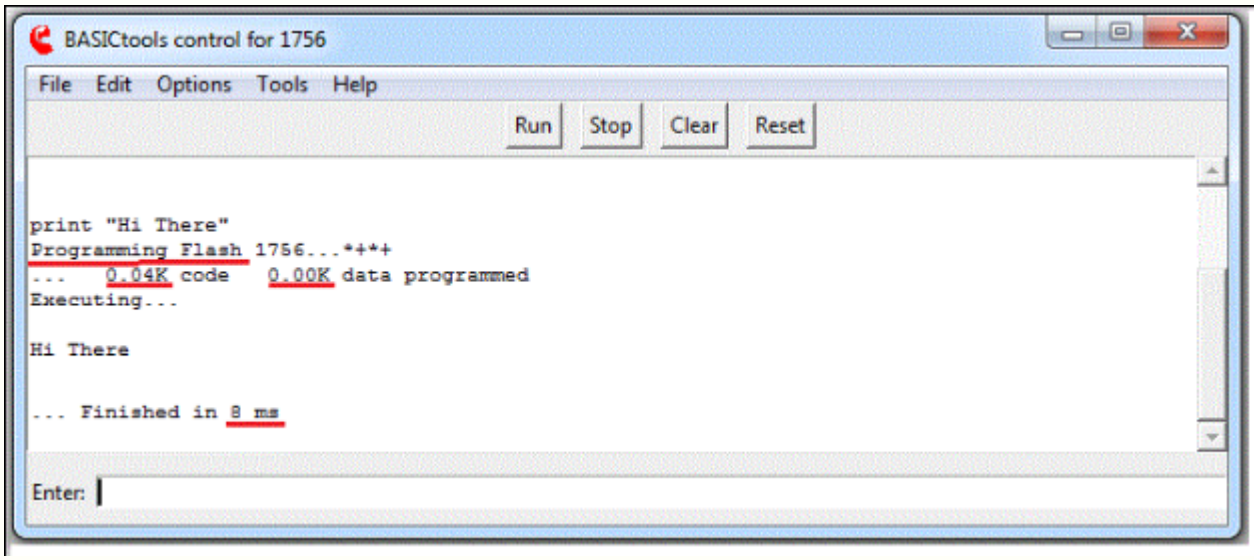


Now RUN the program



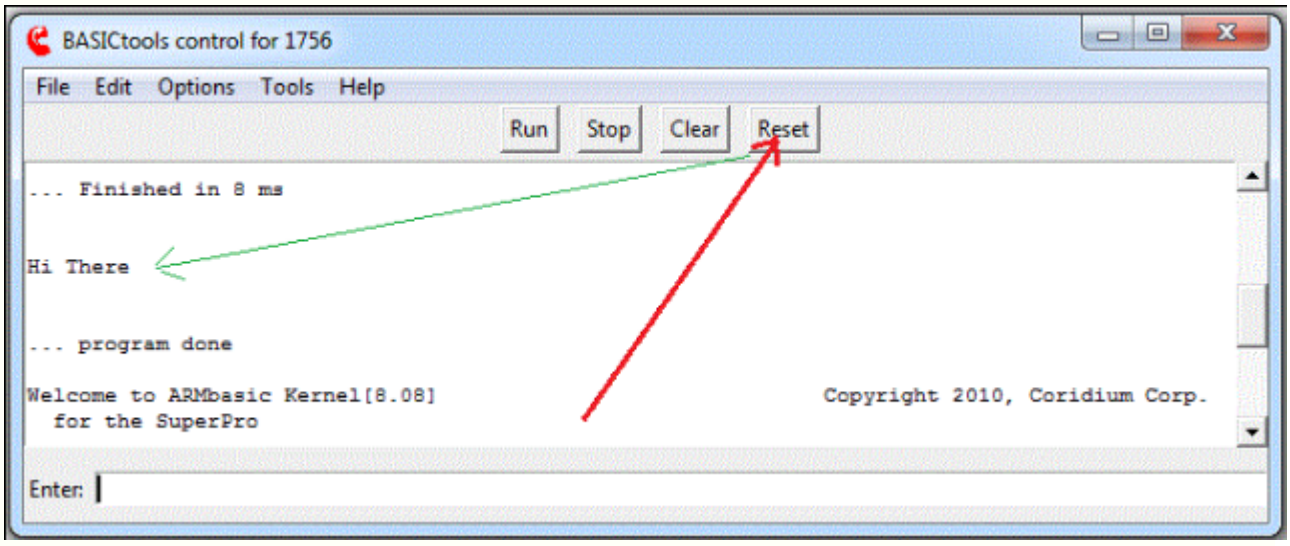
Which you can do by either typing RUN or hitting the RUN button at the top of the screen.

And see the results



You can notice a number of things. First the program is compiled and then written into Flash memory, and your program takes 40 bytes of code and less than 10 bytes of data space. Next the program will be executed, as evidenced by the output of "Hi There" to the console. ARM also reports back how long the program executed, in this case 8 msec, which is mostly startup time.

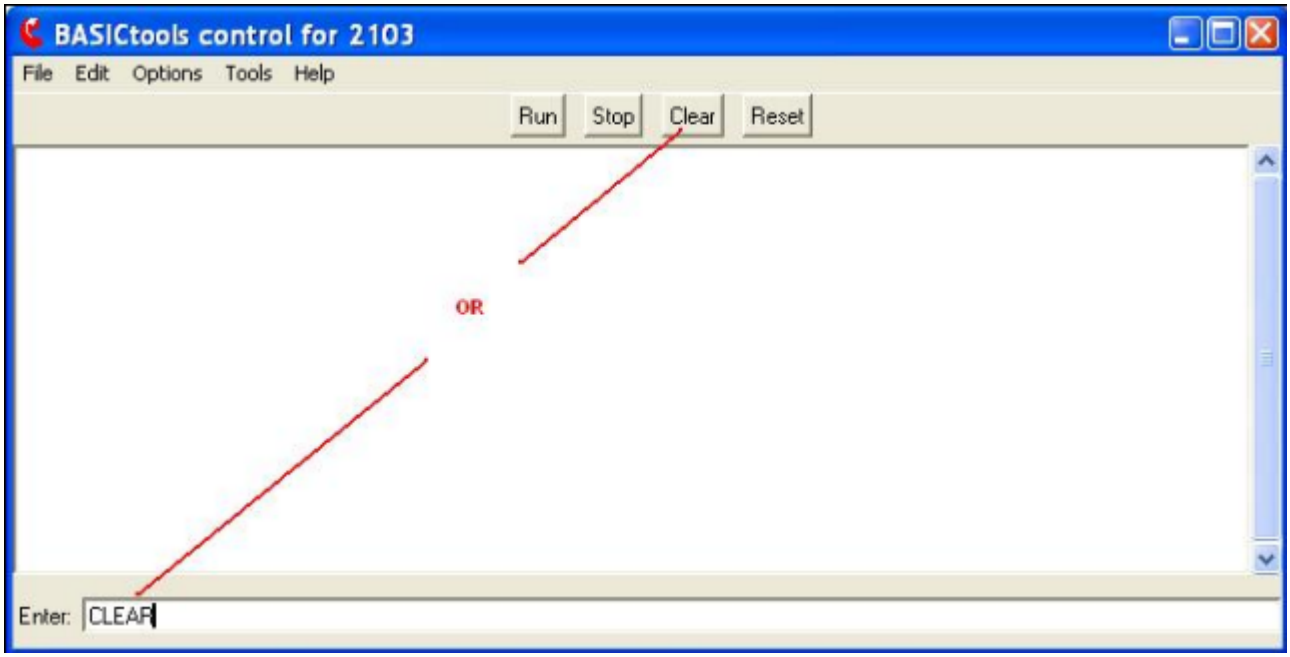
Also your program is now saved in the ARM Flash memory. And it will be executed the next time the board is RESET. So try that...



Now wiggle some pins from a program

Step 4: Programming the IO

Clear previous program



To begin a new program, you should CLEAR the previous one. You can do this with either the button or by typing clear.

A program that uses IO

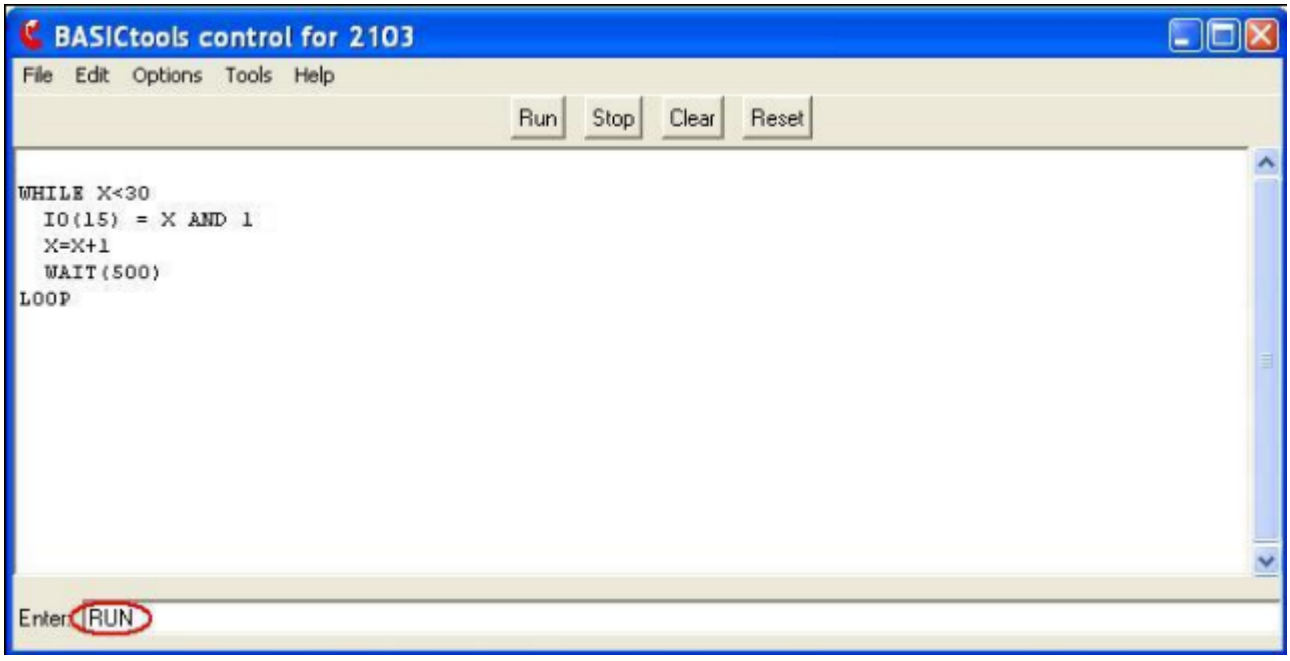
For the LPC11U37 - ARM Stamp and LPC1114 the LED is connected to P0(1).

```
WHILE X<30
   IO(1) = X and 1    ' IO() sets pin direction and state
    X=X+1
   WAIT(500)
 LOOP
```

For the SuperPRO and PROplus. Use the following. LED is connected to P2(10).

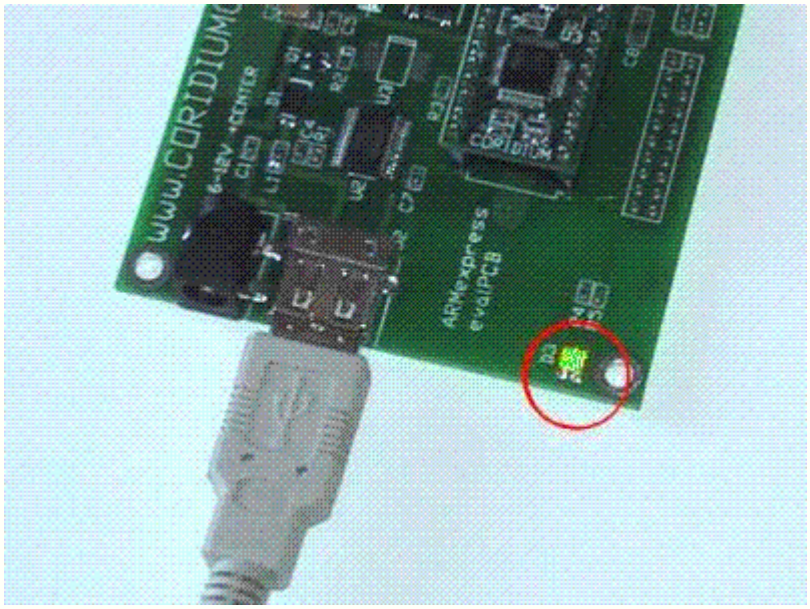
```
                        ' port 2 starts at 64
WHILE X<30
   IO(64+10) = X and 1
    X=X+1
   WAIT(500)
 LOOP
```

Now RUN the program

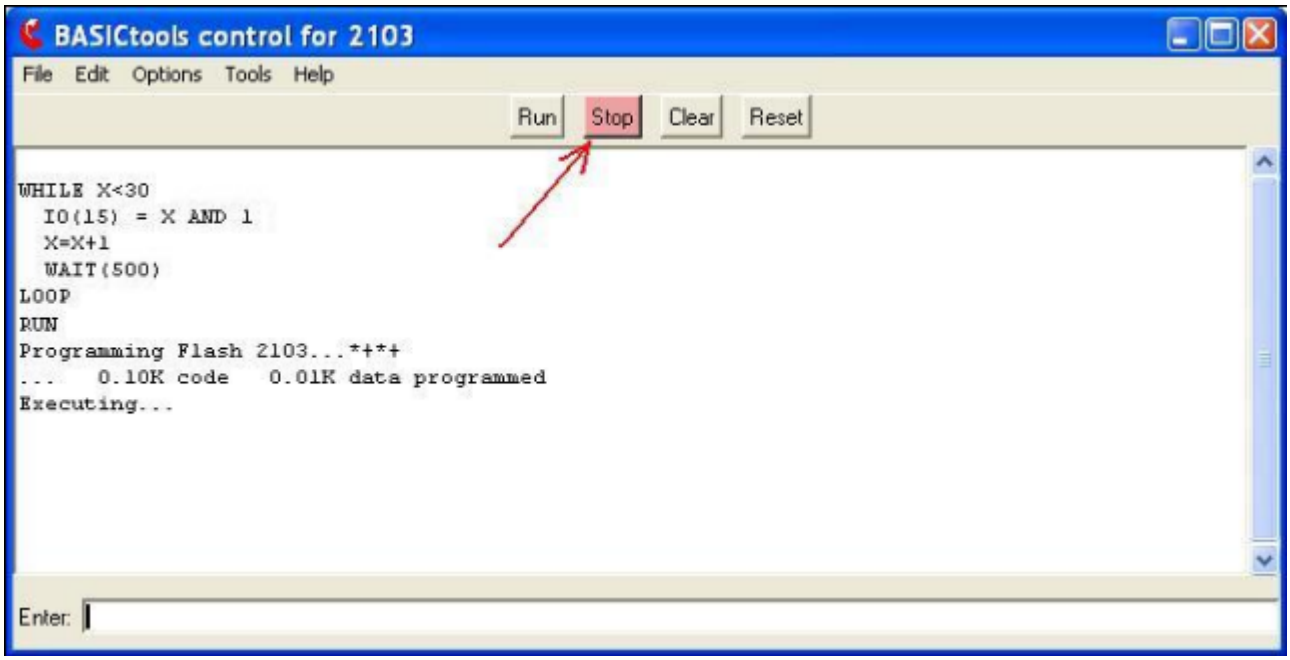


The LED on the PCB should pulse 15 times.

And see the results



Stop the program



To stop a running program simply press the Stop button.

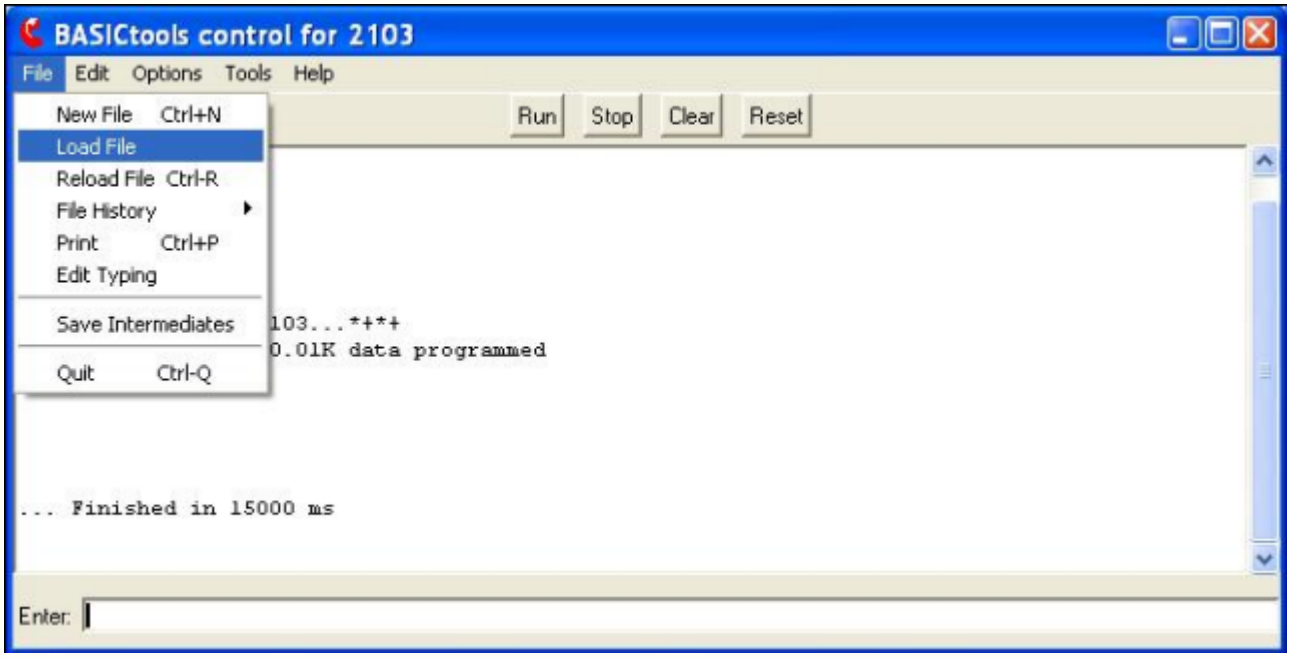
On to Step 5

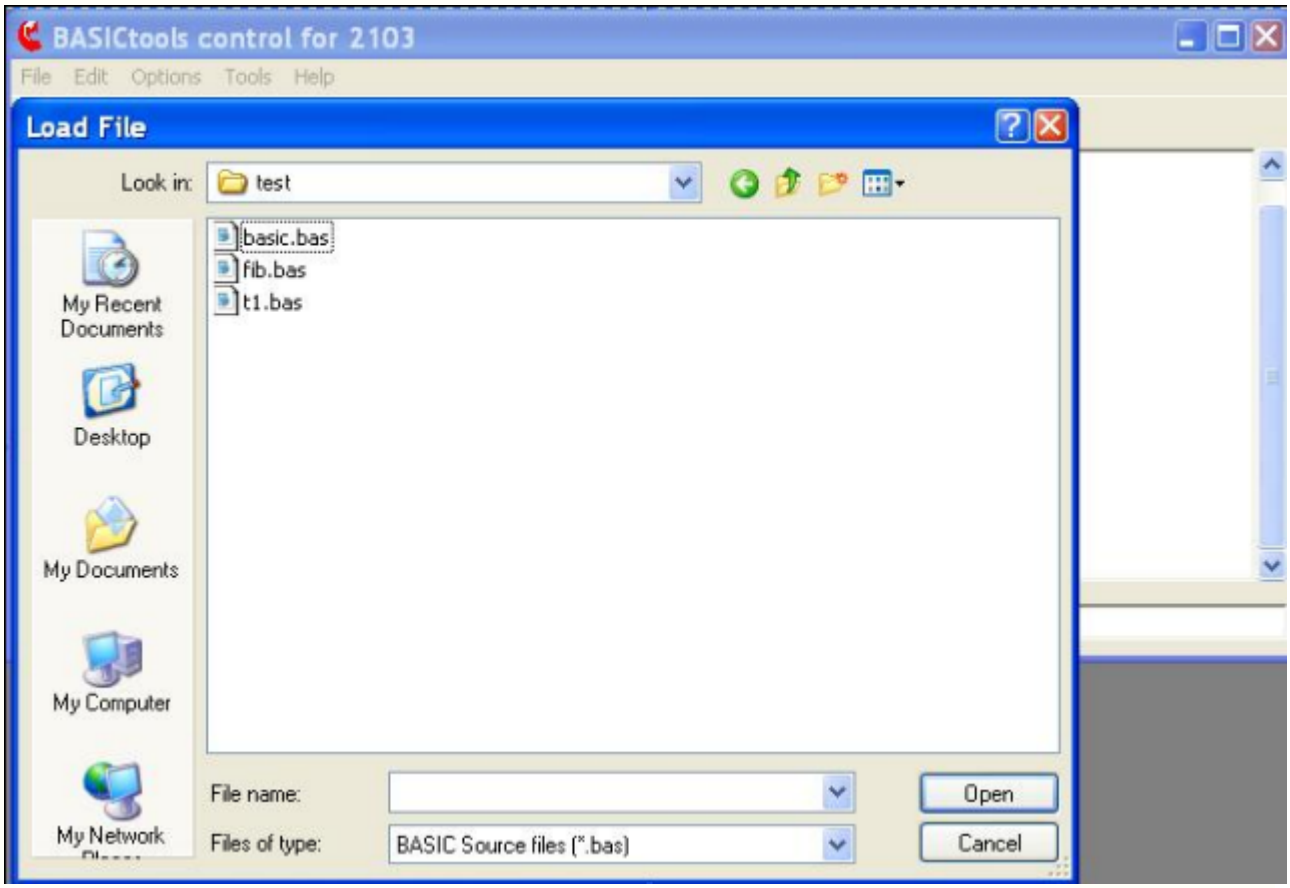
Step 5: More Complex Programming

Choose a File

While the Enter line can be useful for small programs or quickly checking out hardware, you will probably soon need to write larger programs. The way to do this is with a text editor. We don't enforce any text editor on you, you can choose your favorite. We tend to use the **Crimson Editor**, though a number of users are liking **NotePad Plus (NPP)**. Once you've typed up your program you can load that with BASICtools. It is easier to create a larger program with a text editor and then to Load File. You can link BASICtools to your favorite editor with the options (see the next section), or launch the original Windows Notepad if no editor is chosen.

Also the Enter line is limited in that #include <library> may be used, but the general pre-processor #include and other #directives should be avoided when typing a program a line at a time.





You're now ready to start tackling your application. Check with the **Coridium Forum** for files and help from other users of ARMbasic products.

For more details on the BASICtools IDE check the **next page**.

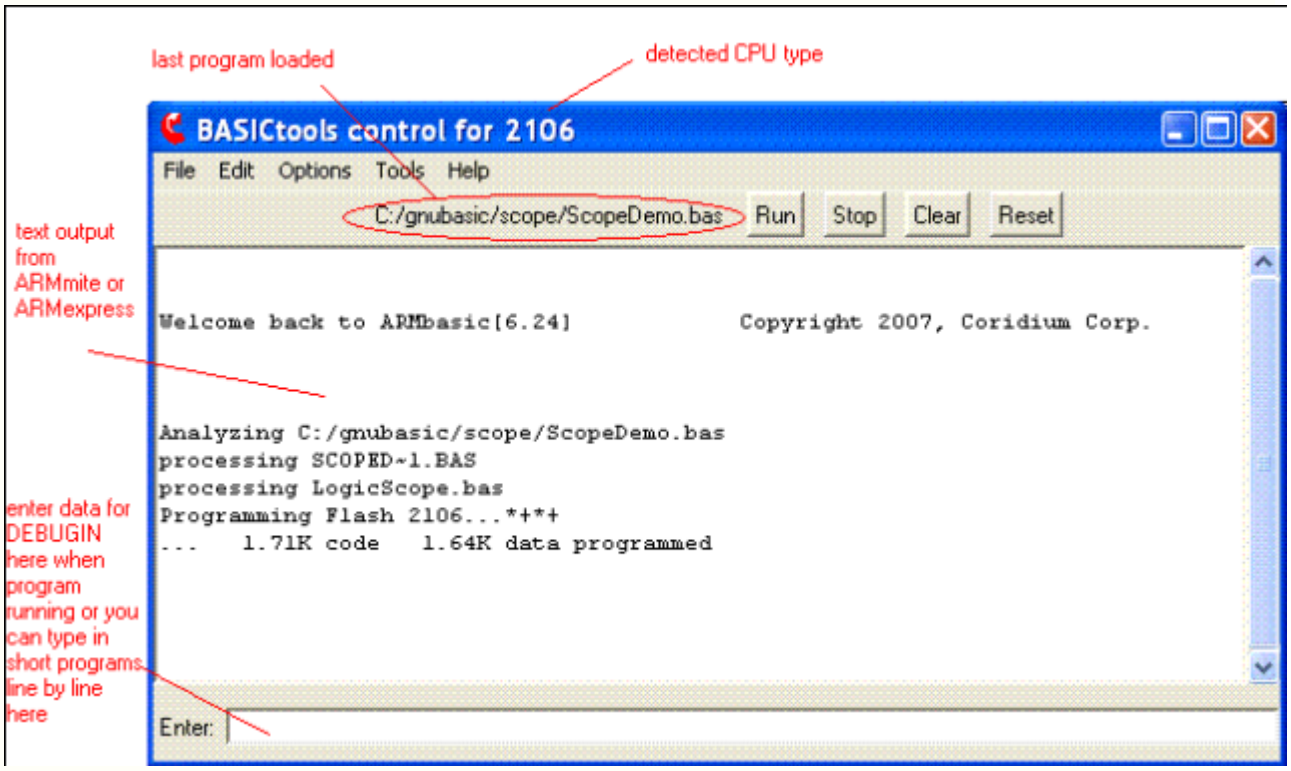
BASICtools Features

BASICtools startup

When BASICtools starts up, it will STOP any user program. So if you find yourself with a program flooding the PC serial port with data, disconnect the serial connection, and reconnect and then choose the same serial port under the Options menu.

This will clear and initialize the serial driver. Reopening the serial port will also stop any user program from starting. A good idea is to erase that program, and a simple way to do that is to enter a 1 line program like PRINT and hit run which will erase the runaway program.

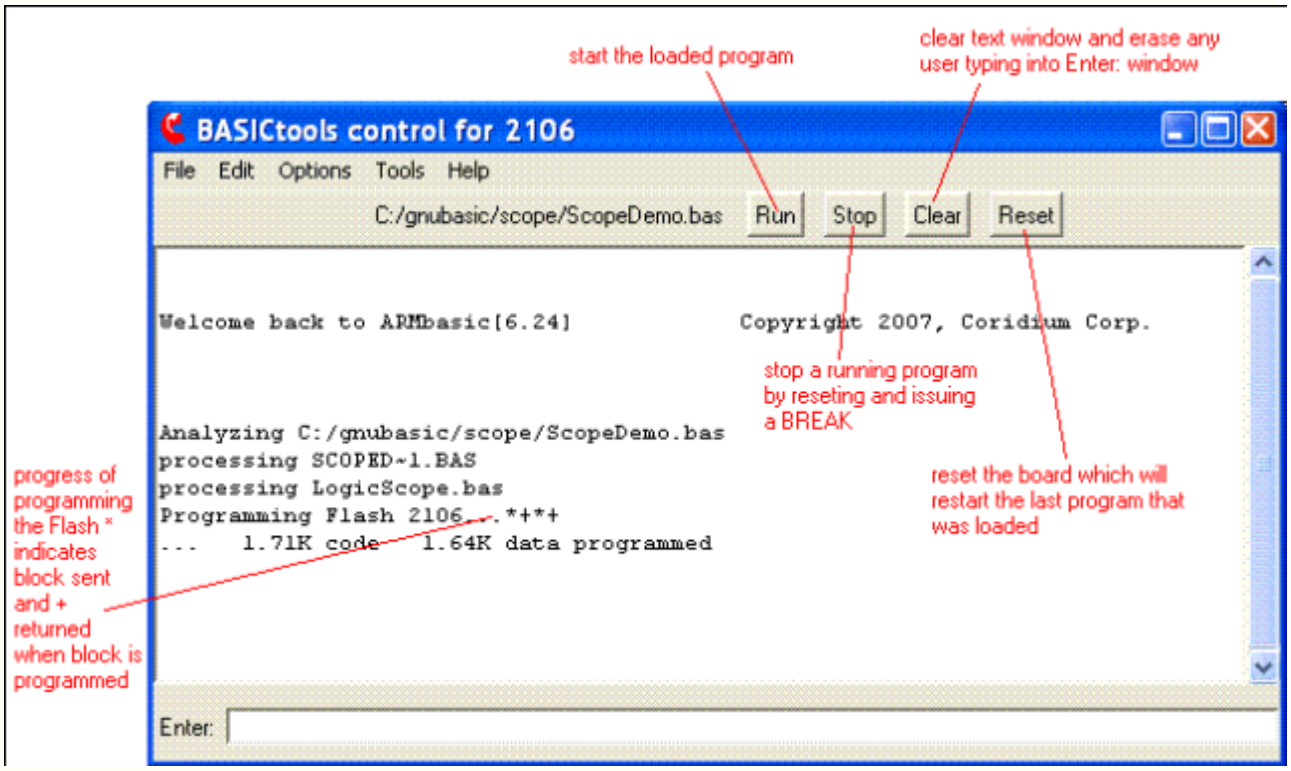
BASICtools Layout



keyw ords: enter line debugin type BASIC commands

Buttons

..



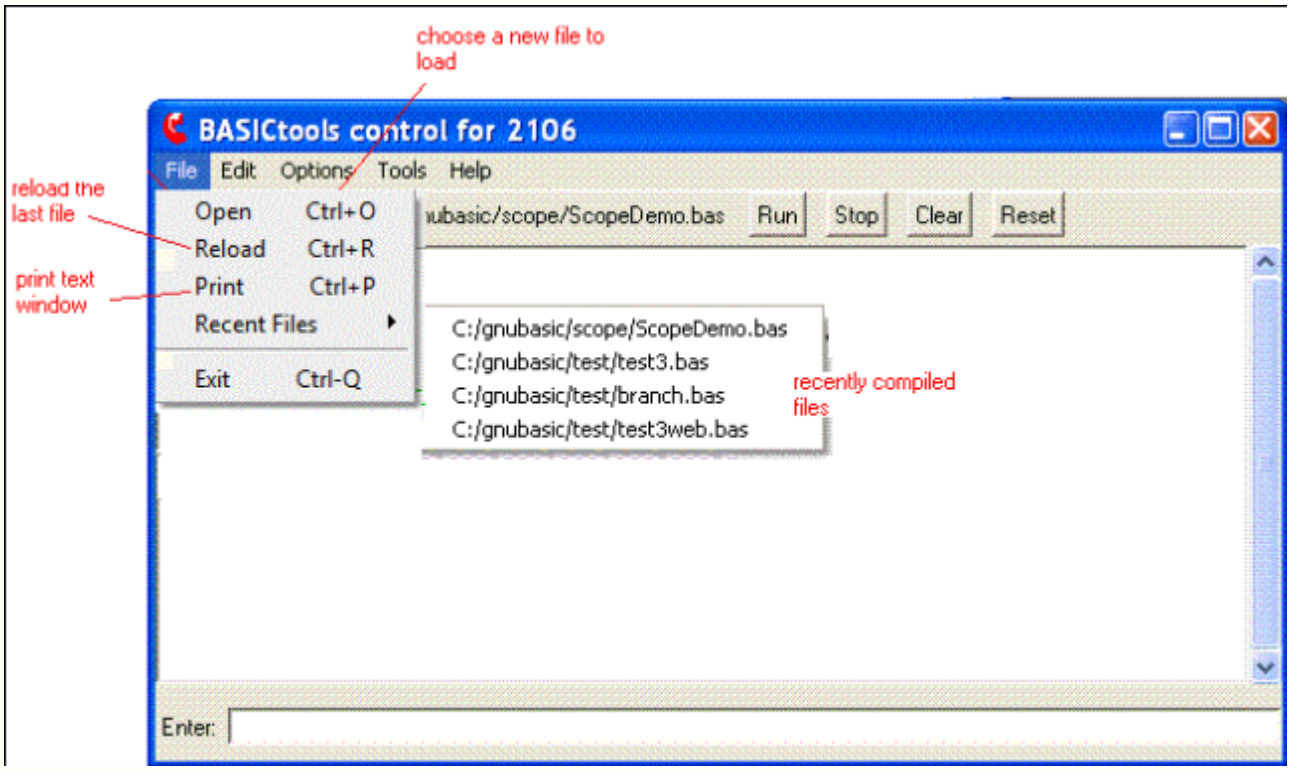
The CLEAR button only erases the display screen and the buffer on the PC of statements you have typed into the Enter window.

To erase the program, load a new program, either a line at a time or using the Load menu.

keyw ords: reset button stop button run button clear button

File Menu

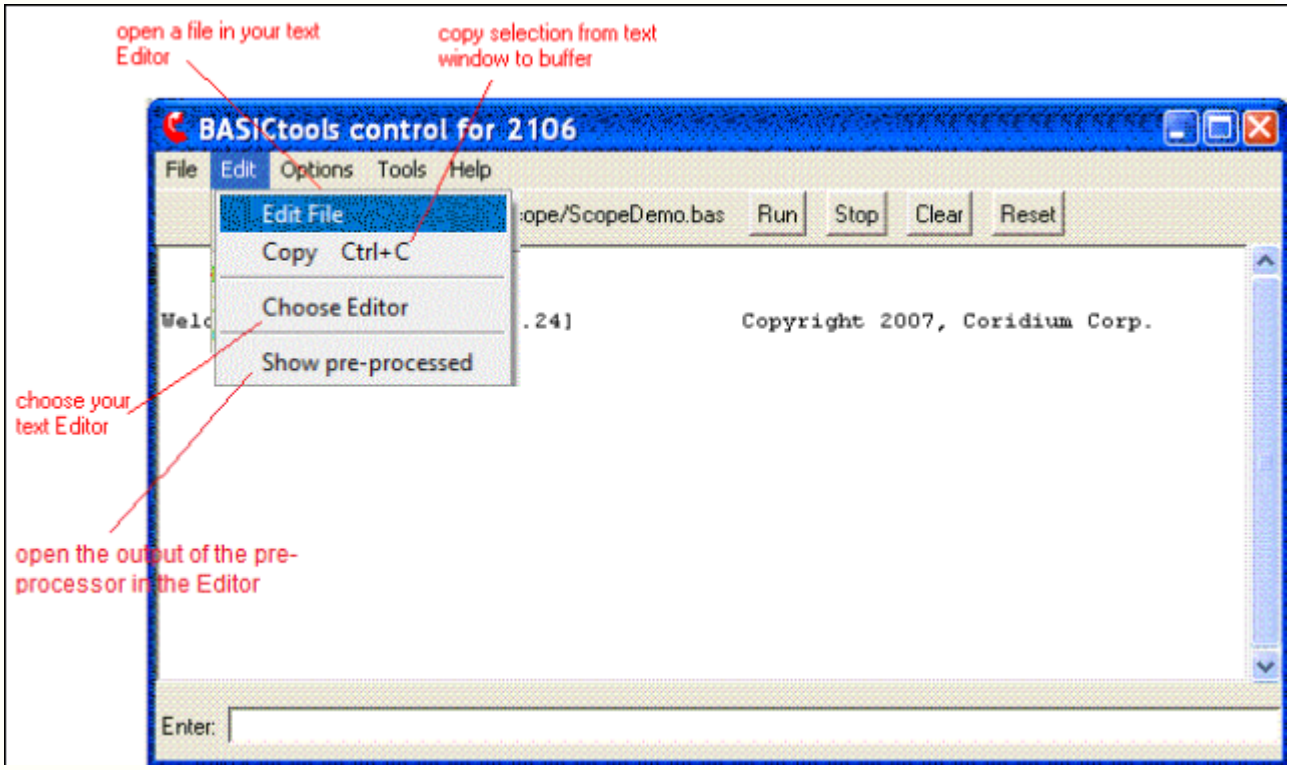
..



file load file reload file print save file quit

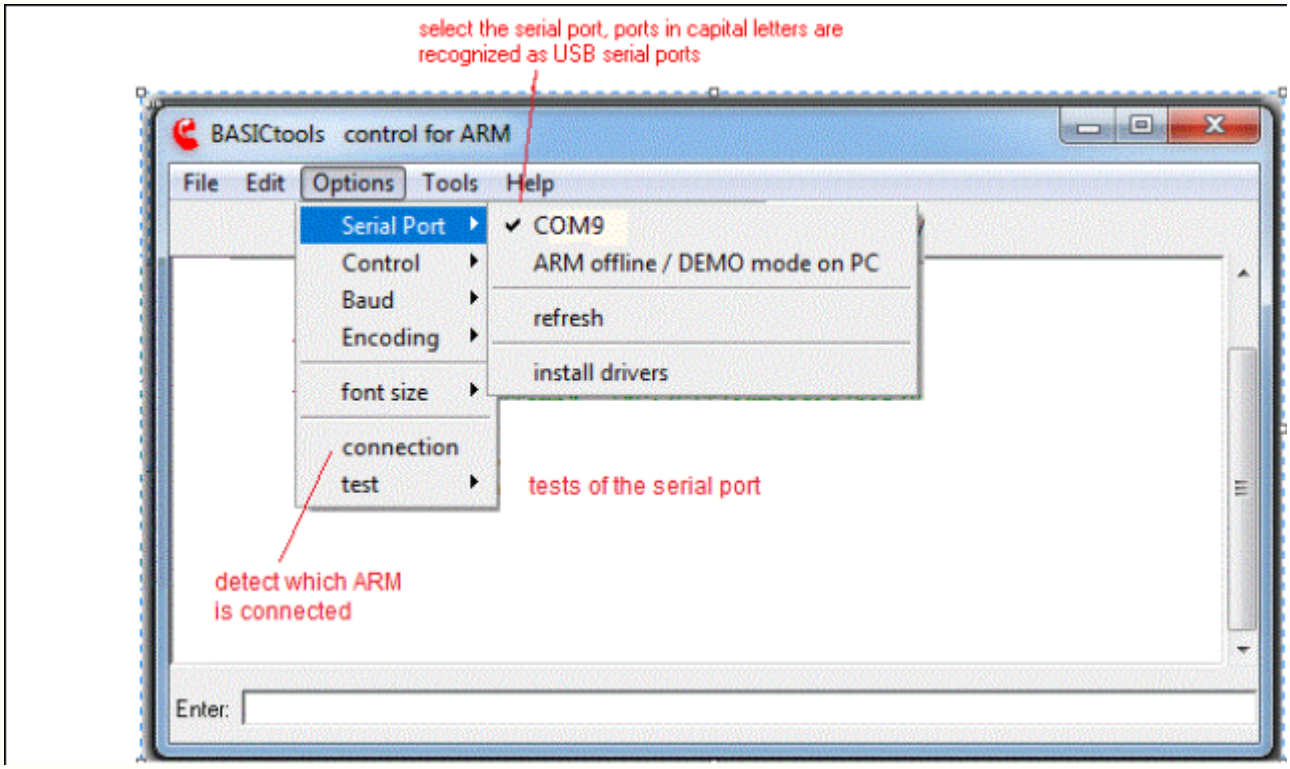
Edit Menu

..



keyw ords: edit choose editor

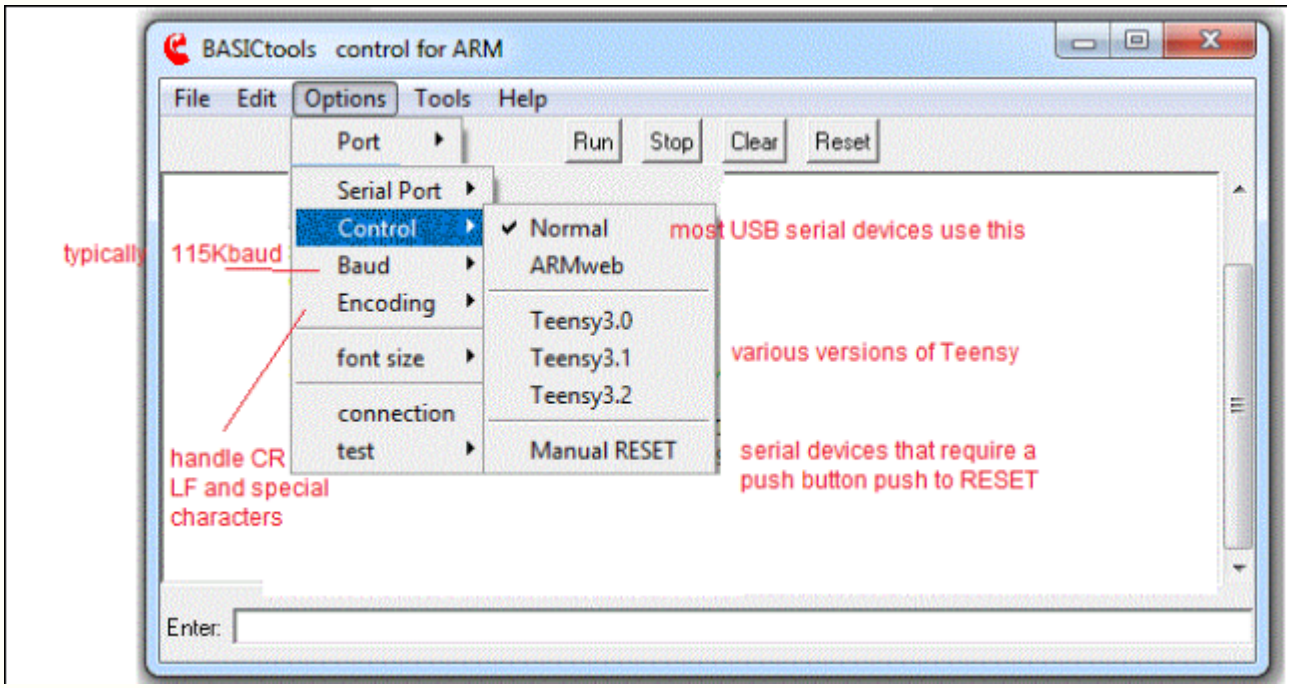
Options Menu



Refresh will check for serial devices again, it is useful if you plugged a device in after starting BASICtools.

keyw ords: options port baud new line char mode PC compile control throttle

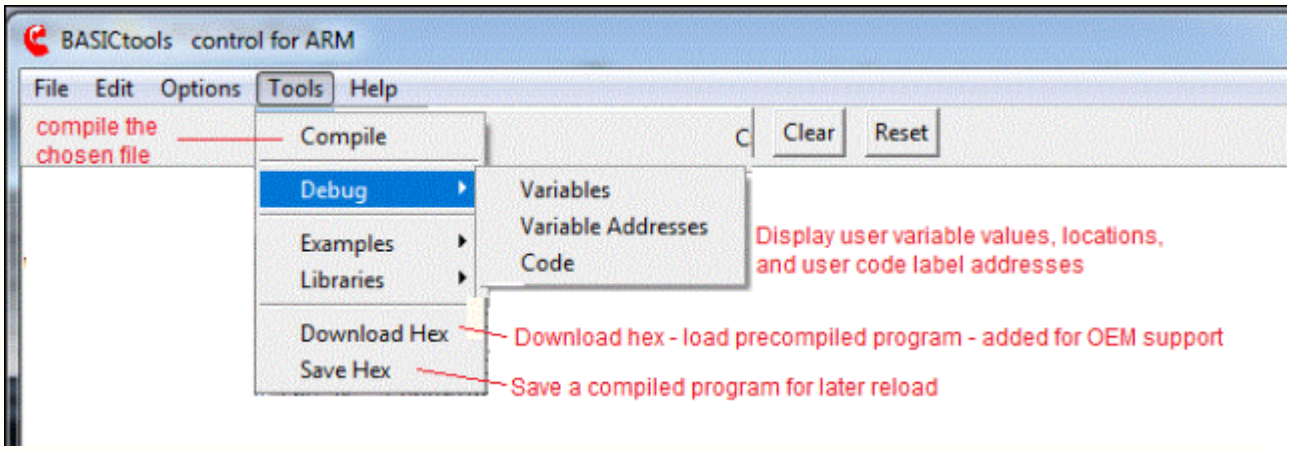
Control Menu



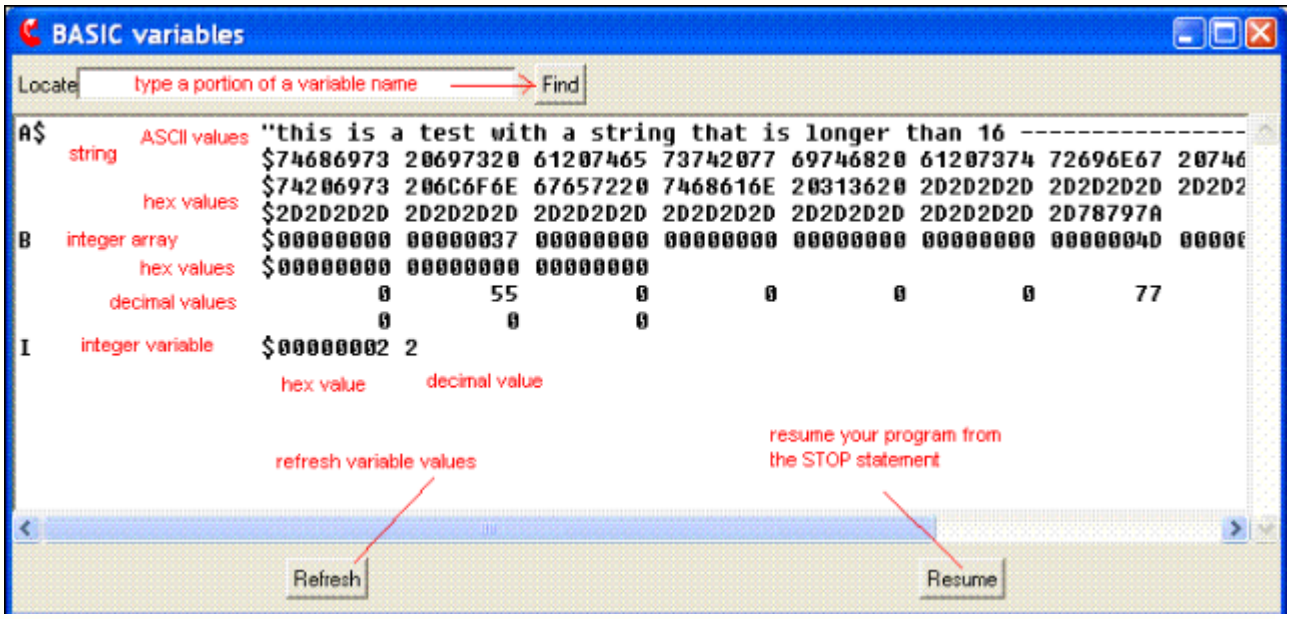
keyw ords: options port baud new line char mode PC compile control throttle

BASIC variable viewer

Open this window from the Tools Menu (variables)

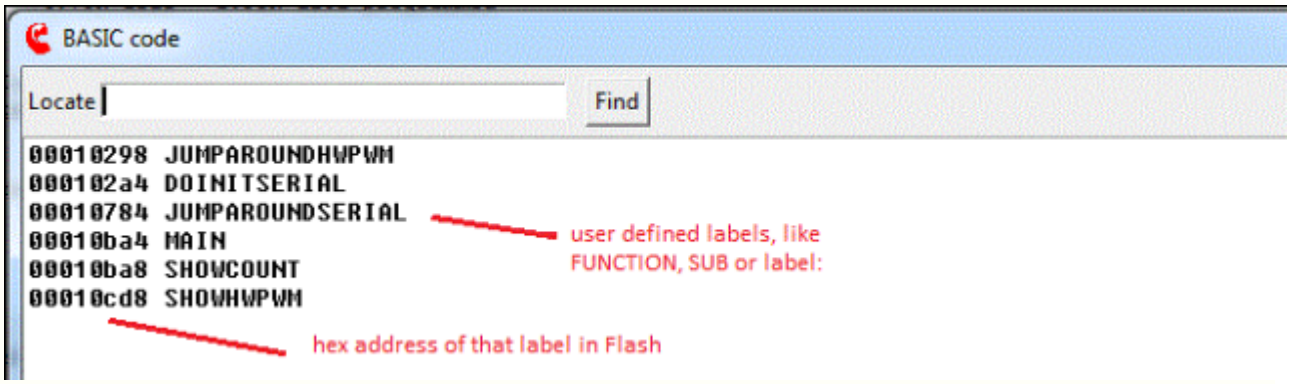


variables window



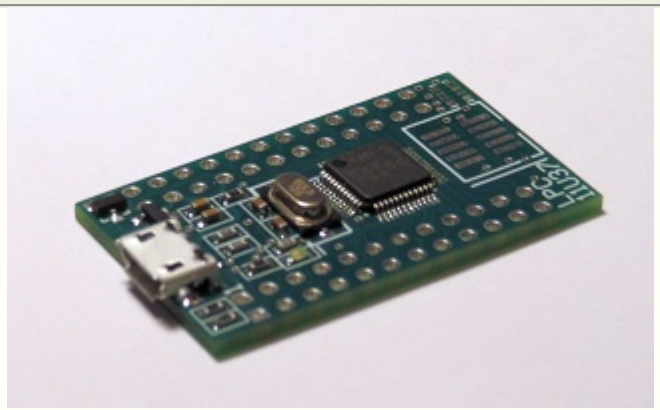
This page is active when your program ENDs or hits a STOP statement or has been STOPed with the button.

code window



keyw ords: variable dump breakpoint STOP view memory

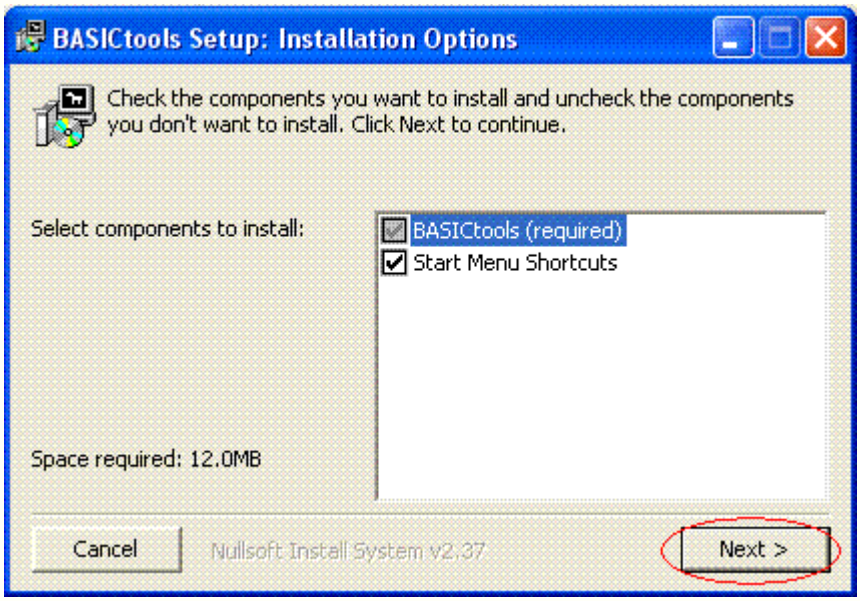
ARMmite, PRO family and BASICchip Getting Started



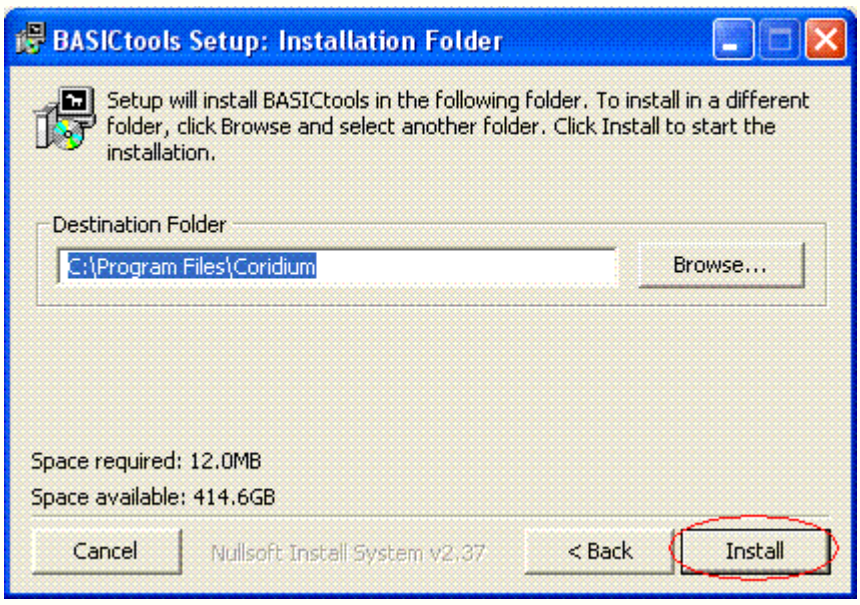
Step 1: Install Software

Coridium Products use a BASIC Compiler that runs on the PC. Coridium supplies BASICtools which includes a terminal emulator and IDE that is specifically designed for the PRO family, BASICchip, ARMmite and Ethernet boards. Also, a number of help files and documents about the ARMbasic will be installed on the machine at this time. This installer is meant for Windows 10, WIndows 8, Windows 7, or Windows XPboth 32 and 64 bit version.

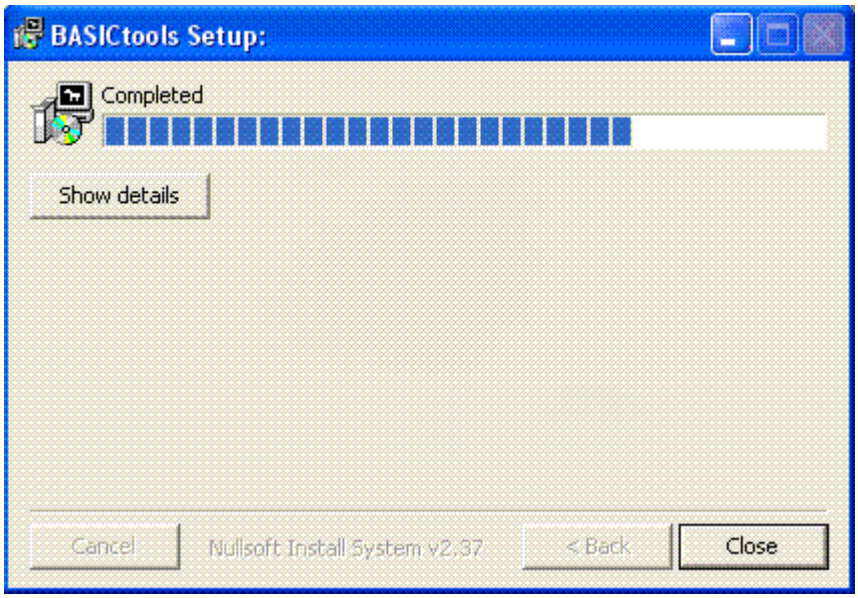
Download **www.coridium.us/files/setupBASIC.exe** from the web, and run the program to start the installation.



Click **Next** to get started.



Accept the defaults and **Install**. You may chose a different target directory.



The installation will now run, and when it finishes hit **Close** .

And its as easy as that.

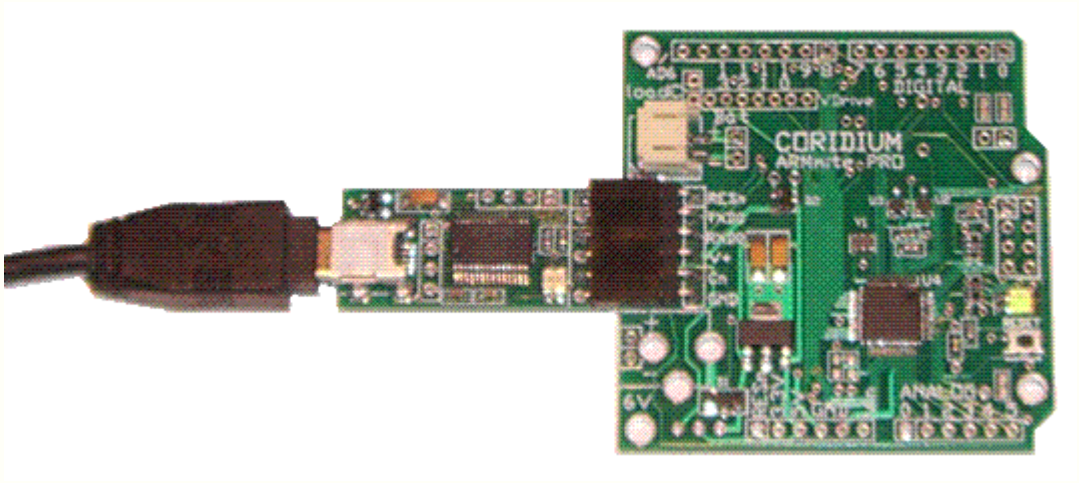
On to Connecting to SuperPRO

On to connect PRO family boards

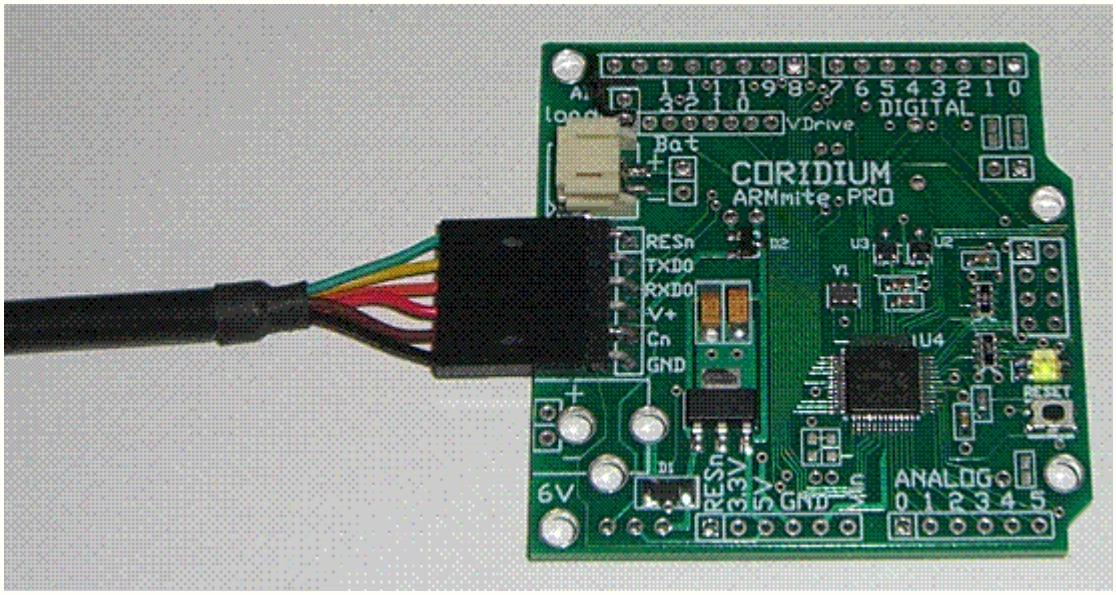
Step 2: Connect USB on the PRO family

Connect Coridium USB Dongle to ARMmite PRO

The ARMmite PRO Eval Kit comes with a USB dongle and cable. This dongle and cable allows you to connect the ARMmite PRO directly to a computer equipped with USB. When connected to a PC, power is supplied by the PC, the optional power connection is not required, but both may be safely connected.

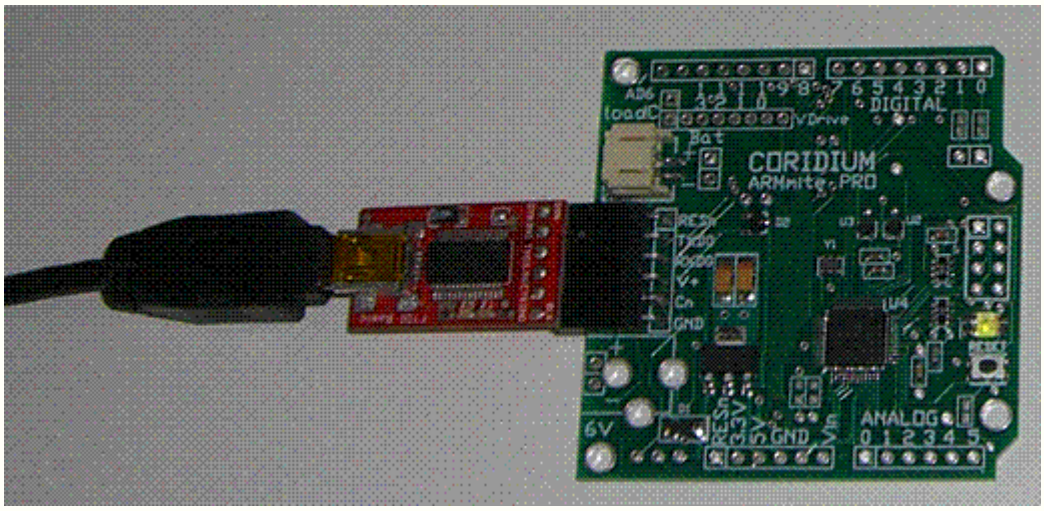


Connect FTDI cable to ARMmite PRO



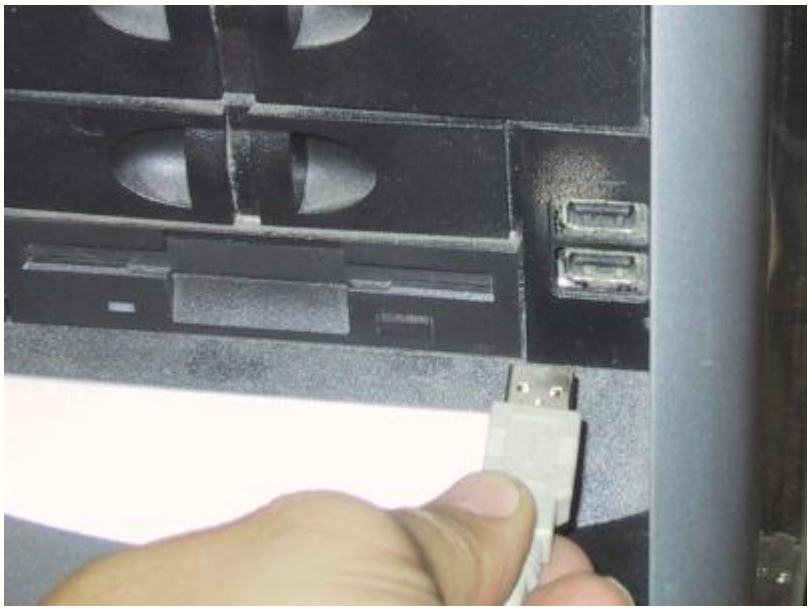
Connect black wire to GND. This cable is available at **Digikey** or the **Makershed**. This cable connects RTS to RESETn, BASICtools support this.

Connect SparkFun USB Dongle to ARMmite PRO



USB dongle from **Sparkfun** shown

Connect USB Cable to Computer



Locate the USB jack on your computer and plug the other end of the cable into it.

Wow did I really take that picture on a computer with floppies. USB connectors have become more standard since then.

Please Consult Installation Guides

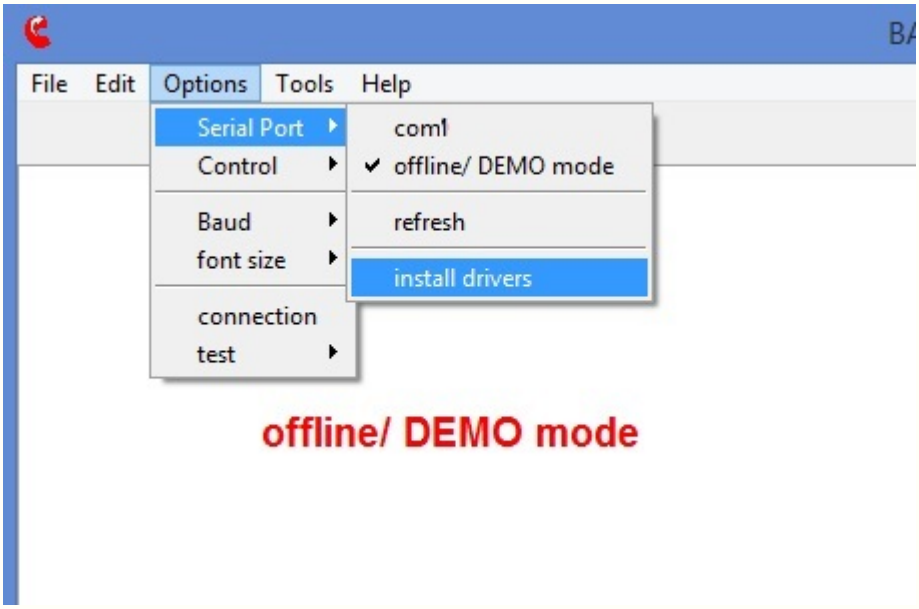
Most PC's will sound a tone that indicates a new USB device has been connected. Most Windows Vista and 7 systems will either include the FTDI device driver or are able to download it automatically from the network.

If your system is unable to do that.

Install drivers for Coridium devices --

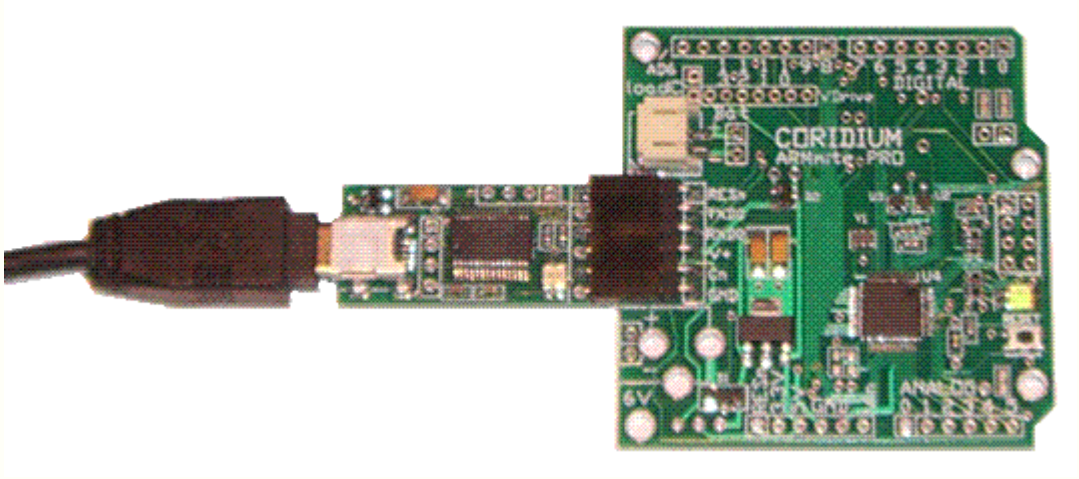
If no serial ports are active in your system you will be prompted to install drivers

Or if there are other serial devices in your system, you can install drivers from BASICtools menus



We try to keep up to date with the FTDI drivers for the chip we use, but up to date details are at the **www.ftdichip.com** VCP drivers page.

Driver Installation Complete, Confirm USB Connection



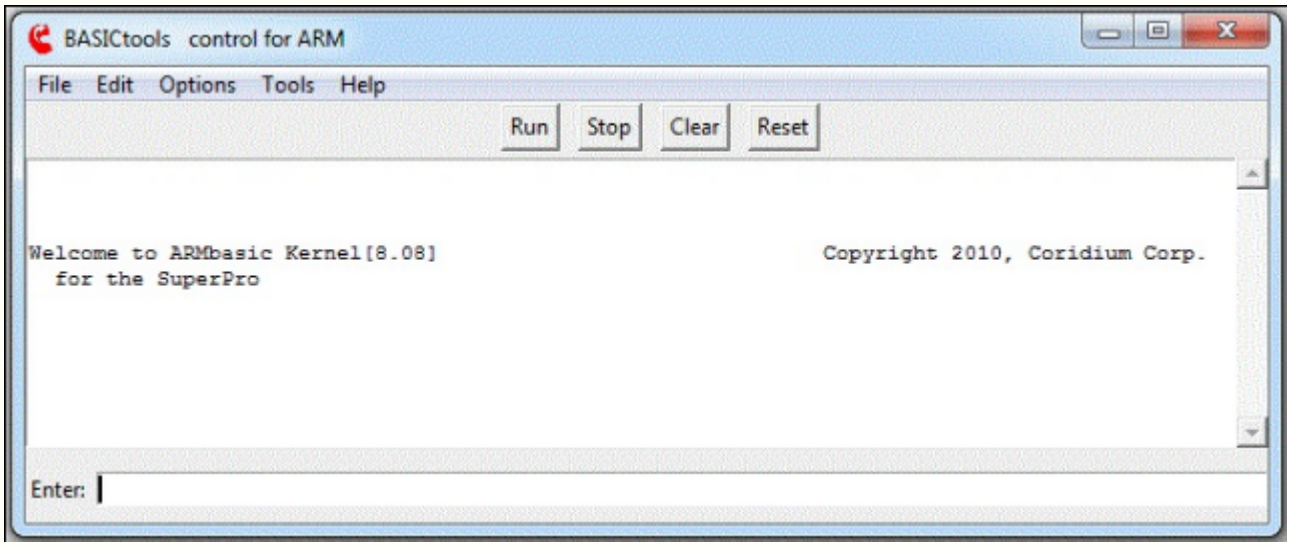
The ARMmite PRO will be powered from the USB bus, when using either the Coridium Dongle or the FTDI 5V cable. It may also be connected to a 6-7V DC power source simultaneously.

To verify connection with the USB and PC the LED on the Eval PCB should light up.

On to Step 3

Step 3: Writing your first Program with BASICtools

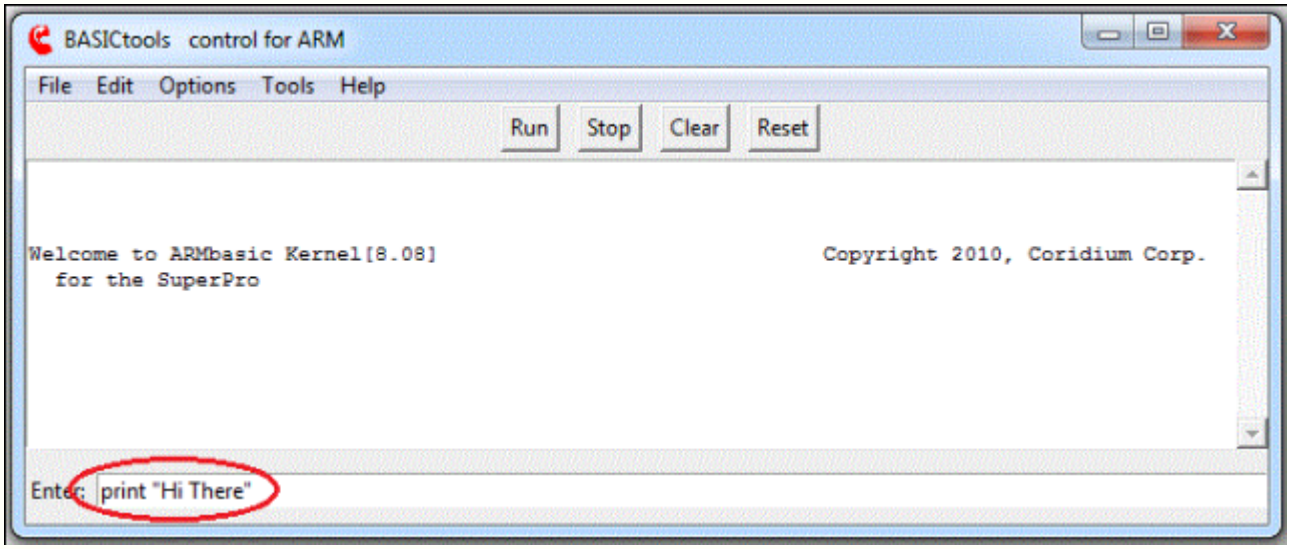
Start the BASICtools from the Start-Menu or from the Desktop Icon. You should see a welcome message which has been sent from the ARM processor-



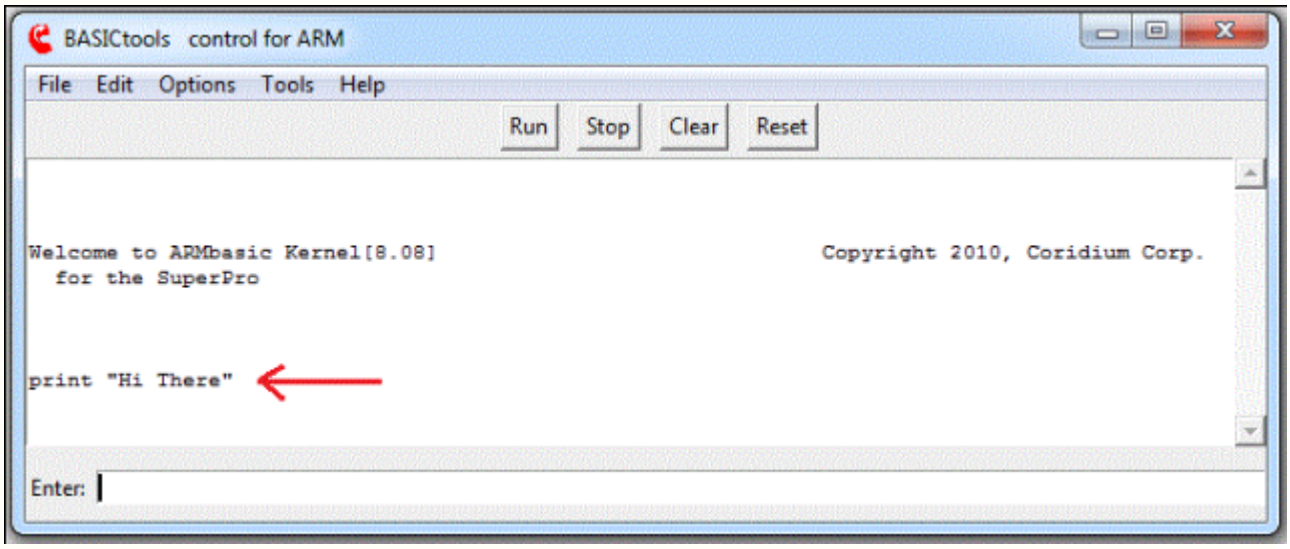
If you do not see this welcome, even after pushing the RESET button, then communication has not been established.

- check cables
- check power supply
- check COM port choice in BASICtools -> Options
- check baud rate in BASICtools -> Options
- on non-Coridium Boards, remove any BOOT select jumpers, press RESET again
- if still not working, check the**Trouble Shooting Section**

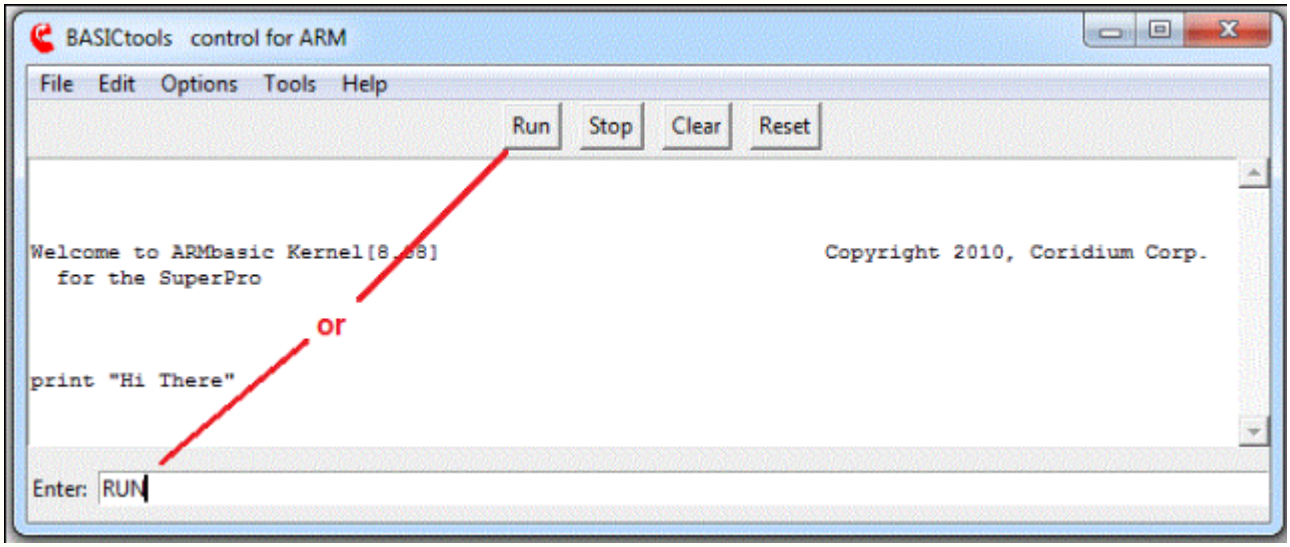
The traditional "Hi World" program



So type something like the traditional PRINT "Hi There"
When you hit the ENTER key it will be sent to the compiler on the PC and be echoed back in the console window. (below)

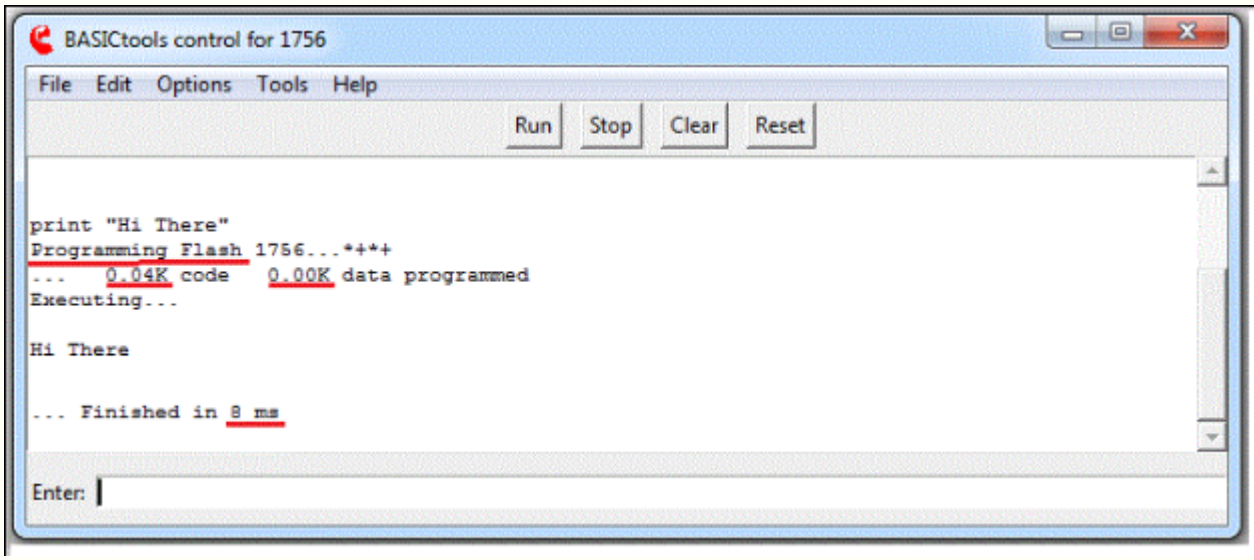


Now RUN the program



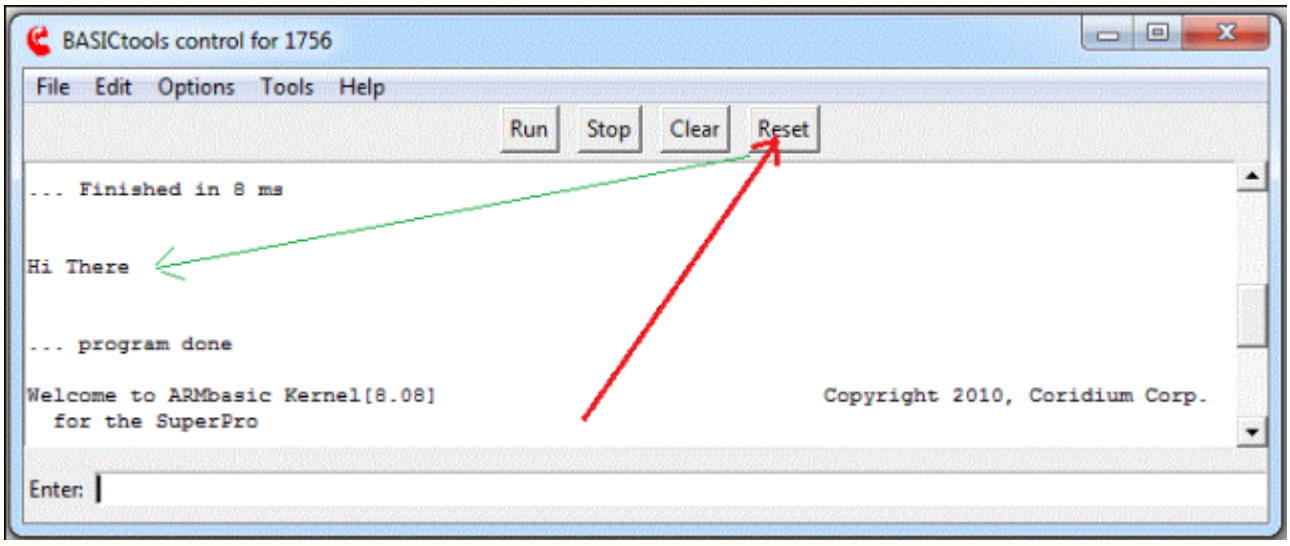
Which you can do by either typing RUN or hitting the RUN button at the top of the screen.

And see the results



You can notice a number of things. First the program is compiled and then written into Flash memory, and your program takes 40 bytes of code and less than 10 bytes of data space. Next the program will be executed, as evidenced by the output of "Hi There" to the console. ARM also reports back how long the program executed, in this case 8 msec, which is mostly startup time.

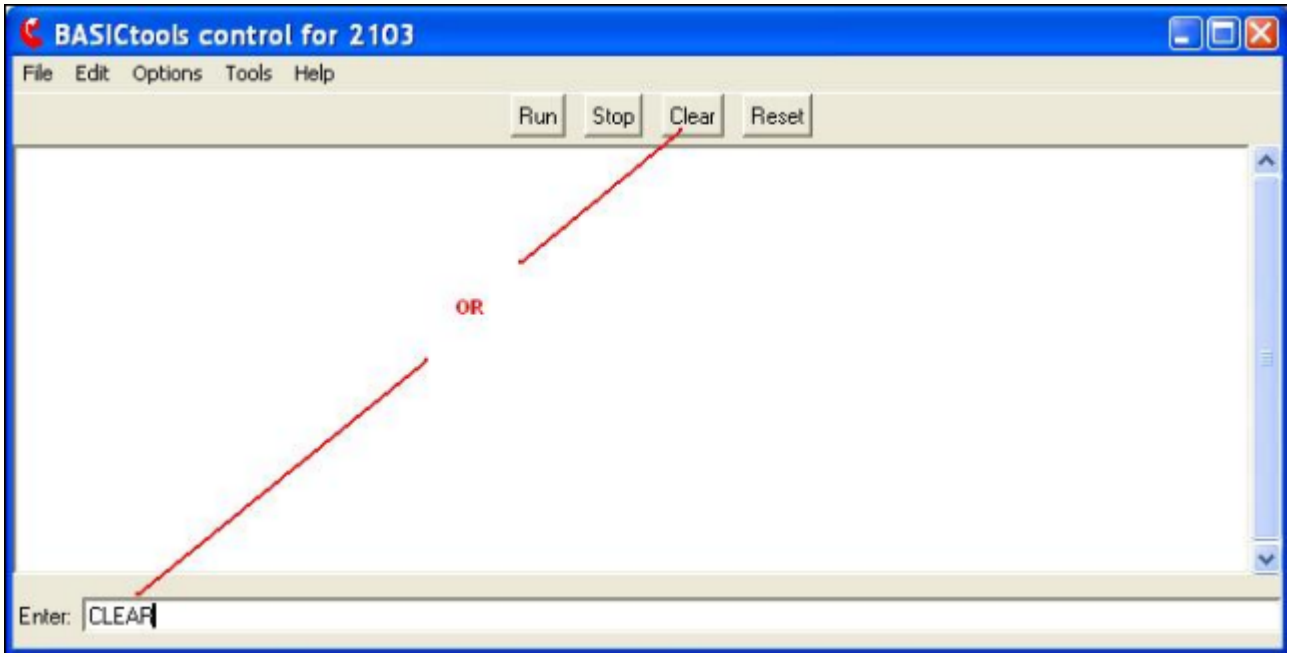
Also your program is now saved in the ARM Flash memory. And it will be executed the next time the board is RESET. So try that...



Now wiggle some pins from a program

Step 4: Programming the IO

Clear previous program



To begin a new program, you should CLEAR the previous one. You can do this with either the button or by typing clear.

A program that uses IO

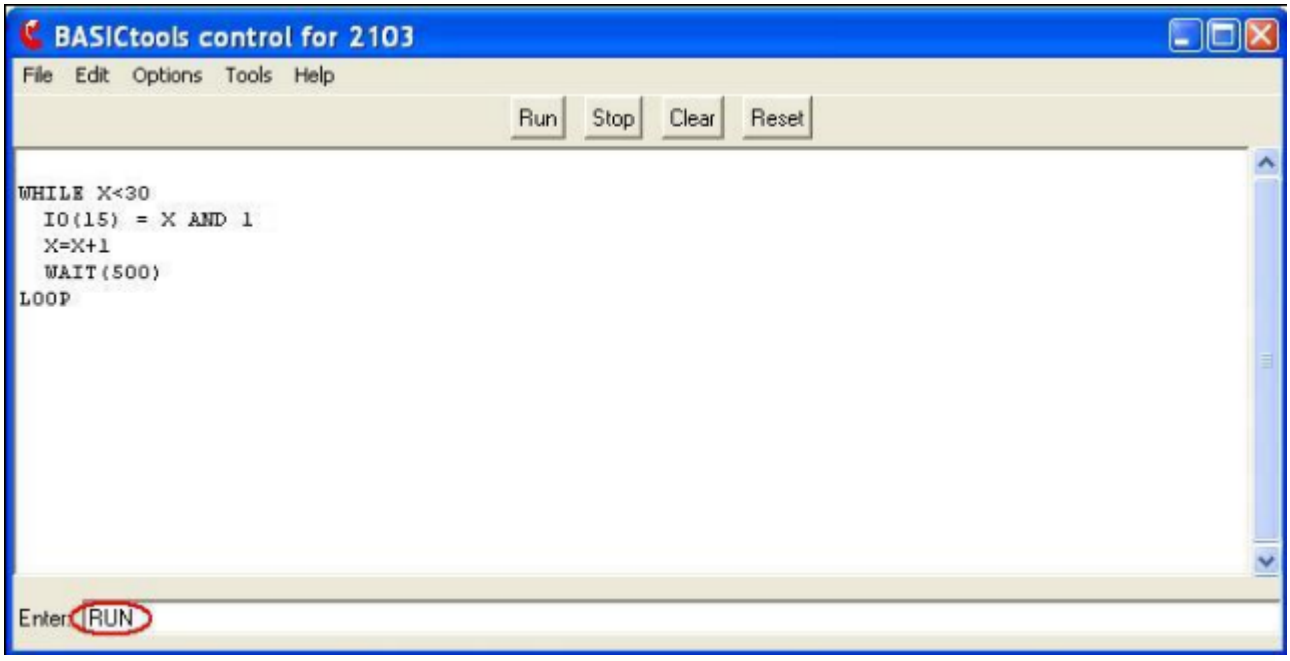
For the LPC11U37 - ARM Stamp and LPC1114 the LED is connected to P0(1).

```
WHILE X<30
   IO(1) = X and 1    ' IO() sets pin direction and state
    X=X+1
   WAIT(500)
 LOOP
```

For the SuperPRO and PROplus. Use the following. LED is connected to P2(10).

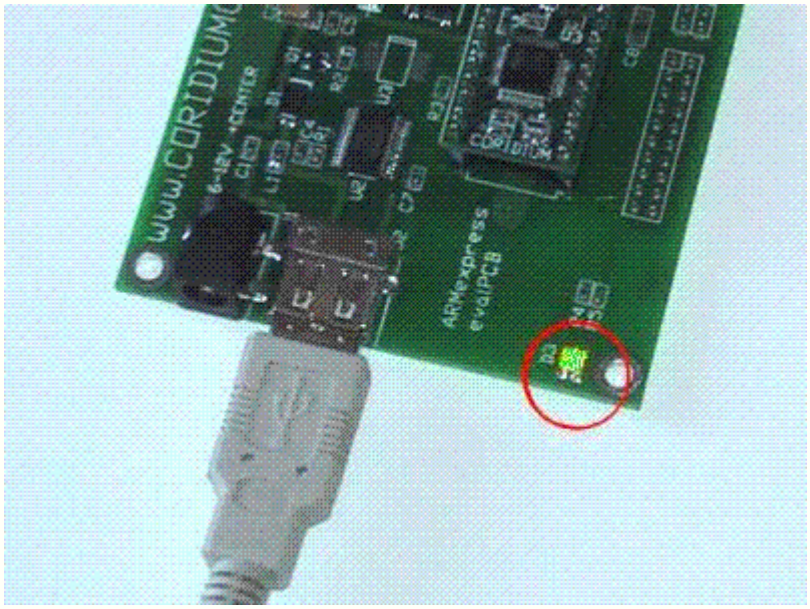
```
                        ' port 2 starts at 64
WHILE X<30
   IO(64+10) = X and 1
    X=X+1
   WAIT(500)
 LOOP
```

Now RUN the program

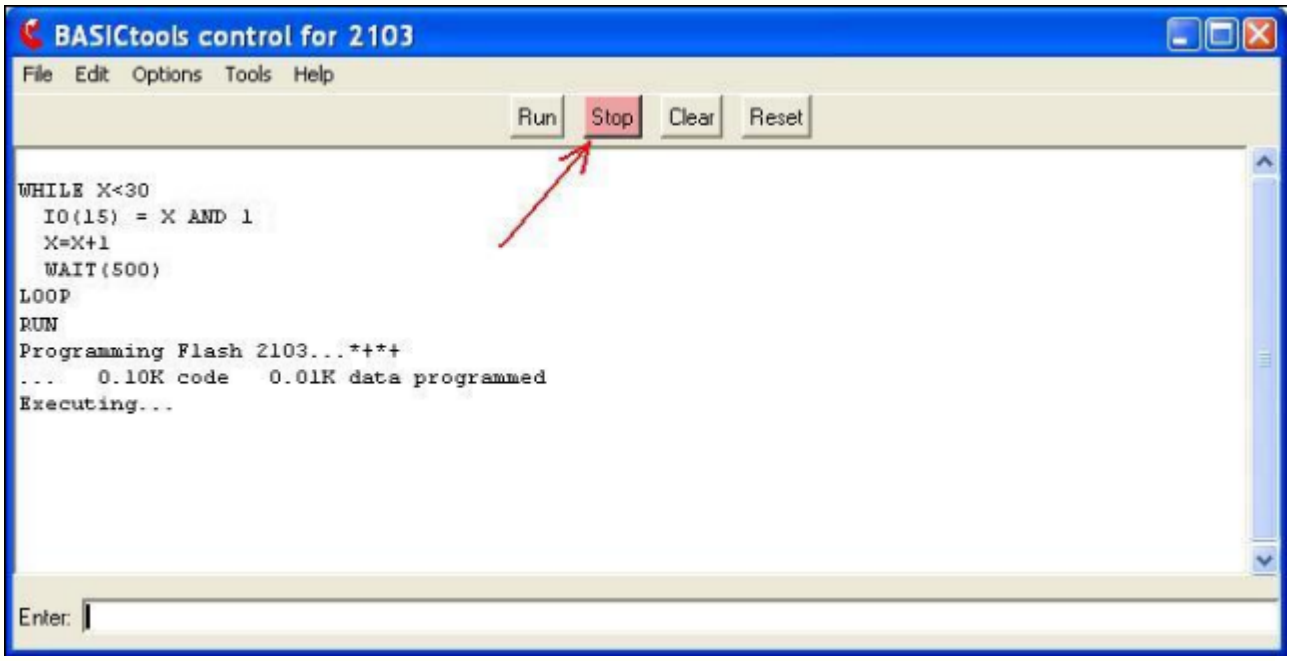


The LED on the PCB should pulse 15 times.

And see the results



Stop the program



To stop a running program simply press the Stop button.

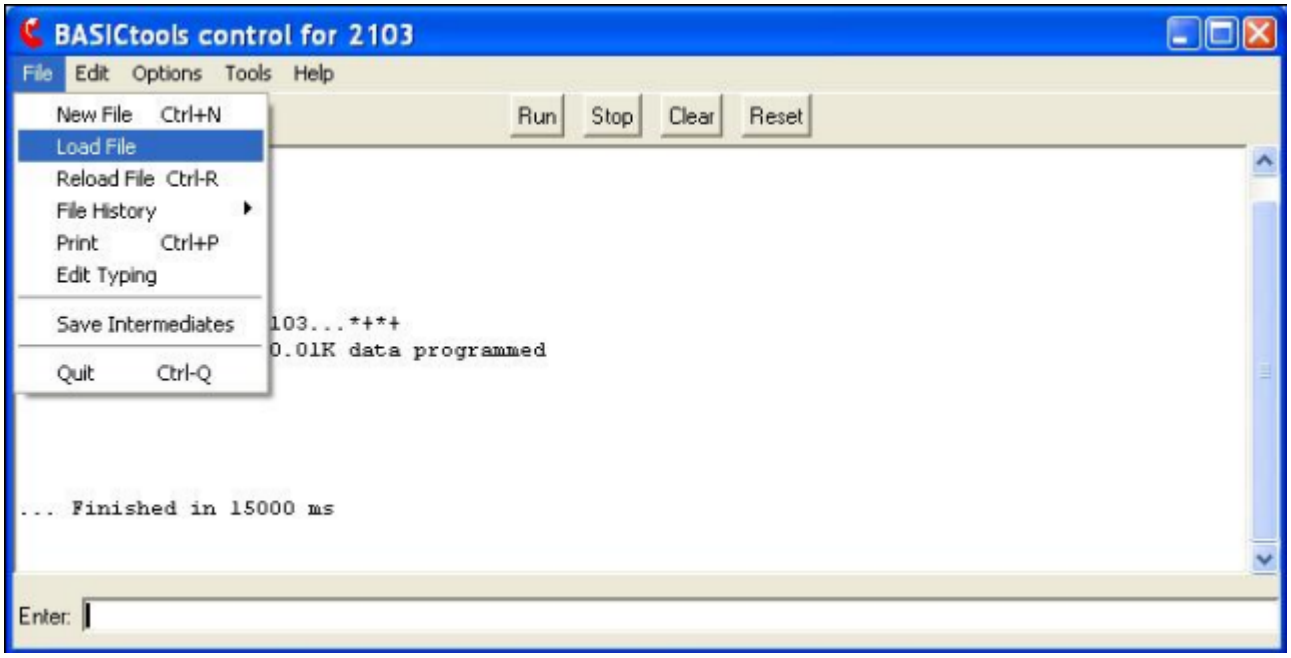
On to Step 5

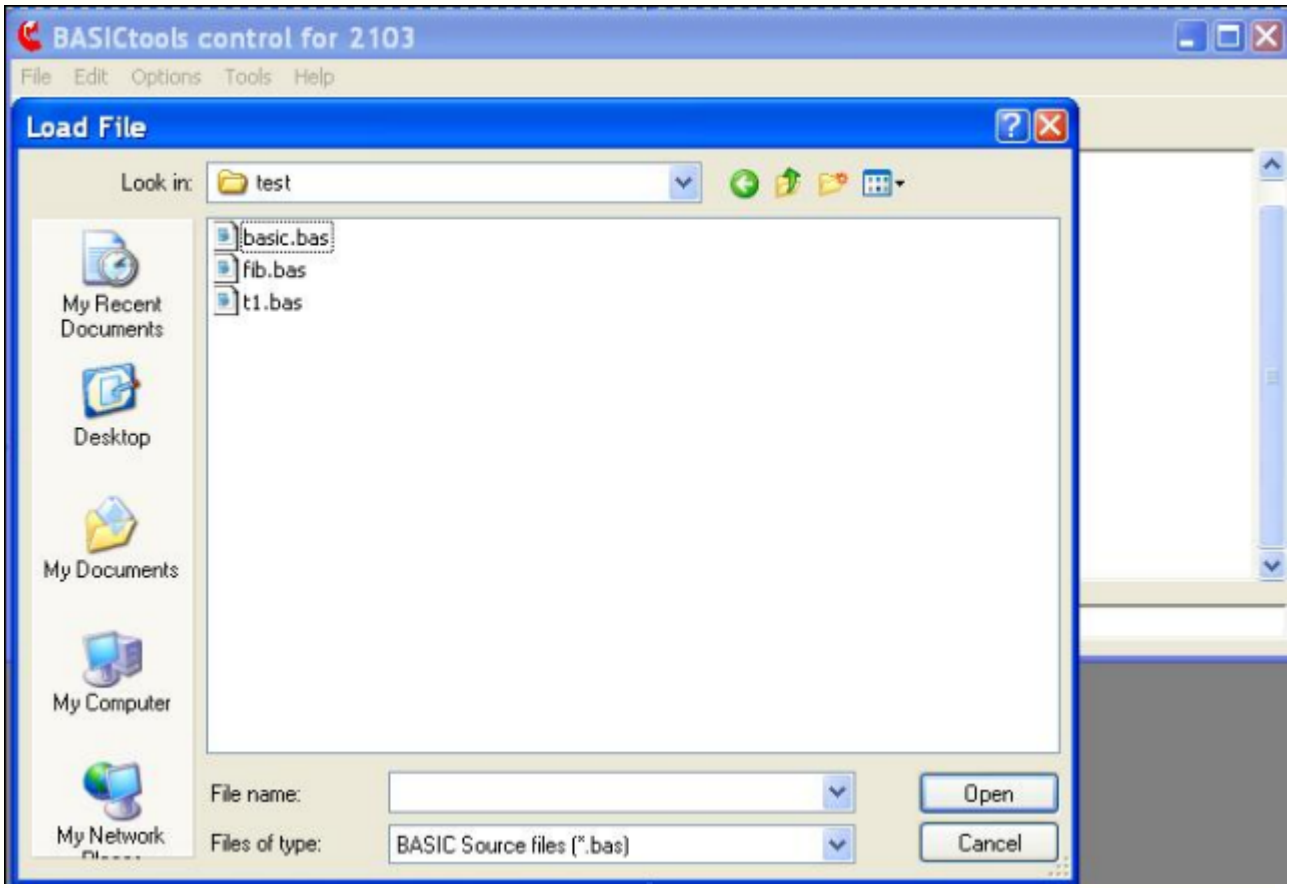
Step 5: More Complex Programming

Choose a File

While the Enter line can be useful for small programs or quickly checking out hardware, you will probably soon need to write larger programs. The way to do this is with a text editor. We don't enforce any text editor on you, you can choose your favorite. We tend to use the **Crimson Editor**, though a number of users are liking **NotePad Plus (NPP)**. Once you've typed up your program you can load that with BASICtools. It is easier to create a larger program with a text editor and then to Load File. You can link BASICtools to your favorite editor with the options (see the next section), or launch the original Windows Notepad if no editor is chosen.

Also the Enter line is limited in that #include <library> may be used, but the general pre-processor #include and other #directives should be avoided when typing a program a line at a time.





You're now ready to start tackling your application. Check with the **Coridium Forum** for files and help from other users of ARMbasic products.

For more details on the BASICtools IDE check the **next page**.

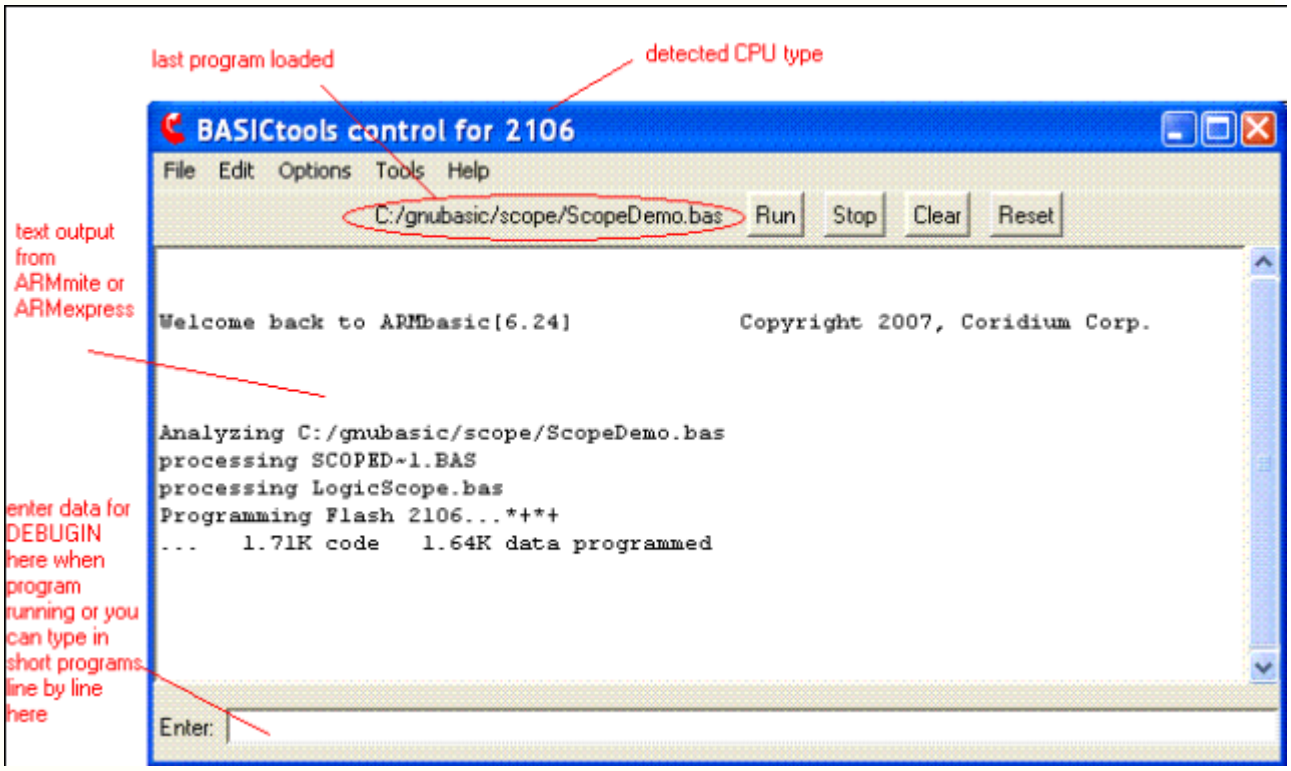
BASICtools Features

BASICtools startup

When BASICtools starts up, it will STOP any user program. So if you find yourself with a program flooding the PC serial port with data, disconnect the serial connection, and reconnect and then choose the same serial port under the Options menu.

This will clear and initialize the serial driver. Reopening the serial port will also stop any user program from starting. A good idea is to erase that program, and a simple way to do that is to enter a 1 line program like PRINT and hit run which will erase the runaway program.

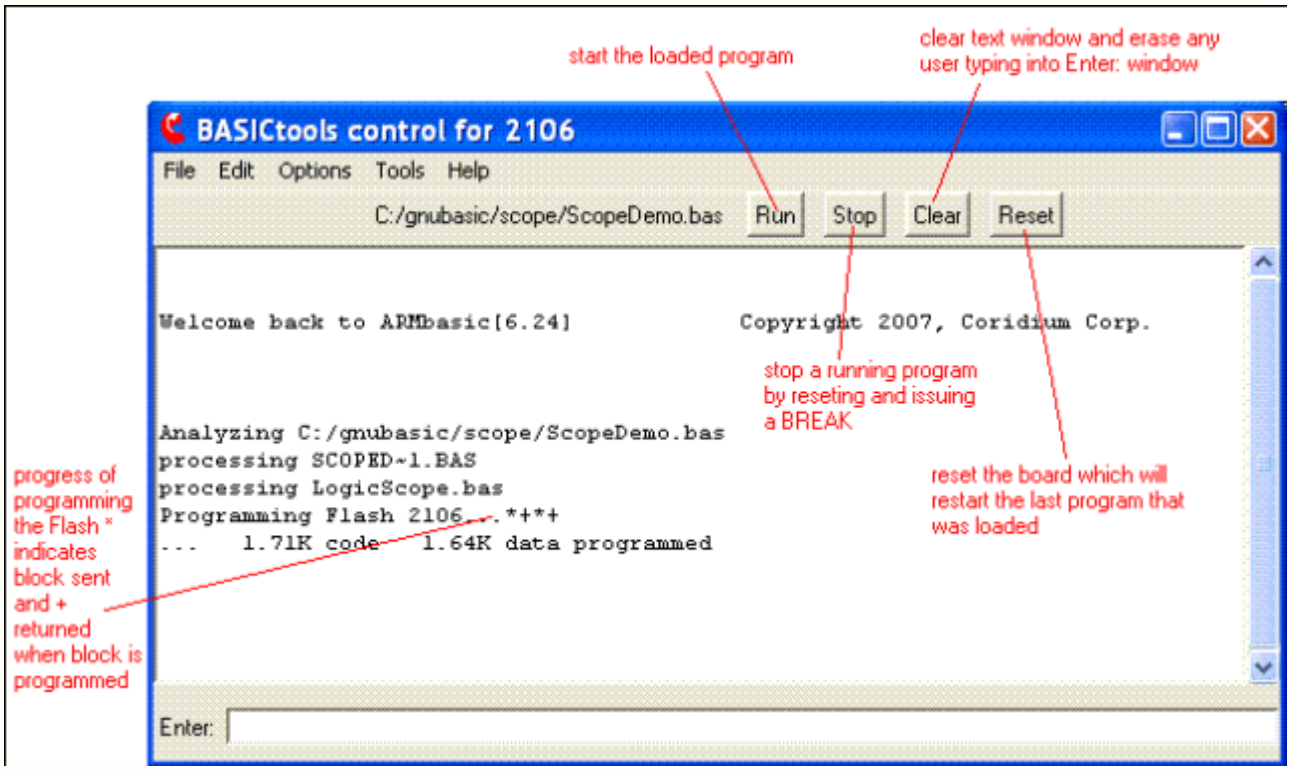
BASICtools Layout



keyw ords: enter line debugin type BASIC commands

Buttons

..



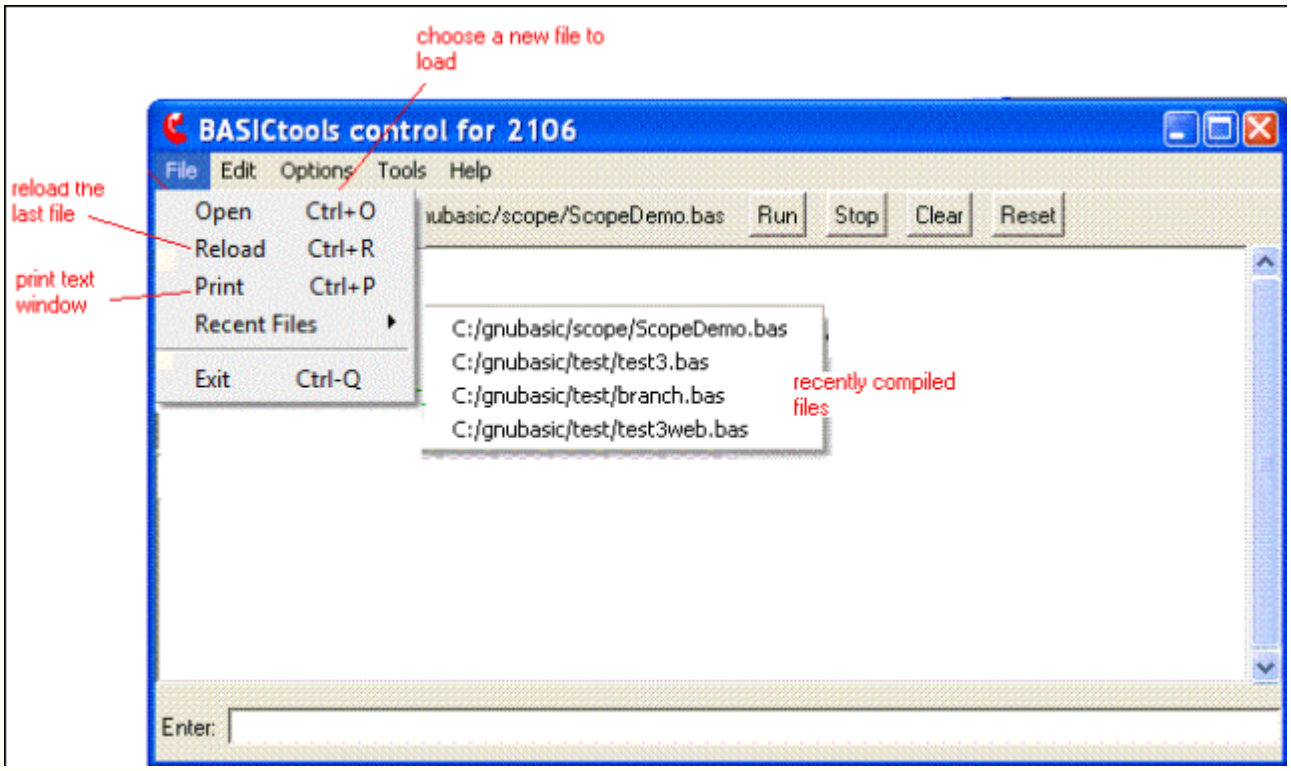
The CLEAR button only erases the display screen and the buffer on the PC of statements you have typed into the Enter window.

To erase the program, load a new program, either a line at a time or using the Load menu.

keyw ords: reset button stop button run button clear button

File Menu

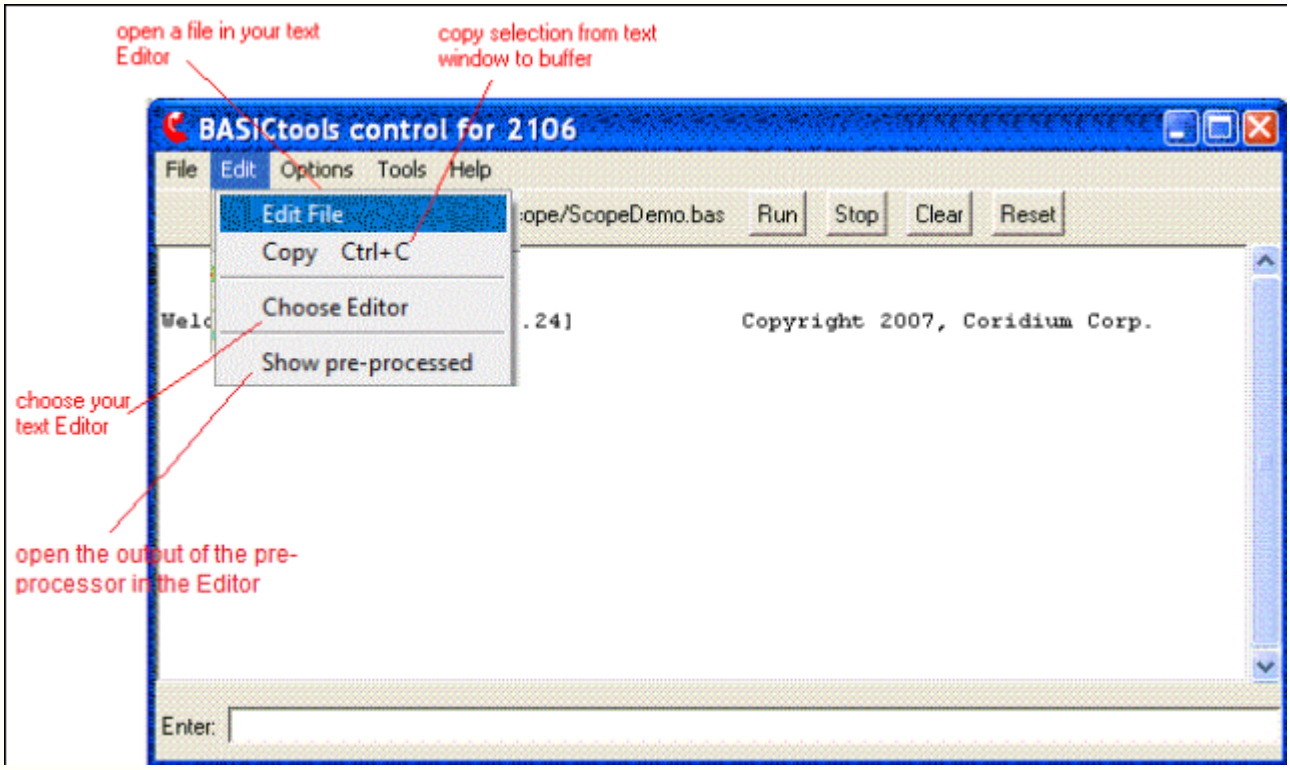
..



file load file reload file print save file quit

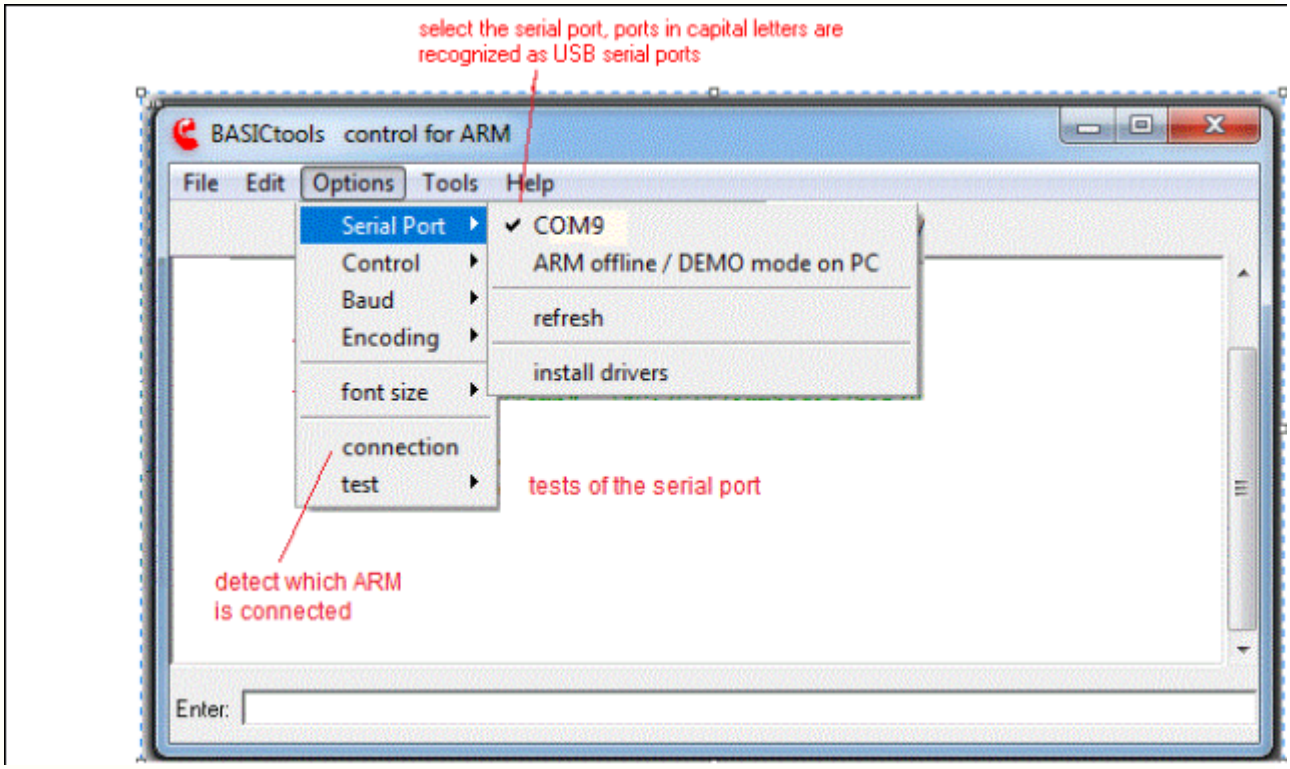
Edit Menu

..



keyw ords: edit choose editor

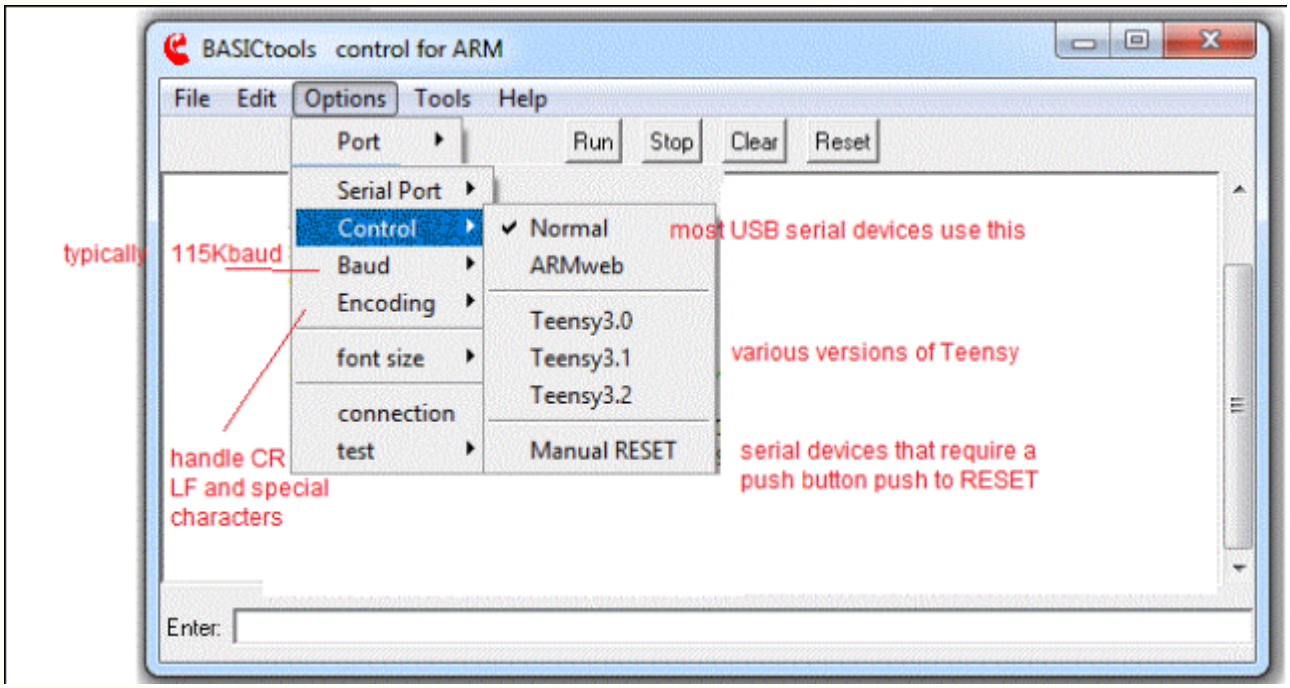
Options Menu



Refresh will check for serial devices again, it is useful if you plugged a device in after starting BASICtools.

keyw ords: options port baud new line char mode PC compile control throttle

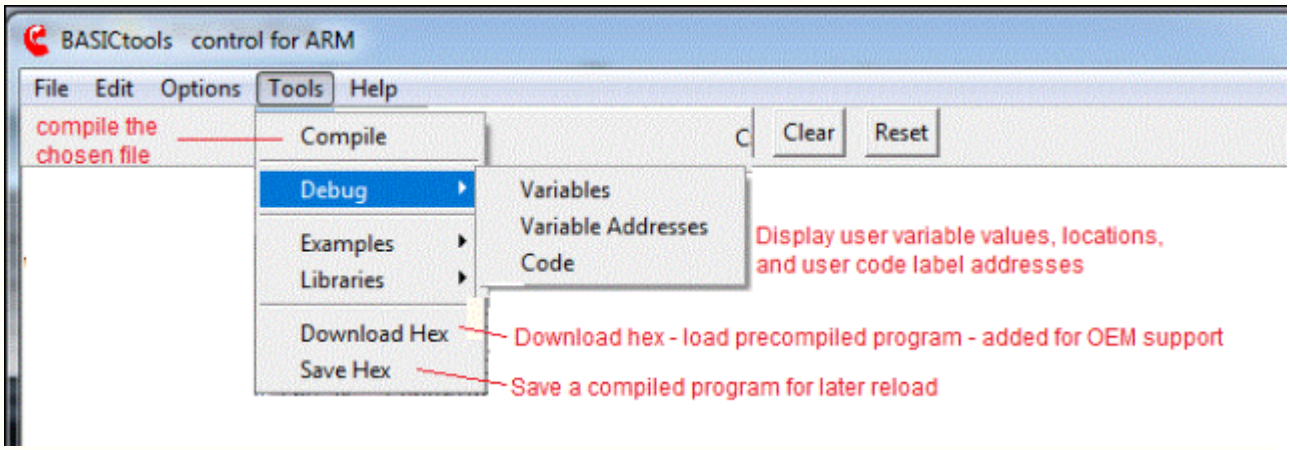
Control Menu



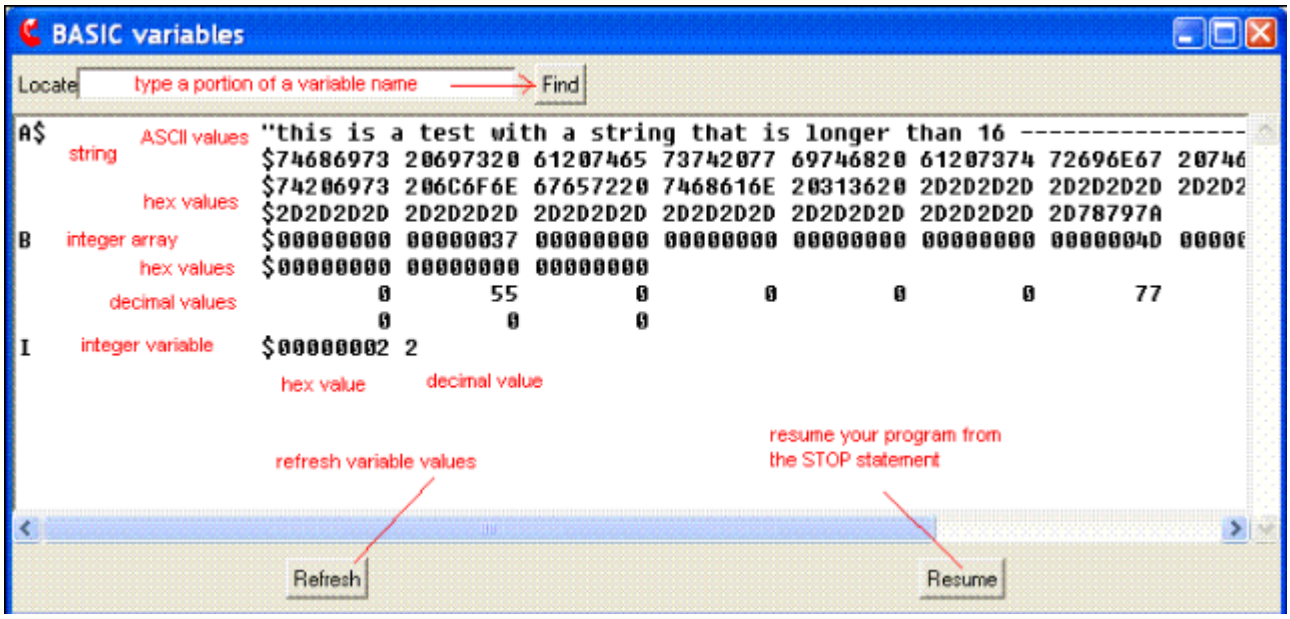
keyw ords: options port baud new line char mode PC compile control throttle

BASIC variable viewer

Open this window from the Tools Menu (variables)

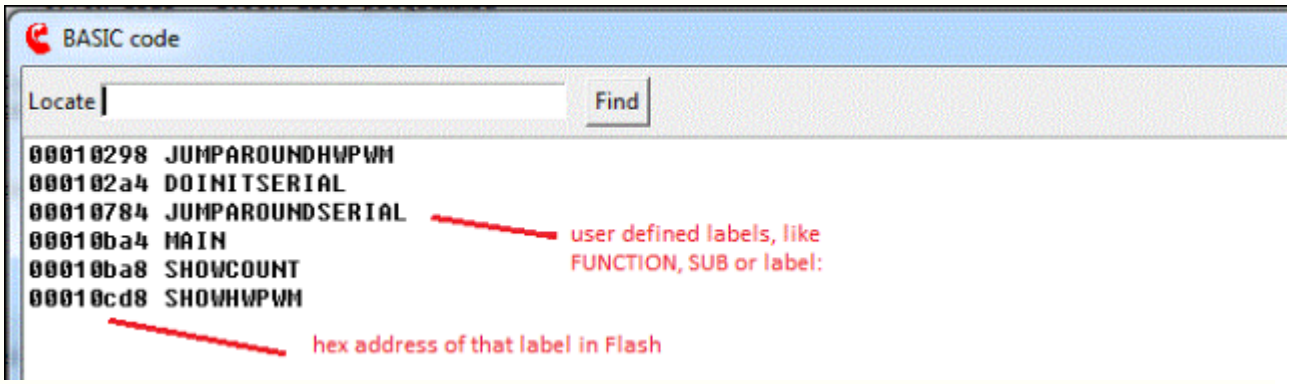


variables window



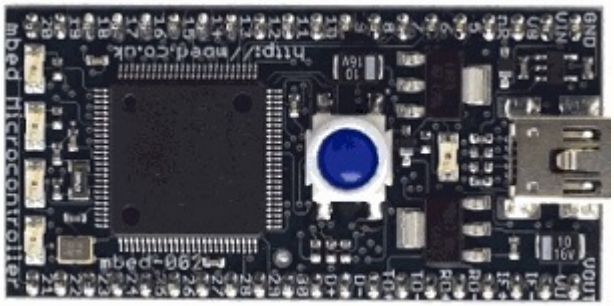
This page is active when your program ENDs or hits a STOP statement or has been STOPed with the button.

code window



keyw ords: variable dump breakpoint STOP view memory

Ethernet Getting Started



Getting Started

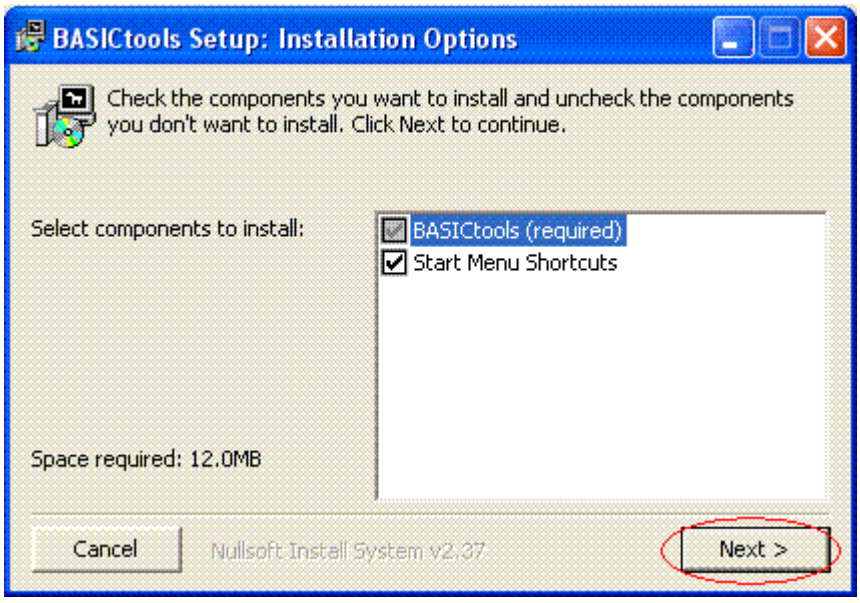
- **Install Software** (BASICtools)
- **Install Firmare** (mbed BASIC binary)
- **Connect Ethernet**
- **USB connection for ARMweb**
- **Writing programs with BASICtools**

Step 1: Install Software

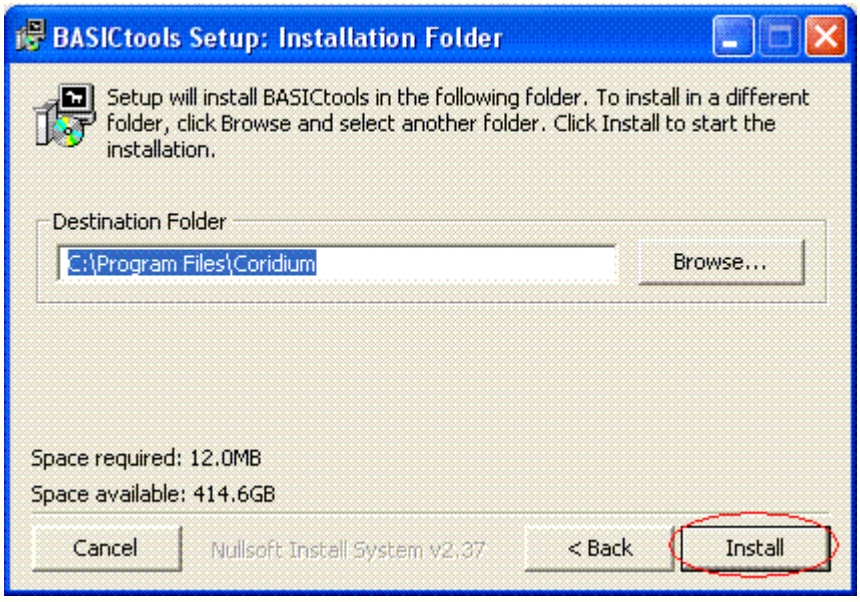
Actually much of the software you need for the **ARMweb** is already on your computer. The interface to the **ARMweb** is through any web-browser. That's why we call this **Simply Connected™** technology.

A simple **ARMbasic** compiler runs on the ARMweb. While you can write short BASIC programs with this interface, the compiler is there to support BASIC that is embedded into the HTML of the web pages served by the ARMweb. Your main BASIC program should be debugged and loaded via BASICtools over a USB connection.

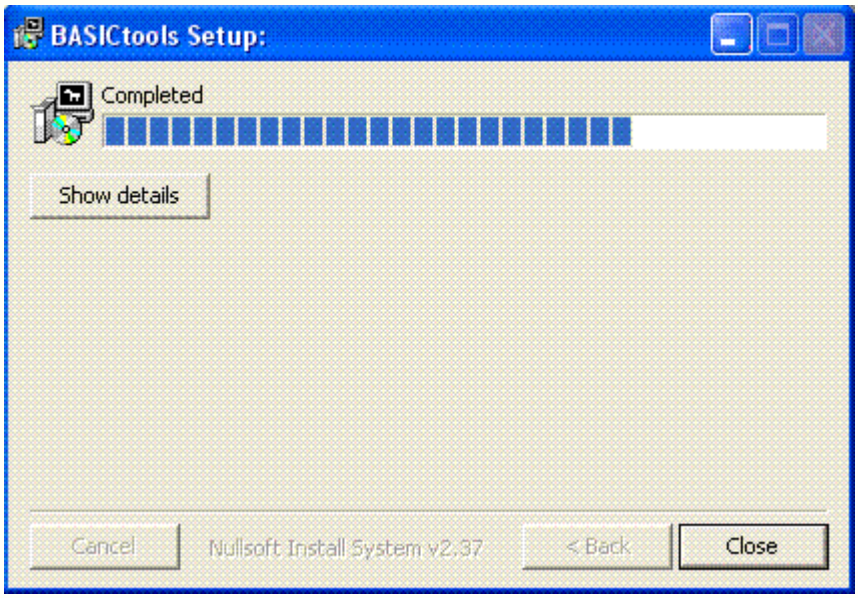
You will want to run the setupBASIC installation, to get access to documentation about ARMbasic and the PC based main BASIC compiler.



Click **Next** to get started.



Accept the defaults and **Install**. You may chose a different target directory.



The installation will now run, and when it finishes hit **Close** .

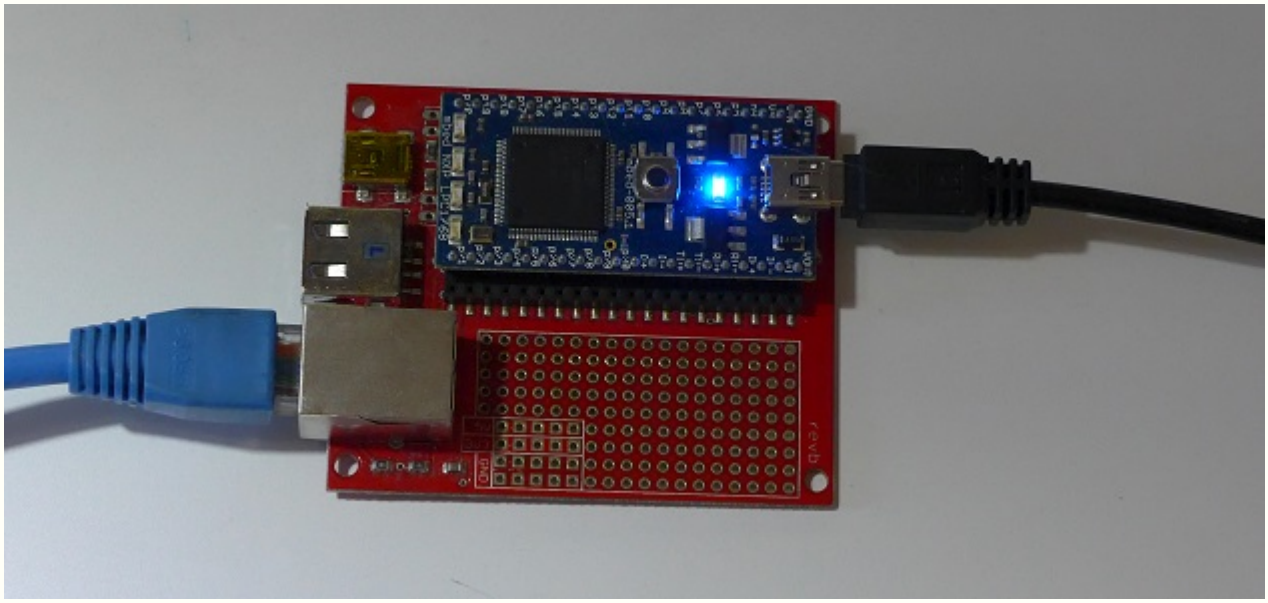
And its as easy as that.

On to Step 2

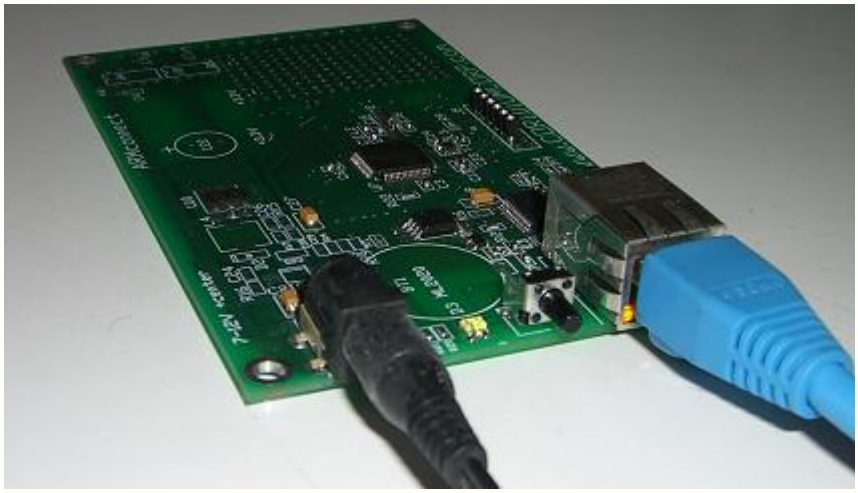
Step 2: Connect Power and Ethernet

Connect Ethernet Cable to ARMweb PCB

You should see a bluue LED on the mBed board and green LED connect light on the lower left side of the Ethernet cable indicate a connection was made. Also your hub normally has a similar type of connection indicator. There should also be some traffic indicated on the right side as the ARMweb looks for a DHCP.



For the mbed LPC1768 power can be supplied from the USB port. The picture above shows the Ethernet connection using a breakout board from Cool Components (unfortunately no longer in production). The Ethernet connection can also be made with a MAGJACK breakout board or mbed Application board from SparkFun. .



The primary power for the ARMweb is 3.3V provided from a linear regulator. The input power for the PCB may be 5V regulated supply or a 6-9V unregulated supply, with a current rating of 250 mA or more. The connector is a standard 2.5mm barrel connector with the + positive side of the supply in the center. A good choice for this power is this **5V regulated supply from SparkFun**

If you don't see the LEDs lit, check your power connections (you should see at least 6V of the + side marked on C1 with an unregulated supply or 5V with a regulated supply, and 3.3V as marked in the prototype area).

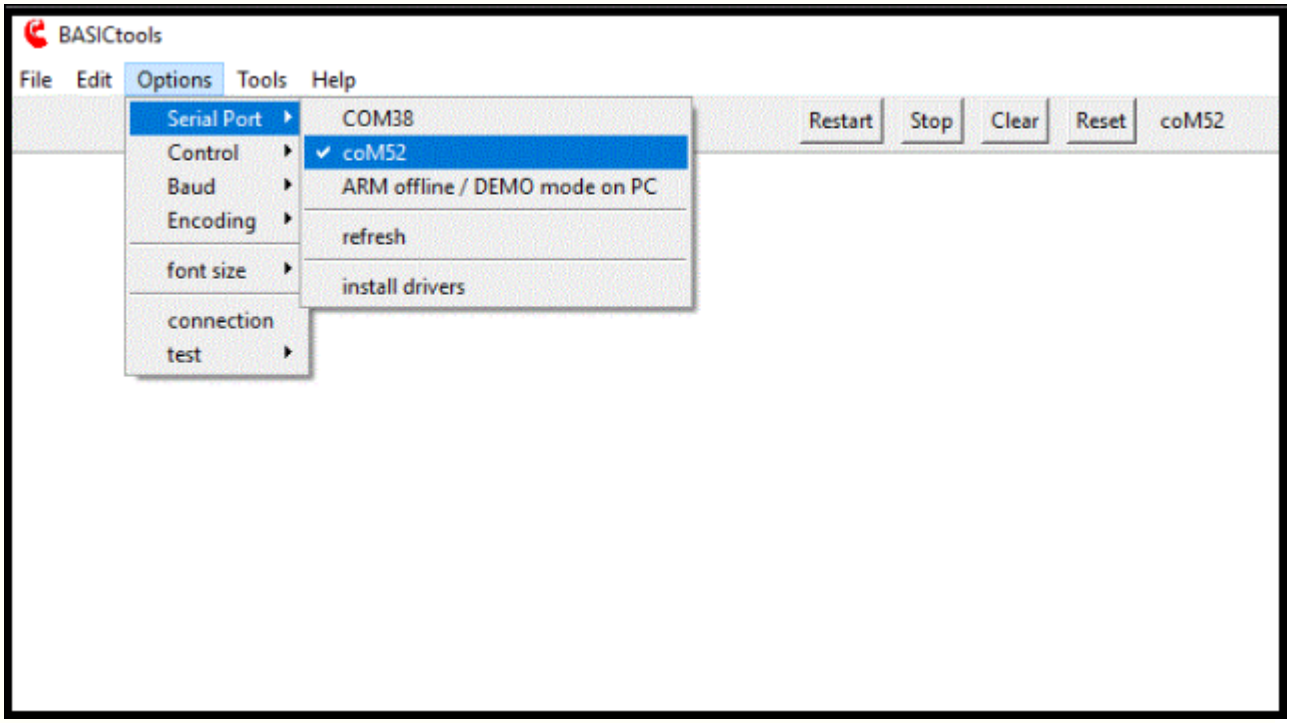
USB connection

You need a serial connection to debug BASIC programs as well as network issues. On the mbed 1768 board

this is through the USB connector on the top of the board.

This can be our **USB dongle** (specific ARMweb version when ordering separately) or some other TTL serial connection. Details on **making the USB connection** here.

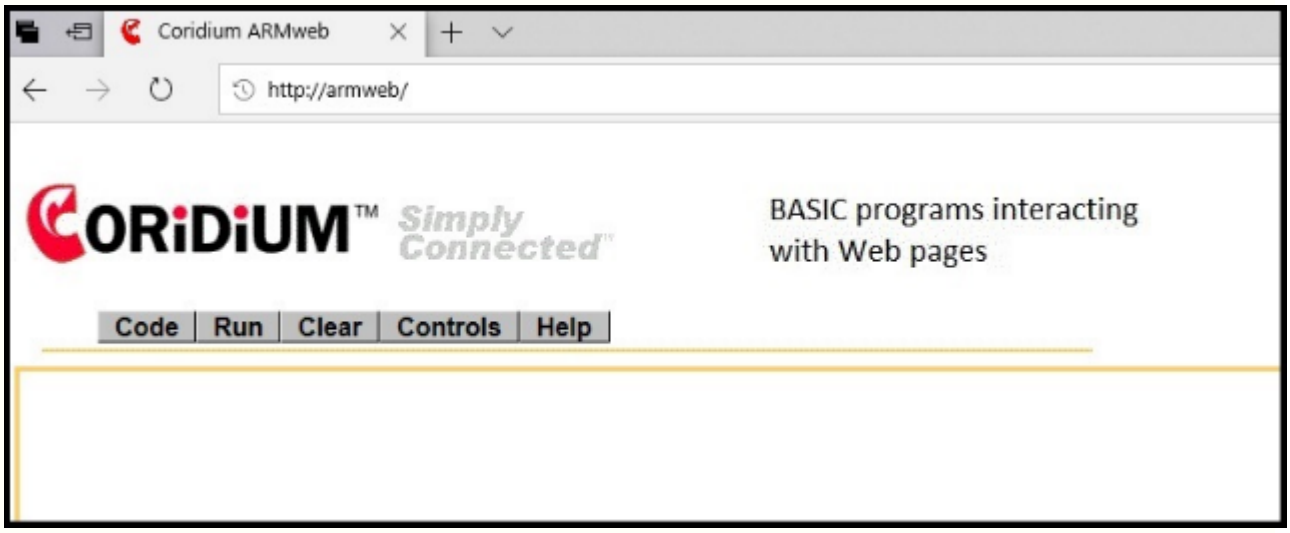
Below is the picture you should see for the mbed 1768 board, with the capital M in coM indicating an mbed serrial device.. If you do not see a coMxx serial device you may need to install the mbed serial device driver (in \Program Files (x86)\Coridium\Windows_drivers.



Finding the card on the network (larger network) -- NetBIOS name service

The ARMweb will configure itself with an IP address assigned by a DHCP server. IP addresses are the way networks organize themselves. If there is no DHCP server found, the ARMweb can provide limited DHCP services in a Diagnostic mode, assuming a single connection on Ethernet with a PC using either a hub or cross-over cable (see the Diagnostic section below).

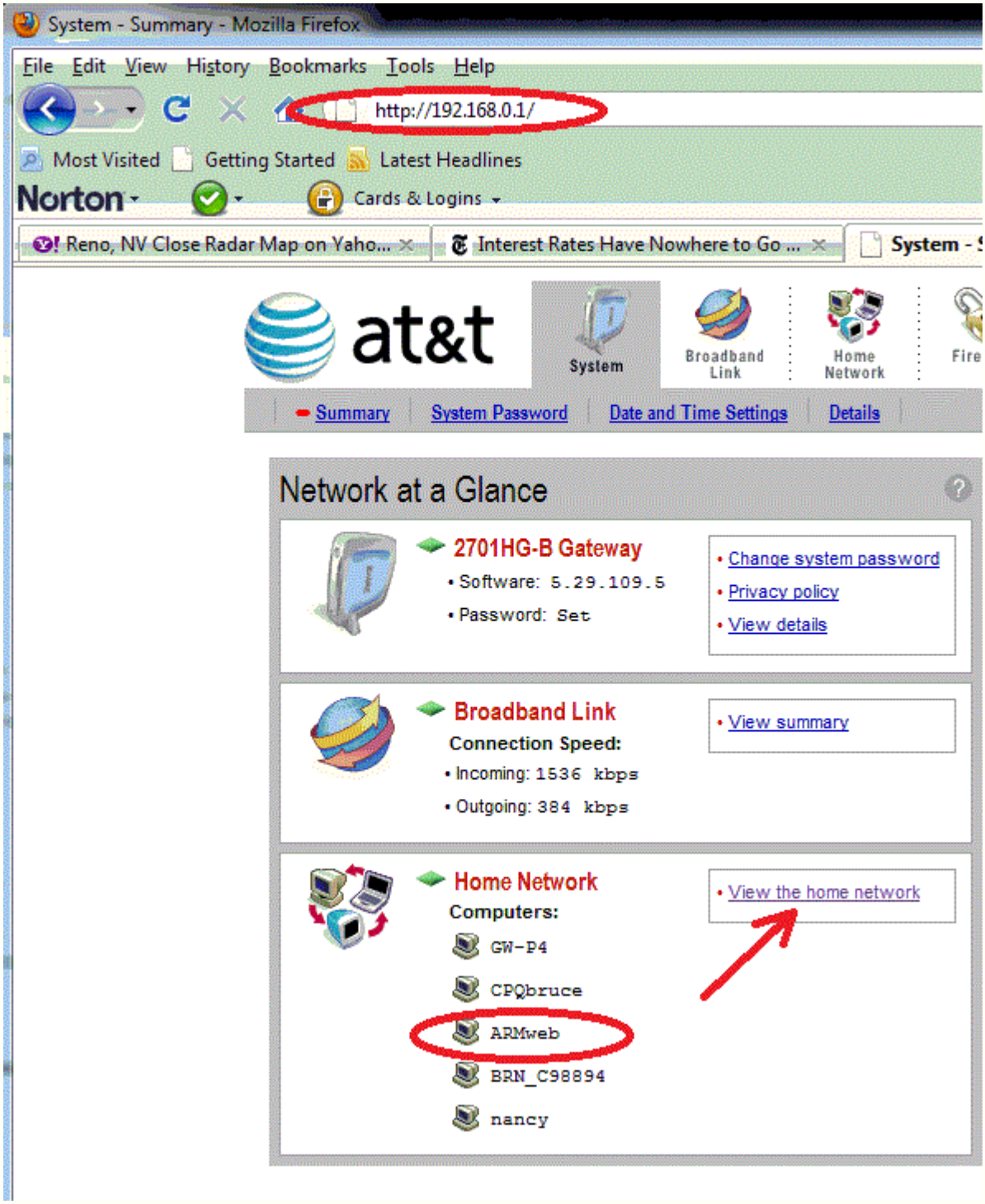
Assuming a DHCP server is available and you are running on a Windows machine, you can use the Windows NetBIOS Name Service. In which case you can find the ARMweb initially with **http://armweb**. **Note that some administrators disable NetBIOS name service** .



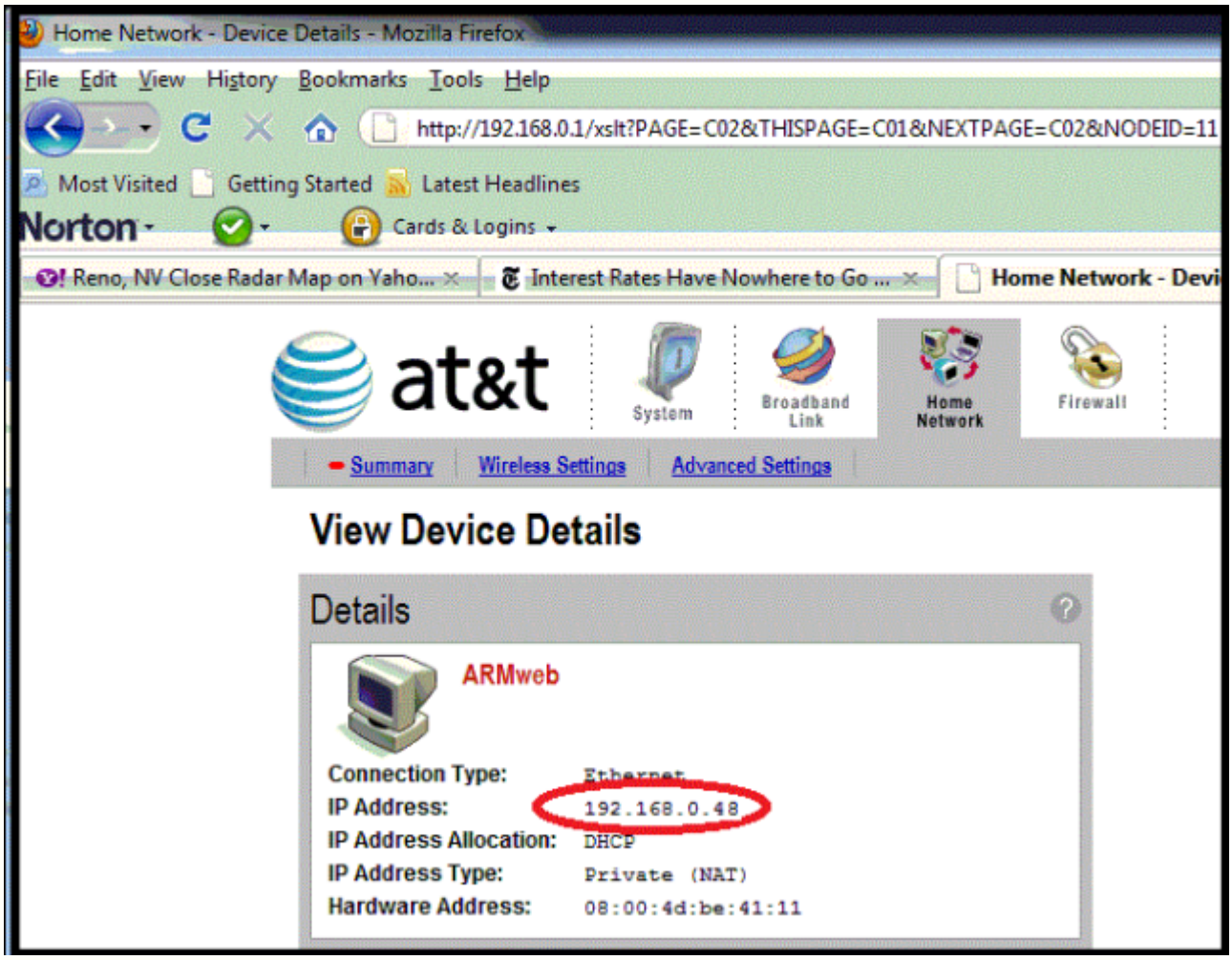
Finding the card using the DHCP server

On most home networks your DHCP will be your internet connection, and its address will share the first 3 bytes with the IP address of your PC. And the final byte being 1. The IP address of your PC is available from the control panel or by typing **IPCONFIG** at a DOS command line. Common values for the DHCP server are 192.168.1.1 or 192.168.0.1 as in the example below.

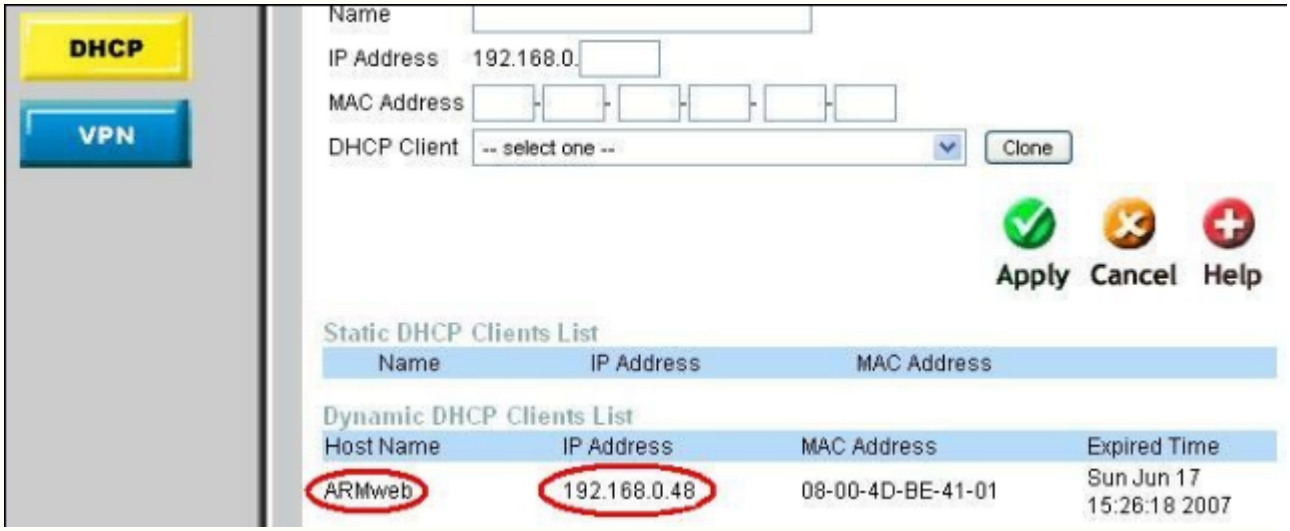
You can navigate to the DHCP server using that IP address from a browser as below.



Most DHCP servers will list client machines which have been assigned an IP address. This 2wire server indicates it on the details view of the home network, and details for the device



Another example is the display from a Dlink Firewall that is also providing DHCP services.



So in this case the ARMweb can be found at **http://192.168.0.48**

Now that you have the IP address of the ARMweb

You can go onto configuration settings, or writing simple programs using BASICtools.

But for this web interface navigate using a browser to **http://w.x.y.z** where w.x.y.z is the IP address of the ARMweb (192.168.0.48 in this example).

DHCP assignment vs. fixed IP addressing

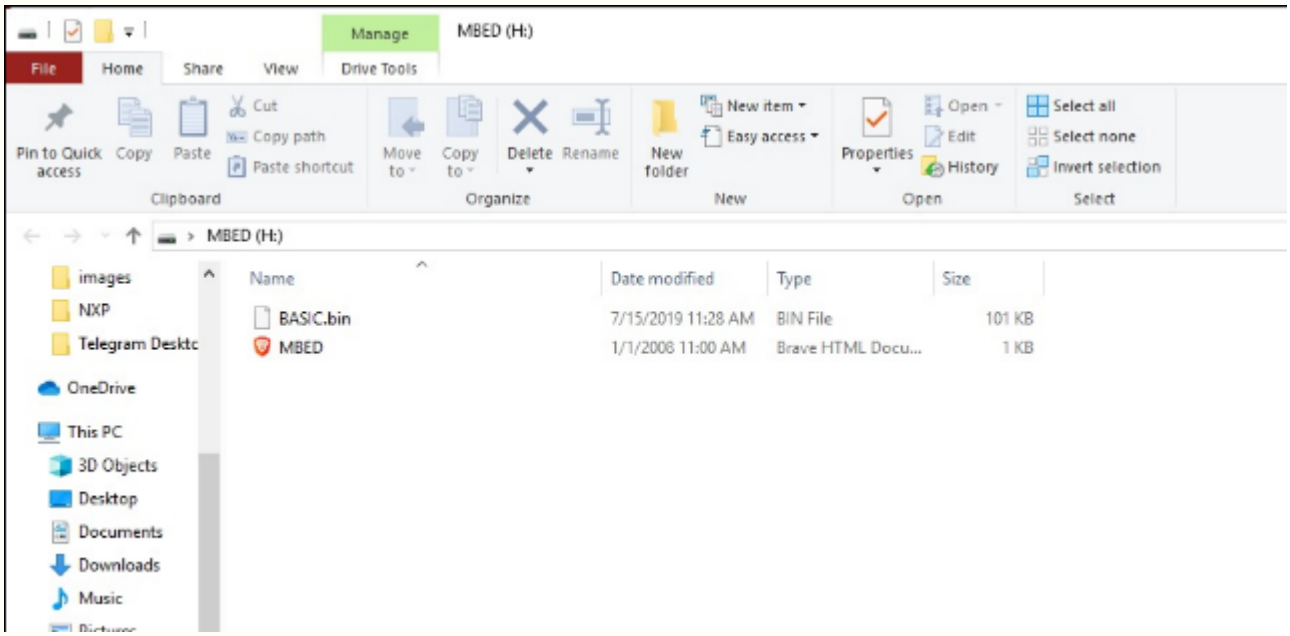
We routinely allow the DHCP server to assign an initial address, but will can a fixed IP address in the final setup. One reason to assign a fixed IP, is to make sure that the IP address assigned never changes, for instance following a power outage. **Details** on setting a fixed IP address.

On to Step 3

USB connection for BASICtools

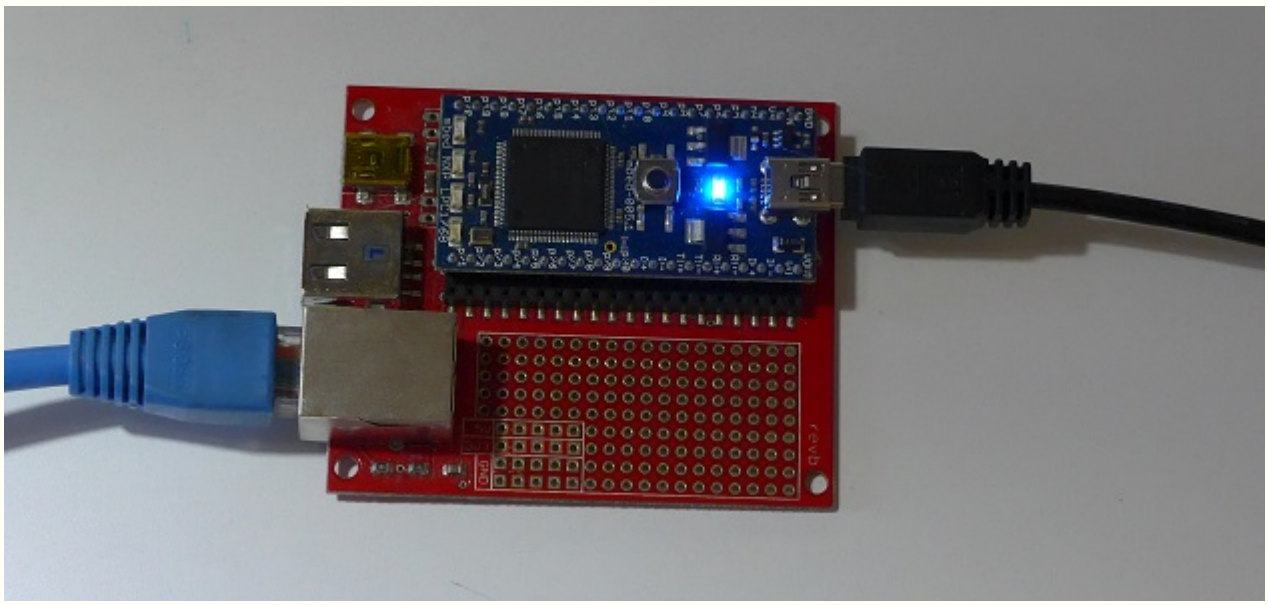
The main user program is loaded through BASICtools via a USB connection. The attachment of the USB and power supply is shown below.

Initially you will have to load BASIC firmware onto the LPC1768. These boards use the mbed protoco, so when you plug the board in for the first time, an MBED disk will appear to the files manager (below)

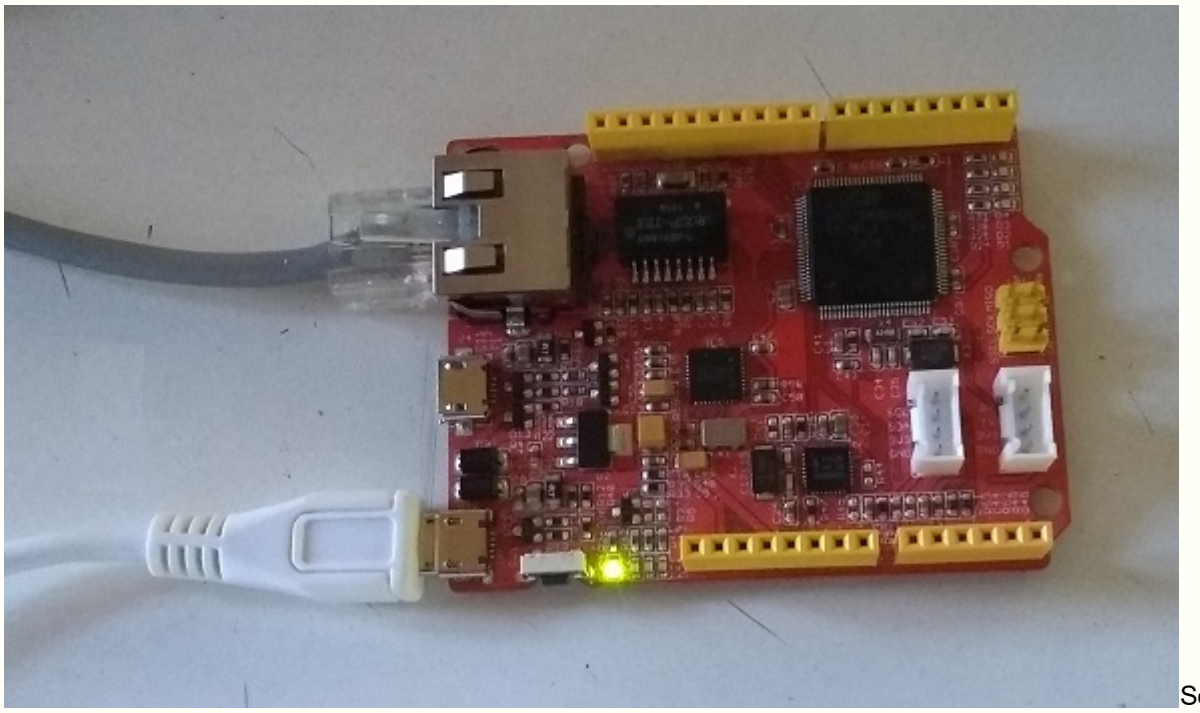


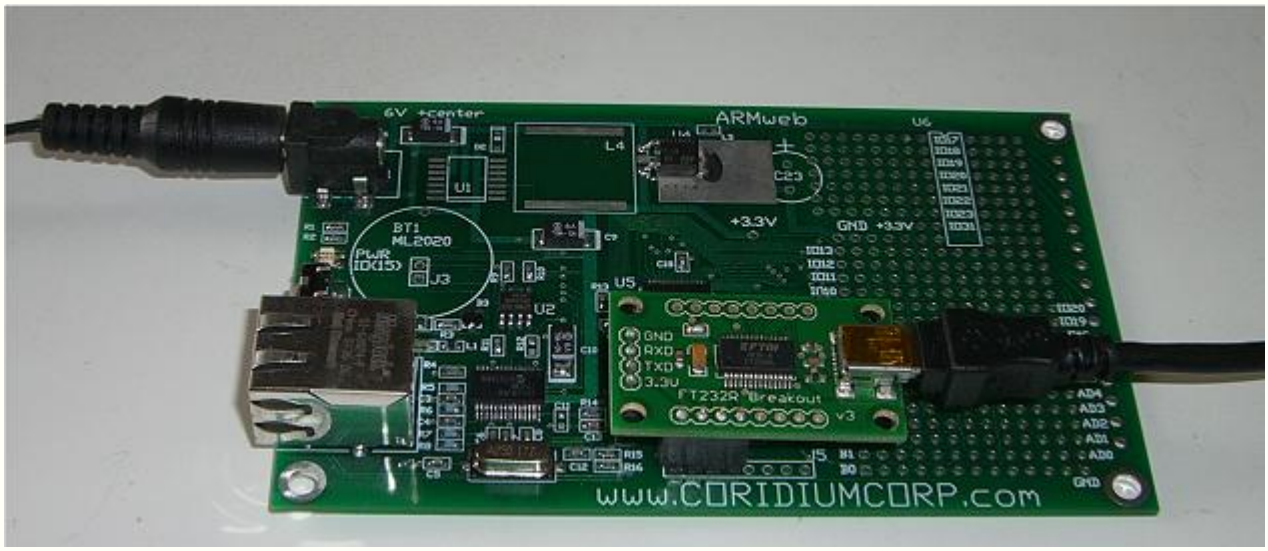
Depending on the mbed firmware version (running on a different CPU LPC11U35 or LPC43xx), you may see a .bin file. Delete any of these and copy the BASIC.bin firmware you downloaded or were emailed from Coridium. And also depending on mbed firmware you may need to reset the board via push button. Not for older ARMweb/ LPC2138 boards use**these directions** .

While an Ethernet connection is not required, if it exists and there is a DHCP server, the ARMweb will boot faster (otherwise each reset the 10 second timeout waiting for DHCP service will occur).

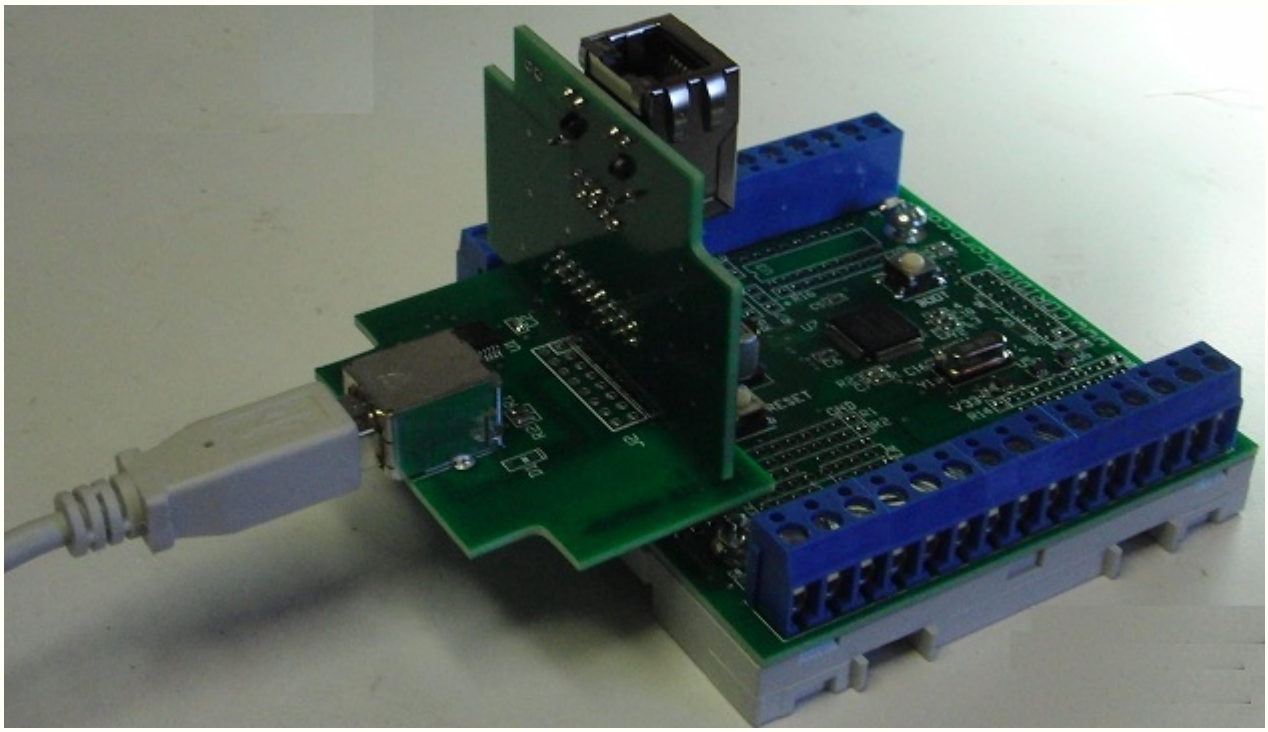


mbed LPC1768


Seeed Studio ArchPro



ARMweb


DINkit - Ethernet.

BASIC and Web page interaction

BASIC can be embedded in the web page served by webBASIC. That BASIC code can access global variables of the user program running on the ARM. BASIC embedded in the web page can **NOT** call a FUNCTION or SUB..

The user (client) can also interact with an webBASIC program via the CGI mechanism.

USB drivers

Most PC's will sound a tone that indicates a new USB device has been connected. Most Windows 7, 8 and 10 systems will either include the FTDI device driver or are able to download it automatically from the network.

If your system is unable to do that. Run the FTDI driver installation setup in the \Program Files\Coridium\Windows_drivers directory. This will install the proper drivers for the FTDI chips we use for interfacing to the USB.

Up to date details are at the **www.ftdichip.com** VCP drivers page.

Continue with the some programming examples.

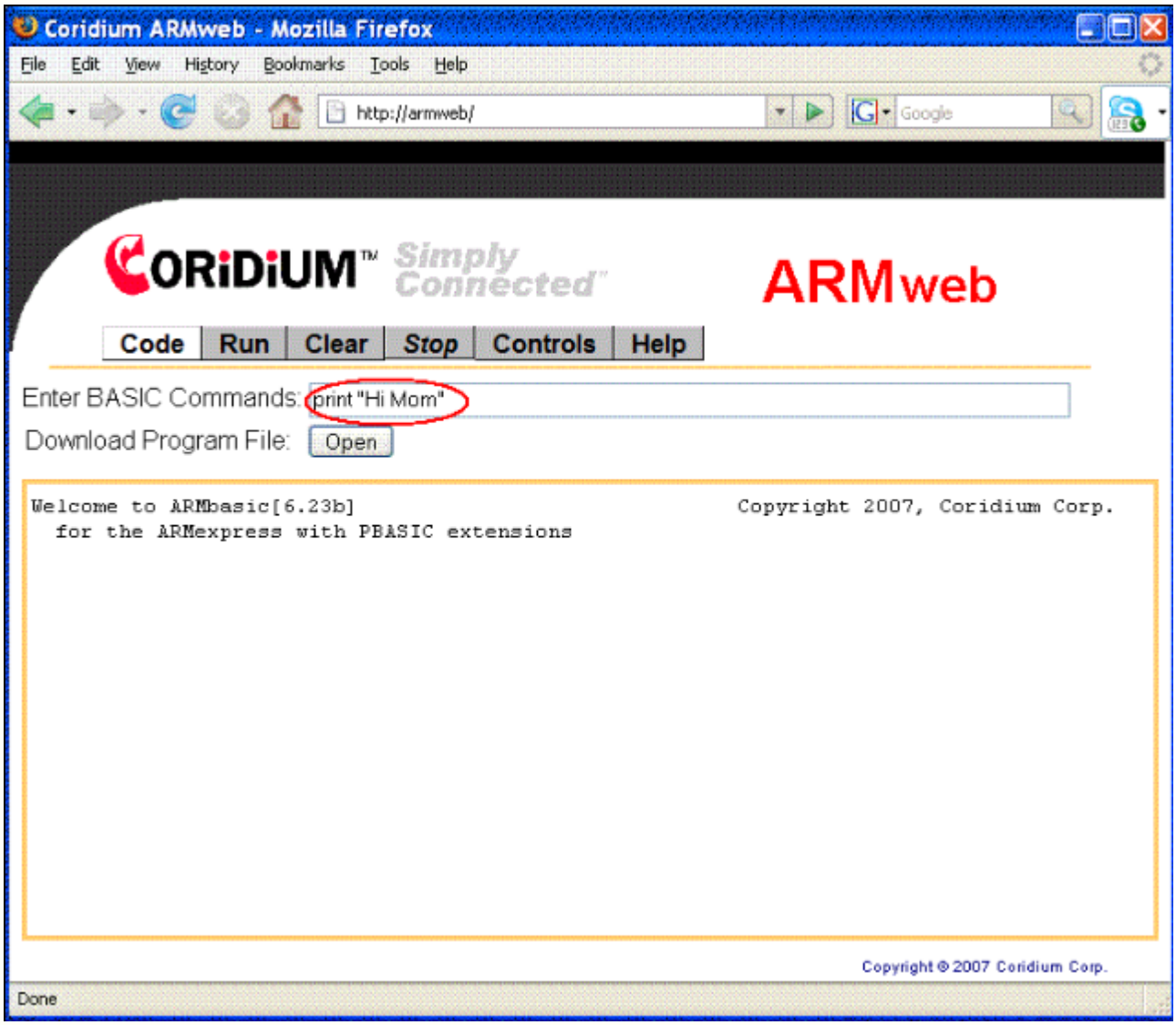
or

More details on webBASIC...

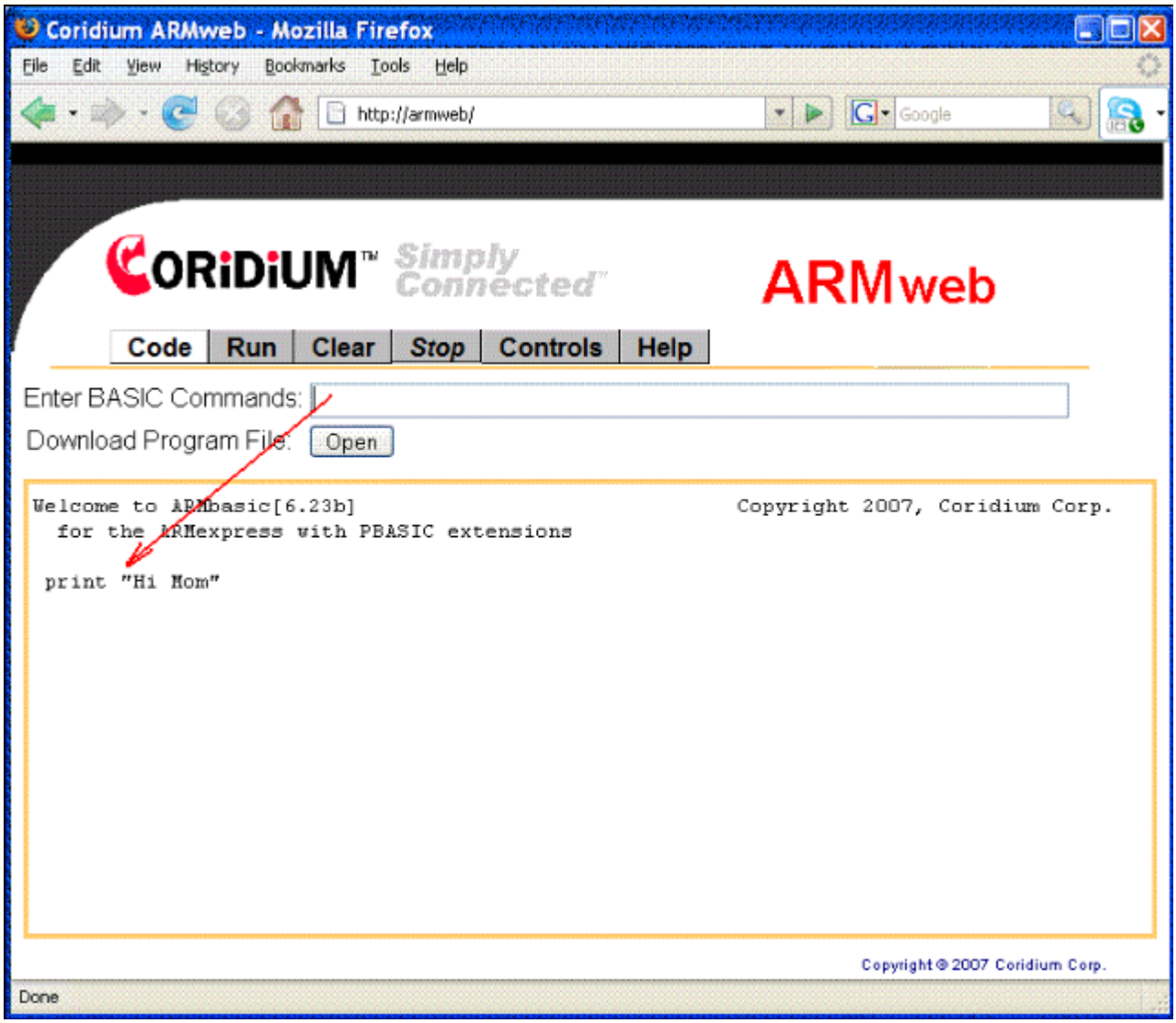
Step 3: Writing a simple Program with the web interface

The traditional "Hi Mom" program

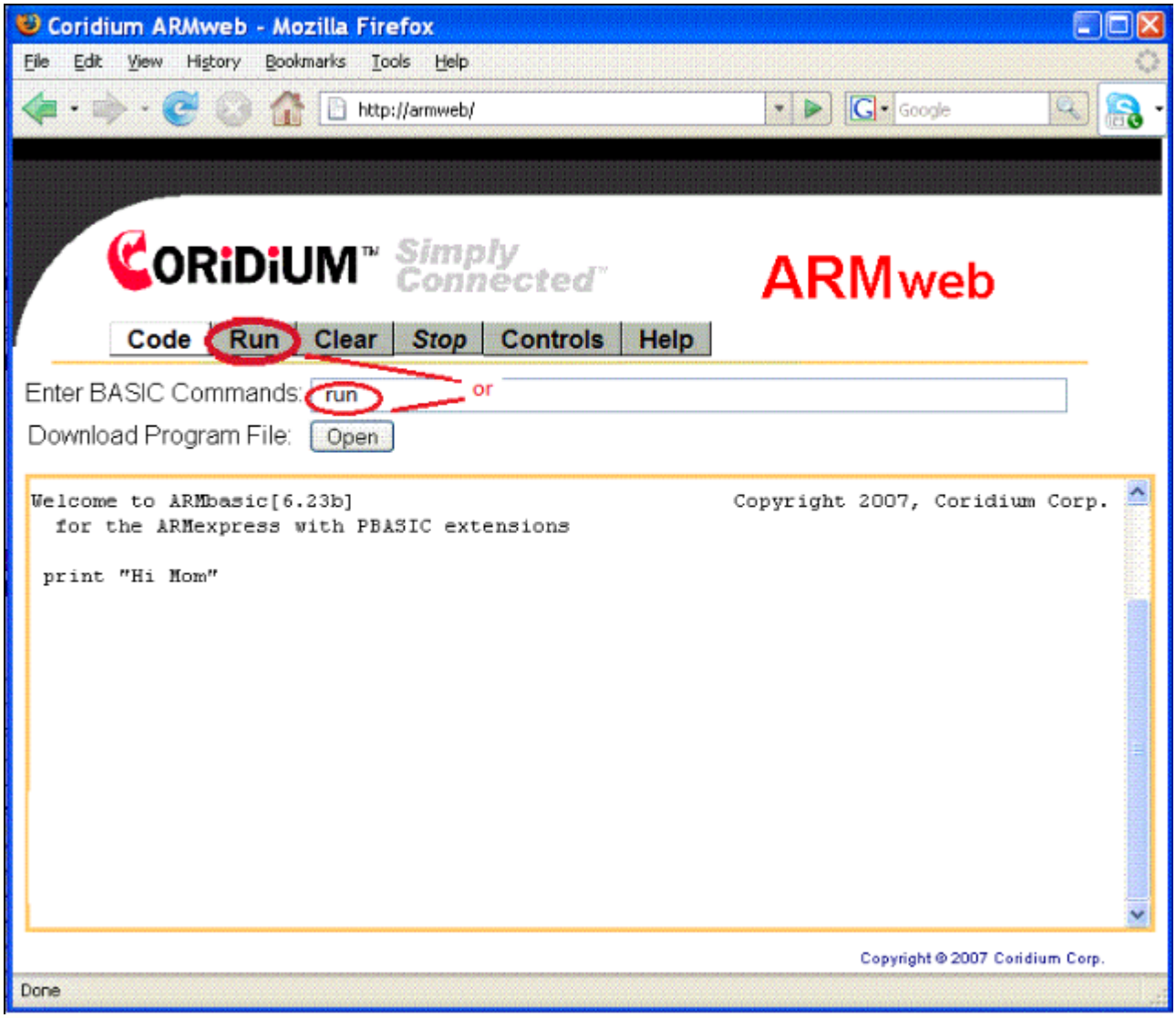
This section describes writing programs with the web interface, which is fine for small programs. But you will really want to use the USB interface to write larger programs, covered in the **next section** .



So type something like the traditional PRINT "Hi Mom"
When you hit the ENTER key it will be sent to the ARMweb and be echoed back in the console window. (below)

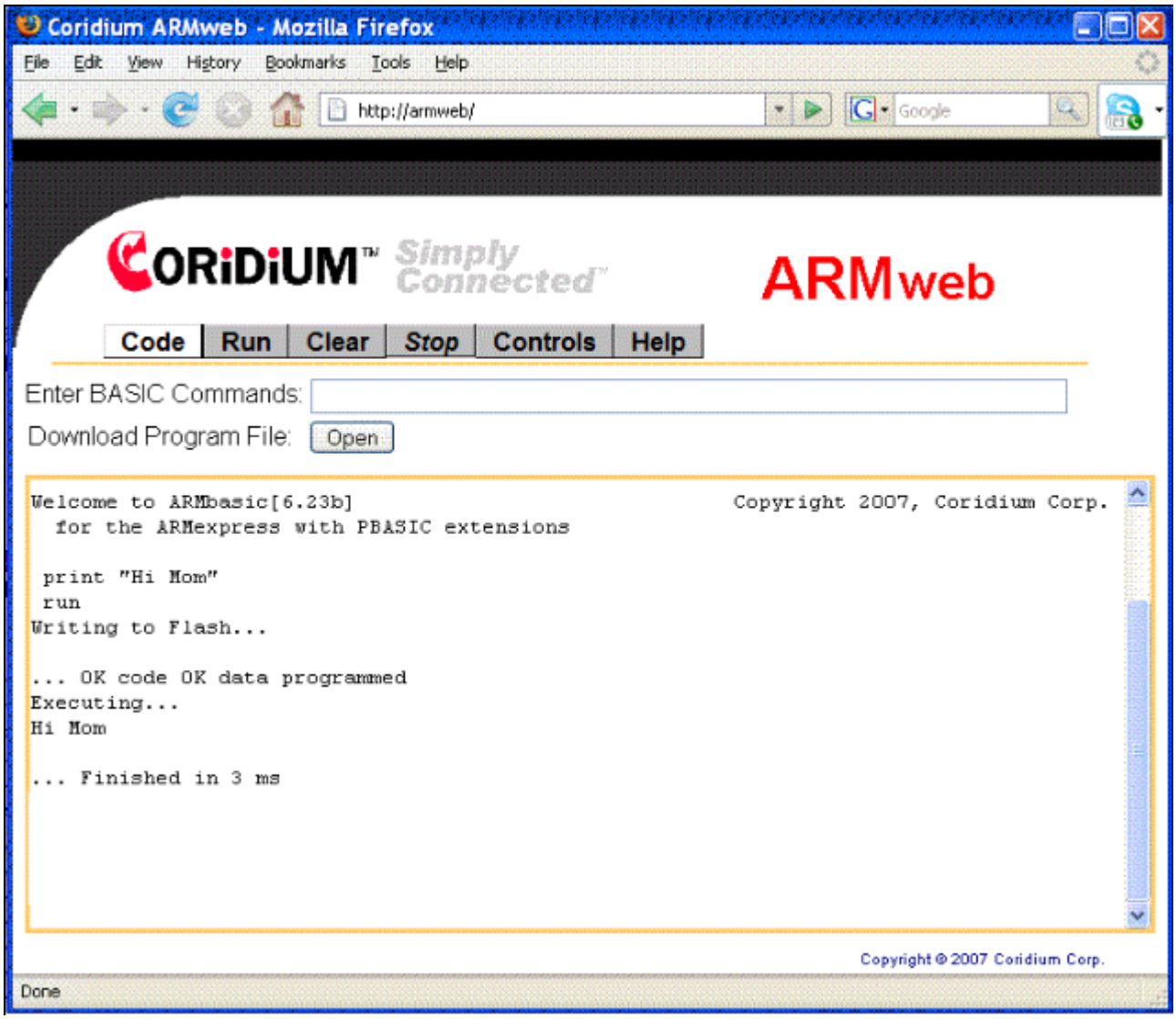


Now RUN the program



Which you can do by either typing RUN or hitting the RUN button at the top of the screen.

And see the results

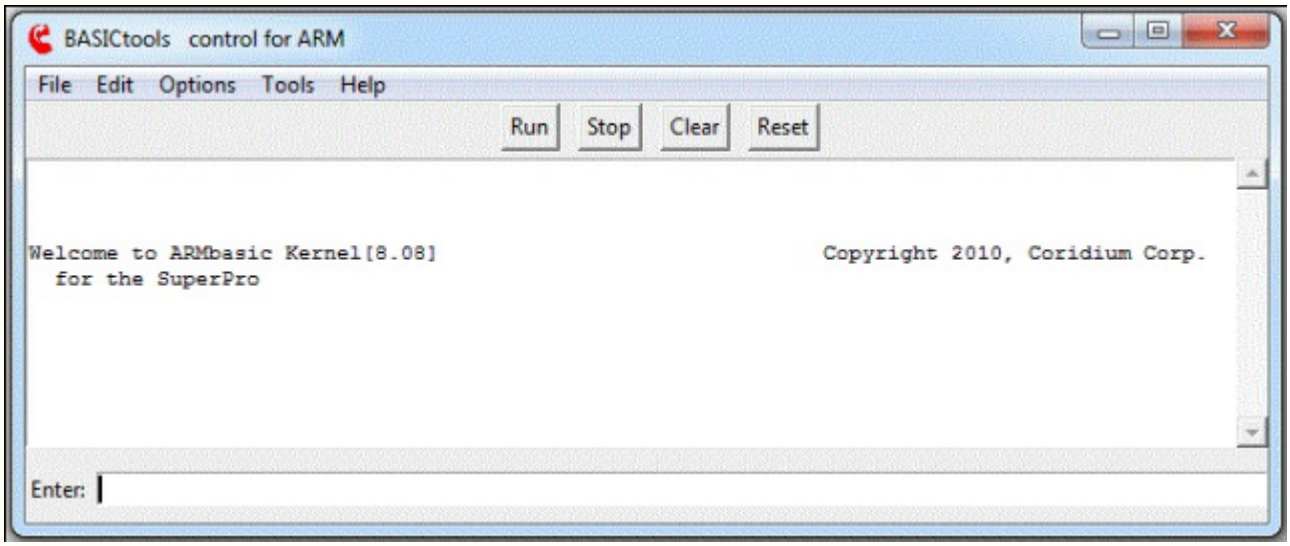


You can notice a number of things. First the program is compiled and then written into Flash memory, and your program takes 0K of code and 0K of data space.
Next the program will be executed, as evidenced by the output of "Hi Mom" to the console.
ARMweb also reports back how long the program executed, in this case 3 msec

On to the next Step

Step 3: Writing your first Program with BASICtools

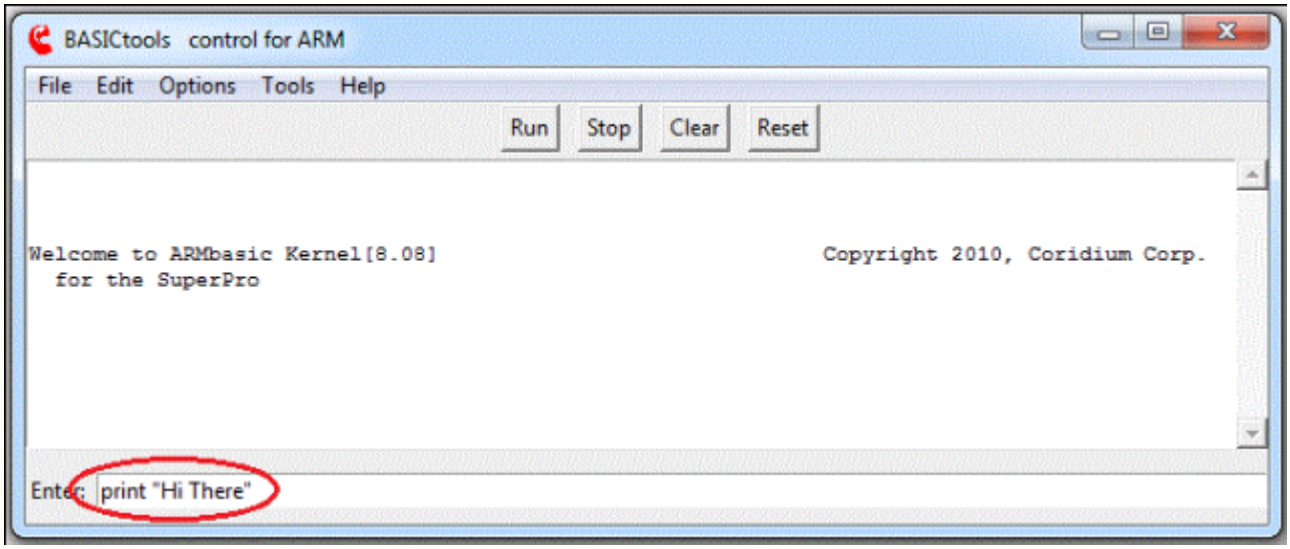
Start the BASICtools from the Start-Menu or from the Desktop Icon. You should see a welcome message which has been sent from the ARM processor-



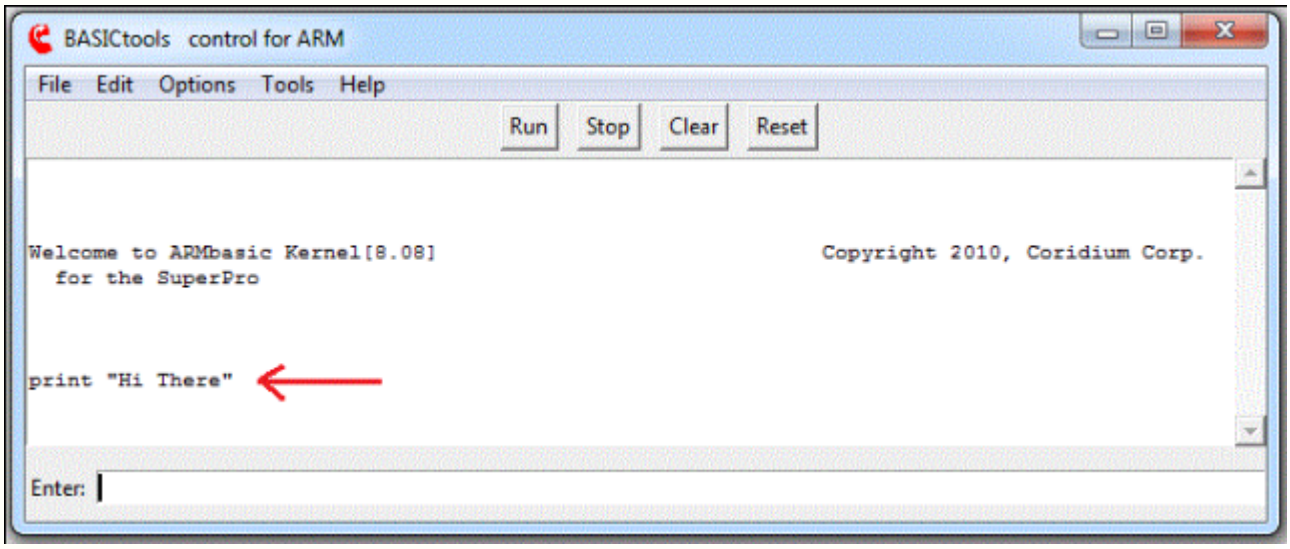
If you do not see this welcome, even after pushing the RESET button, then communication has not been established.

- check cables
- check power supply
- check COM port choice in BASICtools -> Options
- check baud rate in BASICtools -> Options
- on non-Coridium Boards, remove any BOOT select jumpers, press RESET again
- if still not working, check the**Trouble Shooting Section**

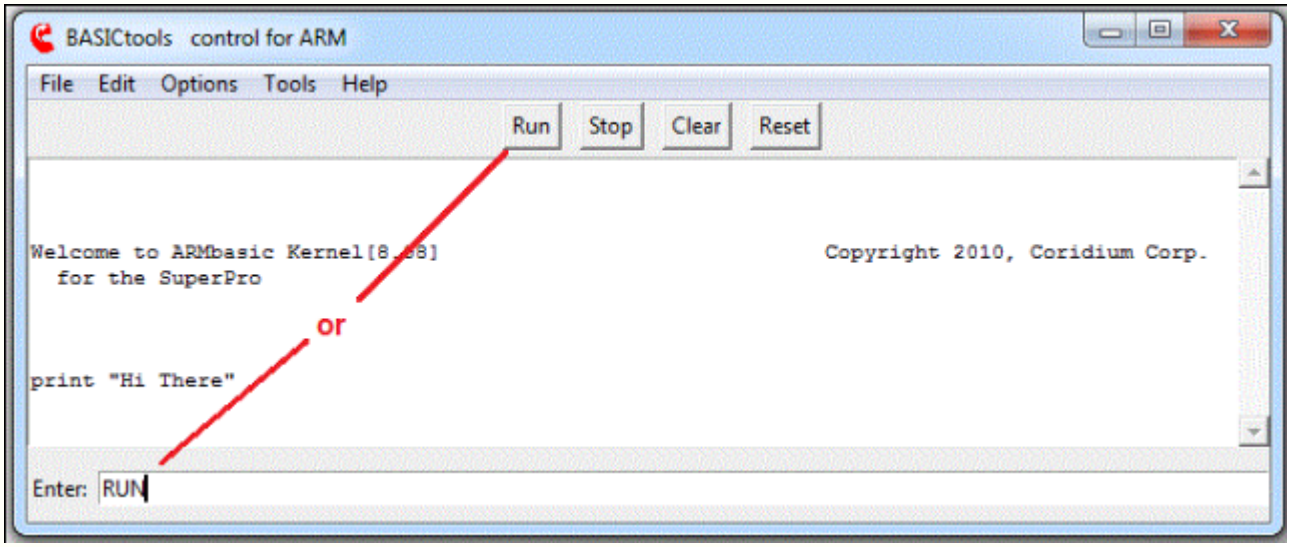
The traditional "Hi World" program



So type something like the traditional PRINT "Hi There"
When you hit the ENTER key it will be sent to the compiler on the PC and be echoed back in the console window. (below)

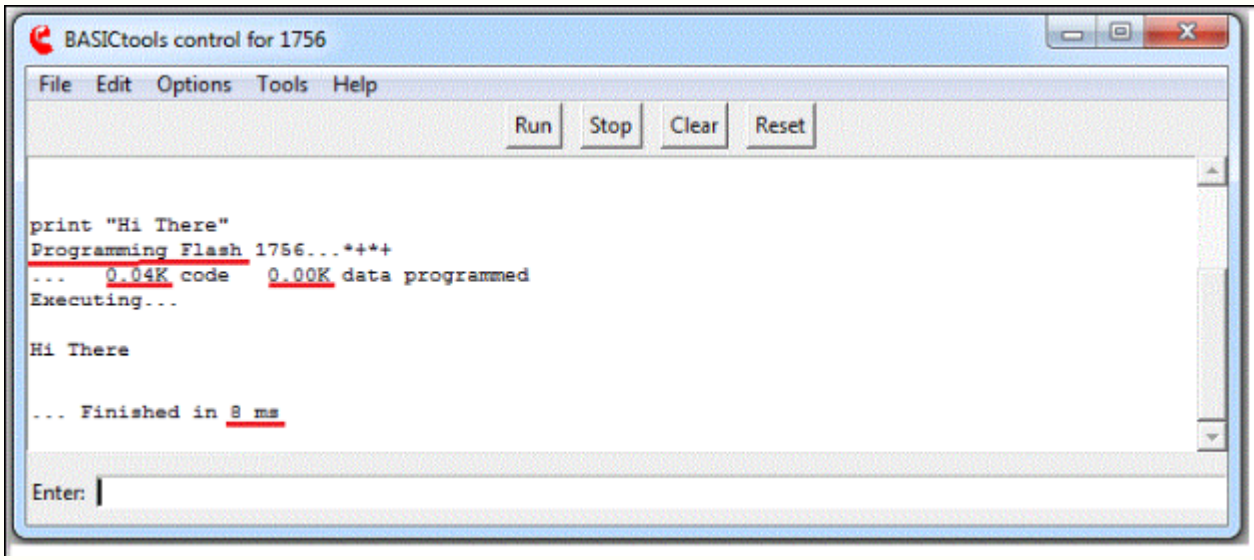


Now RUN the program



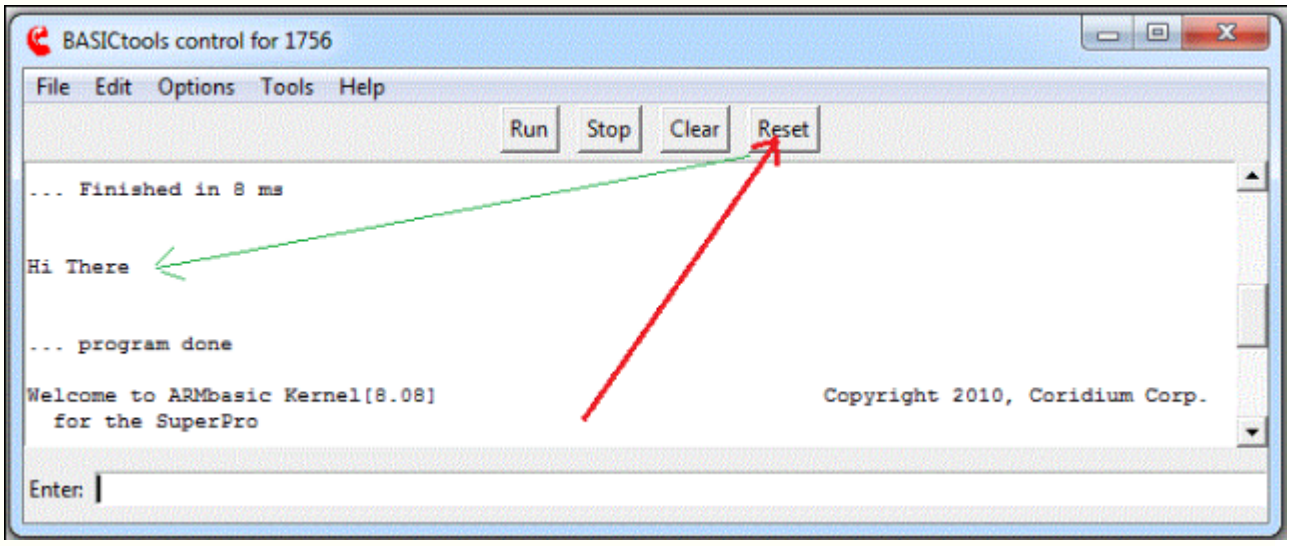
Which you can do by either typing RUN or hitting the RUN button at the top of the screen.

And see the results



You can notice a number of things. First the program is compiled and then written into Flash memory, and your program takes 40 bytes of code and less than 10 bytes of data space. Next the program will be executed, as evidenced by the output of "Hi There" to the console. ARM also reports back how long the program executed, in this case 8 msec, which is mostly startup time.

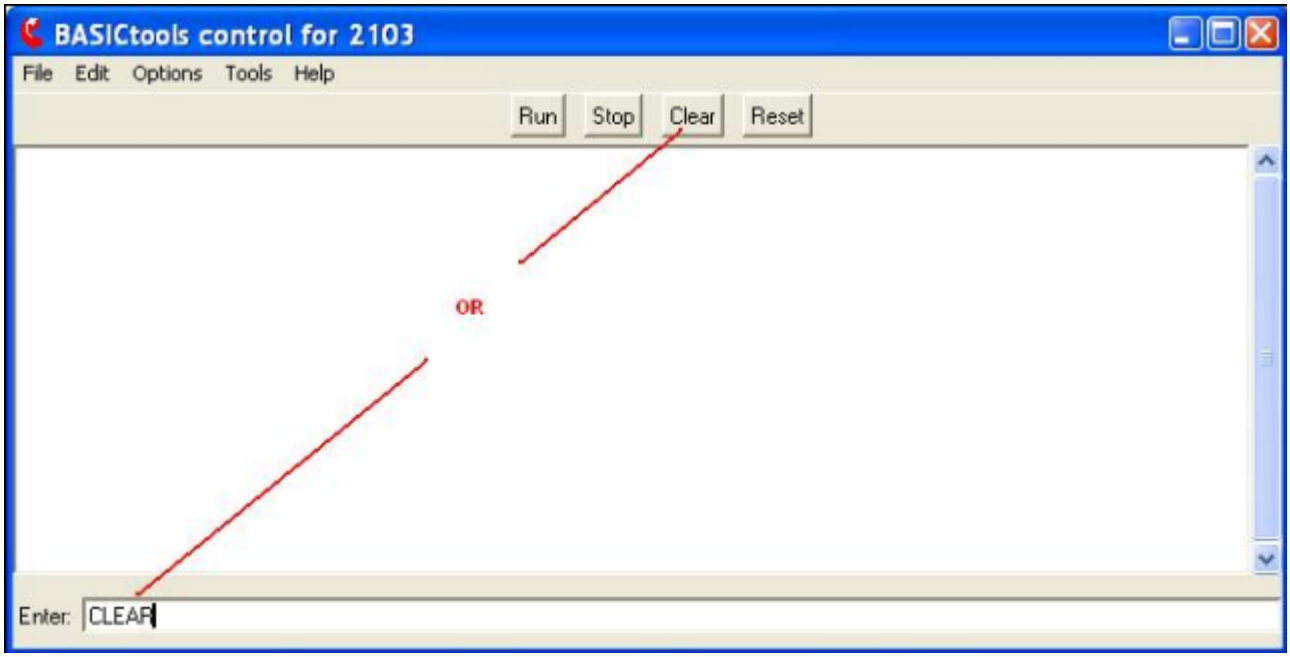
Also your program is now saved in the ARM Flash memory. And it will be executed the next time the board is RESET. So try that...



Now wiggle some pins from a program

Step 4: Programming the IO

Clear previous program



To begin a new program, you should CLEAR the previous one. You can do this with either the button or by typing clear.

A program that uses IO

For the ARMweb type the following program in the console window.

```
                        ' enable pin P0(7) is connected to the LED
WHILE X<30
  IO(7) = X AND 1       ' drive pin 7 high when x is odd, low when x is even
  X=X+1
  WAIT(500)
LOOP
```

For the DINkit type the following program in the console window.

```
                        ' enable pin P0(15) is connected to the LED
WHILE X<30
  IO(15) = X AND 1      ' drive pin 15 high when x is odd, low when x is even
  X=X+1
  WAIT(500)
LOOP
```

For the ARMweb and DINkit to drive the port 1 pins. On all versions ARMweb and DINkit, you can use the following

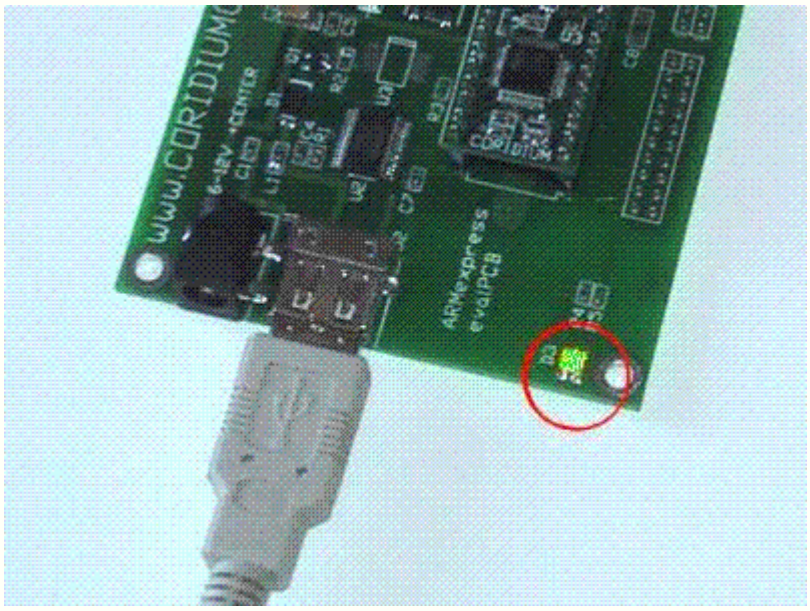
```
#include <LPC21xx.bas>

FIO1DIR = FIO1DIR OR (1<<16)    ' enable the pin as an output
WHILE X<30
   P1(16) = X and 1
    X=X+1
   WAIT(500)
 LOOP
```

Use the following if you have firmware 7.52 or later (includes floating point support)

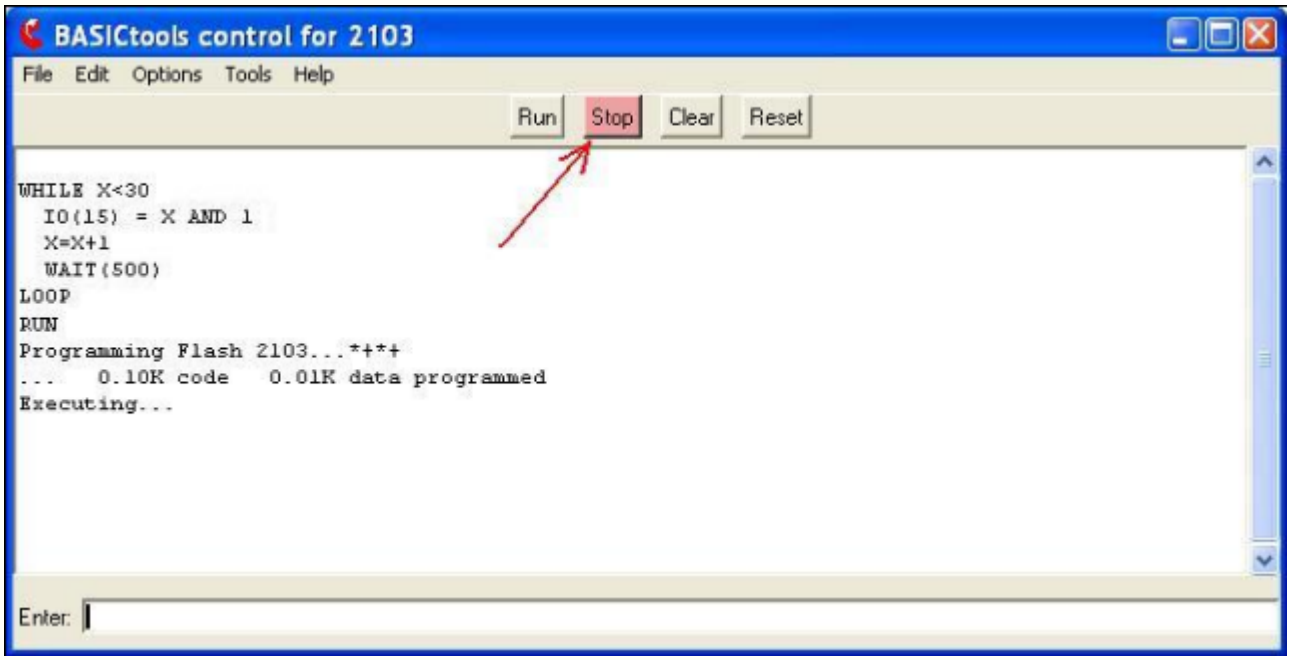
```
                        ' port 1 starts at 32
 WHILE X<30
    IO(32+16) = X and 1
     X=X+1
    WAIT(500)
  LOOP
```

The LED on the PCB should pulse 15 times.

And see the results



Stop the program



To stop a running program simply press the Stop button.

On to Step 5

ARMweb C support

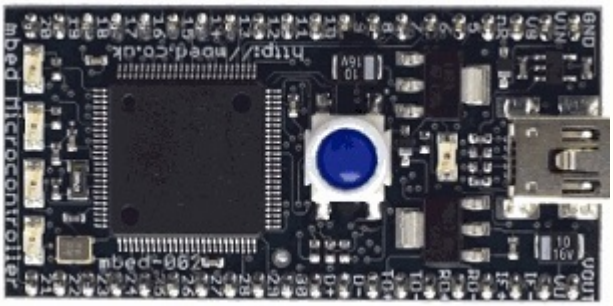
FreeRTOS

We have posted at the FreeRTOS web site a version of FreeRTOS that has been ported to the ARMweb. This open source system is available to our users.

Coridium will provide C support based on either FreeRTOS or on our proprietary system for a fee for custom programming.

The FreeRTOS will support a web server interface, but it does not include the HTML inline BASIC compiler.

MBED Getting Started



Getting Started

- Install Software
- Install Software
- Connect USB to MBED
- Writing your first program
- Writing programs with IO

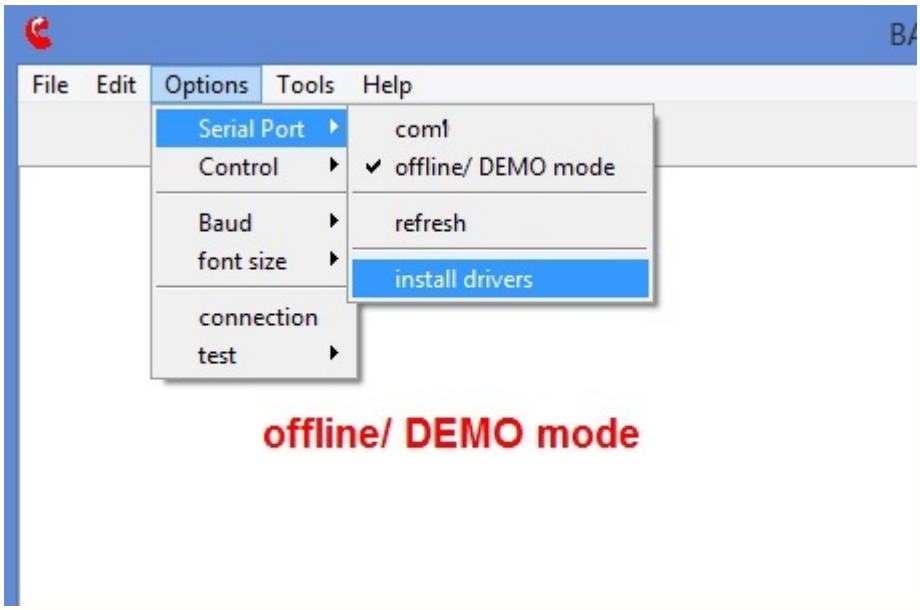
Step 1: Install Software

To start using MBED with BASIC, download and run **www.coridium.us/files/setupBASIC.exe** , directions on the **install steps here** .

Install serial drivers for MBED --

If no serial ports are active in your system you will be prompted to install drivers

Or if there are other serial devices in your system, you can install drivers from BASICtools menus

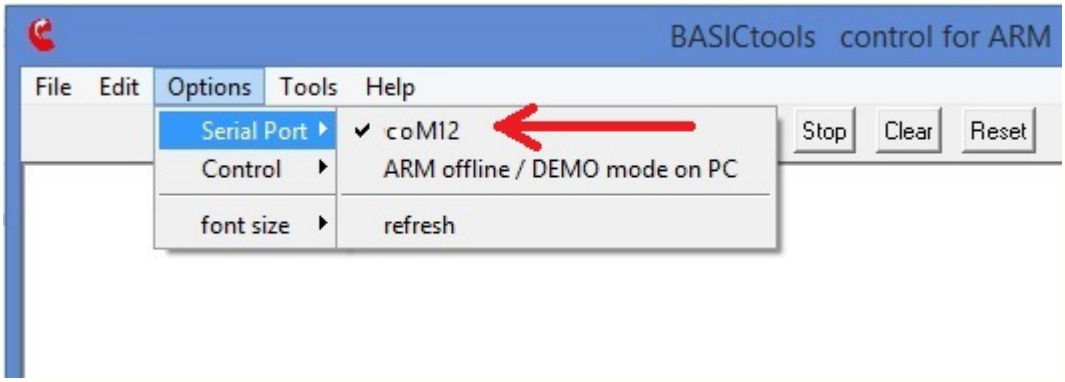


On to connect MBED boards

Step 2: Connect USB to MBED

Start BASICtools

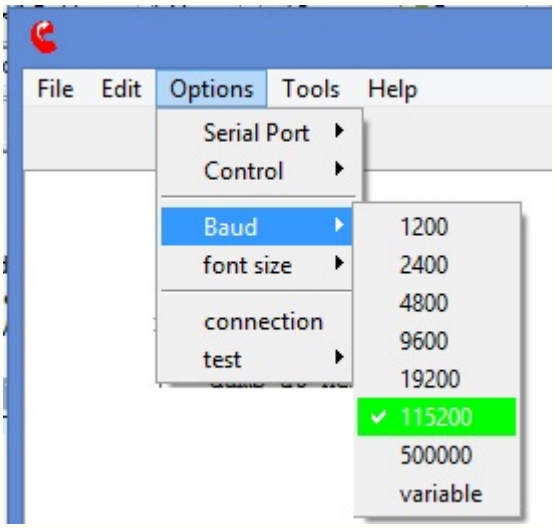
If there are no serial devices listed that have only the last letter capitalized, then the MBED drivers for a USB serial device have not been installed, go back to step 1 and install the drivers. There may be other comx ports listed (all lower case), which are not MBED USB serial devices. Any COMx devices (all caps) listed are FTDI serial ports. If there is a coMx device listed skip to the end of the page. Select that serial port.



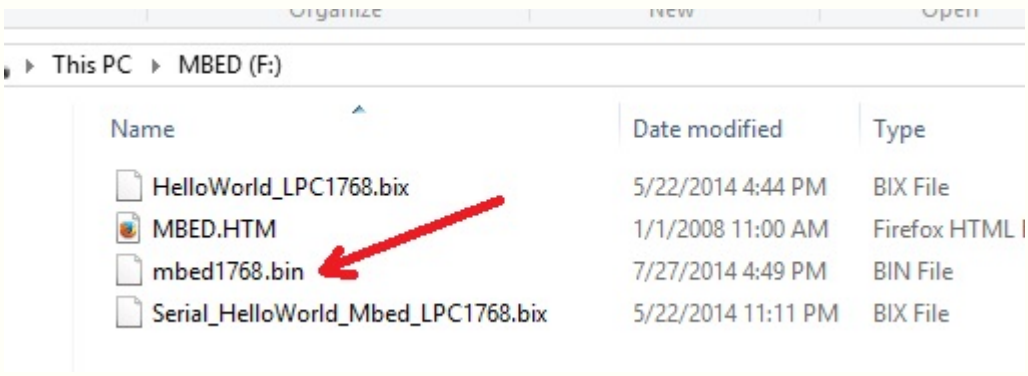
If you do not see this ComN device, then communication has not been established.

- check cables
- make sure you installed the MBED serial device drivers
- push the Reset Button

Now change the baud setting for the serial port. Earlier Coridium devices used 19.2Kbaud as a default rate, but newer ones and MBED use 115Kbaud

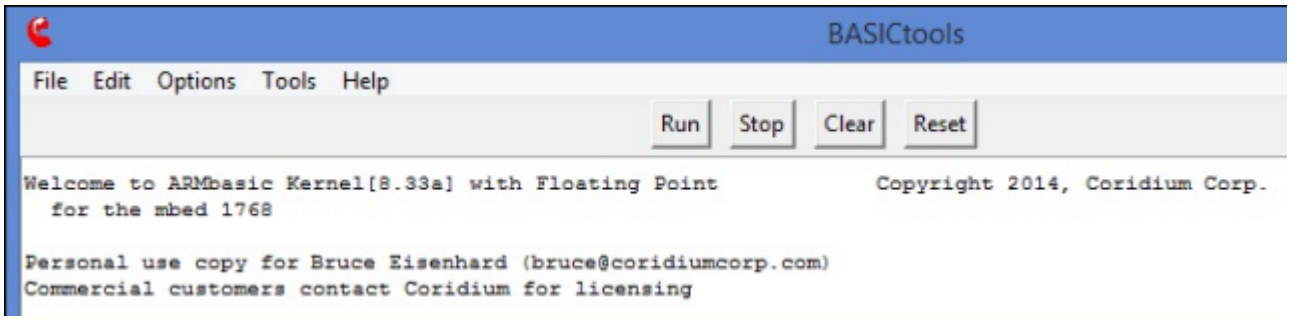


Install BASIC firmware on the MBED



You were emailed a mbedXXXX.bin file from Coridium. Copy this file onto the MBED directory, make sure it is the only .bin file in that directory. Then press the RESET button.

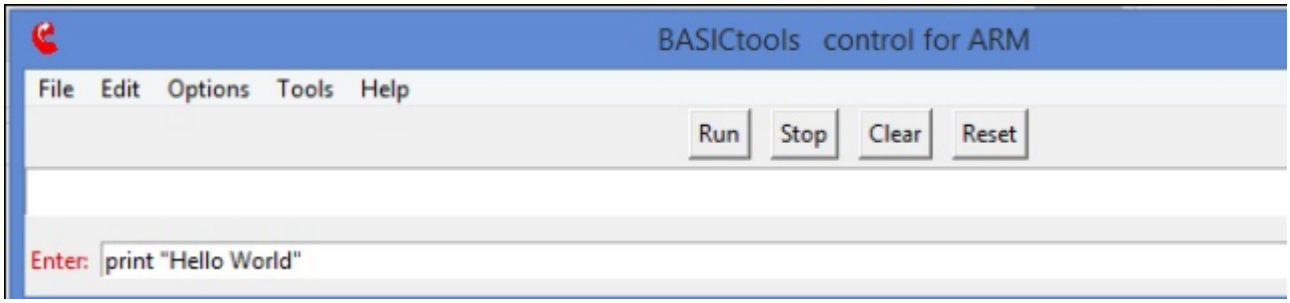
You should see a welcome message in the BASICtools window



On to Step 3

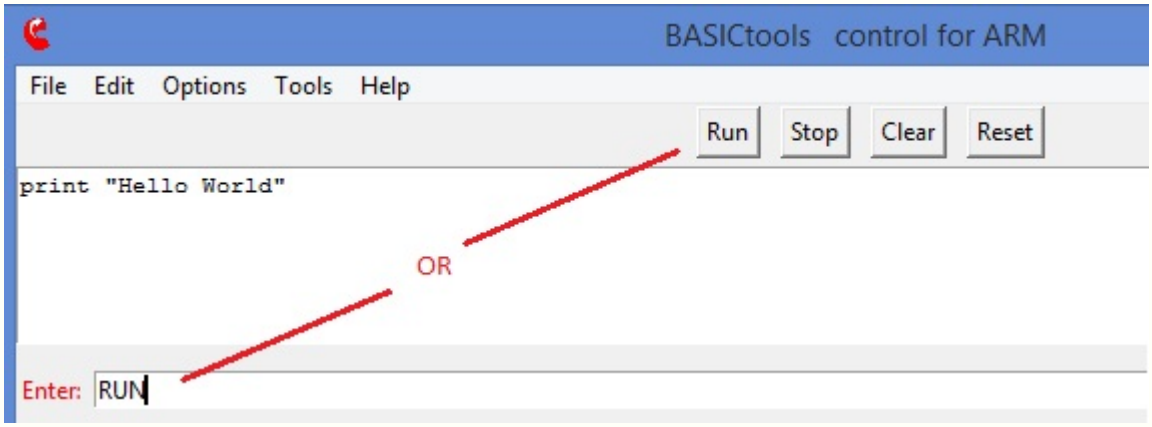
Step 3: Writing your first Program with BASICtools

The traditional "Hello World" program



So type something like the traditional PRINT "Hello World"
When you hit the ENTER key it will be sent to the compiler on the PC and be echoed back in the console window. (below)

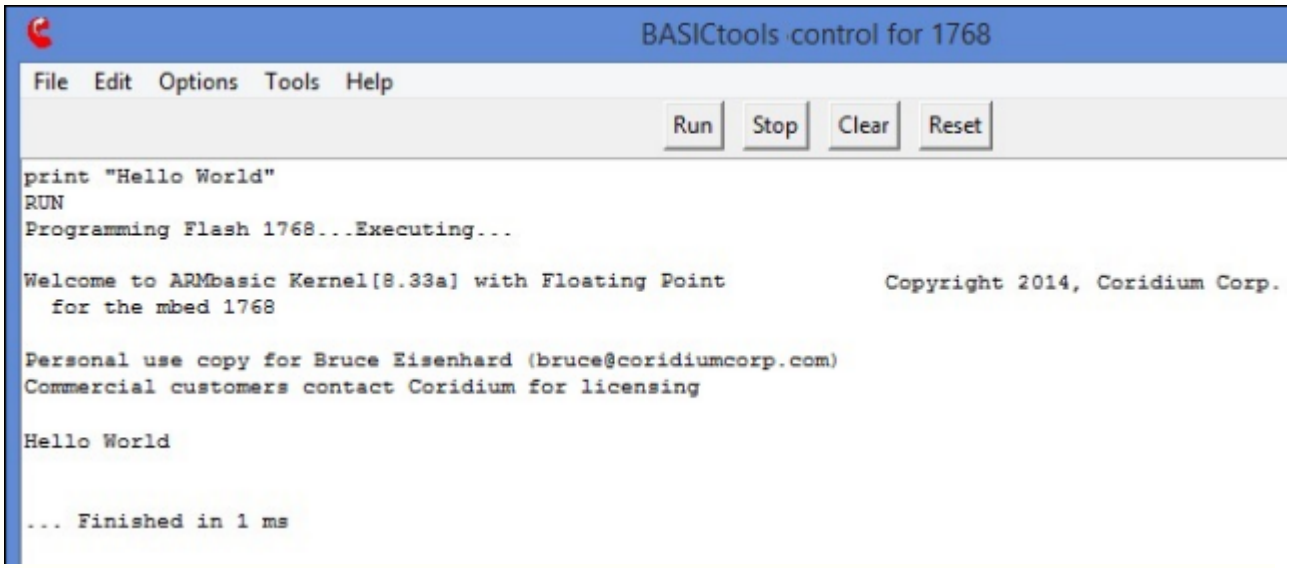
Now RUN the program



Which you can do by either typing RUN or hitting the RUN button at the top of the screen.

The program is compiled on the PC and downloaded into MBED Flash and then executed.

And see the results



You can notice a number of things. First the program is compiled and then written into Flash memory.

Next the program will be executed, as evidenced by the output of "Hello World" to the console.
ARM also reports back how long the program executed, in this case 1 msec, which is mostly startup time.

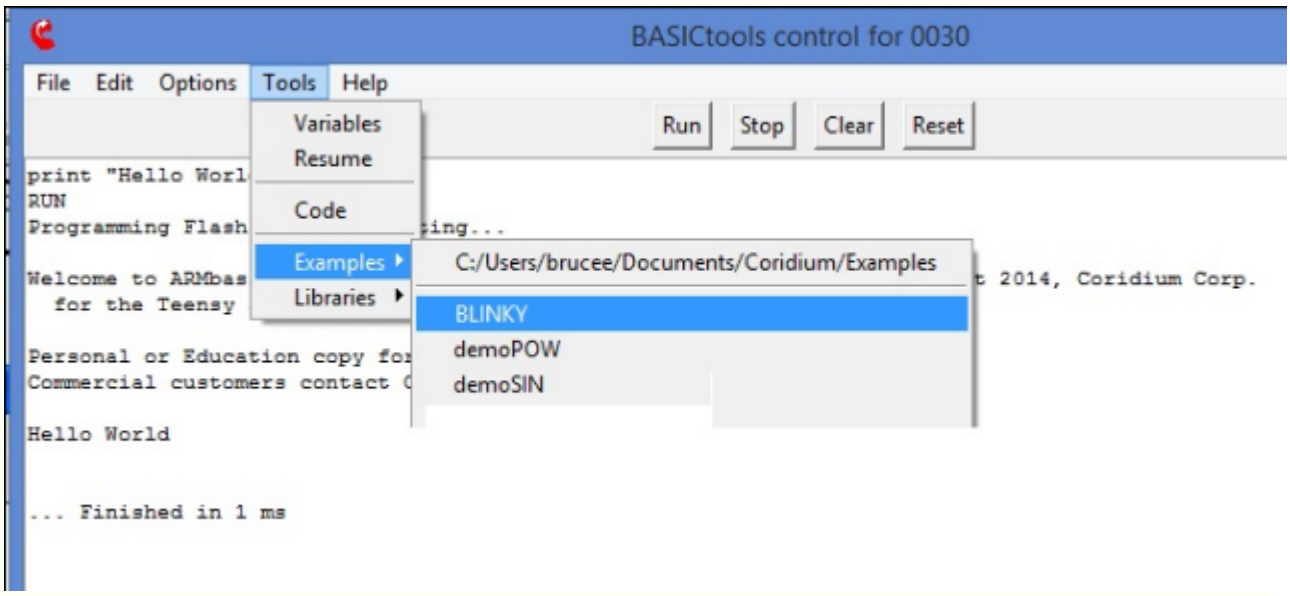
Also your program is now saved in the ARM Flash memory. And it will be executed the next time the board is powered on

Now wiggle some pins from a program

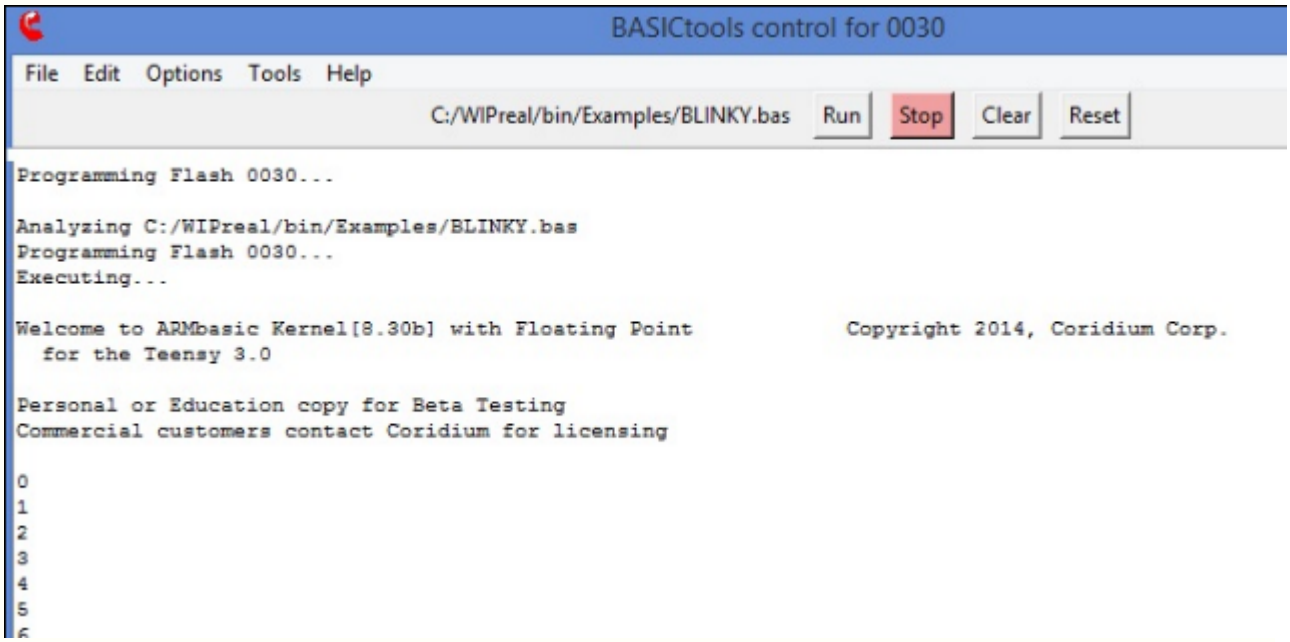
Step 4: Programming the IO

A program that uses IO

A simple BLINKY program is one of the examples, a program that flashes the LED. Load that program from the Examples



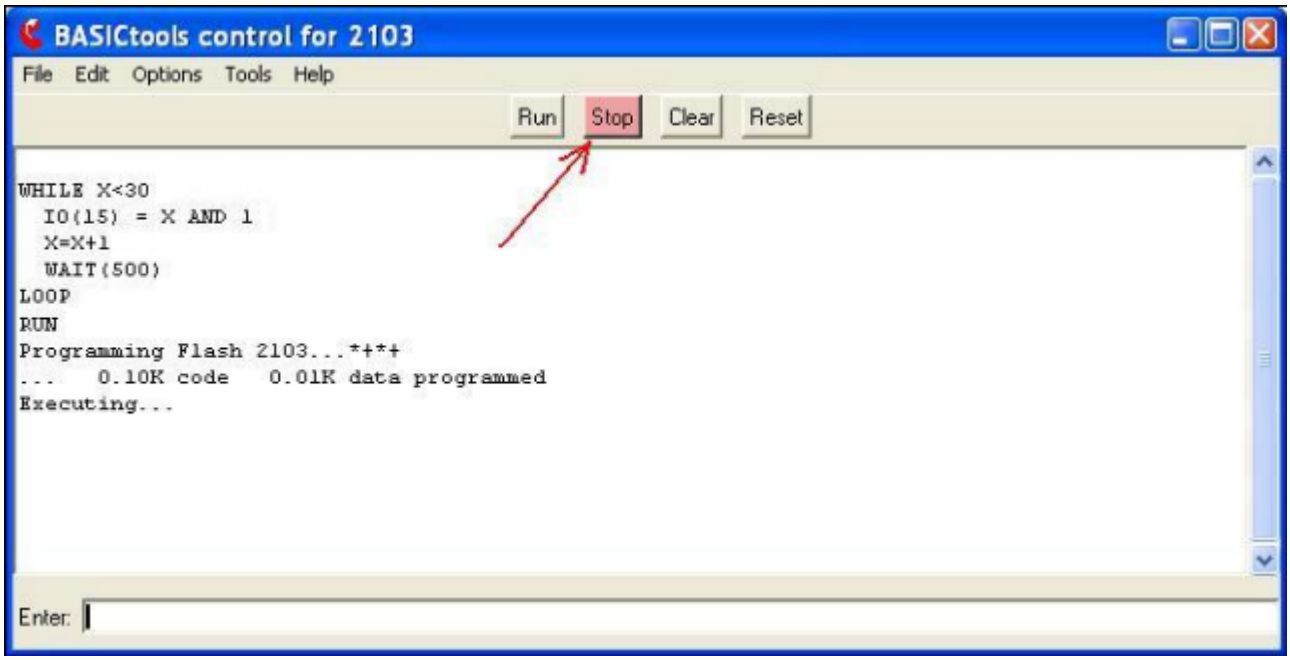
Now LOAD and RUN the program



This program counts to 29 turning the four blue LEDs on and off with each count.

To look at the source choose to Edit the file from that menu.

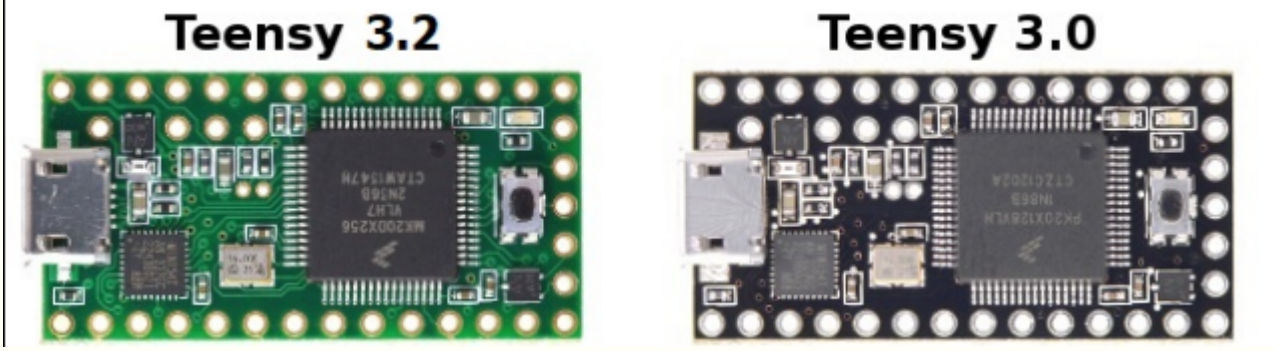
Stop the program



To stop a running program simply press the Stop button.

On to Step 5 -- details of the BASICtools program

Teensy Getting Started



Getting Started

- Install Software
- Connect USB to Teensy
- Writing your first program
- Writing programs with IO

Step 1: Install Software

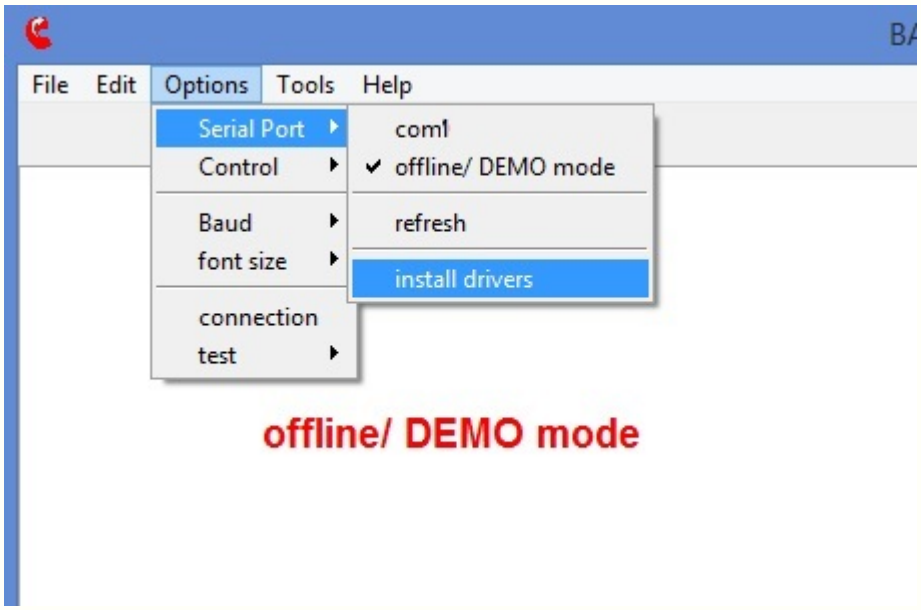
To start using Teensy with BASIC, have the latest version of Teensyduino installed. We have checked it with version 1.18-rc3. You can find the latest software releases from **PJRC here** . You can now install drivers from BASICtools follow the steps below.

Download and run **www.coridium.us/files/setupBASICbeta.exe** , directions on the **install steps here**.

Install drivers for teensy --

If no serial ports are active in your system you will be prompted to install drivers

Or if there are other serial devices in your system, you can install drivers from BASICtools menus

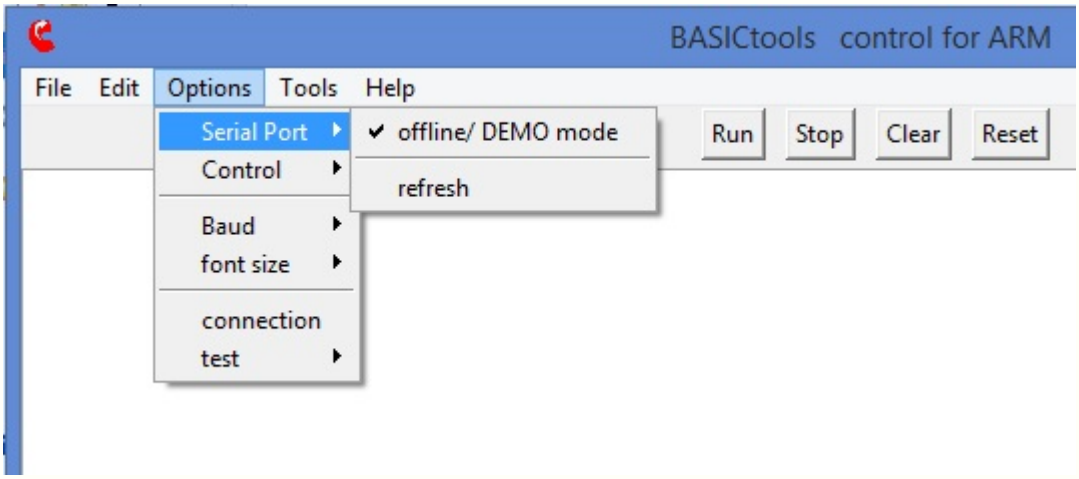


On to connect Teensy boards

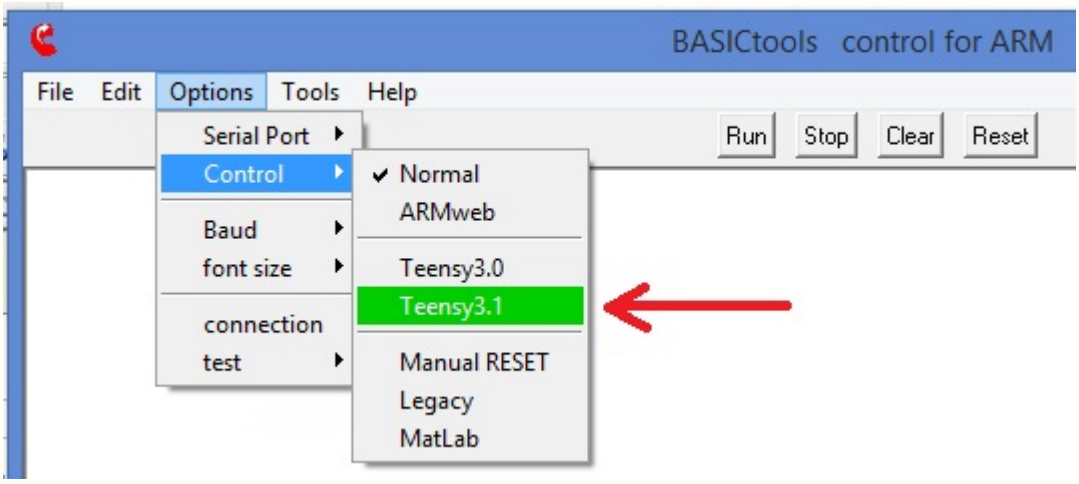
Step 2: Connect USB to Teensy

Connect USB to Teensy

Start BASICtools



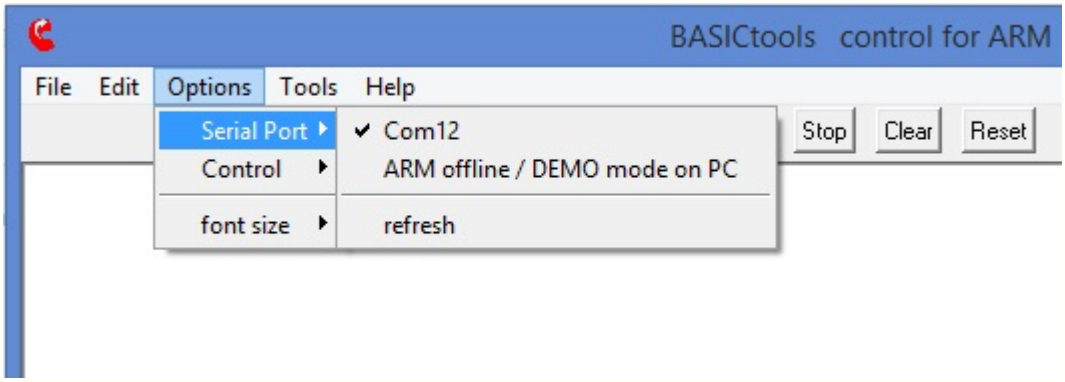
If there are no serial devices listed that have only the first letter capitalized, then the last Teensy program did not enable the USB serial device. There may be other comN ports listed (all lower case), which are not Teensy USB serial devices. Any COMN devices (all caps) listed are FTDI serial ports. If there is a ComN device listed skip to the end of the page.



Select the Control Tab, and select the Teensy device you are using.

These 2 popups below will appear. To initially load a program that enables the USB serial device push the button on the Teensy.

After the button is pushed the Options menu will now reflect Teensy options and will show a ComN port (first letter capitalized) as a Teensy USB serial port.



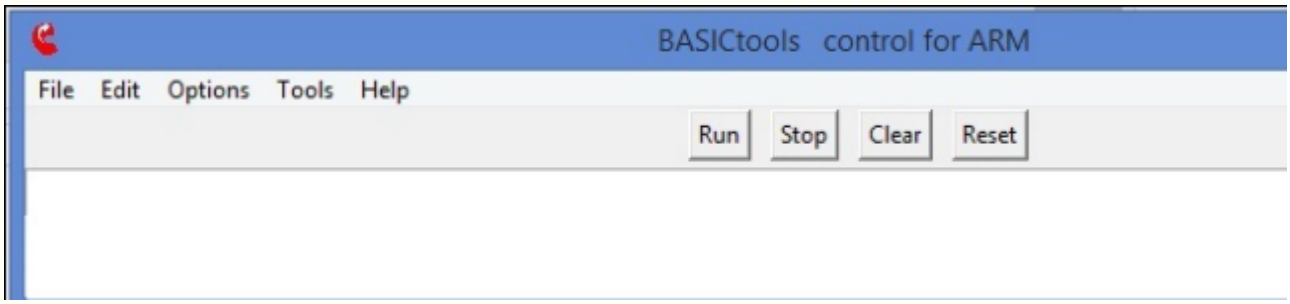
If you do not see this ComN device, then communication has not been established.

- check cables
- make sure you choose the correct device Teensy3.0 / Teensy3.1
- push the Activate Button

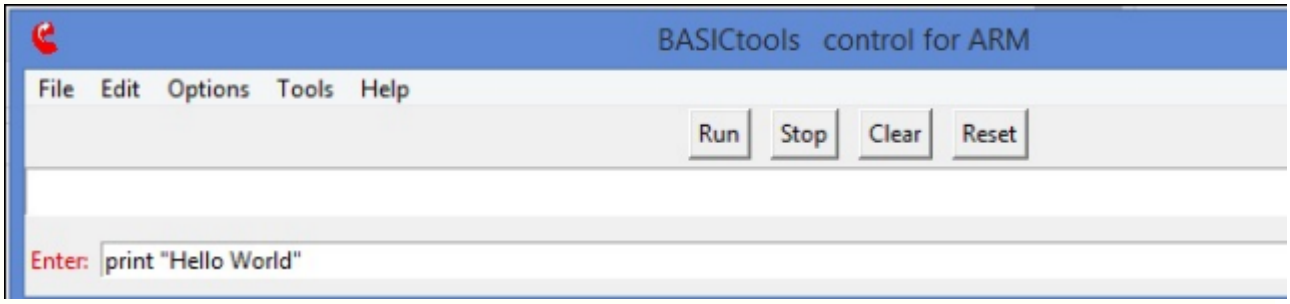
On to Step 3

Step 3: Writing your first Program with BASICtools

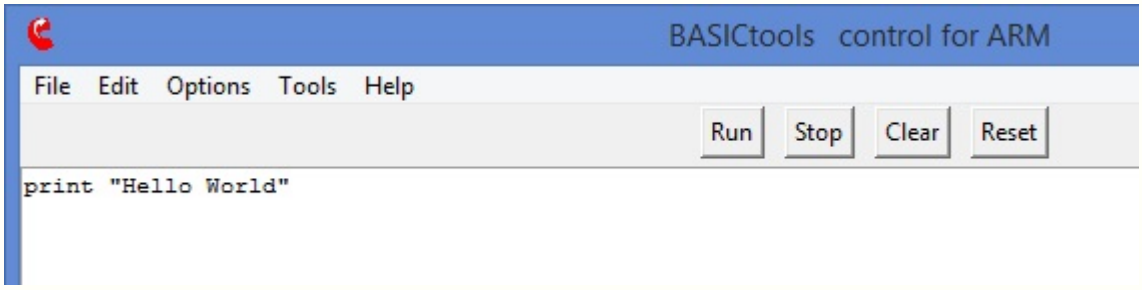
Start the BASICtools from the Start-Menu or from the Desktop Icon. You will not see a welcome message which the Teensy until you start the first program-



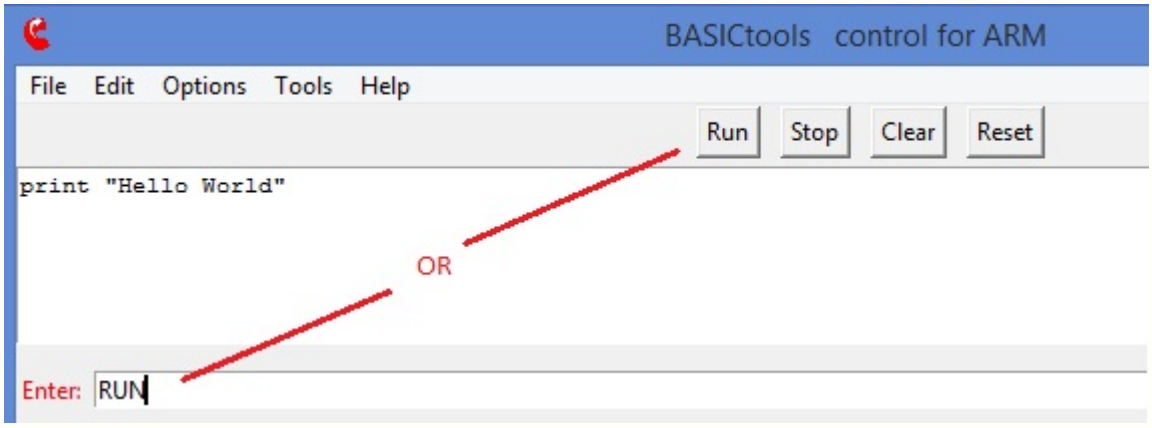
The traditional "Hello World" program



So type something like the traditional PRINT "Hello World"
When you hit the ENTER key it will be sent to the compiler on the PC and be echoed back in the console window. (below)



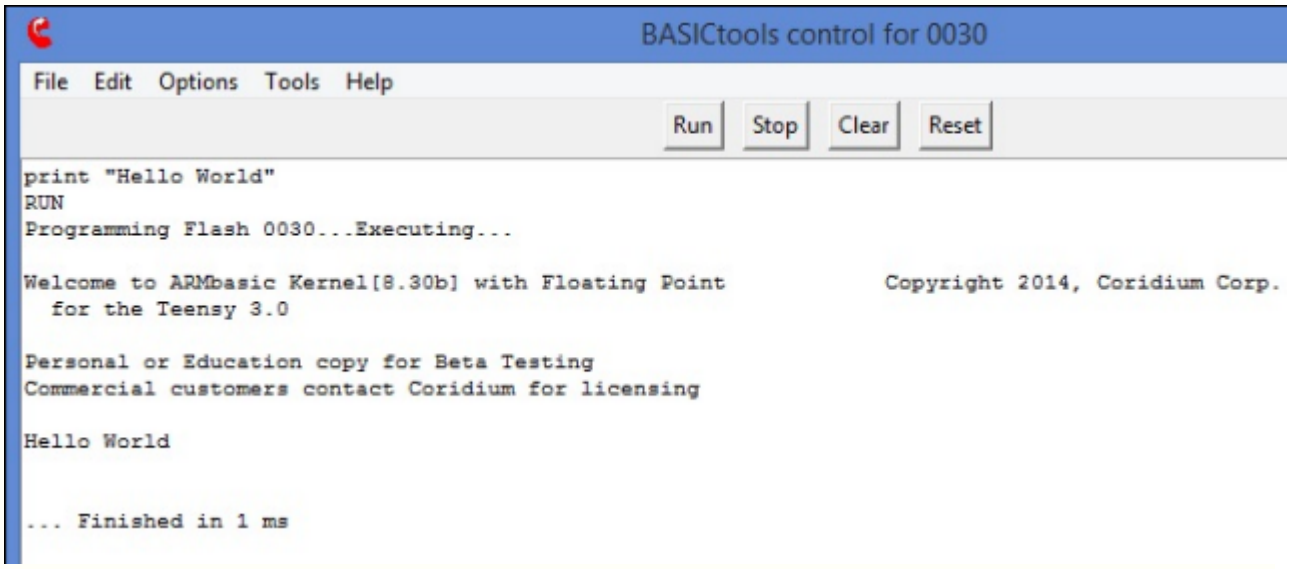
Now RUN the program



Which you can do by either typing RUN or hitting the RUN button at the top of the screen.

The Teensy.exe file will be called to download the program (you should not need to push the Activate button) and a "Teensy is Downloading" popup screen will be active while the program is being loaded and starts running.

And see the results



You can notice a number of things. First the program is compiled and then written into Flash memory.

Next the program will be executed, as evidenced by the output of "Hello World" to the console.
ARM also reports back how long the program executed, in this case 1 msec, which is mostly startup time.

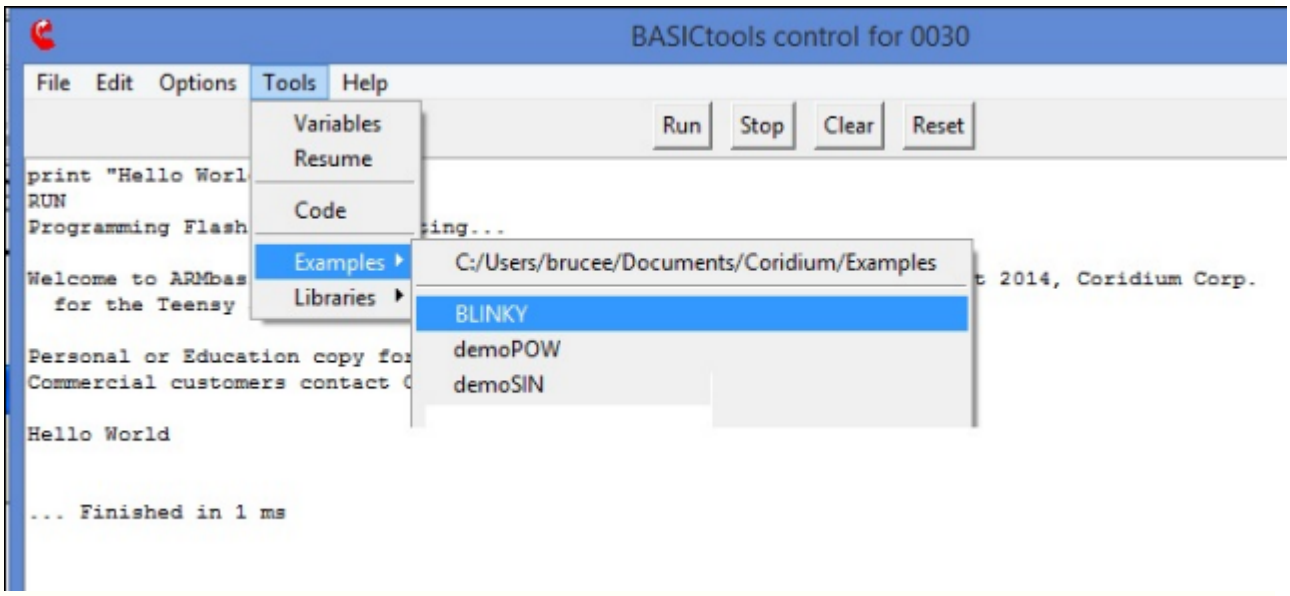
Also your program is now saved in the ARM Flash memory. And it will be executed the next time the board is powered on

Now wiggle some pins from a program

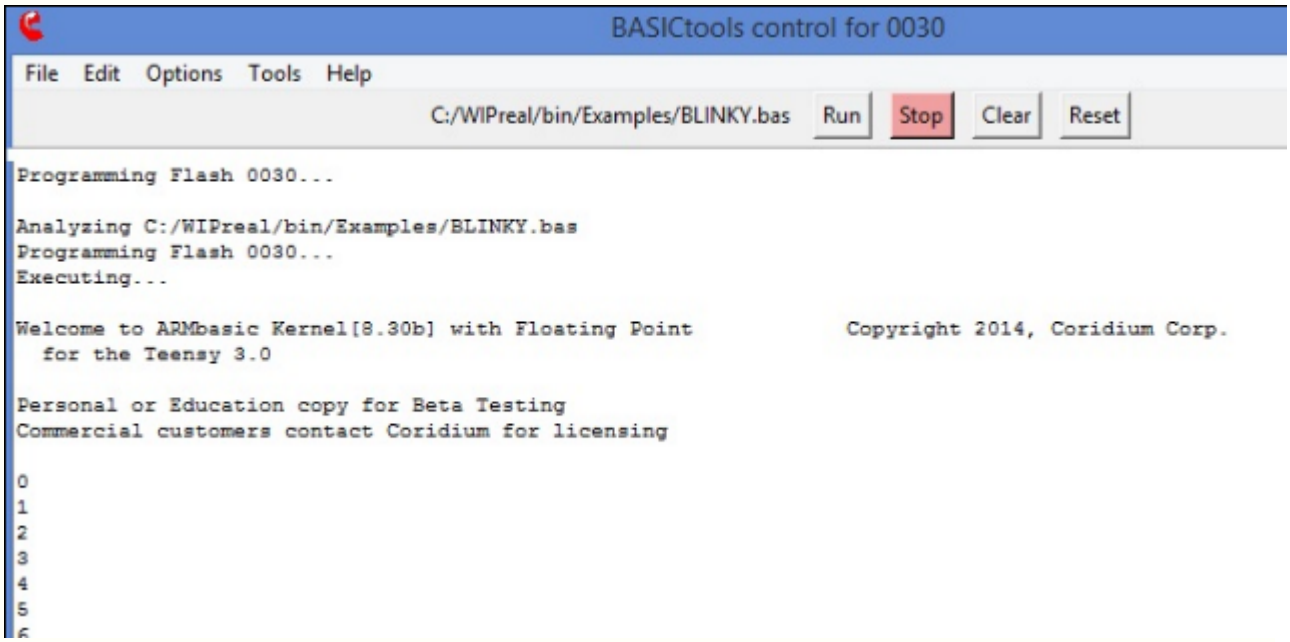
Step 4: Programming the IO

A program that uses IO

A simple BLINKY program is one of the examples, a program that flashes the LED. Load that program from the Examples



Now LOAD and RUN the program



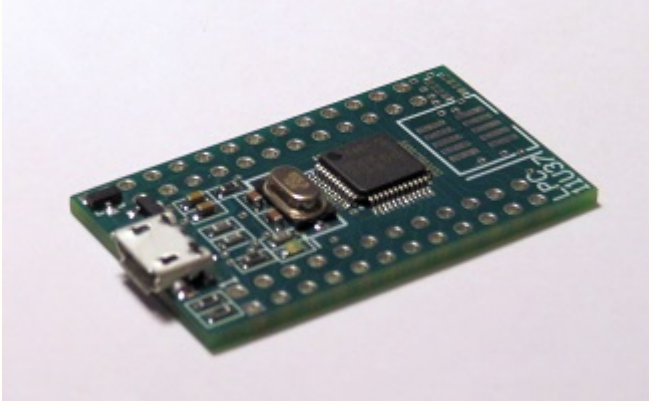
This program counts to 29 turning the LED on or off with each count.

To look at the source choose to Edit the file from that menu.

To stop a running program simply press the Stop button.

On to Step 5 -- details on BASICtools program

The Compiler



About

ARMbasic is a 32-bit BASIC compiler for **ARM** processors. It was started to create a portable, alternative to hardware debuggers, but has quickly grown into a powerful programmable controller tool, already including support for asynchronous serial, I2C, SPI, PWM, timer and counter operations. It is run on ARM CPUs from NXP ranging from the 50 MHz 32KB BASICchip to the 100 MHz 256KB SuperPRO.

ARMbasic is simple to use, with a USB connection to the PC for programming. The target applications include control functions, so performance and a powerful set of hardware routines have been included. The language has a minimum of overhead when compared to larger general purpose languages.

Aside from having a syntax the most compatible possible with MS-VisualBASIC, **ARMbasic** introduces several new features such as hardware specific routines, string support, limited pointers and many others.

ARMbasic is written in ANSI-C compiled with GCC.

Main Features

Simplicity

- Many control applications can be accomplished in a very small program
- **ARMbasic** can be installed in minutes, and be solving your control problems just as quickly
- While BASIC is considered a simplistic language, **ARMbasic** with built-in hardware functions and the speed of compiled code can be a higher performance solution than many more complex languages
- As it is an incremental compiler, it has the feel of an interpreter. Its quick and easy to debug its programs. Why learn a new development system, you can either enter programs directly from the console or use any text editor that you are already familiar with.

BASIC Compatibility

- **ARMbasic** from Coridium is not a "new" BASIC language. It is not required of you to learn anything new if you are familiar with any Microsoft-BASIC variant. Even if you don't have knowledge of the BASIC language, its constructs are easy to learn and easy to use.
- **ARMbasic** is case-insensitive; scalar variables don't need to be dimensioned or declared before use; MAIN function is not required. Syntax follows much of that of Microsoft-Visual BASIC

Most of the PBASIC IO functions have been added

- INPUT and OUTPUT control pin direction
- IO(x) reads or writes a pin state and control input/output direction
- HIGH and LOW control pin output values
- I2C on any of the pin pairs
- SPI using any group of 2/3 pins
- Hardware based PWM
- Software PWM on any pin with 256 levels
- PULSIN and PULSOUT will measure or output a pulse
- SHIFTIN, SHIFTOUT can be used for SPI or MicroWire devices
- OWIN and OWOUT support one-wire devices
- SERIN, SEROUT can be used for low duty cycle asynchronous serial ports on any pin up to 115Kbaud
- RCTIME will measure a capacitive delay

Support for 32-bit variables and Strings

- Integer: (32-bit math)
- Single (32-bit IEEE 754 floating point)
- String support

Arrays

- Static arrays supported, up to 32KB in size on the ARMexpress, 4KB on the ARMmite

Memory Limits

- All arrays, variables and strings are allocated from a 33KB space on the ARMexpress, 5KB on the ARMmite
- Code will include user programs, constant strings (used in expressions or PRINT), DATA constants.
- On the SuperPRO 128KB is available for user programs, and an additional 8KB is available for DATA constants and constant strings. In addition 120KB of Flash can be written and functions as non-volatile memory. Note that Flash may be written a minimum of 100K times.
- On the BASICchip 20KB is available for user programs.

Direct Hardware Access

- Uses the same syntax as C-pointers

Debugging support

- The ease and speed of an interpreter.
- Dump of variables used
- compiled breakpoint

Included with any module

- The **BASICchip** and **PRO** families compile their programs on the PC and they are downloaded using BASICtools, that compiler is part of the utilities available for download from Coridium

Requirements

All versions

- **ARMbasic** for the BASICchip, PRO family, ARMmite, and ARMweb runs on Windows and is controlled by a USB port..
- The **ARMbasic** compiler runs on the Windows PC, all versions Windows XP and later.
- The **ARMbasic** compiler runs on the ARMweb hardware platform and can use a browser for simple programs.
- BASICtools is a simple IDE which controls compilation and download of programs to the ARM hardware.
- TclTerm is a terminal emulation program written in Tcl, and has been ported to Windows. Other terminal emulators may be used, if they allow control of DTR/RTS, or they can be run in **Legacy mode** .
- Documentation is available in both Windows CHM format, PDF and HTML.

Installing

Windows 7

Windows Vista

- Follow the installer instructions which are also outlined in the Getting Started section. The compiler is run on the PC and hex code is downloaded and stored in Flash on the ARM chip.
- Connection to a PC is done with a serial port, details in the corresponding **Getting Started Section** .

Windows Vista 64bit version

- The Windows XP installer BASICtools and TclTerm interface program works for WinXP x64, but the drivers specific for x64 and the FTDI interface must be used.

Windows XP

- Follow the installer instructions which are also outlined in the Getting Started section. The compiler runs on the PC.
- Connection to a PC is done with a serial port, details in the corresponding **Getting Started Section** .

Windows XP 64bit version

- The Windows XP installer BASICtools and TclTerm interface program works for WinXP x64, but the drivers specific for x64 and the FTDI interface must be used.

Windows 2000

- The Windows XP installer, BASICtools and TclTerm interface program works for Win2000.

Windows 98

- Win98 is no longer supported, if you have an old machine install Win2000 on it.

Linux

- Currently an installer is not supported, but only the documentation and a terminal emulator are required.
- A command line interface has been developed for Windows as an example of how to do the same in Linux. The necessary files and sources can be found in the **files section of the Yahoo ARMexpress Group** . There is an effort to port this to Python going on, contact Coridium if you would like to help.

Others

- To communicate with the ARM, a connection to a serial port is required
- The documentation is available in HTML format so anything with a browser should be capable of using it.
- Parallels on Mac OS X runs with the WinXP utilities. OS X version of Tcl does not currently support serial devices so we have not been able to port our utilities to run natively on the Mac.

.

Running

Windows version

- A desktop icon and start-menu links should be created by the installer, use them to open the console directly into the directory where the tools are stored
- see Getting Started section

Linux version

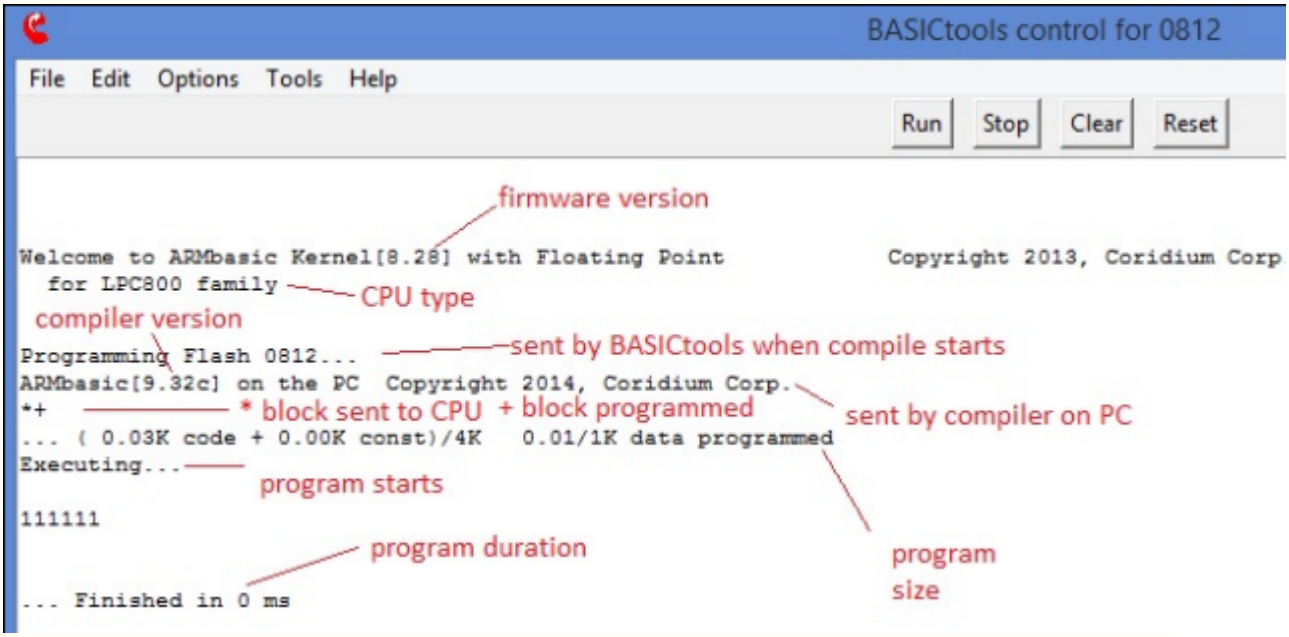
- port not done, though the source is available
- contact Coridium if you want to help build a Linux version, we aren't Linux experts but have provided native versions of the compiler
- BASICtools is written in Tcl and runs on Linux

Mac version

- runs on Parallels using WinXP

DOS version

- no direct support for this



ARMbasic and other BASICs

ARMbasic and Visual BASIC have different goals. Visual BASIC is a general purpose language that includes access to various elements of Microsoft Windows and its application programs. **ARMbasic** is a small language aimed at controlling hardware with some communication abilities with host systems. Wherever practical **ARMbasic** is a proper subset of Visual BASIC. Some elements of earlier BASICs do not apply to Visual BASIC, but still do in ARMbasic. These elements include keywords such as RUN and CLEAR.

Data Types

- Visual BASIC has a rich set of data types as well as object oriented extensions.
- In ARMbasic the default data type is 32 bits (SIGNED INTEGER), and also supports arrays of SIGNED INTEGERS and STRINGs of bytes terminated by 0. and 32 bit IEEE 754 floating point numbers.

Changed due to ambiguity

- FOR..NEXT is ambiguous for negative STEP. To clarify negative steps use DOWNTO.

Design differences

- One goal of ARMbasic is to be a simple, easy to use language, but still be a powerful tool for controlling hardware. For this reason a simple subset of BASIC has been chosen, with extensions for hardware control.
- Only single dimension arrays are supported.

Pre-Processor

- This is a very powerful tool available to C programmers, but not available in many BASICs
- The C-preprocessor (CPP) has been integrated into BASICtools

Differences from PBASIC

ARMbasic version 7 has been shipping and it abandons the script style commands of PBASIC hardware routines in favor of Visual BASIC like functions and subroutines in separate libraries accessed by #include.

32-bits vs. 16-bits

- ARMbasic is written for 32-bit hardware, and cannot utilize code which depends on 16-bit truncation.
- The default data type is 32 bit integers, rather than 16 bits in PBASIC.

Changed due to ambiguity

- FOR..NEXT is ambiguous for negative STEP. To clarify negative steps use DOWNTO
- The PBASIC syntax of IN0, DIR0, OUT0 has problems with parameterization. It is replaced by the use of IN(0), DIR(0) and OUT(0).
- The formatted input of many PBASIC words will in many cases hang waiting for input if it is not of the proper form. Its better to accept any or all input and then parse it later, but PBASIC does not have that ability. A simple set of string functions have been added to **ARMbasic** to interpret input

Design differences

- Integer variables do not need to be declared. This is common to most other BASICs. ARMbasic does not require simple variables to be declared before use. As of version 6.23 of the Windows ARMbasic compiler allows the use of DIM xxx AS INTEGER to declare simple variables, and will enforce that all variables be declared by DIM after that first DIM declaration.
- As there is much more variable space available, simple BIT, NIBBLE, BYTE types are not supported. Arrays of BYTE also called strings **are** supported
- Normal BASIC array declarations are supported using DIM. Unlike PBASIC syntax.
- PIN declaration is replaced by treating pins as an array IN(x) vs. INx. This makes parameterization of pins simpler.
- The standard CONST syntax of most BASICs is used instead of PBASIC CON syntax
- Multiple statements on a single line are not supported
- The standard PRINT is used and its syntax is used in place of PBASIC DEBUGOUT
- Simple statements must be completed on a single line, run on statements are not supported
- The $ suffix can be used to declare strings using the DIM statement
- Strings use a null (char 0) terminator .
- CLEAR is used to reset all variables and reset the stack.
- In an interpreter there is an advantage to having functions such as &\ |\ ^\ ** *\ DIG and DCD But these are easily done in a compiled BASIC and have no performance or space penalty.

```
x = NOT (a AND b)       ' equivalent to a &\ b

x = a * b >> 16         ' equivalent to a ** b (for 16 bit values)

x = a * b >> 8          ' equivalent to a */ b (for 16 bit values)

x = y /1000 mod 10   ' equivalent to y DIG 4

x = 1<<6                ' equivalent to DCD 6
```

- HYP, TAN and NCD are not implemented in ARMbasic
- Many differences will be handled in the PBASIC translator pre-process step (under development)
- -$hex values are not supported

Design simplifications

- Only 1 statement per line is allowed
- run-on statements are not allowed (continuation to the next line)
- Formatted input is replaced with elementary string functions
- PRINTF is part of the firmware and can be called from ARMbasic

Archaic commands

- DTMFOUT is not supported.
- ON and BRANCH should be coded using SELECT CASE.
- LOOKUP can use arrays or strings.
- LOOKDOWN should be coded using SELECT CASE
- GET, PUT can be replaced with arrays

Preprocessor for BASIC

Most BASICs do not have a pre-processor. **ARMbasic** does not include one as part of the standard language, but a version of the CPP has been included as part of the utilities.

The CPP (C preprocessor) is a very powerful tool, most users use just a fraction of the features, but if you want the full story check **this document from gcc** .

As of version 5.14 of BASICtools, the type of CPU is used to generate a #define LPCxxxx that is added before any of the user source code. This can be used to write code good for any of the CPUs as is done in RTC.bas and HWPMW.bas libraries.

These are the most common directives that apply to use with **ARMbasic**: Unlike **ARMbasic** these keywords and any parameters used in them ARE CASE SENSITIVE. The pre-processor is run on the PC, so it is not available when using the built-in compiler of the ARMweb. However the compiler with preprocessor can be used to generate files that can be downloaded to the ARMweb (use the Save Intermediates check box in the Files menu of BASICtools).

#include "filename"

#include <filename>

#define

#ifdef

#ifndef

#if

#if (defined)

#else

#elif

#endif

#undef

#error

#warning

CPP operation

The CPP is a multi-step process carried out automatically by the BASICtools program. All operations are done in a temp file directory created at %temp%Coridium/temp. All files in this directory will be deleted when a File>>Load is performed by BASICtools.

It starts with your source file, and it will be copied into the %temp%/Coridium directory. When copied all comments will be stripped. All included files will be also copied into this temp directory. Then the CPP will be run on the files in that temp directory creating a __temp.bpp file that is the result of all the pre-processor operations. This __temp.bpp file will be combined with other information as __temp.bas and then compiled by ARMbasic.exe and its output is __temp.out. This __temp.out file is a modified Intel hex format of the code generated by the source BASIC program. __temp.out will be downloaded to the ARM.

In addition __temp.bat and __errors.tmp files will be created. __temp.bat is a batch file used in the compilation process. Errors from the compile or any of its steps will be contained in __errors.tmp.

Frequently Asked Questions

ARMbasic questions:

What is ARMbasic?

ARMbasic is a compiler included in a family of modules using the ARM CPU from Coridium Corp. The compiler runs on the ARM processor for the ARMweb products or on the PC for the other ARM products.

ARMbasic has a syntax generally compatible with Visual BASIC,

ARMbasic is written in ANSI C, compiled with GCC.

Who is responsible for ARMbasic?

Coridium Corp. distributes and maintains **ARMbasic**. They can be contacted at www.coridiumcorp.com .

Why should I use ARMbasic?

ARMbasic has innumerable advantages over the alternatives.

- It's fast.
- It produces compiled machine code not interpreted tokens.
- It's simple.
- It has powerful hardware functions built-in for the popular serial control busses.
- It's cost effective.
- It's easy to use
- Did we say it's fast?

Why should I use ARMbasic rather than GCC?

There's no question that some problems require more complex languages. But many control problems are quite simple and this is what **ARMbasic** exceeds at. In many cases **ARMbasic** will run faster than a compiled C program. How is that possible, you ask? The answer is that **ARMbasic** has only global scope, there is no stack frame in the majority of the user code. Control transfers are faster than procedure calls of C or Java. **ARMbasic** is a compromise of speed and code size, but it compares favorably to programs written in C.

How fast is ARMbasic?

The fastest loops use the WHILE ... LOOP, with a simple loop running 7 million iterations per second.
Loops take a number of instructions to execute, when running simple instructions such as X= X+1, it will run at speeds exceeding 13 million lines per second.

How compatible is ARMbasic with Windows Visual-BASIC code?

ARMbasic uses Visual BASIC syntax where compatible. Its unlikely you'll be porting a Visual BASIC application to ARMbasic, but if you do let us know about it.
Being a subset of Visual BASIC opens a larger audience of programmers to this tool, including those who may not have thought they'd be writing code for programmable controllers. .

Does ARMbasic support Object Oriented Programming?

ARMbasic does not support Object Oriented Programming.

Variable Scope

- Most labels and variables are global in ARMbasic. The advantage is that there is little stack overhead which gives greater performance.

- As of version 6.24, of the PC compiler a local scope for functions has been added.

Floating Point Math

- ARMbasic uses both 32 bit integer math and 32 bit IEEE 754 floating point math. The difference in performance is approximately 10:1 for operations on INTEGERs vs. operations on SINGLEs.

Why have any of the compiler on the ARM?

The original ARMexpress had the compiler completely on the ARM, and this was the heritage of where the compiler came from and why it came into existence. But the intention was always to have an ARMweb product, and for that product to support adding ARMbasic statements into a web page that are executed on the fly. The only reasonable way to do that was to be able to compile those statements at runtime during page service, and that means the compiler has to live at least on the ARMweb.

The 2103 group of products uses a very small ARM memory chip, so the runtime and hardware libraries that are used by the **ARMbasic** are all that is included there.

Another side-effect of the compiler being on chip, is that it had to be small, and the smallest compilers are of the recursive-decent type, which includes the **ARMbasic** compiler. What this means is that the syntax of the language is included in the source of the compiler parser. An advantage of these compilers is the size and normally they are also pretty fast. Some of the bad things are you can break the compiler with some odd coding styles. As there is a stack being used for parsing, you can make that overflow with statements that cause a lot of recursion like-

```
x =
(((((((((((((((((((((((((((((((((((((((((((((((((((((((((((((((((((((((((((((((((((((1))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))
)))))))))))))))))))
```

But why would you need to write any code like that? Another "feature" of recursive decent compilers is that error recovery can be poor. The way we chose to do this is to have any error reset the parser to the "outermost" state. What this means is that if an error occurs inside a loop like

```
DO
  x  x =2
  y = 3
LOOP
```

will cause an error on the LOOP statement as well as the x x = 2 statement, as the loop has been broken as the parser returns to the outermost state. Yes this causes errors on good statements, but its a prudent choice from our perspective. You don't want the compiler guessing what you meant and correcting your code (I believe PL1 tried that to comical results).

What are the planned future features for ARMbasic?

- more string functions
- more serial busses
- more hardware functions
- networking
- analog functions
- let us know what you need

Can ARMbasic be customized?

Coridium Corporation is aimed to produce high performance modules based on the latest technologies. Currently this includes the ARM processor. But Coridium also has the engineering resources to customize our designs for the specific needs of our OEM customers. This may include an interface to a specific peripheral chip with language extensions added to the ARMbasic. It may also include an FPGA solution to extend the capabilities of both the hardware and software.

So if you need something special, but want the ease of use of **ARMbasic** , tell us about your application. We are quick to respond, and have designed a custom hardware software combination that delivered prototypes in a couple of weeks, and production volumes within a month.

What volumes make sense for customization? It depends on the complexity, but at a few hundred units the numbers begin to pencil out.

-

Revision History

Revision History:

6.06

ARMbasic initial release summer of 2006

This version of hardware uses open drain IOs on IO(5) and IO(6), this will be changed in future versions.

6.07

Generalized the operation of the I2CIN (backward compatible) and I2COUT.

Optimized all index operations (includes arrays, input/output and strings). Gave 3x performance improvement for these types of operations. Now no difference in using constants or expressions for indexes.

Added the ability to use SIN and SOUT pins for SERIN, SEROUT, BAUD(), RXD() and TXD() as pin 16.

Corrected STRCOMP function.

6.08

Extended break timeout on RESET to 0.5 second.

Accept either CR or LF to terminate a line.

SLEEP now goes into a power down mode using alarm function to wake up.

DEBUGIN string added

Enforce proper declaration of strings and arrays

Multiple string concatenations allowed per line

noted an error - BAUD rate for port 16 can not be changed currently.

6.09

Added string$ support as an Outputlist in hardware functions (zero terminated or constant string)

Expanded the space available for programs to 56K.

6.10

Support for ARMmite.

6.11

BAUD rate setting for port 16 (the hardware serial port) is now allowed. The ARMexpress transceivers limit speed to 19.2Kb, but the ARMmite can run up to 942Kb on port 16.

6.12

Expanded symbol table on ARMexpress, and also allow PC to compile for ARMexpress, which allows much larger symbol table.

6.13

Added SPIMODE and SPIBI.

6.14

Fixed a bug affecting ARMexpress only in large programs with certain GOSUBs. The bug resulted in programs restarting at the GOSUB.

6.15

Improved SPI performance.

6.16

Improved SERIN performance to accept 115.2 Kb streams. There is still a 30 uSec startup for SERIN, and RXD() has better performance as long as the pin is not changed.

6.17

Added HWPWM for 8 channels, though there is a bug that times for channel 7 and 8 are swapped. Added send of + character after Flash has been written, this was done as XON/XOFF was overrunning, and this is used to handshake with BASICtools.

6.18

Fixed HWPWM swap of channel 7 and 8. Added gets() like support for SPIIN, SERIN, OWIN and I2CIN. Also added I2CSPEED for slower I2C devices. Corrected subtract followed by divide bug.

6.19

Added I2CSPEED to slow down I2C operations for older parts or long cables. DATA statements can contain negative numbers now. 32 bit constants on ARMmite or when using PC compiler. On ARMexpress compiler constants while still limited to 16 bits are sign extended. ARMexpress was reporting missing labels, but ARMmite was not, now fixed. Allow for multiple strings in data lists of SEROUT, I2COUT,... Corrected error reporting of strings missing a final ". DEBUGIN now accepts negative numbers. INTERRUPT keyword added. Support for ARMexpress LITE.

6.20

Support for STOP as a breakpoint.

6.21

SERIN_TIMEOUT added. Support for Wireless ARMmite. HWPWM supports duty cycles up to 40 seconds. Baud rates for SERIN/OUT 16,baud works again.

6.22

Support for ARMweb.

6.23

Refinements for ARMweb and STRSTR, STRCHR and TOUPPER string functions. SERIN, RXD was filtering ESCAPE and ctl-C characters on pin 16 (UART0). This has been corrected.

6.24

Added DIM name AS INTEGER and SUB .. ENDSUB local scope (as this is an ARMbasic.exe feature it is backward compatible to 6.17 and later firmware versions.

Firmware changes: only look for ESC/ctl-C for 500/1000 msec after reset (1000 for wireless versions). RND function added (uses an LCG algorithm). HWPWM now uses times in microseconds rather than duty-cycles.

6.25

Added FUNCTION ... END FUNCTION, BYREF and BYVAL parameters for SUB/FUNCTION. This change affects the compiler on the PC or ARMweb.

7.05

Support for old and new firmware versions (new firmware moves built-in functions into #include'd libraries).

fixes to FUNCTIONs and SUBs. Null strings ("") allowed. String constants can be used in string BYREF calls. DIM enforcement of variable declarations once used. VB style CALLs to FUNCTION/SUB, i.e. CALL keyword is optional. Access to hardware registers via * is optimized.

7.09

Firmware support and PC compiler support for interrupts (both are required).

Improved PC compiler generation of constants.

7.10

Minor fixes in PC compiler for calls to SUB/FUNCT with constant strings, flag embedded chr(0) in string expressions. Improved some error messages in PC compiler.

7.11

Support for VB style CGIIN, MAIL, UDPIN and UDPOUT for ARMweb.

7.13

Support for BAUD0 to change UART0 speed. TXD0 subroutine syntax supported.

Support for UART1 added with BAUD1, RXD1 and TXD1

Support for FREAD, WRITE to Flash.

Both PC compiler and firmware are required

7.17

reorganization for generic compiler.

7.18

fix for ARMexpress LITE AD. Inline TIMER code added. and improved constant generation.

7.20

Improved call/return. Expanded *pointer handling, and added ADDRESSOF operator.

TX FIFO enabled.

7.41

changed call/return mechanism for better performance.

8.02

added support for Cortex parts.

8.05

initial bug fixes for Cortex parts.

8.08

added error reporting when an integer operand expected but not found.

8.11

added IEEE 754 floating point, PRINTF, 7.50 version for ARM7 parts, trial versions

8.12

released versions with floating point support, handles multiple IO port in banks of 32, June 2012.

8.15

fixed clock setup for LPC1114, added support for LPC812.

8.20

enabled TX interrupts on all UARTS, fixed sector mapping for LPC175x parts, added SLEEP keyword to compiler to execute WFI instructions, changed SLEEP(x) routine in RTC.bas to IDLE(x).

8.21

fixed FREAD for SuperPRO

8.40

fix printf/sprintf bug changing width when rounding.

9.25

compiler version separated from firmware version

9.39

added ASM32 and allow 32 bit ops in ASM

9.40

added byte, halfword

9.41

checks for unreachable code

9.42

allow forward reference to SUB with no parameters

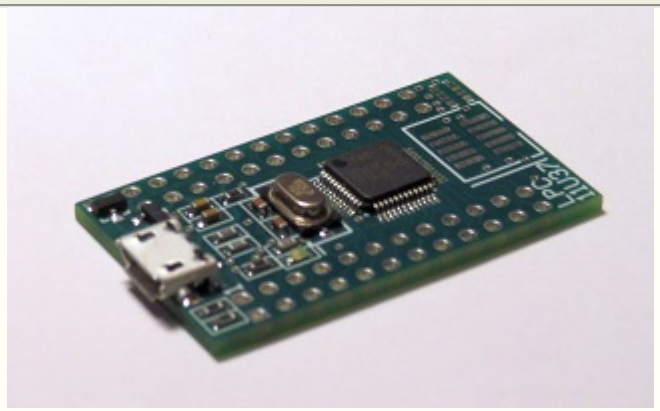
Notices

NO WARRANTY

1. THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. CORIDIUM PROVIDES THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

2. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL CORIDIUM BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

The Language



The Language

- Pre Processor
- Simple Statements
- Compound Statements
- Other Statements
- Functions
- Operators
- Data Types
- Alphabetical Keyword List

Alphabetical Keyword List

- **HIGH**

Preprocessor for BASIC

Most BASICs do not have a pre-processor. **ARMbasic** does not include one as part of the standard language, but a version of the CPP has been included as part of the utilities.

The CPP (C preprocessor) is a very powerful tool, most users use just a fraction of the features, but if you want the full story check **this document from gcc** .

As of version 5.14 of BASICtools, the type of CPU is used to generate a #define LPCxxxx that is added before any of the user source code. This can be used to write code good for any of the CPUs as is done in RTC.bas and HWPMW.bas libraries.

These are the most common directives that apply to use with **ARMbasic**: Unlike **ARMbasic** these keywords and any parameters used in them ARE CASE SENSITIVE. The pre-processor is run on the PC, so it is not available when using the built-in compiler of the ARMweb. However the compiler with preprocessor can be used to generate files that can be downloaded to the ARMweb (use the Save Intermediates check box in the Files menu of BASICtools).

#include "filename"

#include <filename>

#define

#ifdef

#ifndef

#if

#if (defined)

#else

#elif

#endif

#undef

#error

#warning

CPP operation

The CPP is a multi-step process carried out automatically by the BASICtools program. All operations are done in a temp file directory created at %temp%Coridium/temp. All files in this directory will be deleted when a File>>Load is performed by BASICtools.

It starts with your source file, and it will be copied into the %temp%/Coridium directory. When copied all comments will be stripped. All included files will be also copied into this temp directory. Then the CPP will be run on the files in that temp directory creating a __temp.bpp file that is the result of all the pre-processor operations. This __temp.bpp file will be combined with other information as __temp.bas and then compiled by ARMbasic.exe and its output is __temp.out. This __temp.out file is a modified Intel hex format of the code generated by the source BASIC program. __temp.out will be downloaded to the ARM.

In addition __temp.bat and __errors.tmp files will be created. __temp.bat is a batch file used in the compilation process. Errors from the compile or any of its steps will be contained in __errors.tmp.

#define

Syntax

#define *IDname*

or

#define *IDname* *expression*

or

#define *IDname*(param,...) *expression* (param,...)

Description

This statement directs the pre-processor to replace the word *IDname* with *expression* in the file before compiling. This replacement can also contain parameters that will be replaced in corresponding positions as defined in *expression*.

It may also be used to control #ifdef

Example

```
#define COMPILETHIS


#ifdef COMPILETHIS

#endif
```

Differences from other BASICs

- similar function in PBASIC
- no equivalent in Visual BASIC, but may be done with C-pre-processor

See also

- **#ifdef**

#else #elif #endif

Syntax

#if *expression*

#else

#endif

or

#if (defined *name*)

#elif *expression*

#endif

or

#if (defined *name*)

#endif

Description

These statements complete or extend #if statements.

These statements may nest. And unlimited #elif are allowed.

Example

```
#if someNAME == 3

#elif someNAME == 4


#elif (defined COMPILETHIS)  || (defined COMPILETHAT)

#else

#endif
```

Differences from other BASICs

- only #else available in PBASIC
- no equivalent in Visual BASIC, but may be done with C-pre-processor

See also

- **#define**

#ifdef

Syntax

#ifdef *IDname*

#endif

or

#ifndef *IDname*

#endif

Description

This statement directs the pre-processor to copy the contents of file between the ifdef and the endif into the source to be compiled by the BASIC compiler, if *IDname* is defined . #ifndef copies the statements if *IDname* has not been defined.

These statements may nest.

Example

```
#define COMPILETHIS

#ifdef COMPILETHIS

  ' ... will now be included

#endif
```

Differences from other BASICs

- no equivalent in PBASIC
- no equivalent in Visual BASIC, but may be done with C-pre-processor

See also

- **#define**

#if

Syntax

#if *expression*

#endif

or

#if (defined *name*)

#endif

Description

This statement directs the pre-processor to copy the contents of file between the if and the endif into the source to be compiled by the BASIC compiler, if *expression*is TRUE (non-zero).

#if (defined name) is equivalent to #ifdef, and can be used for more complex defines.

These statements may nest.

Example

```
#if someNAME == 3

#endif


#if (defined COMPILETHIS) || (defined COMPILETHAT )

#endif
```

Differences from other BASICs

- similar function in PBASIC
- no equivalent in Visual BASIC, but may be done with C-pre-processor

See also

- **#define**

#include

Syntax

#include " *filename*"

#include <filename>

Description

This statement directs the pre-processor to copy the contents of *filename* into the source to be compiled by the BASIC compiler. After that file is copied, the compilation continues on with the next statement in the original program.

These statements may nest, as one file can include another which can include another...

When filename is enclosed in " ", the directory of the main BASIC program is searched. The filename may contain a relative path, and remember that path is always relative to the directory of the main BASIC program.

When the filename is enclosed in < >, the Program Files/Coridium/BASIClib directory is searched.

Normally #include statements are near the beginning of the BASIC program so that FUNCTIONs and SUBs can be defined before their first use. When this is the case a MAIN: should be used so that code does not try to execute the FUNCTION or SUB code inline.

Example

```
' include the module that controls VDRIVE

#include "Vdrive.bas"

' compiler picks up here
```

Differences from other BASICs

- no equivalent in PBASIC
- no equivalent in Visual BASIC, but may be done with C-pre-processor

See also

- **#ifdef**
- **MAIN:**

#undef

Syntax

#undef *IDname*

Description

This statement directs the pre-processor to forget the word *IDname* for pre-processing.

So #ifdef *IDname* will now evaluate to FALSE.

Example

```
#define COMPILETHIS
  '...
#ifdef COMPILETHIS
  '... will now be included
#endif

#undef COMPILETHIS
#ifdef COMPILETHIS
  '... will now not be included
#endif
```

Differences from other BASICs

- no equivalent in PBASIC
- no equivalent in Visual BASIC, but may be done with C-pre-processor

See also

- **#ifdef**

#warning #error

Syntax

#warning *Message*

or

#error *ErrorMessage*

Description

#warning will issue a warning message visible in the progress window of BASICtools.

#error will generate a compiler error and prevent the BASIC program from being downloaded.

Example

```
#define COMPILETHIS

#ifdef COMPILETHIS
   '...
#else
  #error  No code available for this option
#endif
```

Differences from other BASICs

- similar function in PBASIC
- no equivalent in Visual BASIC, but may be done with C-pre-processor

See also

- **#ifdef**

Simple Statements

Statements are limited to a single line in ARMbasic, and are limited to 256 characters per line.

assignment

Syntax

lvalue = expression

Description

This statement changes the value of the variable, string, array element or hardware register *lvalue* with that of *expression*.

Example

```
DIM AB(10) AS STRING

AB = "this is a string"
AB(8) = "1"   ' makes it    this is 1 string

IO(0) = 1       'set pin 0 to be high

x = 100+(x*z-3)
```

Differences from other BASICs

- none from PBASIC
- some BASICs allow the archaic LET to precede this statement

See also

- **Mathematical Functions**

GOSUB CALL

Syntax

GOSUB label

GOSUB (expression)

[GOSUB] function/sub

or

CALL label

[CALL] function/sub

CALL (expression)

Description

GOSUB is supported for backward compatibility, now **FUNCTIONs and SUBs** would be the preferred method.

Execution jumps to a subroutine marked by line label. Always use **RETURN** to exit a GOSUB, execution will continue on next statement after GOSUB.

label may be defined as any valid identifier followed by a colon, and can be defined before or after the GOSUB. Do not include the colon in the GOSUB.

GOSUB or CALL preceeding a FUNCTION or SUB is optional, and is allowed for backward compatibility with other BASICs. When GOSUB/CALL a FUNCTION the return value is discarded.

CALL (expression) will compute the expression and then call the resulting address.

Forward declarations-

BASIC does not have a mechanism for defining SUB or FUNCTION ahead of their actual definition. But for callback from an interrupt, a mechanism for calling a SUB has been added. The SUB can not accept any parameters, and can not be a FUNCTION returning a value.

Example

```
'  original BASIC syntax

MAIN:
GOSUB message
END

message:
PRINT "Welcome!
return
```

```
'  example of callback

callbacks:                'a label that is never called, but prevents unreachable code warning
   GOSUB SUB11       ' declare SUB11 as a SUB that will be called before it is declared
```

```
SUB sample
   GOSUB SUB11       ' call SUB11 before it is defined
ENDSUB

'...

SUB SUB11            ' eventually define SUB11
   PRINT 1234
ENDSUB

'...
```

Differences from other BASICs

- CALL used in Visual BASIC and version 7.00 makes the CALL optional for FUNCTION/SUB like VB
- GOSUB used in PBASIC

See also

- **GOTO**
- **RETURN**

comment

Syntax

' comment

Description

Comments in ARMbasic can follow a single quote character. All text after the single quote to the end of the line is ignored by the compiler.

Example

```
AB = "this is a string"       '  double quotes are for strings, including single character strings
x = x + 1                     ' this is a comment for the instruction to increment x

'  this entire line is a comment
```

Differences from other BASICs

- none from VisualBASIC
- none from PBASIC
- most early BASICs used the REM statement, which **ARMbasic** does not support

See also

- **Simple Statements**

END

Syntax

END

Description

END is used to terminate the program.

When the **ARMbasic** is used in a control application, the END would not normally be encountered. As most control applications would be a loop, as when a program ends it would require the user to restart or a reboot.

There is an implied END added to any program. When a program ENDs, the last state of variables, IOs and IO controls is maintained. If a program is then RUN again those states will probably be different than running the program by hitting RESET. RESET sets all variables to 0, and all IOs to inputs. When a program is restarted from RUN, the variables will be set to 0, but the last IO state will be maintained.

Example

```
PRINT "An unrecoverable error has occurred "
END
```

Differences from other BASICs

- none

See also

- **STOP**
- **SLEEP**

EXIT

Syntax

EXIT

Description

Leaves a code block such as a **DO...LOOP**, **FOR...NEXT**, or a **WHILE...LOOP** block.

Example

```
'e.g. the print command will not be seen

DO
      EXIT  ' Exit the DO...LOOP
      PRINT "i will never be shown"
LOOP
```

Differences from other BASICs

- None

See also

- **DO**
- **FOR**
- **WHILE**

GOSUB CALL

Syntax

GOSUB label

GOSUB (expression)

[GOSUB] function/sub

or

CALL label

[CALL] function/sub

CALL (expression)

Description

GOSUB is supported for backward compatibility, now **FUNCTIONs and SUBs** would be the preferred method.

Execution jumps to a subroutine marked by line label. Always use **RETURN** to exit a GOSUB, execution will continue on next statement after GOSUB.

label may be defined as any valid identifier followed by a colon, and can be defined before or after the GOSUB. Do not include the colon in the GOSUB.

GOSUB or CALL preceeding a FUNCTION or SUB is optional, and is allowed for backward compatibility with other BASICs. When GOSUB/CALL a FUNCTION the return value is discarded.

CALL (expression) will compute the expression and then call the resulting address.

Forward declarations-

BASIC does not have a mechanism for defining SUB or FUNCTION ahead of their actual definition. But for callback from an interrupt, a mechanism for calling a SUB has been added. The SUB can not accept any parameters, and can not be a FUNCTION returning a value.

Example

```
'  original BASIC syntax

MAIN:
GOSUB message
END

message:
PRINT "Welcome!
return
```

```
'  example of callback

callbacks:                'a label that is never called, but prevents unreachable code warning
   GOSUB SUB11       ' declare SUB11 as a SUB that will be called before it is declared
```

```
SUB sample
   GOSUB SUB11       ' call SUB11 before it is defined
ENDSUB

'...

SUB SUB11            ' eventually define SUB11
   PRINT 1234
ENDSUB

'...
```

Differences from other BASICs

- CALL used in Visual BASIC and version 7.00 makes the CALL optional for FUNCTION/SUB like VB
- GOSUB used in PBASIC

See also

- **GOTO**
- **RETURN**

GOTO

Syntax

GOTO label

Description

Transfers code execution to a label.

GOTO's should be avoided and replaced with the more modern structures such as **DO...LOOP**, **FOR...NEXT**, and **WHILE...LOOP**.

But occasionally GOTO can be the best way to change the program flow.

Example

```
GOTO message


message:
PRINT "Welcome!
```

Differences from other BASICs

- none from Visual BASIC
- none from PBASIC

See also

- **GOSUB**

DEBUGIN *variable*

Syntax

DEBUGIN *variable* | *string*

Description

Normally the programs running on an ARM are running stand-alone and without direct human input. However, during the bring up phase a programmer may want to try different values. So a simplified replacement of the normal BASIC INPUT has been added, called DEBUGIN.

INPUT is used to control the direction of one of the IO pins.

DEBUGIN has a limited edit capacity: it allows to erase characters using the backspace key. If a better user interface is needed, a custom input routine should be used.

DEBUGIN may also read a string from the control serial port.

DEBUGIN does NOT support SINGLE variables.

On the ARMweb, this command is available only on the debug USB port.

Example

```
while 1
  debugin a
  print a*10
loop
```

Differences from other BASICs

- DEBUGIN can take numbers in hexadecimal, binary or decimal format by using &Hhex, $hex %bin
- PBASIC has a more complex DEBUGIN
- other BASICs calls this function INPUT

PRINT

Syntax

PRINT [*expressionlist*] [(, | ;)] ...

Description

Prints *expressionlist* to screen.

Expressionlist can be constant string, constant numbers, variables, string variables or expressions consisting of variables and numbers. Separated by either , or ;

Using a comma (,) as separator or in the end of the *expressionlist* will place the cursor in the next column (every 5 characters), using a semi-colon (;) won't move the cursor. If neither of them are used in the end of the *expressionlist*, then a new-line will be printed.

PRINT statements send data out the serial port. There is a 256 byte FIFO in the UART driver, once that is filled BASIC will wait for space to be available. PRINT requires interrupts to be enabled (the default case) as this buffer is emptied on the TX register available interrupt.

expressionlist will be evaluated as INTEGER or SINGLE depending on the type of the first variable or constant in each expression.

INTEGER expressions will display as -2147483648 to 2147483647

SINGLE expressions will display 0.0 and the range from 0.100000 through 9999999. and the corresponding negative range. Numbers outside that range will display as 1.000000E-39 to 9.999999E+38. Numbers out of range of valid values will display as Inf (infinity) or Nan (not a number). Numbers are also rounded before printing to the 7th significant digit.

Example

```
DIM AB(10) AS STRING
" new-line"Hello World!"" no new-line
PRINT "Hello";AB; "!";
PRINT

" column separator
PRINT "Hello!", "World!"

PRINT "3+4 =",3+4

y=4321
x=1234
PRINT "sum=",x+y
```

Differences from other BASICs

- none from Visual BASIC
- PBASIC uses DEBUGIN and a non-standard syntax

See also

- DEBUGIN **the opposite function that receives user input**

RETURN

Syntax

RETURN

inside function-
RETURN *expression* | *string-expression*

Description

RETURN is used to return control back to the statement immediately following a previous **GOSUB** call. When used in combination with GOSUB, A GOSUB call must always have a matching RETURN statement, to avoid stack

If the RETURN is inside a function, an integer, single or string expression is expected.

RETURN will exit a FUNCTION or SUB even when inside a component statement such as WHILE, FOR, SELECT ...

If a RETURN is executed without a corresponding GOSUB or CALL, a Prefetch Abort error will stop your program.

Example

```
PRINT "Let's Gosub!"
GOSUB MyGosub
PRINT "Back from Gosub!"
END

MyGosub:
PRINT "In Gosub!"
RETURN
```

Differences from other BASICs

- a subset of the RETURN of Visual BASIC
- none from PBASIC

See also

- **GOSUB**.

Compound Statements

Compound Statements

- DO...LOOP
- FOR...NEXT
- IF...THEN
- SELECT CASE
- WHILE...LOOP
- EXIT

DO...LOOP

Syntax

```
[DO] WHILE condition
   [statement block]
LOOP

DO
   [statement block]
[LOOP] UNTIL  condition

DO
   [statement block]
LOOP
```

Description

Repeats a block of statements until/while the *condition* is met. The three above syntaxes show the different types. The DO .. LOOP without a WHILE or UNTIL will loop forever, unless an EXIT statement is executed.

An EXIT is the proper way to leave a DO...LOOP, you can also use a GOTO or RETURN statement.

Example

```
'This will continue to print "hello" on the screen until the condition (a > 10) is met.

a = 1
DO
      PRINT "hello"
       a += 1
LOOP UNTIL a > 10
```

Differences from other BASICs

- Some BASICs allow interchangeability of UNTIL as the equivalent of NOT WHILE

See also

- **EXIT**
- **FOR...NEXT**
- **WHILE...LOOP**

FOR...NEXT

Syntax

FOR *counter* = *startvalue* TO *endvalue* [STEP *stepvalue*]
  [*statement block*]
NEXT [*counter*]

FOR *counter* = *startvalue* DOWNTO *endvalue* [STEP *stepvalue*]
  [*statement block*]
NEXT [*counter*]

Description

A FOR [...] NEXT loop initializes *counter* to *startvalue*, then executes the *statement block*'s, incrementing *counter* by *stepvalue* until it reaches *endvalue*. If *stepvalue* is not explicitly given it will set to 1.

If the DOWNTO is used, then the counter is decremented by the stepvalue or 1 if none is specified.

An EXIT is the proper way to leave a FOR...NEXT, you can also use a GOTO or RETURN statement.

Example

```
PRINT "counting from 3 to 0, with a step of -1"
FOR i = 3 DOWNTO 0 STEP 1
      PRINT "i is "; i
NEXT i
```

Differences from other BASICs

- PBASIC does not use DOWNTO, and must specify a negative step
- PBASIC does not allow the variable in the NEXT statement (while this is not necessary it is good coding practice)

See also

- **STEP**
- **NEXT**
- **DO...LOOP**
- **EXIT**

DOWNTO

Syntax

FOR *counter* = *startvalue* DOWNTO *endvalue* [STEP *stepvalue*]
[*statement block*]
NEXT [*counter*]

Description

This has been added for FOR loops that count down, which are ambiguous when *startvalue* or *endvalue* are variables.

Example

```
PRINT "counting from 3 to 0, with a step of -1"
FOR i = 3 DOWNTO 0 STEP 1
      PRINT "i is "; i
NEXT i
```

STEP

Syntax

FOR *iterator* = *initial_value* TO *end_value* STEP *increment*

Description

In a **FOR** statement, STEP specifies the increment of the loop iterator with each loop.
If no STEP value is specified in the FOR loop the default of 1 is used. The STEP value should always be positive, for DOWNTO it is subtracted from the index.

Example

```
FOR I=1 TO 10 STEP 2
NEXT

FOR I=120 DOWNTO 10 STEP 20
NEXT
```

See also

- **FOR**

TO

Syntax

FOR *iterator intial_value* TO *ending_value*
...
NEXT [*iterator*]

SELECT *case_comparison_value*
CASE *lower_bound* TO *upper_bound*
...
END SELECT

Description

The TO keyword is used to define a certain numerical range. This keyword is valid only if used with **FOR ... NEXT** and **SELECT / CASE** .

In the first syntax, the TO keyword defines the initial value of the iterator in a FOR statement, and the ending value.

In the second syntax, the TO keyword defines lower and upper bounds for CASE comparisons.

Example

```
" this program uses bound variables along with the TO keyword to create an array, store random

FOR it = minimum_temp_count TO maximum_temp_count

  " display a message based on temperature using our min/max danger zone bounds
  SELECT array( it )
      CASE min_low_danger TO max_low_danger
        COLOR 11
        PRINT "Temperature" ; it ; " is in the low danger zone at" ; array( it ) ; " degrees!"
      CASE min_medium_danger TO max_medium_danger
        COLOR 14
        PRINT "Temperature" ; it ; " is in the medium danger zone at" ; array( it ) ; " degrees!"
      CASE min_high_danger TO max_high_danger
        COLOR 12
        PRINT "Temperature" ; it ; " is in the high danger zone at" ; array( it ) ; " degrees!"
       CASE ELSE
        COLOR 3
        PRINT "Temperature" ; it ; " is safe at" ; array( it ) ; " degrees."
  END SELECT

NEXT it

SLEEP
```

Differences from other BASICs

- none

See also

- **FOR...NEXT**
- **SELECT CASE**

IF...THEN

Syntax

IF *expression* THEN *statement(s)* [ELSE *statement(s)*]

IF *expression* [THEN]
statement(s)
[ELSEIF *expression* [THEN]
statement(s)]
[ELSE
statement(s)]
ENDIF

Description

IF...THEN is a way to make decisions. It is a mechanism to execute code only if a condition is true, and can provide alternative code to execute based on more conditions.

The syntax allows single line IF..THEN, or multi-line versions that end with ENDIF.

The single line version only allows simple statements. To use nested IFs the multi-line version must be used.

Version 7.00 allows ENDIF or END IF

Example

```
'e.g. here is a simple "guess the number" game using if...then for a decision.

PRINT "guess the number between 0 and 10"

DO 'Start a loop
      PRINT "guess"
      DEBUGIN y                    'Input a number from the user
      IF x = y THEN
            PRINT "right!" 'He/she guessed the right number!
             EXIT
      ELSEIF y > 10 THEN 'The number is higher then 10
            PRINT "The number cant be greater then 10! Use the force!"
      ELSEIF x > y THEN
            PRINT "too low" 'The users guess is to low
      ELSEIF x < y THEN
            PRINT "too high" 'The users guess is to high
      ENDIF
LOOP 'Go back to the start of the loop
```

Differences from other BASICS

- none

See also

- **DO...LOOP**
- **SELECT CASE**

SELECT [CASE]

Syntax

SELECT [CASE] *expression*
[CASE *expressionlist*]
    [*statements*]
[CASE ELSE]
    [*statements*]
ENDSELECT

Description

Select case executes specific code depending on the value of an expression. If the expression matches the first case then it's code is executed otherwise the next cases are compared and if one case matches then its code is executed. If no cases are matched and there is a 'case else' on the end then it will be executed, otherwise the whole select case block will be skipped.

Syntax of an expression list:
expression [{TO *expression* | *relational operator expression*}][, ...]

example of expression lists:

```
CASE "A"                    ' the "A" is equivalent to &H41,   multi-character strings can not be used in CASE statements
CASE 5 TO 10
CASE > "e"
CASE 1, 3 TO 10
CASE 1, 3, 5, 7, 9
```

The SELECT expression must be an INTEGER, and the CASE expressions must also be INTEGER.

You can exit a SELECT/ CASE with a GOTO or RETURN statement.

Example

```
PRINT "Choose a number between 1 and 10: "
DEBUGIN choice
SELECT choice
CASE 1
      PRINT "number is 1"
CASE 2
      PRINT "number is 2"
CASE 3, 4
      PRINT "number is 3 or 4"
CASE 5 TO 10
       PRINT "number is in the range of 5 to 10"
CASE <= 20
       PRINT "number is in the range of 11 to 20"
CASE ELSE
      PRINT "number is outside the 1-20 range"
ENDSELECT
```

Differences from other BASICs

- SELECT CASE is used in Visual BASIC
- SELECT is used in PBASIC
- either is allowed in **ARMbasic**
- Visual BASIC uses an optional IS before relational operators

- ENDSELECT is used to terminate the SELECT in both **ARMbasic** and PBASIC
- END SELECT (separate words) are used in Visual BASIC and are allowed in **ARMbasic**

See also

- **IF...THEN**

TO

Syntax

FOR *iterator intial_value* TO *ending_value*
...
NEXT [*iterator*]

SELECT *case_comparison_value*
CASE *lower_bound* TO *upper_bound*
...
END SELECT

Description

The TO keyword is used to define a certain numerical range. This keyword is valid only if used with **FOR ... NEXT** and **SELECT / CASE** .

In the first syntax, the TO keyword defines the initial value of the iterator in a FOR statement, and the ending value.

In the second syntax, the TO keyword defines lower and upper bounds for CASE comparisons.

Example

```
" this program uses bound variables along with the TO keyword to create an array, store random

FOR it = minimum_temp_count TO maximum_temp_count

  " display a message based on temperature using our min/max danger zone bounds
  SELECT array( it )
      CASE min_low_danger TO max_low_danger
        COLOR 11
        PRINT "Temperature" ; it ; " is in the low danger zone at" ; array( it ) ; " degrees!"
      CASE min_medium_danger TO max_medium_danger
        COLOR 14
        PRINT "Temperature" ; it ; " is in the medium danger zone at" ; array( it ) ; " degrees!"
      CASE min_high_danger TO max_high_danger
        COLOR 12
        PRINT "Temperature" ; it ; " is in the high danger zone at" ; array( it ) ; " degrees!"
       CASE ELSE
        COLOR 3
        PRINT "Temperature" ; it ; " is safe at" ; array( it ) ; " degrees."
  END SELECT

NEXT it

SLEEP
```

Differences from other BASICs

- none

See also

- **FOR...NEXT**
- **SELECT CASE**

WHILE...LOOP

Syntax

[DO] WHILE*condition*
    [*statements*]
LOOP

Description

WHILE [...] LOOP will repeat the *statements* between WHILE and LOOP, while the *condition* is true.

If the *condition* isn't true when the WHILE statement begins, none of the *statements* will be run.

The DO is optional in ARMbasic.

WHILE loops have the lowest overhead of all looping constructs, so they are faster than FOR or DO loops

An EXIT is the proper way to leave a WHILE...LOOP, you can also use a GOTO or RETURN statement.

Example

```
' a good way to time operations

x = 1000000

while x
       x -= 1   ' decrement x
       y = y * z    ' do something to measure here -- like multiply for instance
loop

-------------------------------

  ... Finished in 181 msec  (on PROplus)

so with loop overhead of 121 msec, that's 60 nsec to multiply 2 numbers
```

Differences from other BASICs

- Visual BASIC uses the syntax DO WHILE ... LOOP, which is allowed by **ARMbasic**
- PBASIC also requires the DO
- Some BASICs use WHILE ... WEND

See also

- **DO...LOOP**
- **EXIT**

Other Statements

__ASM__
CLEAR
CONST
DIM
END
label:
MAIN
ON
RUN
STOP

__ASM__

Syntax

__ASM__ (*expression*)

__ASM32__ (expression)

Description

__ASM__ tells the compiler to compile the *expression* into the program

expression is a number, and represents the 16 bit opcode that is added to the code. If the value of expression is a 32 bit constant (>= &H10000), then 2 16 bit values are added to the code. This handles long jumps

We currently don't have a symbollic assembler and the user has to generate the *expression* that represents an ARM Thumb opcode. Details on the **Thumb** opcodes can be found **here** . Recently we have created a macro library that covers ARM instructions we have used or think are commonly used. It will be part of the release, or can be found at the **forum** .

__ASM32__ will evaluate as a 32 bit word and is aligned to a word boundary and added to the code. The alignment is done by inserting a NOP if necessary, but the intention of this is for PC relative constants which must be word aligned. While it can be used for general assembly opcodes the simple __ASM__ is better.

To save some needed assembly statements, __ASM__ can now be used as an rvalue. This actually does nothing and the result is whatever is in R7

A number of ARM instructions have been encoded by the #include file ASMinlineBASIC.bas

Example

```
#include "ASMinlineBASIC.bas"      ' get macros for most ARM instructions

...

 ASM_LDR_SPI(7,60)                ' r7 now has PC for routine that INTERRUPT SUB interrupted
 savePC =__ASM__                   ' save it in savePC
...
 ASM_LDR_PC(6,2)                  ' load R6 with PC relative constant &H12345678
 __ASM32__(&H12345678)
```

under the hood --

For the Cortex parts, the BASIC compiler saves the value computed in the last statement in R7. These registers uses are subject to change.

R4 is used in index operations and pointers

R6 is normally the second operand in expression evaluation, and is combined with R7 to produce the last value of an expression

R5 is used in some string operations

R9 can be used as a temporary variable in complex expressions that require intermediate values to be saved on the stack.

Differences from other BASICs

- no equivalent in Visual BASIC
- no equivalent in PBASIC

See also

- **EXIT**

CONST

Syntax

CONST *symbolname* = *value*

CONST *arrayname* = { value [, valuelist] }

valuelist = value [, value] ' up to 60 values per line

value = integernumber | floating*point*number

CONST *bytearrayname* AS BYTE = { value [, valuelist] }

CONST *stringname* = "any string"

Description

Declares compiler-time constant symbols that can be an integer.

More complex CONST can now be handled by **#define -- see pre-processor**

under the hood-

Single constants do not take up any program space as they are maintained by the PC Compiler.

Constants can be 32 bit values using the PC ARMbasic compiler and can be integer or floating point numbers. but constants are limited to 16bit values for the on chip ARMweb compiler. A floating point number is any number that contains a decimal point, without a decimal point numbers are assumed to be integers. If you want the floating point representation of an integer an example would be 123. , -455. , 0. , 122.0 or 0.0 .

Array constants can be 8 or 32 bit values and are stored in high memory along with constant strings and do take up Flash space available on the ARM. You can use the constants in your program as if it were an array.

Constant arrays can span more than one line, the compiler, it will keep reading lines until the final } is found. This feature was added in version 8.16c of the compiler.

Constants when used are not type checked or type converted, so if you try to assign an integer constant to a single variable, you will get odd looking results.

Example

```
CONST REPS = 20

CONST repeatedstring = "something used more than once"

DIM X as SINGLE
DIM I

CONST  factorials = { 1., 1., 2., 6., 24. , 120. , 720., 5040. , 40320. , 362880. , 3628800. , 39916800. ,
 479001600. ,
                    6227020800. , 87178291200. , 1307674368000. , 20922789888000.
, 355687428096000. ,
                    6402373705728000. ,121645100408832000. , 2432902008176640000. ,
5109094217170940000. ,
                    112400072777761000000. , 2585201673888500000000. ,
```

```
620448401733239000000000. ,
                    155112100433310000000000000. , 403291461126606000000000000. ,
10888869450418400000000000000. ,
                    304888344611714000000000000000. , 884176199373970000000000000000000. ,
26525285981219100000000000000000000.,
                    82228386541779200000000000000000000. ,
26313083693369400000000000000000000000. ,
                    86833176188118900000000000000000000000.
, 29523279903960400000000000000000000000000. }

PRINT repeatedstring

FOR I=0 to  REPS
   X = factorials (i)
   PRINT I,X
NEXT I

PRINT repeatedstring


-------------------------------------------------

something used more than once
0       1.
1       1.
2       2.
3       6.
4       24.
5       120.
6       720.
7       5040.
8       40320.
9       362880.
10       3628800.
11       39916800.
12       479001600.
13       6227020800.
14       87178291200.
15      1307674368000.
16      20922789888000.
17      355687428096000.
18      6402373705728000.
19      121645100408832000.
20      2432902008176640000 1
something used more than once
```

Differences from other BASICs

- Visual BASIC allows more complex CONST declarations
- syntax in PBASIC is *symbolname* CON *value*

See also

- **Preprocessor**

DIM

Syntax

Declaring Integers:
DIM *symbolname* [AS INTEGER] ' INTEGER type is assumed if no type is specified

Declaring Floating point (requires firmware 7.50 or 8.11)
DIM *symbolname, symbolname2, symbolname3* AS SINGLE

Declaring Arrays:
DIM *symbolname* (*max_element*)
DIM *symbolname* (*max_element*) AS BYTE

Declaring Strings:
DIM *symbolname$* (*max_element*)
DIM *symbolname* (*max_element*) AS STRING

Description

Declares a named variable and allocates memory to accommodate it. Though **ARMbasic** does not require the declaration of integer variables, DIM is used to declare floating-point singles or arrays, arrays of integers, or strings (arrays of bytes). The size is the *max_element* in the array plus 1. This allows indexing from 0 to *max_element* .

For backward compatibility strings may have the last character the dollar sign $. Multiple variables of the same type may be defined on a single DIM statement, that includes both individual variables and arrays of the same type.

Memory for simple variables is allocated from the start of a heap, and strings or arrays are allocated from the top or end of the heap. Strings are packed as bytes and always word aligned, you must allow enough space to accommodate the expected maximum size of the string plus 1 byte for a termination (0) character. String operators rely on the terminator. Strings may not exceed 255 characters.

Simple INTEGER variables will be automatically declared on first use, unless you use DIM *symbolname* AS INTEGER. At which point all subsequent integers must be declared using a DIM. Classic BASIC declared all variables automatically, but VB requires them all to be declared with a DIM statement. The advantage in VB is that if you mistype a variable name it will be flagged as undefined rather than becoming a newly automatically declared INTEGER variable. As the authors (read me) came from classic BASIC, the automatic declaration is default, but if you don't like that just use a DIM to declare an INTEGER variable early in the program. That statement should be after the #includes as many libraries use auto-declaration.

Floating point SINGLE use a single word of storage and are allocated from the start of the heap.

SUB and FUNCTION procedures also use DIM between SUB/FUNCTION .. ENDSUB/ENDFUNCTION. Those variables will be local to the procedure. Using DIM here does not change whether all subsequent integers must be declared using a DIM or not. In other words the state whether DIM is required is saved upon entering a SUB procedure and is restored at the ENDSUB.

Arrays of 8bit values can be declared AS BYTE. BYTE arrays are not limited to 255 characters. However, string operations and functions ARE still limited to 255 bytes.

Example

```
DIM a$ (10)
DIM b (20)  AS STRING
```

```
DIM c (300)  AS BYTE


a$  = "Hello World"
b  = "... from ARMbasic!"
c = a$ + b     ' you can mix STRING and BYTE arrays, but the results are limited to 255 characters
and are truncated at the first 0

print c                 ' displays Hello World... from ARMbasic
```

Differences from other BASICs

-
- Like Visual BASIC the first element uses an offset of 0, but also memory is allocated for 0, 1 to *size* elements. This is backward compatible with earlier BASICs which indexed from 1 to *size* .
- PBASIC uses the syntax *symbolname* VAR WORD | BYTE [(*size*)]

See also

label:

Syntax

name :

Description

GOTO and GOSUB go to a *label*. Somewhere in the code is that target *label*. A label can be any valid variable *name* followed by a colon : . A *label* can be the only element on a line.

MAIN: is a special case of label that will start execution of the program at somewhere other than the first line of code.

Example

```
'...
GOSUB sayHello

'...
END

'...

sayHello:
PRINT "Hello"
RETURN
```

Differences from other BASICs

- none from Visual BASIC
- none from PBASIC

See also

- **MAIN**

MAIN

Syntax

MAIN:

Description

Normally an **ARMbasic** program will start at the first statement in the BASIC source. This can be changed by having a MAIN: somewhere else in the program. When a MAIN: does exist, the program will begin at this point.

MAIN: is useful for programs that use FUNCTIONs or SUBs and have those FUNCTIONs or SUBs at the beginning of the source. This also includes FUNCTIONs or SUBs that are #include'd in the source.

Statements before the MAIN will not be executed, unless they are called or GOTO'd. But those statements can be used to define variables or SUB or FUNCTION.

A MAIN is not required in a program, but the compiler assumes that there will be one. To get around that you can have a label at the start of the program and execution will begin there.

Any inline code before MAIN will be flagged by the compiler as **UNREACHABLE** . If you do not have a MAIN and want to execute code inline, place a label (any name OK) at the start of your code.

Example

```
SUB SUB1:
  PRINT "Hello from sub1"
END SUB

MAIN:
GOSUB SUB1
END
```

```
'or another example:

X = 1234     '  this defines X as an INTEGER variable without using a DIM

MAIN:

PRINT x   ' will print 0, because while X is defined the assignment of 1234 never happened
```

Differences from other BASICs

- none from Visual BASIC
- none from PBASIC

See also

- **EXIT**

__MAP__

Syntax

__MAP__ [CODE const_expression] [CONST const_expression] [DATA const_expression] [STRING const_expression]

Description

__MAP__ tells the compiler to reserve space, by advancing the 4 pointers. In all cases it must be a constant expression. There is no error checking, so the user is responsible for the validity of the values.

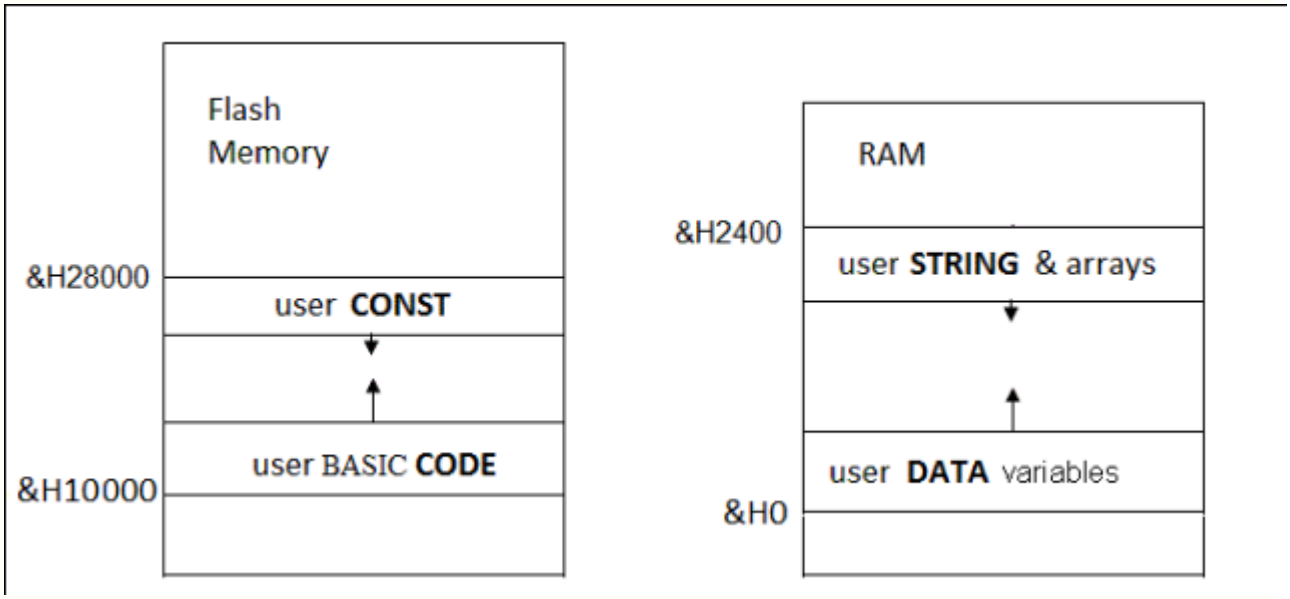
All of these values are byte values. For CODE sets the compiler program counter. CONST sets the pointer for constant strings and CONST arrays, it counts down from the end of Flash memory. Both of these addresses are byte addresses and are absolute.

DATA sets the pointer for INTEGER and SINGLE variables. STRING sets the pointer for arrays of BYTE, INTEGER or FLOAT and counts down from the end of the variable space. These are byte addresses, but are relative to the space that BASIC allocates for variables.

When used with no parameters, the compiler will report where the various pointers are left following the compile.

This allows you to reserve space for **Flash Write** , or for implimenting program overlays.

Example



The values for the above map represent are based on the SuperPRO with 8.25 firmware.

The first program writes code to the first Flash sector then calls code in the second Flash sector

```
share_x = 22
share_y = 33

print "start the program and call overlay"
```

```
gosub start1
print "we are back"

print share_x

end ' if you don't have an END here or a loop the program will drop through into unprogrammed memory

__MAP__ CODE &H11000 CONST &H27000

start1:

print "in the overlay"

return
```

This program when run shows

```
start the program and call overlay
in the overlay
we are back
22
```

Now overlay the second Flash sector with this program

```
DIM share_x          ' to access global variables in the original program, they must be declared in the same order
DIM share_y

    ' if you declare anything that would produce code before the __MAP__ you will overlay the 1st sector -- which you don't want to do

__MAP__ CODE &H11000 CONST &H27000

start1:

'print "THIS IS A NEW OVERLAY"

share_x = share_y

return
```

Now when run it shows, and it shows the access to share_x was changed

```
start the program and call overlay
we are back
33
```

Differences from other BASICs

- no equivalent in Visual BASIC
- no equivalent in PBASIC

See also

- **EXIT**

ON (only on ARM7 parts with integer BASIC firmware)

ON is NOT used with BASICchip, PROstart, PROplus and SuperPRO see INTERRUPT SUB

ON is no longer supported on the Floating point versions of firmware, Use the INTERRUPT SUB or revert to the integer version of firmware can be found in the **Coridium Forum**.

Syntax

ON TIMER *msec label*

or

ON EINT0|EINT1|EINT2 RISE|FALL|HIGH|LOW *label*

Description

These statements will initialize interrupt service routines so that when the interrupt occurs the code at label will be executed. *Label* must have been pre-defined and can either be a SUB (without parameters) or code beginning with a *label*: and ending in a RETURN. The interrupt response time is approximately 3 usec. Other interrupts may make this time longer.

TIMER interrupts will occur every *msec* milliseconds. *msec* may be a variable or constant, expressions are not allowed. The value for *msec* must be greater than 1. If TIMER interrupts are used, then only 4 hardware PWM channels are available.

EINT0 and EINT2 are 2 pins that will interrupt when the defined event occurs. RISE and FALL are the preferred method and will generate interrupts on rising or falling edges on those 2 pins. HIGH and LOW are supported, but if the pin remains in that state interrupts will be continuously generated.

EINT1 is connected to the RTS line of the PC, and is normally high, so it can be used by a program on the PC to interrupt the ARMmite, rather than having to reset the board. EINT1 is also available on the ARMexpress modules (pin 21), and should also be kept normally high if used.

Each time the ON statement is executed the interrupt will be initialized, so it is possible to change routines within the program. Multiple interrupts can be used, but they are serviced in the order received, and each interrupt service routine will complete before the next one is handled (interrupts that occur while one is being serviced will be handled after the current interrupt is processed).

Interrupt routines should normally be short and simple. The state of the other user BASIC code will be restored after the interrupt, with the exception of **string** functions, which should **NOT** be done inside an interrupt. PRINT statements use strings, so other than a temporary debug to see if the interrupt occurs, they should not be inside an interrupt routine.

To disable the interrupt use the following #define

```
#define VICIntEnClear  *&HFFFFF014

#define TIMERoff    VICIntEnClear = &H20
#define EINT0off    VICIntEnClear = &H4000
#define EINT1off    VICIntEnClear = &H8000
#define EINT2off    VICIntEnClear = &H10000
```

ON added in version 7.09

The LPC2106 based ARMexpress supports ONLY ON LOW, due to hardware limitations.

ON is a statement that is executed, so if multiple ON statements are in a program the last statement executed will be active command.

Cortex M3 and M0 do not support ON, but use INTERRUPT SUB

Example

```
IO15up = 0              ' serves to declare IO15up

 '...
SUB IO15count
  IO15up = IO15up + 1
ENDSUB

 '...
main:

ON EINT2 RISE IO15count

IO15up = 0
while 1
  if IO15up <> lastIO15count then
    print IO15up
    lastIO15count = IO15up
  endif

 '...

loop
```

```
every20msec:
  checkIO0 = checkIO0 + (IO(0) and 1)
  IO0samples = IO0samples +1
RETURN

 '...
main:

ON TIMER 20 every20msec

 '...

PRINT "Percentage of time IO0 is HIGH =", 100*checkIO0 / IO0samples

 '...
```

Differences from other BASICs

- no equivalent in VB
- no equivalent in PBASIC

See also

- **GOTO**
- **RETURN**

STOP

Syntax

STOP

Description

Halt execution of the program.

STOP functions like a breakpoint when under control of BASICtools. When the STOP is executed the BASIC program halts execution, but allows BASICtools to dump variable values. Also in BASICtools RUN will resume execution at the statement following STOP.

You may also read and write memory locations using @ and ! For these commands, hexadecimal numbers are assumed, you should not type $ or &H.

@ hex-address [hex-number] ' will display hex-number locations starting at hex-addr, if hex-number is omitted, 8 locations are displayed.

! hex-address hex-value ' will write hex-value into the location hex-address

^ ' will continue user BASIC program execution

STOP will normally STOP a running program. But if your program causes an exception (read/write address that does not exist, illegal instruction or others), typically a FAULT will be displayed and an address where it occurred. You can look into the Code addresses to see which routine it was in.

But what if the program does not STOP. We call these runaway programs. These programs typically are flooding the serial communication channel, and this can cause issues with the driver or BASICtools Tcl program, having a hard time keeping up, In these cases, the program can usually be STOP'd by disconnecting the USB connection and reconnecting and then going into the Options Menu choose Serial Port and reselect the COM port you were using. In this case the disconnect lets Windows clear the serial driver, and Tcl to close that serial port. Re-opening the serial port will start the board up again, but will send a character to halt the user program.

Example

```
'If pin 2 is low halt the processor
IF IO(2) = 0 THEN
  PRINT "Processor Stopped"
  PRINT "Press Reset to Continue"
  STOP
ENDIF
```

Differences from other BASICs

- none from Visual BASIC
- none from PBASIC, though the breakpoint features are not supported

See also

- EXIT

Debugging

ARMbasic is an incremental compiler, meaning that you can enter a portion of a program, run it, check some variable values, enter some more code and run it again... This operates much like an interpreter, so that debugging of code can be done very quickly.

There are a number of operations that aid during the debug phase of programming an ARMexpress. These allow you to read and write memory locations which also includes the registers of the many ARM peripherals.

Debugging Functions

! - write to memory
@ - read memory
CLEAR
LIST
STOP
RUN

@ (dump memory)

Syntax

@ [expression] [length]

Description

This command will dump ARM memory starting at *expression* for optional *length* words. It is useful for debugging direct control of the ARM peripherals, or looking at memory. If *expression* is omitted and *length* was non-zero , then the next *length* of memory will be displayed.

Normally @ *expression length* will be used first, with following pages displayed by typing @ without the *expression*.

Expression and *length* are hex values without the leading &H or $..

Example

The following example displays the area of ARM memory corresponding to the PWM registers. Memory address on the left, followed by 4 or 8 words of memory displayed in hex.

```
@e0014000 10

EO014000: 00000000 00000001 04BFE6BB 0000E663 0000A516 00000000 00000000
00000000
E0014020: 00000000 00000001 04BFE6BB 0000E663 0000A516 00000000 00000000
00000000
```

The display above shows the 16 words following hex address e0014000. With the display showing addresses to the left of the : and 32bit word values to the right. Versions of firmware before 8.11 and 7.53 do not show the addresses.

Differences from other BASICs

- non-existent in Visual BASIC or PBASIC

See also

- **! set memory**

! (set memory)

Syntax

! hex-number hex-number2

Description

This command will write *hex-number2* into location *hex-number* in ARM memory. It is useful for debugging direct control of the ARM peripherals.

Expression can only be a hex value without the leading $ or &H and no spaces between the ! and the hexvalue. The ARMmite does not list the address or the ASCII values.

This function will be added in version 7.47 for ARM7 and 8.07 for Cortex parts. And also requires BASICtools 5.9 or later.

Example

The following example displays the area of ARM memory corresponding to the PWM registers. Memory address on the left, followed by 4 or 8 words of memory displayed in hex.

```
@e0014000

E0014000: 00000000 00000001 04BFE6BB 0000E663 0000A516 00000000 00000000
00000000

!e0014000 1234567

@e0014000

E0014000: 01234567 00000001 04BFE6BB 0000E663 0000A516 00000000 00000000
00000000
```

Differences from other BASICs

- non-existent function in Visual BASIC or PBASIC

See also

- **@ (dump memory)**

CLEAR

Syntax

CLEAR

Description

This is a BASICtools command that erases the screen and the buffer in PC memory. It is useful when you are typing in short programs a line at a time.

It does not affect the current BASIC program in ARM memory, to erase that program, you can hit or type clear, and then RUN a short program like

PRINT

That will replace the program in ARM memory with a short do nothing program.

It should NOT be used as a statement inside a BASIC program.

Example

Example

```
PRINT "hi there"
RUN
```

```
hi there
```

```
CLEAR
```

Differences from other BASICs

- same as many BASICs
- no equivalent in Visual BASIC
- no equivalent in PBASIC

See also

- RUN

DEBUGIN *variable*

Syntax

DEBUGIN *variable* | *string*

Description

Normally the programs running on an ARM are running stand-alone and without direct human input. However, during the bring up phase a programmer may want to try different values. So a simplified replacement of the normal BASIC INPUT has been added, called DEBUGIN.

INPUT is used to control the direction of one of the IO pins.

DEBUGIN has a limited edit capacity: it allows to erase characters using the backspace key. If a better user interface is needed, a custom input routine should be used.

DEBUGIN may also read a string from the control serial port.

DEBUGIN does NOT support SINGLE variables.

On the ARMweb, this command is available only on the debug USB port.

Example

```
while 1
  debugin a
  print a*10
loop
```

Differences from other BASICs

- DEBUGIN can take numbers in hexadecimal, binary or decimal format by using &Hhex, $hex %bin
- PBASIC has a more complex DEBUGIN
- other BASICs calls this function INPUT

LIST

Syntax

LIST

Description

When typing commands into BASICtools a line at a time, use LIST to see what was typed.

Those lines can be captured into a file for further editing either by cut and paste or opening the Edit File option in the Edit menu.

This command is not used by the BASIC compiler, so it should not be included in a file to be compiled

Example

```
for i=1 to 10
  print i
next i

  '...
```

```
LIST
for i=1 to 10
  print i
next
```

RUN

Syntax

RUN

Description

RUN will compile the program and write it into Flash Memory. Then it will execute the program which has been saved.

Now that the program is in Flash it will be executed when the board is either reset or powered on.

BASICtools can STOP a program that is being executed from Flash.

RUN is a command line function, it should NOT be included in a BASIC program. It is equivalent to the RUN button in the BASICtools. Your BASIC program will start automatically when the ARM is reset.

Example

```
PRINT "hi there"
RUN
CLEAR
```

Differences from other BASICs

- same as many BASICs
- no equivalent in Visual BASIC
- no equivalent in PBASIC, done with the editor

See also

- CLEAR

STOP

Syntax

STOP

Description

Halt execution of the program.

STOP functions like a breakpoint when under control of BASICtools. When the STOP is executed the BASIC program halts execution, but allows BASICtools to dump variable values. Also in BASICtools RUN will resume execution at the statement following STOP.

You may also read and write memory locations using @ and ! For these commands, hexadecimal numbers are assumed, you should not type $ or &H.

@ hex-address [hex-number] ' will display hex-number locations starting at hex-addr, if hex-number is omitted, 8 locations are displayed.

! hex-address hex-value ' will write hex-value into the location hex-address

^ ' will continue user BASIC program execution

STOP will normally STOP a running program. But if your program causes an exception (read/write address that does not exist, illegal instruction or others), typically a FAULT will be displayed and an address where it occurred. You can look into the Code addresses to see which routine it was in.

But what if the program does not STOP. We call these runaway programs. These programs typically are flooding the serial communication channel, and this can cause issues with the driver or BASICtools Tcl program, having a hard time keeping up, In these cases, the program can usually be STOP'd by disconnecting the USB connection and reconnecting and then going into the Options Menu choose Serial Port and reselect the COM port you were using. In this case the disconnect lets Windows clear the serial driver, and Tcl to close that serial port. Re-opening the serial port will start the board up again, but will send a character to halt the user program.

Example

```
'If pin 2 is low halt the processor
IF IO(2) = 0 THEN
  PRINT "Processor Stopped"
  PRINT "Press Reset to Continue"
  STOP
ENDIF
```

Differences from other BASICs

- none from Visual BASIC
- none from PBASIC, though the breakpoint features are not supported

See also

- EXIT

Data Abort

Prefetch Abort

Undefined Routine

Description

Data Aborts are generated when a user's BASIC program accesses non-existent memory. One way is accessing an array with an index that is larger than available RAM space. Another is using a pointer for hardware access, but with a value that does not correspond to a valid location.

Prefetch aborts indicate an attempt to access an instruction from non-existent memory. Prefetch aborts can occur when RETURNing when a sub/function had not been called.

Undefined Routine, which indicates a call or return to non-existent code. This error will occur if you RETURN when there has not been a GOSUB, the equivalent of a return stack underflow. This also may occur when interrupts are used with firmware prior to version 7.30

The number reported (in hex) is the program address where the illegal access was detected. Refer to the memory maps for more information, generally user code starts at &H3000.

FUNCTIONs and SUBroutines

Sub Programs

- FUNCTION
- SUB
- ENDFUNCTION
- ENDSUB

BYREF

Syntax

```
FUNCTION name (parameter list)  [AS INTEGER | AS STRING | AS SINGLE]
  parameter list = parameter [, parameter list]
  parameter  =  [BYVAL] paramname [AS INTEGER]
                |  [BYVAL] paramname AS SINGLE
                | [BYVAL] paramname(size) AS STRING
                |   BYREF paramname AS STRING
                |   BYREF paramname() AS STRING
                |   [BYREF] paramname() [AS INTEGER]
                |   [BYREF] paramname() AS SINGLE
```

Description

Used as a modifier in parameter declarations for FUNCTIONs or SUBs.

BYREF passes a pointer to the parameter to the FUNCTION or SUB. This allows a function to read AND write the original source parameter.

An advantage in BYREF use with STRINGs, is that extra space is not required and the STRING does not have to be copied for the FUNCTION or SUB procedure. Constant strings may be passed BYREF, but any code that attempts to modify a constant string will cause a Data Abort.

BYVAL is assumed if nothing specified for single variables or STRINGs

Arrays of INTEGERs or SINGLEs can be passed, but are always passed BYREF, so the BYREF modifier is optional.

Example

```
function toupper(a(100) as string) as string
  dim i as integer

  for i=0 to 100
    if a(i)=0 then exit
    if a(i) <= "z" and a(i) >= "a" then a(i) = a(i) - &H20
  next i
  return a
end function

main:

print toupper("asdf")                ' will print ASDF
```

Differences from other BASICs

- simplification of Visual BASIC
- no equivalent in PBASIC

See also

- FUNCTION
- SUB

BYVAL

Syntax

FUNCTION *name* (parameter list) [AS INTEGER | STRING | SINGLE]
parameter list = parameter [, parameter list]
parameter = [BYVAL] paramname [AS INTEGER]
| [BYVAL] paramname AS SINGLE
| [BYVAL] paramname(size) AS STRING

Description

Used as a modifier in parameter declarations for FUNCTIONs or SUBs.

When BYVAL is used a copy of the parameter will be used in the FUNCTION or SUB. And the FUNCTION or SUB procedure can change the copy of the parameter, BUT not the original. BYVAL is the default declaration and is assumed if the BYVAL or BYREF keyword are not used, except for arrays.

Example

```
function toupper(a(100) as string) as string
  dim i as integer

  for i=0 to 100
    if a(i)=0 then exit
    if a(i) <= "z" and a(i) >= "a" then a(i) = a(i) - &H20
  next i
  return a
end function

main:

print toupper("asdf")                ' will print ASDF
```

Differences from other BASICs

- simplification of Visual BASIC
- no equivalent in PBASIC

See also

- FUNCTION
- SUB

PARAMARRAY

Syntax

SUB | FUNCTION *name* (PARAMARRAY paramname)

Description

Used to declare a variable number of parameters in FUNCTIONs or SUBs.

When used a variable number of INTEGER parameters can be passed BYVAL to a FUNCTION or SUB and accessed as elements of an array named paramname.

1 to 15 parameters may be passed. and the number of parameters - 1 is passed in paramname(15).

Example

```
FUNCTION sumof (PARAMARRAY z)
  DIM i,sum
  sum = 0
  FOR i=0 to z(15)
    sum += z(i)
  NEXT
  return sum
ENDFUNCTION

main:

print sumof(1,2,3)           ' prints 6
print sumof(5,6,7,8,9,10)    ' prints 45
```

Differences from other BASICs

- simplification of Visual BASIC
- no equivalent in PBASIC

See also

- FUNCTION
- SUB

FUNCTION *name* (optional parameters)

Syntax

FUNCTION *name* [AS INTEGER | STRING | SINGLE] ' a FUNCTION without parameters

or

FUNCTION *name* (parameter list) [AS INTEGER | STRING | SINGLE]
parameter list = parameter [, parameter list]
| PARAMARRAY paramarray
parameter = [BYVAL] paramname [AS INTEGER]
| [BYVAL] paramname AS SINGLE
| [BYVAL] paramname(size) AS STRING
| BYREF paramname AS STRING
| BYREF paramname [AS INTEGER]
| BYREF paramname AS SINGLE

Description

FUNCTIONs are an extension of SUB that will return a value. If no type for the FUNCTION is specified, then INTEGER is assumed.

The FUNCTION .. ENDFUNCTION construct allows for a second scope of variables. Scope meaning the region in which code can see a set of labels. ARMbasic has a global scope and a local scope for any variable declared with DIM inside an FUNCTION. Local scope variables will be only accessible from within that FUNCTION procedure (the local scope).

Simple parameters are assumed to be called BYVAL if not specified. In BYVAL calls, a copy of the parameter is passed to the FUNCTION procedure. Integer or string parameters may be called BYREF which means a pointer to the integer/string is passed, and changes to that integer/string can be made by code inside the FUNCTION procedure. STRINGs may be passed either BYVAL or BYREF, but arrays must be passed BYREF.

Code labels for goto/gosub declared within the SUB procedure are also in the local scope. Call to global labels are allowed within a FUNCTION ... END FUNCTION , but that global label must be defined BEFORE the FUNCTION ... END FUNCTION .

A FUNCTION should end with a RETURN expression, as FUNCTIONs return a value, if that is not needed use a SUB.. A FUNCTION may also be called with a GOSUB, but the returned value is ignored.

FUNCTION without any parameters can be called with or without (), an empty parameter list.

Recursive calls with parameters or local variables are not supported. And ENDFUNCTION or END FUNCTION syntax are allowed.

Program structure:

FUNCTIONs should be arranged ahead of the MAIN: body of code. In many cases they will be part of #include files at the beginning of the user ARMbasic code. If FUNCTIONs are located at the start of a program a MAIN: must be used.

FUNCTIONs can access global variables that have been declared before the FUNCTION, this declaration can either be implicit or use a DIM.

FUNCTIONs must be defined before they are called.

Example

```
function toupper(a(100) as string) as string
  dim i as integer

  for i=0 to 100
    if a(i)=0 then exit
    if a(i) <= "z" and a(i) >= "a" then a(i) = a(i) - &H20
  next i
  return a
end function

main:

print toupper("asdf")                ' will print ASDF
```

Differences from other BASICs

- simplification of Visual BASIC
- no equivalent in PBASIC

See also

- **DIM**
- **PARAMARRAY**
- **GOSUB**
- **ENDSUB**
- **MAIN:**

SUB *name* (optional parameters)

Syntax

SUB *name* ' a SUB without parameters

or

SUB *name* (parameter list)
parameter list = parameter [, parameter list]
| PARAMARRAY paramarray
parameter = [BYVAL] paramname [AS INTEGER]
| [BYVAL] paramname(size) AS STRING
| BYREF paramname AS STRING
| BYREF paramname [AS INTEGER]

Description

GOSUB goes to a *label*. , but can also go to a defined SUB procedure.

The SUB.. ENDSUB construct allows for a second scope of variables. Scope meaning the region in which code can see a set of labels. ARMbasic has a global scope and a local scope for any variable declared with DIM inside an SUB. Local scope variables will be only accessible from within that SUB procedure (the local scope).

Simple parameters are assumed to be called BYVAL if not specified. In BYVAL calls, a copy of the parameter is passed to the SUB procedure. Integer or string parameters may be called BYREF which means a pointer to the integer/string is passed, and changes to that integer/string can be made by code inside the SUB procedure. STRINGs may be passed either BYVAL or BYREF, but arrays must be passed BYREF.

Code labels for goto/gosub declared within the SUB procedure are also in the local scope. Call to global labels are allowed within a SUB .. ENDSUB, but that global label must be defined BEFORE the SUB ... ENDSUB.

SUB without any parameters can be called with or without (), an empty parameter list

Recursive calls with parameters or local variables are not supported. And ENDSUB or END SUB syntax are allowed.

Program structure:

SUB procedures should be arranged ahead of the MAIN: body of code. In many cases they will be part of #include files at the beginning of the user ARMbasic code. If SUBs are located at the start of a program a MAIN: must be used.

SUB procedures can access global variables that have been declared before the SUB, this declaration can either be implicit or use a DIM.

An implied RETURN is compiled at the ENDSUB, but code may also return to the caller with RETURN

SUBs must be defined before they are called.

Example

```
SUB sayHello
   DIM I as INTEGER      ' this variable is local to the sayHello SUB procedure
```

```
    FOR I=1 to 3
       PRINT "Hello"
    NEXT I

ENDSUB
 '...

MAIN:
 '...
I = 55
PRINT I                    ' will display 55

GOSUB sayHello

PRINT I                    ' will still display 55, as this is the global I, different from sayHello local I
 '...
```

Differences from other BASICs

- simplification of Visual BASIC
- no equivalent in PBASIC

See also

- **DIM**
- **PARAMARRAY**
- **GOSUB**
- **ENDSUB**
- **MAIN:**

ENDFUNCTION | END FUNCTION

Syntax

ENDFUNCTION

ENDFUNCTION or END FUNCTION syntax are allowed

Description

ENDFUNCTION terminates a FUNCTION procedure

FUNCTIONs must be defined before they are called.

Example

```
function toupper(a(100) as string) as string
  dim i as integer
  dim l as integer
  l = len(a)
  for i=0 to l
    if a(i) <= "z" and a(i) >= "a" then a(i) = a(i) - &H20
  next i
  return a
end function

main:

print toupper("asdf")                ' will print ASDF
```

Differences from other BASICs

- simplification of Visual BASIC
- no equivalent in PBASIC

See also

- **DIM**
- **GOSUB**
- **SUB**
- **MAIN:**

ENDSUB | END SUB

Syntax

ENDSUB

ENDSUB or END SUB syntax are allowed

Description

ENDSUB terminates a SUB procedure

SUBs must be defined before they are called.

Example

```
SUB sayHello
   DIM I as INTEGER      ' this variable is local to the sayHello SUB procedure

   FOR I=1 to 3
      PRINT "Hello"
   NEXT I

ENDSUB
 '...

MAIN:
 '...
I = 55
PRINT I                  ' will display 55

GOSUB sayHello

PRINT I                  ' will still display 55, as this is the global I, different from sayHello local I
```

Differences from other BASICs

- simplification of Visual BASIC
- no equivalent in PBASIC

See also

- **DIM**
- **GOSUB**
- **SUB**
- **MAIN:**

Operators List

Operator List

- & (String concatenation)
- * (Multiplication)
- + (Addition)
- + (String concatenation)
- - (Negation)
- - (Subtraction)
- / (Division)
- < (Less than)
- <= (Less than or equal)
- <> (Inequality)
- = (Equality)
- > (Greater than)
- >= (Greater than or equal)
- ABS
- ADDRESSOF
- AND (Conjunction)
- IF(ternary)
- MOD (Integer modulo)
- NOT (Bit-wise complement)
- OR (Disjunction, Inclusive Or)
- << (Shift-left)
- >> (Shift-right)
- XOR (Exclusive Or)

& (String concatenation)

Syntax

string1 & string *2*

Description

The concatenation returns a string made of sticking both variables together. If some of the variables are not strings, the **STR** function is called automatically to convert the variable to a string.

Multiple concatenations per line are supported, and the strings can include string functions such as LEFT, RIGHT, HEX and STR. Also if a constant or integer is used it will be automatically converted to a string, as if it had been enclosed in a STR().

Example

```
DIM A(20) as STRING
DIM C(20) as STRING
A ="The result is: "
B=1243
C=A & B
PRINT C
SLEEP
```

The output would look like:

```
The result is: 1243
```

Differences from other BASICs

- same as Visual Basic functions
- no equivalent in PBASIC

See also

- **+ String Concatenation**
- **String Functions**

* (Multiplication)

Syntax

*argument1 * argument2*

Description

The multiplication operator is used to multiply two numbers, and is the inverse of division, /. The arguments *argument1* and *argument2* can be any valid numerical expression.

Example

```
n = 4 * 5
PRINT n
SLEEP
```

The output would look like:

```
20
```

Differences from other BASICs

- None

See also

- / (Division)
- + (Addition)
- Mathematical Functions

+ (Addition)

Syntax

argument1 + *argument2*

Description

The addition operator is used to find the sum of two numbers. Addition, +, is the inverse of subtraction, -. The arguments *argument1* and *argument2* can be any valid numerical expression.

Example

```
n = 454 + 546
PRINT n
SLEEP
```

The output would look like:

```
1000
```

Differences from other BASICs

- None

See also

- **- (Subtraction)**
- **Mathematical Functions**

+ (String concatenation)

Syntax

string1 + *string2*

Description

The concatenation operator takes two string variables and returns a string made of sticking both strings together.

Multiple concatenations per line are supported, and the strings can include string functions such as LEFT, RIGHT, HEX and STR. Also if a constant or integer is used it will be automatically converted to a string, as if it had been enclosed in a STR().

Example

```
DIM  A(20)
DIM  B(20)
DIM  C(30)

A="Hello,"
B=" World!"
C=A +B
PRINT C
SLEEP
```

The output would look like:

```
Hello, World!
```

Differences from other BASICs

- PBASIC does not have string function support
- Similar to Visual BASIC

See also

- **& String Concatenation**
- **String Functions**

- (Negation)

Syntax

- number

Description

The negation operator is used to give the negative value of *number*. *number* can be any valid numerical expression.

Example

```
PRINT -5
n = 6543256
n = - n
PRINT n
SLEEP
```

The output would look like:

```
-5
-6543256
```

Differences from other BASICs

- None

See also

- **Mathematical Functions**

- (Subtraction)

Syntax

argument1 - *argument2*

Description

The subtraction operator is used to find the difference between two numbers. Subtraction, -, is the inverse of addition, **+**. The arguments *argument1* and *argument2* can be any valid numerical expression.

Example

```
n = 4 - 5
PRINT n
SLEEP
```

The output would look like:

```
-1
```

Differences from other BASICs

- None

See also

- **+ (Addition)**
- **Mathematical Functions**

> (Greater than)

Syntax

expressionLEFT > *expressionRT*

Description

The > (Greater-than) Operator evaluates two expressions, compares them and returns the resulting condition. The condition is false (0) if the left-hand side expression is less than or equal to the right-hand side expression, or true (1) if it is greater than the right-hand side expression.

Example

The **<= (Less-than Or Equal) Operator** is complement to the > (Greater-than) Operator, and is functionally identical when combined with the **NOT (Bit-wise Complement) Operator**:

```
IF( 420 > 69 ) THEN PRINT "( 420 > 69 ) is true."
IF NOT( 420 <= 69 ) THEN PRINT "not( 420 <= 69 ) is true."
```

Differences from other BASICs

- none

See also

- **<**
- **<=**
- **<>**
- **>**
- **>=**
- **Mathematical Functions**

/ (Division)

Syntax

argument1 / *argument2*

Description

The division operator is used to divide (or to find the ratio of) two numbers and return an integer or floating point result. Division is the inverse of multiplication, *. The arguments *argument1* and *argument2* can be any valid numerical expression.

Example

```
DIM x AS SINGLE

x=123.456
PRINT n / 5
n = 600000 / 23
PRINT n
PRINT x/n
```

The output would look like:

```
0
26086
4.732654E-02
```

Differences from other BASICs

- None with PBASIC
- Visual BASIC returns a floating point result

See also

- *** (Multiplication)**
- **Mathematical Functions**

< (Less than)

Syntax

expressionLEFT < *expressionRT*

Description

The < (Less-than) Operator evaluates two expressions, compares them and returns the resulting condition. The condition is false (0) if the left-hand side expression is greater than or equal to the right-hand side expression, or true (1) if it is less than the right-hand side expression.

Example

The **>= (Greater-than Or Equal) Operator** is complement to the < (Less-than) Operator, and is functionally identical when combined with the **NOT (Bit-wise Complement) Operator**:

```
IF( 69 < 420 ) THEN PRINT "( 69 < 420 ) is true."
IF NOT( 69 >= 420 ) THEN PRINT "not( 69 >= 420 ) is true."
```

Differences from other BASICs

- none

See also

- **<**
- **<=**
- **<>**
- **>**
- **>=**
- **Mathematical Functions**

<= (Less than or equal)

Syntax

expressionLEFT <= *expressionRT*

Description

The <= (Less-than) or Equal Operator evaluates two expressions, compares them and returns the resulting condition. The condition is false (0) if the left-hand side expression is greater than the right-hand side expression, or true (1) if it is less than or equal to the right-hand side expression.

Example

The **> (Greater-than) Operator** is complement to the <= (Less-than or Equal) Operator, and is functionally identical when combined with the **NOT (Bit-wise Complement) Operator**:

```
IF( 69 <= 420 ) THEN PRINT "( 69 <= 420 ) is true."
IF NOT( 60 > 420 ) THEN PRINT "not( 420 > 69 ) is true."
```

Differences from other BASICs

- the =< version of Visual BASIC is also supported
- none from PBASIC

See also

- **<**
- **<=**
- **<>**
- **>**
- **>=**
- **Mathematical Functions**

<> (Inequality)

Syntax

expressionLEFT <> *expressionRT*

Description

The <> (Inequality) Operator evaluates two expressions, compares them for inequality and returns the resulting condition. The condition is false (0) if the left-hand side expression and the right-hand side expression are equal, or true (1) if they are unequal.

Example

In a number guessing game, the <> (Inequality Operator) can be used to check the player's guess with the secret number:

```
guess = 0
      ' ...                  " <- get number from user and store in guess
  IF( guess <> secret_number ) THEN PRINT "Sorry, you guessed wrong. Try again."
      '  ...
```

The **= (Equality) Operator** is complement to the <> (Inequality) Operator, and is functionally identical when combined with the **NOT (Bit-wise Complement) Operator**:

```
IF( 420 <> 69 ) THEN PRINT "( 420 <> 69 ) is true."
IF NOT( 420 = 69 ) THEN PRINT "not( 420 = 69 ) is true."
```

Differences from other BASICs

- none

See also

- **<**
- **<=**
- **<>**
- **>**
- **>=**
- **Mathematical Functions**

= (Equality)

Syntax

expressionLEFT = *expressionRT*

Description

The = (Equality) Operator evaluates two expressions, compares them for equality and returns the resulting condition. The condition is false (0) if the left-hand side expression and the right-hand side expression are unequal, or true (1) if they are equal.

Example

Equality comparisons should not be confused with **Assignments**, both of which also use the "=" symbol:

```
i = 420              " assignment: assign the value of i as 420

IF( i = 69 ) THEN    " equation: compare the equality of the value of i and 69
    PRINT "serious error: i should equal 420"
    END
ENDIF
```

The **<> (Inequality) Operator** is complement to the = (Equality) Operator, and is functionally identical when combined with the **NOT (Bit-wise Complement) Operator**:

```
IF( 420 = 420 ) THEN PRINT "( 420 = 420 ) is true."
IF NOT( 69 <> 69 ) THEN PRINT "not( 69 <> 69 ) is true."
```

Differences from other BASICs

- none

See also

- **<**
- **<=**
- **<>**
- **>**
- **>=**
- **Mathematical Functions**

>= (Greater than or equal)

Syntax

lexpressionLEFT >= expressionRT

Description

The >= (Greater-than) or Equal Operator evaluates two expressions, compares them and returns the resulting condition. The condition is false (0) if the left-hand side expression is less than the right-hand side expression, or true (1) if it is greater than or equal to the right-hand side expression.

Example

The **< (Less-than) Operator** is complement to the >= (Greater-than or Equal) Operator, and is functionally identical when combined with the **NOT (Bit-wise Complement) Operator**:

```
IF( 420 >= 69 ) THEN PRINT "( 420 >= 69 ) is true."
IF NOT( 420 < 69 ) THEN PRINT "not( 420 < 69 ) is true."
```

Differences from other BASICs

- the => version of Visual BASIC is also supported
- none from PBASIC

See also

- **<**
- **<=**
- **<>**
- **>**
- **>=**
- **Mathematical Functions**

AND

Syntax

number1 AND *number2*

Description

In BASIC all logic operators are context sensitive. The default is to perform a bitwise operation on 32 bit numbers. So that

```
number1 = &H55555555
number2 = &H33333333

print HEX(number1 AND number2)
```

Result is 11111111. The function **HEX** (param) returns a string representing the hex value param.

But is the context includes a comparison then the AND operation becomes a logical operation

```
IF number1 > 5 AND number2 > 5 THEN statement1
```

So if the 2 conditions are TRUE then statement1 is executed.

If you are going to mix logical and bitwise operators then use () in the expression to evaluate the bitwise operations first, see the example below.

Example

```
x=&H55
y=&H33
if (x and y)=&H11 and x>y then print "OK"


' This will print OK
```

Differences from other BASICs

- none from Visual BASIC
- PBASIC AND is always logical, and & is bit-wise

See also

- **OR**
- **XOR**
- **NOT**

NOT

Syntax

NOT *expression*

Description

Not, at its most primitive level, is a operation, a logic function that takes one bit and returns a inverted bit. This function returns true if the bit is false, and false if the bit is true. This also holds true for conditional expressions in **ARMbasic** . When using "Not" encased in an If block, While loop, or Do loop, the output will behave quite literally:

```
IF NOT condition1 THEN expression1
```

Is translated as:

```
IF condition1 = 0   THEN perform expression1
```

When given a expression, number, or variable that return a number that is more than a single bit, Not is performed bit-wise. A bit-wise operation performs a logic operation for every bit.
The Boolean math expression below describes this:

```
00001111 NOT
-------- equals
11110000
```

Notice how in the resulting number of the operation, reflects an NOT operation performed on each bit of the expression.

When used with conditions NOT becomes a logical operation.

```
if NOT x>5 then print "x is less than or equal to 5"
'-------- equivalent to
if x <= 5 then print "x is less than or equal to 5"
```

In the above example if x is 7 and you PRINT NOT x>5 would print 0, and print 1 if x is 3.

If using NOT in a complex statement, it has a very low precedence and it is best to force the operation of the NOT by using parenthesis.

Example

```
' Using the NOT operator on a numeric value


numeric_value = 15 '00001111

'Result = -16 =     11111111111111111111111111110000
PRINT NOT numeric_value
END
```

```
' Using the NOT operator on conditional expressions

n1 = 15
n2 = 25

IF NOT n1 = 10 THEN PRINT "N1 is not equal to 10"
IF NOT n2 = 25 THEN PRINT "N2 is not equal to 25"
IF (NOT n2 = 25) or n2=10 THEN PRINT "N2 is not equal to 25 or is equal to 10"
END

' This will output "Numeric_Value1 is not equal to 10" because
' the first IF statement is false.
' It will not output the result of the second IF statement because the
' condition is true.
```

Differences from other BASICs

- None

See also

- **AND**
- **OR**
- **XOR**

OR

Syntax

number OR *number*

Description

In BASIC all logic operators are context sensitive. The default is to perform a bitwise operation on 32 bit numbers. So that

```
number1 = &H55555555
number2 = &H33333333

print HEX(number1 OR number2)
```

Result is 77777777. The function **HEX** (param) returns a string representing the hex value param.

But is the context includes a comparison then the AND operation becomes a logical operation

```
IFnumber1 > 5 OR number2 > 5 THEN statement1
```

So if either of the 2 conditions are TRUE then statement1 is executed.

If you are going to mix logical and bitwise operators then use () in the expression to evaluate the bitwise operations first, see the example below.

Example

```
x=&H55
y=&H33
if (x or y)=&H77 or x<y then print "OK"


' This will print OK as X OR Y is &H77
```

Differences from PBASIC

- PBASIC OR is always logical, and | is bit wise

See also

- **AND**
- **XOR**
- **NOT**

<<

Syntax

number << *places*

Description

<< shifts all bits in the argument *number* INTEGER to the left by argument *places*. This has the effect of multiplying the argument *number* by two for each shift given in the argument *places*. Both arguments, *numbers* and *places* are integers. This is easiest to see in a binary number. For example %0101 << 1 return the binary number %01010. In base 10 numbers this looks like 5 << 1 and returns 10.

Shift operations can not be used on SINGLE variables

Example

```
FOR i = 1 TO 10
      PRINT 1 << i
NEXT i
SLEEP
```

The output would look like:

```
2
4
8
16
32
64
128
256
512
1024
```

Differences from other BASICs

- none

See also

- >>

>>

Syntax

number >> *places*

Description

>> shifts all bits in the argument *number* INTEGER to the right by argument *places*. This has the effect of dividing the argument *number* by two for each shift given in the argument *places*. Both arguments, *numbers* and *places* are integers. This is easiest to see in a binary number. For example %0101 >> 1 return the binary number %010. In base 10 numbers this looks like 5 >> 1 and returns 2.

If the *number* variable is signed, the sign bit is recopied into its place after the shift.

Shift operations can not be used on SINGLE variables

Example

```
FOR i = 1 TO 10
      PRINT 1000 >> i
NEXT i
SLEEP
```

The output would look like:

```
500
250
125
62
31
15
7
3
1
0
```

Differences from other BASICs

- none

See also

- <<

XOR

Syntax

number XOR *number*

Description

XOR, at its most primitive level, is a Boolean operation, a logic function that takes in two bits and outputs a resulting bit. If given two bits, this function returns true if ONLY one of the bits are true, and false for any other combination. The truth table below demonstrates all combinations of a Boolean XOR operation:

```
Bit1    Bit2    Result
 0       0       0
 1       0       1
 0       1       1
 1       1       0
```

In ARMbasic, XOR is done as a bit-wise operation, unless it is part of a logical expression. Logical expressions are those that compare values. And when part of a logical expression the XOR is true if either logical expression are true, but not if both are true.

```
bit wise  00001111 XOR 00011110  is equal to  00010001
```

Notice how in the resulting number of the operation, reflects an XOR operation performed on each bit of the top operand, with each corresponding bit of the bottom operand.

Example

```
x=&H55
y=&H33
if (x xor y)=&H66 or x<y then print "OK"


' This will print OK as X XOR Y is &H66
```

Differences from PBASIC

- PBASIC XOR is always logical, and ^ is bit wise

See also

- AND
- OR
- NOT

Compound Operators

Description

Version 8.10 added compound operators. These are a special assignment statement that applies the operator to the lvalue that will be assigned.

lvalue+= 1

The statement above will increment the variable, in general a compound operator makes the 2 following statements equivalent.

lvalue op= *expr< /EM>*

lvalue = lvalue op expr

*Compound operators include * + - / AND OR XOR >> <<*

Operator precedence does NOT change with a compound operator so that

```
x += a << b   ' BAD, as the addition will take precedence over the shift
```

should not be used, but you should use

```
x += (a << b)
```

Example

```
i += 1                  ' increment i< /FONT>
a AND= &Hffff0fff       ' mask bits 12 to 15 of a
b <<= 5                 ' shift b left by 5 bits
c OR= &Hf0              ' set bits 4-7 of c
h *= 10                 ' multiply h by 10
```

See also

- *Operator List*

Operator Precedence

Description

When several operations occur in a single expression, each operation is evaluated and resolved in a predetermined order. This called the order of operation or operator precedence. There are three main categories of operators; arithmetic, comparison, and logical. If an expression contains operators from more than one category, arithmetic operators are evaluated first, comparison operators next, and finally logical operators are evaluated last. If operators have equal precedence, they then are evaluated in the order in which they appear in the expression from left to right. Comparison operators all have equal precedence.

The following table gives the operator precedence for each operator in each category. Operators lower on the list have a lower operator precedence. Operators on the right have lower precedence than ALL operators in the column to the left. Arithmetic operators are evaluated before comparison operations, and logical operators are last.

Parentheses can be used to override operator precedence. Operations within parentheses are performed before other operation. However, within the parentheses operator precedence is used.

Arithmetic	Comparison	Bitwise/Logical
- (Negation)	= <> < > <= >=	AND
*, / (Multiplication and division)		OR XOR
MOD (Modulus Operator)		NOT
+, - (Addition and subtraction)		
<<, >> (Shift Bit Left and Shift Bit Right)		

See also

- Operator List

Ternary Operation

Syntax

IF(*expression, true_expr, false_expr)*

Description

Ternary operations allow a shorthand for doing conditional calculations

```
IF   ch >= "a"  THEN ch =  ch AND &H5F
```

can be written as:

```
ch = IF(ch > = "a", ch AND &H5F, ch)        ' this is a crude upshift
```

Differences from other BASICs

- No equivalent in PBASIC
- None from VB

See also

- **AND**
- **OR**
- **XOR**

Type Conversions

Description

The ARMbasic compiler handles many conversions between INTEGER, SINGLE and STRING variables.

input	INTEGER	SINGLE	STRING
INTEGER =		yes	no
SINGLE =	yes		no
STRING =	yes	no*	
FUNCTION (INTEGER param)		yes	no
FUNCTION (SINGLE param)	yes		no
FUNCTION (STRING param)	no	no	
RETURN INTEGER-FUNCTION		yes	no
RETURN SINGLE-FUNCTION	yes		no
RETURN STRING-FUNCTION	no	no	
printf	no	no	no

Simple statements

lvalue = expression

Expression will be evaluated with the type of lvalue and conversions will be made to lvalue's type as allowed above

Complex Statements

IF (expressionA) then RETURN expressionB else RETURN expressionC

ExpressionA is evaluated depending on the first operand found (scanning from left to right)

ExpressionB and ExpressionC will be converted to the type of the FUNCTION as allowed in the table.

IF (expressionA) then lvalue = expressionB

ExpressionA is evaluated depending on the first operand found (scanning from left to right)

ExpressionB will be evaluated with the type of lvalue

Parameter Passing

FUNCTION someFunction (paramA AS typeA, paramB AS typeB, paramC AS typeC) AS funtionType

...

lvalue = someFunction (ExpressionA, ExpressionB, ExpressionC)

ExpressionA, ExpressionB and ExpressionC will be evaluated depending on the type of the parameter as declared.

If the type of someFunction is different than the type of lvalue it will be converted as allowed.

printf

printf is a special built in SUB, there is no type checking or conversion between parameters and the format-string, so if you try to use a %d to print a SINGLE type variable, you will not get the expected results.

INTEGER/STRING

INTEGERs are converted to string type and concatenated with the rest of the string expression.

To convert a STRING to an INTEGER use the built-in VAL function.

Currently you must use SPRINTF to convert a SINGLE to a STRING

There is no built-in conversion from STRING to SINGLE.

See also

- **Operator List**

AddressOf

Syntax

... = *ADDRESSOF(sub/function)* ' get the starting address of the sub/function

or

... = ADDRESSOF (variable/string)

Description

The address of a variable or function can be determined with the ADDRESSOF operator.

Earlier versions of the compiler did not use ( ) around the object, and that syntax is still allowed though not recommended.

The variable, string, sub or function must be declared prior to the use of ADDRESSOF operation.

Example

```
xx = 0

interrupt sub doit
  ' insert code to clear the interrupt
   xx = xx+1
end sub

'...      other code including MAIN go here

VICVectAddr3 =  ADDRESSOF ( doit ) +1    ' setup the 3rd interrupt to execute doit (+1 for Thumb code)
```

Differences from other BASICs

- no equivalent in Visual BASIC
- no equivalent in PBASIC

See also

-

AddressOf

Syntax

... = *ADDRESSOF(sub/function)* ' get the starting address of the sub/function

or

... = ADDRESSOF (variable/string)

Description

The address of a variable or function can be determined with the ADDRESSOF operator.

Earlier versions of the compiler did not use ( ) around the object, and that syntax is still allowed though not recommended.

The variable, string, sub or function must be declared prior to the use of ADDRESSOF operation.

Example

```
xx = 0

interrupt sub doit
  ' insert code to clear the interrupt
   xx = xx+1
end sub

'...      other code including MAIN go here

VICVectAddr3 =  ADDRESSOF ( doit ) +1    ' setup the 3rd interrupt to execute doit (+1 for Thumb code)
```

Differences from other BASICs

- no equivalent in Visual BASIC
- no equivalent in PBASIC

See also

-

Constants

Description

Constants are numbers which cannot be changed after they are defined. For example, 5 will always mean the same number.

In ARMbasic, variable names can be told to be constants by defining them with the **CONST** command.

Such constants are then available globally, meaning that once defined, you can use the word to refer to a constant anywhere in your program.

After being defined with the **CONST** command, constants cannot be altered. If code tries to alter a constant, an error message will result upon code compilation.

Only the first 32 characters of a constant name are used, beyond that they are truncated.

By default, constants are defined by decimal numbers. Versions of the compiler after 7.43 also support VB style hex constants defined by &H, such as &H1000 = 4096.

PBASIC style hex and binary constants may also be used. A PBASIC style hex constant will begin with $, such as $3FAB. PBASIC style Binary constants begin with %, such as %010101111. While decimal constants can be signed, hex and binary constants are always unsigned.

IEEE 754 floating point support has been added. Constants for single numbers can be defined as an option - sign followed by decimal digits for the integer portion, followed by . then the fractional portion. Such as 1234.568 55. 0.0000000003456

Example

```
CONST PI = 3.14159265

CONST FirstNumber = 1
CONST SecondNumber = - 2


PRINT FirstNumber, SecondNumber 'This will print 1      -2
```

See also

- **CONST**

Variables

Syntax

symbolname = *expression* ' automatic declaration for INTEGER

or

DIM *symbolname* AS INTEGER

DIM *floatingpoint* AS SINGLE

Description

Variables are values which can be manipulated. They are referenced using names composed of letters, numbers, and character "_". These reference names cannot contain most other symbols because such symbols are part of the **ARMbasic** programming language. They also cannot contain spaces.

32-bit signed whole-number data type. Can hold values from -2147483648 to 2147483647.

Single variables can hold positive and negative numbers 0.0, 1.0000E-39 to 1.0000E+38. Within this range there are 7 significant digits. Outside this range numbers will print as **NaN -- not a number** or **Inf for infinity** .

Variables are declared automatically on first use. A DIM statement is not required, but can be used. Once a INTEWGER variable is declared using a DIM, **then all following variables must be declared that way** .

Variable names must start with a letter or _ character, but then can contain letters, numbers or the _ character. The name must also contain at least one letter. Only the first 32 characters of a variable name are used, beyond that they are truncated. Also names are not case sensitive.

Example

```
FirstNumber = 1
SecondNumber = -2
ThirdNumber = &H20

PRINT FirstNumber, SecondNumber, ThirdNumber 'This will print 1      -2      32
```

```
DIM FirstNumber AS INTEGER
DIM SecondNumber AS INTEGER
DIM ThirdNumber AS INTEGER

FirstNumber = 1
SecondNumber = -2
ThirdNumber = &H20

PRINT FirstNumber, SecondNumber, ThirdNumber 'This will print 1      -2      32
```

Differences from other BASICs

- similar to Visual BASIC
- different syntax in PBASIC

See also

- DIM

Arrays

Description

Arrays are **Variables** which contain more than one value. The value decided upon is chosen using an index which is an integer value between 0 and the number of elements in the array. In **ARMbasic** , any array must be declared before it's first use using the **DIM** command.

The best way to conceptualize an array is look at it like a spreadsheet. For example, if you had an array called myArray which contained elements (0 to 10), and was filled with random numbers, you could look at it like this:

Index	Data
0	4
1	5
2	2
3	6
4	5
5	9
6	1
7	0
8	4
9	5
10	7

Keep in mind that the numbers in the Data column are completely arbitrary in our example. When you create an array in **ARMbasic** using the DIM command, the elements are all set to 0.

If you were to look at myArray(1), you'd find it's equal to 5. If you were to look at myArray(5),you'd find it equal to 9. In **ARMbasic** , you can for the most part treat arrays with indexes the same as you would all **Variables**.

Arrays can hold BYTE, INTEGER or SINGLE values. Index values are always computed as integers,

so that any SINGLE variable used inside the (index expression) is truncated to an INTEGER.

Example

```
DIM Numbers( 10)
DIM OtherNumbers( 10)

Numbers(1) = 1
Numbers(2) = 2
OtherNumbers(1) = 3
OtherNumbers(2) = 4

GOSUB PrintArray

FOR a = 1 TO 10
 PRINT Numbers(a)
NEXT a

PRINT OtherNumbers(1)
PRINT OtherNumbers(2)
PRINT OtherNumbers(3)
PRINT OtherNumbers(4)
PRINT OtherNumbers(5)
PRINT OtherNumbers(6)
PRINT OtherNumbers(7)
PRINT OtherNumbers(8)
PRINT OtherNumbers(9)
PRINT OtherNumbers(10)

PrintArray:
 FOR i = 1 TO 10
  PRINT otherNumbers(i)
 NEXT i
RETURN
```

See also

- **Strings**
- **DIM**

Strings

Syntax

DIM *symbolname$ (maxlength)* ' kept for backward compatibility

or

DIM *symbolname (maxlength)* AS STRING

or

DIM *symbolname (maxlength)* AS BYTE

Description

A STRING is a special array of BYTE terminated by a byte of 0, and is limited to 256 characters. String operations must be limited to the first 256 characters and there is no runtime check.

Despite the use of the *maxlength* , an implicit **&H0** is added to the end of the STRING, to allow for variable length during program execution. For this reason a &H0 may not be used within a string.

Byte arrays can be used using an allocation as BYTE, and they may exceed 256 characters. They may also contain embedded &H0 elements. But if they do, string operations can not be used. For instance a byte array of &H0, &H1, &H2 can be built as-

```
astr(0) = 0
astr(1) = 1
astr(2) = 2
astr(3) = 3
```

But the following will **NOT** work-

```
astr = chr(0) + chr(1) + chr(2) + chr(3)   ' fails as the first 0 terminates this string operation
```

But it can be done as-

```
astr = chr(1) + chr(1) + chr(2) + chr(3)
astr(0) = 0                                ' replace the first character with a $0
```

STRINGs are not checked for length at run time, so care must be taken to avoid filling it beyond the declared DIM.

Individual characters within a string can be accessed like an array, such as astr(12) returns the character in position 13, with the first element at offset 0.

Single character strings are a special case, and usually replaced by the byte constant representing that character. So "A" can be used interchangeably with &H41 or 63.

Example

```
'  Fixed-length declaration, but value varies during execution
DIM astr (20) as STRING
astr = "Hello"

astr = astr+chr(32)+ "World"
```

```
PRINT astr       '  = "Hello World"
```

Differences from other BASICs

- Similar to Visual BASIC strings. In VB strings can have implied length when declared, but **ARMbasic** requires an explicit length when declared.
- PBASIC has Arrays of BYTEs but no specific strings

See also

- **STR**

ARM Hardware Access

Description

While **ARMbasic** provides access to many hardware functions through various keywords, there are cases where the user may want to program the available control registers directly.

Example

```
DayOfWeek = * (&HE0024034)    ' read the real time clock day of week register

* (&HE0024034) = DayOfWeek    ' write the real time clock day of week register
```

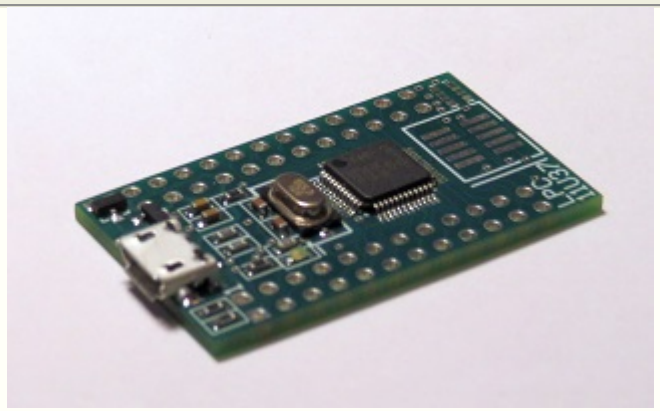
Differences from other BASICs

- no equivalent in Visual BASIC
- no equivalent in PBASIC

See also

- **CPU Register Details**

Converting Data Types



To/From Strings

- ASC implied
- CHR
- HEX
- STR
- VAL

Other automatic conversions

Expression Evaluation

Description

ARMbasic can evaluate an expression as either an integer, string or floating point. The type of evaluation that is performed is determined by the first variable, constant or function in a statement when scanning from left to right. It is simple scan of left to right and () are ignored for determining the type of evaluation that will occur.

Example

```
DIM SINGLEtype as SINGLE
SINGLEtype = 1/2 ' results in 0.500000
INTEGERtype = 1/2 ' results in 0
functionCall ( 0.123 + INTEGERtype) ' passes a SINGLE type to functionCall
functionCall ( 123 + SINGLEtype ) ' passes an INTEGER to functionCall
if 1 = SINGLEtype*2 then print "true"          ' will not print true, as the expression is evaluated as integers
if 1.0 = SINGLEtype*2 then print "true"        ' will print true as 1.0 is a floating point constant and the expression is evaluated as float
```

CHR

Syntax

CHR(*expression*)

Description

CHR returns a single byte string containing the character represented by the **ASCII** code passed to it. For example, CHR(97) returns "a".

Note:
There is no need for a complimentary function, as that type conversion is automatic, see sample code below.

Example

```
PRINT "the character represented by the ASCII code of 97 is:"; CHR(97)    ' will print   a

DIM a(10) as STRING  ' examples of automatic type conversion complimentary to CHR
a ="asdf"

PRINT a(0), chr(a(0))                ' will print    97     a
x = a(0)
PRINT x                               ' will print    97
if x = "a" then PRINT "it is a"        ' will print it is a
```

Differences from other BASICs

- does not exist in PBASIC
- same function exists in Visual BASIC

See also

- **STR**
- **HEX**
- **VAL**

HEX

Syntax

HEX (*expression*)

Description

This returns the hexadecimal string representation of the INTEGER *expression* . Hexadecimal values contain 0-9, and A-F. The size of the result string depends on the integer type passed, it's not fixed.

HEX is not defined for use with SINGLE types

Example

```
DIM text(10) as String

text = HEX(4096)
PRINT "0x";text     ' will display 0x1000
```

Differences from other BASICs

- same function as Visual BASIC
- similar to PBASIC format directive available in SHIFTIN, SERIN, DEBUGIN

See also

- **CHR**
- **STR**
- **VAL**

I2F F2I -- implied function

Syntax

In ARMbasic this is an automatic type conversion between SINGLE and INTEGER

But if you want to do it explicitly, in your code add the following do-nothing #define

```
#define I2F(x)    0.0+x   ' works because 0.0 is the first operand encountered and makes it floating point

#define F2I(y)    0+y     ' works because the first operand is 0 an integer
```

Description

ARMbasic automatically converts INTEGER type variables to/from SINGLE type variables.

In the simplest case, this is done by assigning SINGLE type = INTEGER type or vice versa.

When statements are parsed by ARMbasic, the parser looks for the first variable or constant and will use that elements type to determine whether to use floating point or integer calculations. Index parameters are always treated as INTEGER.

So --

```
DIM SINGLEtype AS SINGLE

SINGLEtype = 1/2     '  results in 0.500000

INTEGERtype = 1/2   '  results in 0

functionCall ( 0.123 + INTEGERtype)   ' passes a SINGLE type to functionCall

functionCall ( 123 + SINGLEtype )      ' passes an INTEGER to functionCall
```

To access bit fields within a SINGLE, you must convert it to an INTEGER without the implied type conversion. The easiest way to do that is

```
INTEGERtype = * (ADDRESSOF SINGLEtype)     ' now you can do bit manipulations with AND, OR, XOR, NOT on INTEGERtype.
```

Differences from other BASICs

- does not exist in PBASIC
- same function exists in Visual BASIC

See also

- **ASCII** table
- **HEX**
- **VAL**

STR

Syntax

STR(*expression*)

Description

STR will convert an integer *expression* into a string.

For example, STR(3) will become "3", or STR(333) will become "333".

Incidentally, this is the opposite of the **VAL** function, which converts a string into a number.

STR is also used in certain routines of the Hardware Library to designate that a series of bytes should be read or written to a string.

Also in the following case the STR function is implied and is not required.

bstring = 333 + " sent" ' will save the ASCI string "333 sent" into b$

The implied STR will work for simple expressions, but anything complex should use STR(), this would include any function call, array element fetches.

For floating point numbers use SPRINTF.

Example

```
DIM bstring (10) AS STRING
a = 8421
bstring = STR(a)
PRINT a, b     ' will display    8421     8421
```

Differences from other BASICs

- same function in Visual BASIC
- similar to DEC formatting function in PBASIC

See also

- **VAL**
- **CHR**
- **HEX**
- **Hardware Library, Function List**

VAL

Syntax

VAL(*string*)

Description

VAL converts a *string* to a decimal number. For example, VAL("10") will return 10. The function parses the string from the left and returns the longest number it can read, stopping at the first non-suitable character it finds.

Incidentally, this function is the opposite of **STR** , which converts a number to a string.

string can contain negative numbers or hex numbers when preceded by a $.

VAL does only INTEGER interpretation

Example

```
DIM astring(20) as STRING
astring = "20xa211"
b = VAL(astring)
PRINT astring, b
```

```
20xa211   20
```

Differences from other BASICs

- similar to Visual BASIC -- VB ignores space and tab characters during the parsing and allows &H or &O
- similar to formatting directives DEC, HEX in PBASIC

See also

- **STR**
- **HEX**
- **CHR**

Additional Reserved Words

The Future
The **ARMexpress** is the first in a new generation of ARM-based controllers. The ARMbasic language has provisions for some of the features for the next members in the family. For this reason a number of words are reserved for future use.

In order to maintain compatibility with future ARMbasic instructions the following words have been reserved.

CODE	
DATA	READONLY
QUIT	WEB
QUITDUMP	WEBGET
QUITNOW	

Obsolete Reserved Words

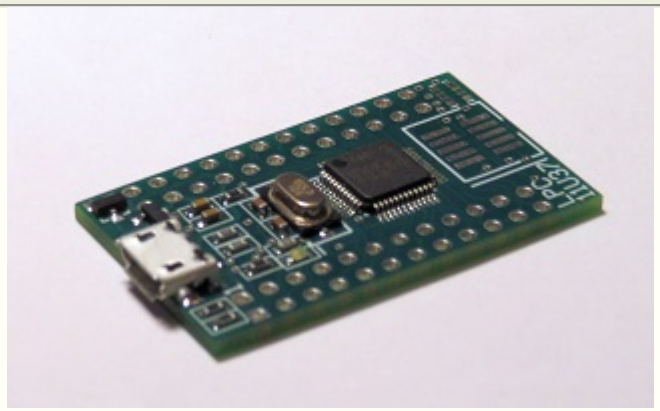
The Past

The ARMbasic language has replaced some older constructs with either other keywords or library calls to make room for floating point.

In order to maintain compatibility with other ARMbasic instructions the following words have been reserved.

BAUD0 BAUD1 RXD0 RXD1 TXD0 TXD1	REV RND DATA READ RESTORE

Runtime Library



Runtime Library

- Math Functions
- String Functions

Byte and HalfWord Functions

RD_BYTE

Syntax

RD_BYTE(*address*)

Description

The value of the byte at *address* is returned. That value is unsigned, so 0 to 255 can be expected.

This is a compiler change and is a feature of compilers 9.36 and later.

Example

```
x = RD_BYTE(&H10000000)     ' load the byte at &H10000000 into x
```

Differences from other BASICs

- no equivalent in Visual BASIC
- no equivalent in PBASIC

See also

- RD_HALF
- WR_BYTE
- WR_HALF
- * WORD ACCESS

RD_HALF

Syntax

RD_HALF (*address*)

Description

The value of the halfword at *address* is returned. That value is unsigned, so 0 to 65535 can be expected.

This is a compiler change and is a feature of compilers 9.36 and later.

Some ARMs allow unaligned accesses, but others that causes a memory fault. The LPC17xx and LPC4xxx allow un-aligned access.

Example

```
x = RD_HALF(&H10000000)     ' load the 16 bit value at &H10000000  and &H10000001 into x
```

Differences from other BASICs

- no equivalent in Visual BASIC
- no equivalent in PBASIC

See also

- RD_BYTE
- WR_BYTE
- WR_HALF
- * WORD ACCESS

WR_BYTE

Syntax

WR_BYTE(*address*, *number*)

Description

The value of the *number* is writen into the byte at *address* .

This is a compiler change and is a feature of compilers 9.36 and later.

Some ARMs allow unaligned accesses, but others that causes a memory fault. The LPC17xx and LPC4xxx allow un-aligned access.

Example

```
WR_BYTE(&H10000000, x)    ' load the 8 bit value of x into &H10000000
```

Differences from other BASICs

- no equivalent in Visual BASIC
- no equivalent in PBASIC

See also

- RD_BYTE
- RD_HALF
- WR_HALF
- * WORD ACCESS

WR_HALF

Syntax

WR_HALF (*address* , *number*)

Description

The value of the *number* is writen into the 2 bytes at *address* .

This is a compiler change and is a feature of compilers 9.36 and later.

Some ARMs allow unaligned accesses, but others that causes a memory fault. The LPC17xx and LPC4xxx allow un-aligned access.

Example

```
WR_HALF(&H10000000, x)    ' load the 16 bit value of x into &H10000000 and &H10000001
```

Differences from other BASICs

- no equivalent in Visual BASIC
- no equivalent in PBASIC

See also

- RD_BYTE
- RD_HALF
- WR_BYTE
- * WORD ACCESS

Mathematical Functions

Mathematical Functions

- ABS
- MOD

ABS

Syntax

ABS (*number*)

Description

The absolute value of a number is its unsigned magnitude. For example, ABS(-1) and ABS(1) both return 1. The required *number* argument can be any valid numeric expression. If *number* is an uninitialized variable, zero is returned.

WIth firmware 8.15 support for floating point has been added and is valid for SINGLE variables, in fact it's the one case where floating point is faster than integer.

Example

```
PRINT ABS ( -1 )
PRINT ABS ( 42 )
PRINT ABS ( N )

N = -69

PRINT ABS ( N )
```

The output would look like:

```
1
42
0
69
```

Differences from other BASICs

- none from Visual BASIC
- none from PBASIC

See also

- **OR**
- **XOR**
- **NOT**

MOD

Syntax

argument1 MOD *argument2*

Description

MOD is the modulus or "remainder" arithmetic operator. The result of MOD is the integer remainder of *argument1* divided by *argument2*.

Both *arguments* should be integers, MOD is not defined for floating point values.

Example

```
PRINT 47 MOD 7
PRINT 56 MOD 2
PRINT 5 MOD 3
```

The output would look like:

```
5
0
2
```

Differences from other BASICs

- none from Visual BASIC
- PBASIC uses //

See also

File operations (SD card)

These functions are implemented in versions of BASIC for the DataLogger and Teensy3.0/3.1. Other versions on Coridium boards are currently under development.

The current version limits the use to having only 1 file open at a time, and only operations in the root directory. File names are limited of up to 8 characters period followed by up to 3 characters. Spaces are NOT allowed in filenames.

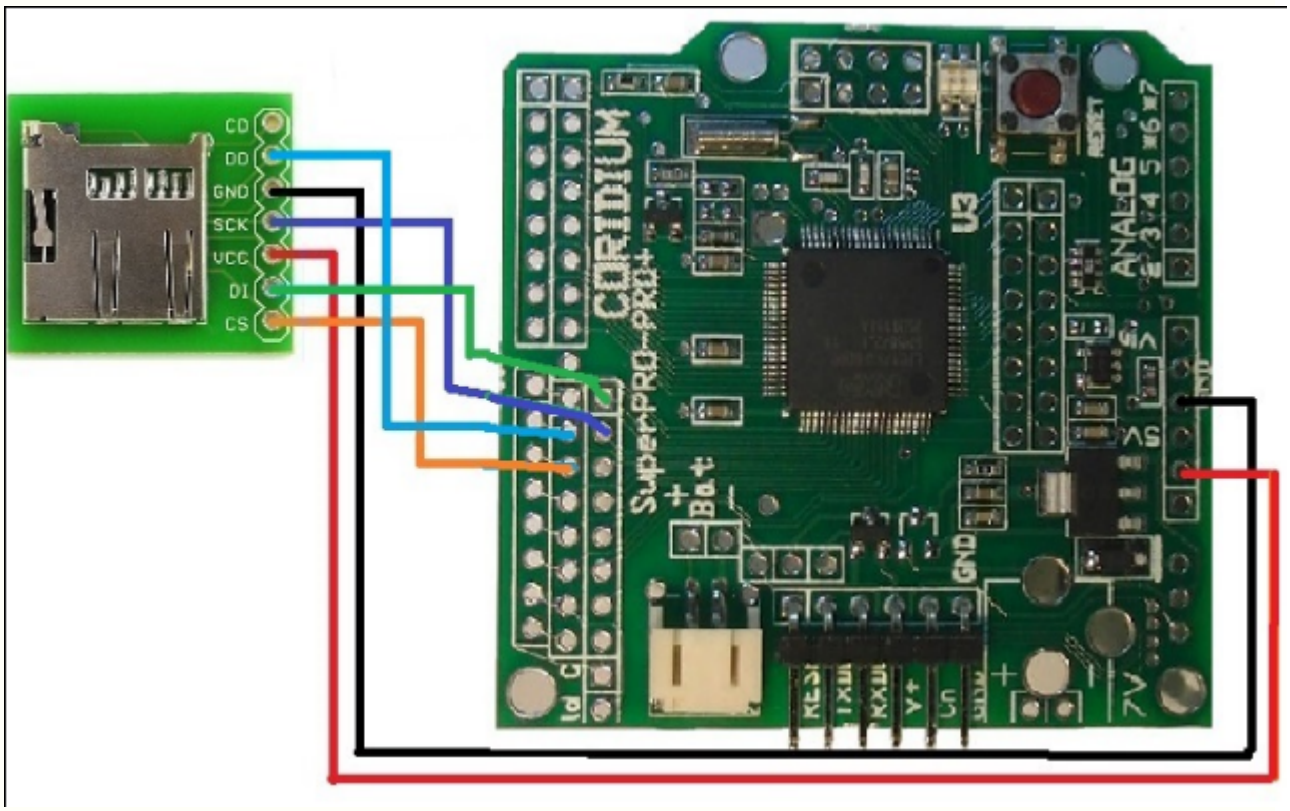
File Functions

- FILEOPEN
- FILEREADDIR
- FILEREADBYTE
- FILEWRITEBYTE
- FILEEOF
- FILECLOSE

Data Logger

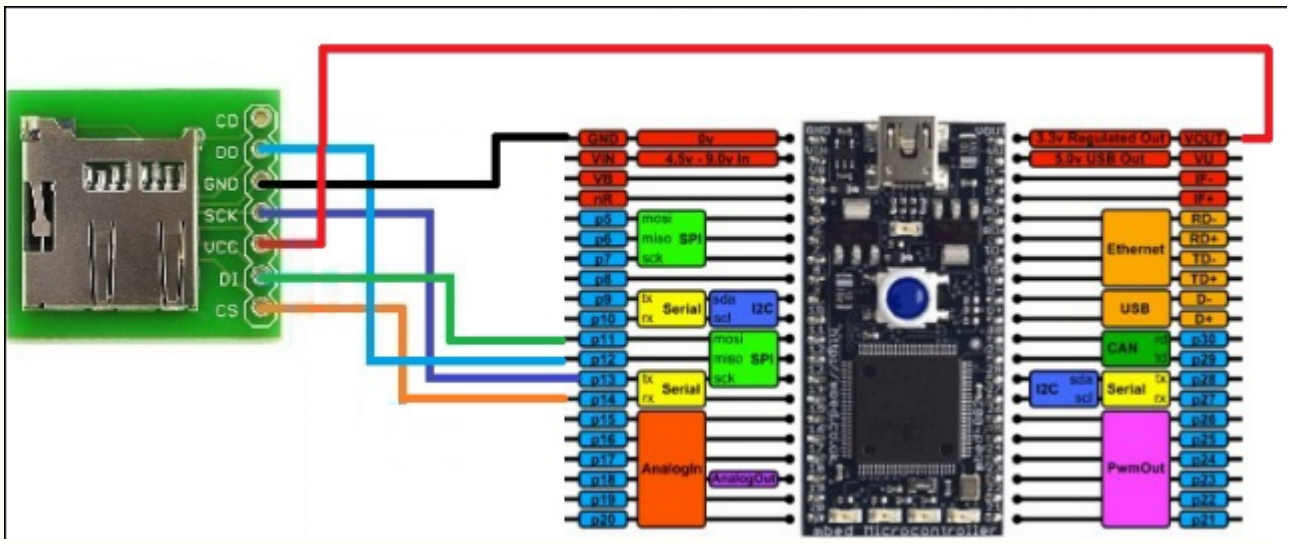
The LPC4330 based DataLogger has an SDcard socket and supports the File operations.

Connections for SuperPRO (unreleased alpha version)



-
  SD DI -- MOSI P0.18
- SD DO -- MISO P0.17
- SD SCK -- SCK P0.15
- SD CS -- SSEL P0.16

Connections for mBed

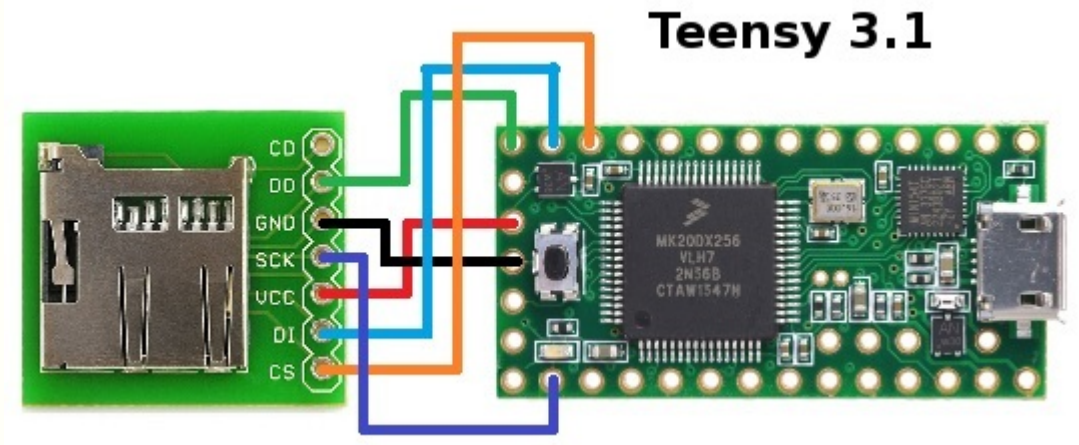


Connections for ArchPro

The connections are made on the SPI connector

-
  SD DI -- MOSI P0.18
- SD DO -- MISO P0.17
- SD SCK -- SCK P0.15
- SD CS -- SSEL P0.16

Connections for Teensy



FILEOPEN

Syntax

FUNCTION FILEOPEN(*"filenameString", param*)

Description

FILEOPEN is built into version 8.30 and later.

The *filename* is a string. File names are limited of up to 8 characters period followed by up to 3 characters. Spaces are NOT allowed in filenames

param is a single character "r" for read, "w" for write, and "a" for append, single characters are treated like the integer value of the ASCII character.

The function returns 1 for a successful open.

In this version ONLY ONE file can be open at a time.

Error codes:

-1 FR_NOT_READY
-2 FR_NO_FILE
-3 FR_NO_PATH
-4 FR_INVALID_NAME
-6 FR_DENIED
-7 FR_EXIST
-8 FR_RW_ERROR
-11 FR_NO_FILESYSTEM
-12 FR_INVALID_OBJECT

Example

```
i = fileopen("logfile.txt", "w")

if i <> 1 then
  printf("OPEN FAILED %d, TRY AGAIN", i)
  i = fileopen("logfile.txt", "w")                    ' try again
endif
```

Differences from other BASICs

- no equivalent Visual BASIC
- no equivalent in PBASIC

See also

- **FILEREADDIR**
- **FILEWRITEBYTE**
- **FILEREADBYTE**
- **FILEEOF**
- **FILECLOSE**

FILEREADDIR

Syntax

FUNCTION FILEREADDIR(*filenumber*) AS STRING

Description

FILEREADDIR is built into version 8.30 and later.

Reads the Nth entry in the directory file list.
Returns the file name or an empty string if there are no more files.

Example

```
dim file_number as integer
dim file_name(16) as string

file_number = 0
file_name = ""

do
 file_name = filereaddir(file_number)
 if file_name <> ""
        print "file "; file_number; " is "; file_name
 endif
 file_number += 1
loop until file_name = ""

```

Differences from other BASICs

- no equivalent Visual BASIC
- no equivalent in PBASIC

See also

- **FILEOPEN**
- **FILEWRITEBYTE**
- **FILEREADBYTE**
- **FILEEOF**
- **FILECLOSE**

FILEREADBYTE

Syntax

FUNCTION FILEREADBYTE(*reserved*)

Description

FILEREADBYTE is built into version 8.30 and later.

This version returns the next byte in the file, the parameter *reserved* is not used. The byte value can be 0 to 255, a value of -1 is returned on error.

Example

```
i = fileopen("logfile.txt", "r")

j=0
while(fileeof(0) = 0 )
 j += 1
 i= filereadbyte(0)
 if i<0
  printf("filereadbyte() failed %d", i)
  goto READ_FAILED
 else
  print chr(i);
 endif
loop

READ_FAILED:
 fileclose(0)
 print
 printf("Read %d bytes",j)
```

Differences from other BASICs

- no equivalent Visual BASIC
- no equivalent in PBASIC

See also

- **FILEOPEN**
- **FILEWRITEBYTE**
- **FILEREADDIR**
- **FILEEOF**
- **FILECLOSE**

FILEWRITEBYTE

Syntax

FUNCTION FILEWRITEBYTE(*character, reserved*)

Description

FILEWRITEBYTE is built into version 8.30 and later.

Writes a character into the open file.

Returns 1 on success or error codes listed in FILEOPEN

Example

```
dim buffer(32) as string

printf("SD Card Example Program")

i = fileopen("logfile2.txt", "w")

if i=1
  buffer = "abcdefghijklmnoa" + chr(13)+chr(10)
  for j=0 to len(buffer)-1
   i = filewritebyte(buffer(j),0)
   if i < 0
    printf("filewritebyte() failed %d", i)
    goto WRITE_FAILED
   endif
  next j

WRITE_FAILED:
  fileclose(0)
  printf("Write %d bytes",j)
 endif
```

Differences from other BASICs

- no equivalent Visual BASIC
- no equivalent in PBASIC

See also

- **FILEOPEN**
- **FILEREADDIR**
- **FILEREADBYTE**
- **FILEEOF**
- **FILECLOSE**

FILEEOF

Syntax

FUNCTION FILEOF(*reserved*) as INTEGER

Description

FILEOF is built into version 8.30 and later.

FILEOF returns 1 when the end of file has been read.

reserved is ignored for now, but may be used in a future version.

Example

See **FILEREADBYTE**

Differences from other BASICs

- no equivalent Visual BASIC
- no equivalent in PBASIC

See also

- **FILEOPEN**
- **FILEWRITEBYTE**
- **FILEREADBYTE**
- **FILEREADDIR**
- **FILECLOSE**

FILECLOSE

Syntax

SUB FILECLOSE(*reserved*)

Description

FILECLOSE is built into version 8.30 and later.

FILECLOSE closes the currently open file . In this version ONLY ONE file may be open at a time

reserved may be used in a future version, and is ignored for now.

Example

```
fileclose(0)
```

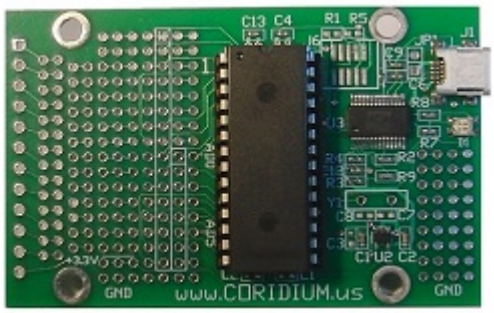
Differences from other BASICs

- no equivalent Visual BASIC
- no equivalent in PBASIC

See also

- **FILEOPEN**
- **FILEWRITEBYTE**
- **FILEREADBYTE**
- **FILEEOF**
- **FILEREADDIR**

User Contributed Floating Point Functions



These functions are implemented using a library written in BASIC that is included at compile time. This library was contributed by Tod W.

Mathematical Functions

- Trig functions
- Inverse Trig
- Hyperbolic functions
- Power functions
- Misc function
- Random Numbers

Trigonometric functions

Syntax

```
#define include_trig
#include <AB_Math.bas>

sin ( radians)

cos ( radians )

tan ( radians )
```

Description

Part of a user contributed library that computes the standard trigonometric functions.

Example

```
#define include_trig
#include <AB_Math.bas>

for i=0 to 90 step 10
  print i,sin(_deg2rad *i),cos(_deg2rad *i),tan((_deg2rad *i)
next i
```

The output would look like:

```
0 0.000000 1.0000000 0.000000
10 0.1736482 0.9848078 0.1763270
20 0.3420201 0.9396926 0.3639701
30 0.5000000 0.8660253 0.5773502
40 0.6427875 0.7660430 0.8390993
50 0.7660430 0.6427875 1.191754
60 0.8660253 0.5000000 1.732051
70 0.9396926 0.3420202 2.747478
80 0.9848078 0.1736483 5.671279
90 1.0000000 0.000000 Inf
```

See also

- Inverse Trig
- Hyperbolic

Inverse Trigonometric functions

Syntax

```
#define include_inverse_trig
#include <AB_Math.bas>

asin ( x)

acos ( x )

atan ( x )

atan2(y , x, hypot)          ' hypot is called by reference and updated by the call

sec ( x)

csc (x)

ctan (x)

asec(x)

acsc(x)

acot(x)
```

Description

Part of a user contributed library that computes the standard trigonometric functions.

Example

```
#define include_inverse_trig
#include <AB_Math.bas>

for i= 0 to 10
  print i,sec(i/10)
next i
```

See also

- Trig functions
- Hyperbolic Trig

Hyperbolic Trigonometric functions

Syntax

```
#define include_hyper
#include <AB_Math.bas>
```

cosh (*radians*)

sinh (*radians*)

tanh (*radians*)

asinh (*x*)

acosh (*x*)

atanh (*x*)

sech (x)

csch (x)

ctanh (x)

asech(x)

acsch(x)

acoth(x)

Description

Part of a user contributed library that computes the standard trigonometric functions.

Example

```
#define include_hyper
#include <AB_Math.bas>

for i= 0 to 90 step 10
  print i,sinh(i/10)
next i
```

See also

- Trig functions
- Hyperbolic

Power functions

Syntax

```
#define include_power_exp_log_sqrt
#include <AB_Math.bas>

sqrt( x)

exp( x )     ' return 2.71828 raised to x power

logn( x )    ' natural log

pow(x, p)  ' returns x raised to p power -- both SINGLE

log ( x )    '  returns value 10 musted be raised to to produce x
```

Description

Part of a user contributed library that computes the standard trigonometric functions.

Example

```
#define include_power_exp_log_sqrt
#include <AB_Math.bas>

for i= 0 to 10
  print i,sqrt(i)
next i
```

See also

- Trig functions
- Hyperbolic Trig

Miscellaneous functions

Syntax

```
#include <AB_Math.bas>

fix( x)       ' return integer value of the SINGLE x

floor( x )    '  return integer <= x

ceil( x )     '  return integer >= x

powi(x, p)   '  returns x raised to INTEGER p power

round( x )   '   returns integer value closest to x
```

Description

Part of a user contributed library that computes the standard trigonometric functions.

Example

```
#include <AB_Math.bas>

for i= 0 to 10
  print i,powi(2.345, i)
next i
```

See also

- Trig functions
- Hyperbolic Trig

Random Numbers

Syntax

```
#define include_rand
#include <AB_Math.bas>

init_random

rand

randb (low, high)
```

Description

Part of a user contributed library that computes the standard trigonometric functions.

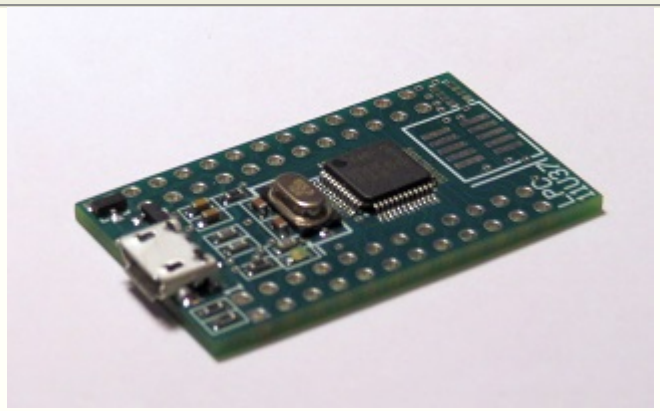
Example

```
#define include_rand
#include <AB_Math.bas>

for i= 1 to 10
  print i, rand, randb (0.2,  5.1)
next i
```

See also

- Trig functions
- Hyperbolic

String Functions



Built-in String Functions	VBSTRING.bas Library (VB style)
CHR	MID
HEX	INSTR
LEFT	UCASE
LEN	LCASE
RIGHT	
STR	STRING.bas Library (C style)
STRCOMP	MIDSTR
VAL	STRCHR
	STRSTR
	TOLOWER
	TOUPPER

String functions may not be nested. What does this mean?

String functions are built using a string accumulator which is a 256 byte buffer. There is only one string accumulator due to memory constraints. The general expression evaluation for integers involves a stack, but it is impractical to have a string stack. So when a string is built from an expression, it uses this string accumulator. String FUNCTIONs also use this string accumulator to return the string value. So string FUNCTIONs can not be used after the first operand in a string expression.

As there is only the single string accumulator, string functions should NOT be performed inside INTERRUPT SUB.

String expressions are parsed left to right, and parenthesis for grouping are not allowed as that is the equivalent of nesting. However a string expression can have any number of strings being combined into a single string. So the following is proper-

```
DIM ast(30) as string
DIM bst(30) as string
DIM cst(30) as string

ast = ast + "abcd" + str(2 + 44 / 33) + str(len(cst)) + "zcxv" + chr(13) + "more stuff" + bst
```

The chr(13) inserts a carriage return into this string so it spans 2 lines. This is proper as strings only have two limitations. First that they are less that 256 bytes, and they are terminated by a 0 or null character.

Note that the str(2 + 44 / 33) involves the integer evaluation stack and is OK as that is a separate entity. Also the str(len(cst)) is valid as that involves a string as stored in memory.

What would not be allowed is something like

ast = "length is " + str(len(cst + bst)) ' THIS IS INVALID NESTING

because cst + bst would have to be evaluated before ast could be built, and there is no room to do that.

ast = "length is " + str(len(cst) + len(bst)) ' allowed as len is called with simple pointers

User FUNCTIONs

Now with the addition of user defined functions, there is the possibility of a nested string function that the compiler can not detect. If a string expression calls a user function, and that user function does ANY string expressions or PRINT statements; then this is a nested string operation. The compiler will not be able to detect this, and its possible to get unexpected string results or even data abort errors.

ast = user_string_function (1,2,3) ' is OK

ast = str (user_integer_function (1,2,3)) ' is OK

ast = "result of " + user_string_function (1,2,3) ' INVALID string nesting

ast = "result of "+ str(user_integer_function (1,2,3)) ' valid only if no string op or PRINT statement in user_integer_function

ast = user_string_function (1,2,3) + " returned" ' is OK, as the string function was the first called

ast = str(user_int_function (1,2,3)) + " returned" ' is OK, as the user function was the first called

VB vs. C style String Functions

VB accesses the first character by Stringname.Chars(0) In ARMbasic that first character is accessed by Stringname(0)

But VB's MID ("This is a string",1,3) returns "Thi".

The existing library of string functions was translated from C, which is always 0 based for the first element. So

MIDSTR ("This is a string",1,3) returns "his"

String Comparisons

Syntax

string1 compare_op string2

string1 = string-variable | byref_string_pointer | string_constant
compare_op = > | >= | = | <> | =< | <
string2 = string1_types | string_functions

Description

This compares the two strings returning -1 if *string1* satisfies the *comparison_op* with *string2*. Returning 0 if the *comparison_op* is not true.

String1 and *String2* may be constant or variable strings. *String2* may also be a FUNCTION of type STRING.

Example

```
DIM text(10) as STRING

text = "BAT"
PRINT text > "BBB"        ' will display 0
PRINT "BBB" <= text       ' will display 0
PRINT text < "BOOT"       ' will display 1

PRINT text > "BAA"        ' will display 1
```

Differences from other BASICs

- similar to Visual BASIC
- no equivalent in PBASIC

See also

- **CHR**
- **STR**
- **VAL**

CHR

Syntax

CHR(*expression*)

Description

CHR returns a single byte string containing the character represented by the **ASCII** code passed to it. For example, CHR(97) returns "a".

Note:
There is no need for a complimentary function, as that type conversion is automatic, see sample code below.

Example

```
PRINT "the character represented by the ASCII code of 97 is:"; CHR(97)    ' will print   a

DIM a(10) as STRING  ' examples of automatic type conversion complimentary to CHR
a ="asdf"

PRINT a(0), chr(a(0))                ' will print    97     a
x = a(0)
PRINT x                               ' will print    97
if x = "a" then PRINT "it is a"        ' will print it is a
```

Differences from other BASICs

- does not exist in PBASIC
- same function exists in Visual BASIC

See also

- **STR**
- **HEX**
- **VAL**

HEX

Syntax

HEX (*expression*)

Description

This returns the hexadecimal string representation of the INTEGER *expression* . Hexadecimal values contain 0-9, and A-F. The size of the result string depends on the integer type passed, it's not fixed.

HEX is not defined for use with SINGLE types

Example

```
DIM text(10) as String

text = HEX(4096)
PRINT "0x";text     ' will display 0x1000
```

Differences from other BASICs

- same function as Visual BASIC
- similar to PBASIC format directive available in SHIFTIN, SERIN, DEBUGIN

See also

- **CHR**
- **STR**
- **VAL**

INSTR ' VB style

Syntax

#include <VBSTRING.bas> ' source in /Program Files/Coridium/BASICclib

FUNCTION INSTR(*start* , *searchee, lookfor*)

Description

This FUNCTION written in BASIC searches the string *searchee* looking for the string *lookfor,* starting at the *start*-th character.

If it is found, the position of the first character of *lookfor* in *searchee* is returned, otherwise 0.

start is based on 1 being the first character, which is consistent with the InStr VB function, but inconsistent with the VB *searchee*.Chars(0) being the first character. The C style STRSTR version of this routine uses 0 as the first character.

Example

```
#include <VBSTRING.bas>

DIM text(10)

text = "HELLO WORLD"
PRINT INSTR(1, text, "LLO")     ' will display  3
```

Differences from other BASICs

- similar function as Visual BASIC
- no equivalent in PBASIC

See also

- UCASE
- MID

LCASE

Syntax

```
#include <VBSTRING.bas>                    ' source in /Program Files/Coridium/BASICclib

FUNCTION LCASE( string ) as STRING
```

Description

This FUNCTION written in BASIC shifts the letters of *string* to lower case .*String* may be a constant or variable string.

Example

```
#include <VBSTRING.bas>

DIM text(10)

text = "HELLO WORLD"
PRINT LCASE(text)     ' will display  hello world
```

Differences from other BASICs

- similar function as Visual BASIC
- no equivalent in PBASIC

See also

- **UCASE**
- **INSTR**

LEFT

Syntax

LEFT(*string*, *characters*)

Description

Returns n-*characters* starting from the left of *string*. *String* may be a constant or variable string.

String functions may not be nested.

```
Astring = LEFT("this is a test",5) + RIGHT(Bstring,3)      ' valid string operation

Astring = LEFT( "this "+bstring,5)                         ' NOT ALLOWED nested operation
```

Example

```
DIM text(20) as STRING
text = "hello world"
PRINT LEFT(text, 5) 'displays "hello"
```

Differences from other BASICs

- none from Visual BASIC
- no equivalent in PBASIC

See also

- RIGHT
- LEN

LEN

Syntax

LEN(*string*)

Description

This returns the length of *string* in characters.*String* may be a constant or variable string.

String functions may not be nested.

Example

```
DIM text$(10) as STRING

text = "0x"+HEX(4096)
PRINT LEN(text)     ' will display 6
```

Differences from other BASICs

- same function as Visual BASIC
- no equivalent in PBASIC

See also

- **CHR**
- **STR**
- **VAL**

MID ' VB style

Syntax

#include <VBSTRING.bas> ' source in /Program Files/Coridium/BASIClib

FUNCTION MID(*string* , *start*, *length*) as STRING

Description

This FUNCTION written in BASIC returning the portion of *string* from the *start* character for *length* characters.

String may be a constant or variable string.

start is based on 1 being the first character, which is consistent with the MID VB function, but inconsistent with the VB search.chars(0) being the first character. The C style MIDSTR version of this routine uses 0 as the first character.

Extracting or setting a single byte in a string can be done with an index STRING(3) refers to the 4th byte of the string (starts from 0).

Example

```
#include <VBSTRING.bas>

DIM text(10)

text = "HELLO WORLD"
PRINT MID(text, 4,5)     ' will display  LO WO
```

Differences from other BASICs

- similar function as Visual BASIC
- no equivalent in PBASIC

See also

- **UCASE**
- **INSTR**

MIDSTR ' C style

Syntax

```
#include <STRING.bas>                    ' source in /Program Files/Coridium/BASIClib

FUNCTION MIDSTR( string , start, length ) as STRING
```

Description

This FUNCTION written in BASIC returning the portion of *string* from the *start* character for *length* characters.

String may be a constant or variable string.

MIDSTR is written in C style with 0 being the first character of the *string*, consistent with VB *string* .Chars(0).

Extracting or setting a single byte in a string can be done with an index STRING(3) refers to the 4th byte of the string (starts from 0).

Example

```
#include <STRING.bas>

DIM text(10)

text = "HELLO WORLD"
PRINT MIDSTR(text, 4,5)     ' will display  O WOR
```

Differences from other BASICs

- similar function as Visual BASIC
- no equivalent in PBASIC

See also

- **TOUPPER**
- **STRSTR**

PRINTF

Syntax

SUB PRINTF ("C format string", ParamArray ...)

built-in SUB that calls the existing printf routine in the firmware for versions 8.11 or later (version 7.51 or later on ARM7 parts)

Description

PRINTF is a way to format printed output. It can be as simple a short string or quite complex specifying field sizes and pad characters for a number. This routine is written in C and is part of the firmware. As of 8.11 firmware BASIC programs can access this powerful C subroutine. But the BASIC compiler just passes the parameters to the C and does NO type checking or even counts of parameters.

Rather than duplicate the documentation of PRINTF, there is a great deal of information for printf in documentation for C such as

C reference for printf

PRINTF in the firmware supports c (BYTE), s (STRING), d (decimal INTEGER), x or X for (hex INTEGER), e, E, f, g, or G (for SINGLE).

The G format is used by the ARMbasic PRINT, it differs a little from the G format of C. It displays 0.0 or the range 0.1000000 through 9999999. and outside that it uses D.DDDDDDE+DD and the negative values as well.

Pad characters of 0 or space can be added to fill out LENGTH or WIDTH and PRECISION. PRECISION is limited to 0 to 8 characters.

To add CR <carriage return> to the output printed, either use PRINT or print a %c with a value of 13

Example

```
PRINTF("a simple string")

i = 1234
PRINTF(" %d",i)                    ' print a decimal integer

DIM x AS SINGLE
x = 12.34
PRINTF(" %1.4f",x)                 ' print a floating point number with 4 digits of precision

PRINTF("%c%c a string a single %1.5f an integer %d a hex number %x and a carriage return
%c",13,13,x,i,i,13)
```

will print out --

a simple string 1234 12.3400

a string a single 12.34000 an integer 1234 a hex number 4D2 and a carriage return

Differences from other BASICs

- similar to String.Format in Visual BASIC
- no equivalent in PBASIC

See also

- **SPRINTF**

RIGHT

Syntax

RIGHT(*string*, *characters*)

Description

Returns n-*characters* starting from the right of the *string*. *String* may be a constant or variable string.

String functions may not be nested.

```
A = LEFT("this is a test",5) + RIGHT(B,3)      ' valid string operation

A = RIGHT("this "+b,5)                         ' NOT ALLOWED nested operation
```

Example

```
DIM text(20) as string

text = "hello world"
PRINT RIGHT(text, 5)      'displays "world"
```

Differences from other BASICs

- this function does not exist in PBASIC
- similar function to Visual BASIC

See also

- LEFT

Single Byte access

Syntax

someString(index)

Description

A string is just an array of bytes, terminated by a 0. Strings are limited to 256 characters (no bounds checking). So individual bytes can be accessed like individual elements in an array.

Extracting or setting a single byte in a string can be done with an index STRING(3) refers to the 4th byte of the string (starts from 0).

Example

```
DIM text(10) as string

text(0) = chr("H")        ' single character strings like "H" are treated like a character constant
text(1) = chr( "E")
text(2) = chr( "L")
text(3) = text(2)         ' copy the previous character
text(4) = text(3) + 3     ' expressions are OK too, the result is truncated to 8 bits
text(5) = 0

PRINT text     ' will display  HELLO
```

Differences from other BASICs

- same as Visual BASIC
- same as PBASIC

See also

- **UCASE**
- **INSTR**

SPRINTF

Syntax

FUNCTION SPRINTF ("C format string", ParamArray ...) AS STRING

built-in FUNCTION that calls the existing SPRINTF routine in the firmware for versions 8.11 or later (version 7.51 or later on ARM7 parts)

Description

SPRINTF is a way to format strings. It can be as simple a short string or quite complex specifying field sizes and pad characters for a number. This routine is written in C and is part of the firmware. As of 8.11 firmware BASIC programs can access this powerful C subroutine. But the BASIC compiler just passes the parameters to the C and does NO type checking or even counts of parameters.

Rather than duplicate the documentation of SPRINTF, there is a great deal of information for printf in documentation for C such as

C reference for printf

SPRINTF in the firmware supports c (BYTE), s (STRING), d (decimal INTEGER, x or X for (hex INTEGER), e, E, f, g, or G (for SINGLE).

The G format differs a little from the G format of C. It copies 0.0 or the range 0.1000000 through 9999999. and outside that it uses D.DDDDDDE+DD and the negative values as well.

Pad characters of 0 or space can be added to fill out LENGTH or WIDTH and PRECISION. PRECISION is limited to 0 to 8 characters.

Special characters can be added using %c and filling that value with something like 13 for carriage return, 9 for TAB or 7 to wring the BELL

Example

```
DIM str(100) AS STRING

i = 1234

DIM x AS SINGLE
x = 12.34

str = SPRINTF("%c%c a string a single %1.5f an integer %d a hex number %x and a carriage return %c",13,13,x,i,i,13)

PRINT str
```

will print --

a string a single 12.34000 an integer 1234 a hex number 4D2 and a carriage return

Differences from other BASICs

- similar to String.Format in Visual BASIC
- no equivalent in PBASIC

See also

- **PRINTF**

STR

Syntax

STR(*expression*)

Description

STR will convert an integer *expression* into a string.

For example, STR(3) will become "3", or STR(333) will become "333".

Incidentally, this is the opposite of the **VAL** function, which converts a string into a number.

STR is also used in certain routines of the Hardware Library to designate that a series of bytes should be read or written to a string.

Also in the following case the STR function is implied and is not required.

bstring = 333 + " sent" ' will save the ASCI string "333 sent" into b$

The implied STR will work for simple expressions, but anything complex should use STR(), this would include any function call, array element fetches.

For floating point numbers use SPRINTF.

Example

```
DIM bstring (10) AS STRING
a = 8421
bstring = STR(a)
PRINT a, b     ' will display    8421     8421
```

Differences from other BASICs

- same function in Visual BASIC
- similar to DEC formatting function in PBASIC

See also

- **VAL**
- **CHR**
- **HEX**
- **Hardware Library, Function List**

STRCHR ' C style

Syntax

#include <STRING.bas> ' source in /Program Files/Coridium/BASICIib

FUNCTION STRCHR(*string* , *char*)

Description

This FUNCTION written in BASIC searches *string* looking for the first instance of *char* .*String* may be a constant or variable string.
If *char* is not found -1 is returned, otherwise the position of *char*.

STRCHR is written in C style with 0 being the first character of the *string*, consistent with VB *string* .Chars(0).

Example

```
#include <STRING.bas>

DIM text(10)

text = "HELLO WORLD"
PRINT STRCHR(text, "W")     ' will display  6
```

Differences from other BASICs

- similar function as Visual BASIC
- no equivalent in PBASIC

See also

- **TOUPPER**
- **STRSTR**

STRCOMP

Syntax

STRCOMP(*string1, string2*)

Description

This compares the two strings returning -1 if *string1 would sort before string2*. Returning 0 if the two strings are equal, and 1 if *string1 would sort after string2.*

String1 and *String2* may be constant or variable strings.

String functions may not be nested.

Example

```
DIM text(10) as STRING

text = "BAT"
PRINT STRCOMP(text, text)          ' will display 0
PRINT STRCOMP(text, "BAT")         ' will display 0 )
PRINT STRCOMP(text, "BOOT")        ' will display -1 )

PRINT STRCOMP(text, "BAA")         ' will display 1
```

Differences from other BASICs

- same function as Visual BASIC
- no equivalent in PBASIC

See also

- **CHR**
- **STR**
- **VAL**

STRSTR ' C style

Syntax

#include <STRING.bas> ' source in /Program Files/Coridium/BASICclib

FUNCTION STRSTR(*searchee, lookfor*)

Description

This FUNCTION written in BASIC searches the string *searchee* looking for the string *lookfor*.

If it is found, the position of the first character of *lookfor* in *searchee* is returned, otherwise -1.

STRSTR is written in C style with 0 being the first character of the *string*, consistent with VB *string* .Chars(0).

Example

```
#include <STRING.bas>

DIM text(10)

text = "HELLO WORLD"
PRINT STRSTR(text, "LLO")     ' will display  2
```

Differences from other BASICs

- similar function as Visual BASIC
- no equivalent in PBASIC

See also

- TOUPPER
- STRCHR

TOLOWER

Syntax

```
#include <STRING.bas>                    ' source in /Program Files/Coridium/BASIClib

FUNCTION TOLOWER( string ) as STRING
```

Description

This FUNCTION written in BASIC shifts the letters of *string* to lower case .*String* may be a constant or variable string.

Example

```
#include <STRING.bas>

DIM text(10)

text = "HELLO WORLD"
PRINT TOLOWER(text)     ' will display  hello world
```

Differences from other BASICs

- similar function as Visual BASIC
- no equivalent in PBASIC

See also

- **TOUPPER**
- **STRSTR**

TOUPPER

Syntax

#include <STRING.bas> ' source in /Program Files/Coridium/BASIClib

FUNCTION TOUPPER(*string*) as STRING

Description

This FUNCTION written in BASIC up shifts the letters of *string* .*String* may be a constant or variable string.

Example

```
#include <STRING.bas>

DIM text(10)

text = "hello world"
PRINT TOUPPER(text)     ' will display  HELLO WORLD
```

Differences from other BASICs

- similar function as Visual BASIC
- no equivalent in PBASIC

See also

- **TOLOWER**
- **STRSTR**

UCASE

Syntax

```
#include <VBSTRING.bas>                    ' source in /Program Files/Coridium/BASIClib

FUNCTION UCASE( string ) as STRING
```

Description

This FUNCTION written in BASIC up shifts the letters of *string* .*String* may be a constant or variable string.

Example

```
#include <VBSTRING.bas>

DIM text(10)

text = "hello world"
PRINT UCASE(text)     ' will display  HELLO WORLD
```

Differences from other BASICs

- similar function as Visual BASIC
- no equivalent in PBASIC

See also

- **LCASE**
- **INSTR**

VAL

Syntax

VAL(*string*)

Description

VAL converts a *string* to a decimal number. For example, VAL("10") will return 10. The function parses the string from the left and returns the longest number it can read, stopping at the first non-suitable character it finds.

Incidentally, this function is the opposite of **STR** , which converts a number to a string.

string can contain negative numbers or hex numbers when preceded by a $.

VAL does only INTEGER interpretation

Example

```
DIM astring(20) as STRING
astring = "20xa211"
b = VAL(astring)
PRINT astring, b
```

```
20xa211   20
```

Differences from other BASICs

- similar to Visual BASIC -- VB ignores space and tab characters during the parsing and allows &H or &O
- similar to formatting directives DEC, HEX in PBASIC

See also

- **STR**
- **HEX**
- **CHR**

pointers

* (ARM peripheral access)

equivalent of PEEK and POKE

Syntax

* *variable*

* *constant*

* (*expression)* ' added in version 8.04 of the compiler

Description

The C pointer syntax is used to give direct access to the ARM peripheral registers and memory.

x = **addr* ' equivalent of PEEK(*addr*) for a word

**addr* = x ' equivalent of POKE(*addr*,x) writes x as a word into address *addr*

This gives the programmer the ability to directly control the ARM hardware. Details on what the registers do can be found in the NXP User Manuals for the corresponding chip (LPC2103 for ARMmite, ARMexpress LITE, PRO, LPC2106 for ARMexpress, LPC2138 for ARMweb, and LPC1751/6 for the PROplus and SuperPRO)

Examples of programming the registers can be found in the BASIClib directory which contains sub-programs that control various hardware functions.

If you prefer the PEEK or POKE versions you can always use the pre-processor

```
#define PEEK(addr)   (*(addr))      ' the () ensure only addr is used in the address calculation
#define POKE(addr,value)   *(addr) = value
```

Pointers only operate on 32 bit WORD values, and should use WORD or 4-byte alligned addresses.

For byte and halfword access see **RD_BYTE** , **WR_BYTE** , **RD_HALF** and **WR_HALF**

Example

```
' from the HWPWM.bas  library

'* ---- Timer 2 ----------------------------------------
#define T2_TCR        * &HE0070004
#define T2_TC         * &HE0070008
#define T2_PR         * &HE007000C
#define T2_MCR        * &HE0070014
#define T2_MR0        * &HE0070018
#define T2_MR1        * &HE007001C
#define T2_MR2        * &HE0070020
#define T2_MR3        * &HE0070024

 T2_PR  = prescale
 T2_TCR = TxTCR_COUNTER_ENABLE          ' Timer1 Enable

 T2_MR3 = cycletime -1
```

```
T2_MCR = &H400    ' rollover when count reaches MR3
```

Differences from other BASICs

- No equivalent in Visual BASIC
- no direct equivalent in PBASIC, CONFIGPIN is a similar function

See also

- **CPU details**
- **ADDRESSOF**
- **RD_BYTE**
- **RD_HALF**
- **WR_BYTE**
- **WR_HALF**

QUICKSORT

Syntax

SUB QUICKSORT(*buffer, number_of_elements*)

Description

QUICKSORT is built into version 8.30 and later.

This version of QuickSort, will sort *number_of_elements* integers in *buffer.*

Example

```
#define MAX_NUMBERS 100

dim numbers(MAX_NUMBERS)   ' BASIC arrays have elements 0 to MAX_NUMBERS
for i=0 to MAX_NUMBERS
  numbers (i)=MAX_NUMBERS-i
next
for i= 0to 10
  print numbers(i)
next
print
print "now sort the numbers"
quicksort(addressof numbers, MAX_NUMBERS+1)
for i=0 to 10
  print numbers(i)
```

next

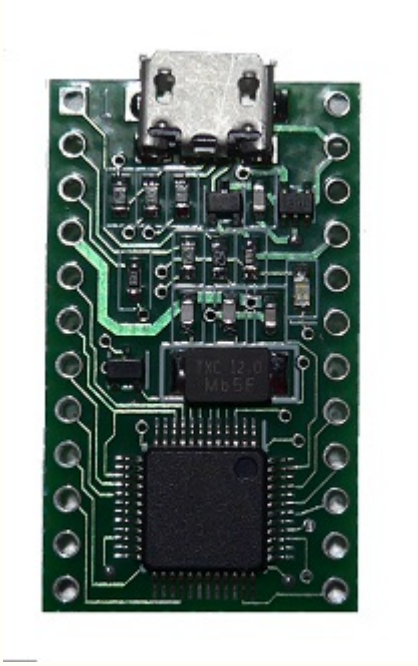
Differences from other BASICs

- no equivalent Visual BASIC
- no equivalent in PBASIC

See also

- **TOUPPER**
- **STRSTR**

Version 7 Hardware Library



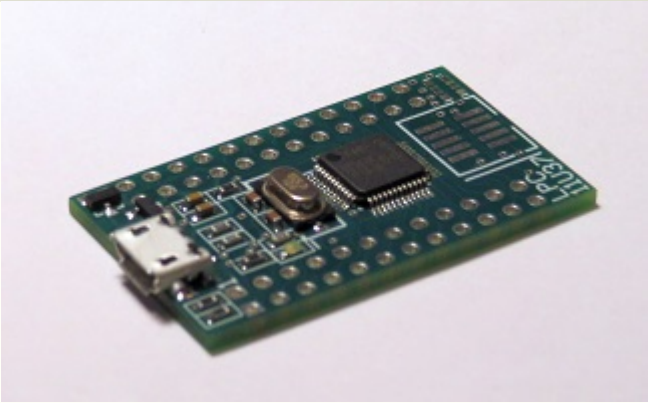
With Version 7, most of the built-in firmware hardware routines have been replaced by ARMbasic routines that can be accessed by **#include <filename>.**

Version 7 is more like Visual BASIC.

Version 8 is the same firmware for ARM Cortex processors.

The Welcome message shows the firmware version level of the ARMexpress Family device. This is displayed when the device is STOPped in the BASICtools or when reset and no user program has been loaded.

Date and Time Functions



Date and Time Functions available on all products

- TIMER
- WAIT
- WAITMICRO

Date and Time Functions that require RTC hardware

- DAY
- HOUR
- MINUTE
- MONTH
- SECOND
- WEEKDAY
- YEAR

The ARMmite, ARMweb, DINkit and SuperPRO have a provision to add a battery to keep these time functions running when power is removed. That battery (ML2020) has to be installed by the user or can be special ordered for larger volume customers.

The BASICchip does not have RTC hardware. The PROplus board does not have the RTC crystal installed.

DAY

The ARMmite, ARMweb, DINkit and SuperPRO have a provision to add a battery to keep these time functions running when power is removed. That battery (ML2020) has to be installed by the user or can be special ordered for larger volume customers.

The BASICchip does not have RTC hardware. The PROplus board does not have the RTC crystal installed.

Syntax

```
#include <RTC.bas>              ' source in /Program Files/Coridium/BASIClib

FUNCTION DAY(value)     ' when called with 0, the current day is returned, otherwise set the current day
```

Description

Function setting or returning the day of the month.

When called with a non-zero value, the DAY is changed.
Range 1 to 28, 29, 30, or 31
(depending on the month and whether it is a leap year).

Example

```
#include <RTC.bas>

DAY (14)

PRINT "This is "; MONTH(0); "/"; DAY(0); "/"; YEAR(0), "at"; HOUR(-1); ":"; MINUTE(-1); ":";
SECOND(-1)
```

The output would look like:

```
This is 4/14/2006 at 13:15:30
```

HOUR

The ARMmite, ARMweb, DINkit and SuperPRO have a provision to add a battery to keep these time functions running when power is removed. That battery (ML2020) has to be installed by the user or can be special ordered for larger volume customers.

The BASICchip does not have RTC hardware. The PROplus board does not have the RTC crystal installed.

Syntax

```
#include <RTC.bas>                    ' source in /Program Files/Coridium/BASIClib

FUNCTION HOUR(value)   'When called with -1 the current value for HOUR is returned.
```

Description

Function setting or returning the hour.

When called with a value >= 0, the HOUR is changed.
Range 0 to 23.

Example

```
#include <RTC.bas>

HOUR (13)

PRINT "This is ";  MONTH(0); "/"; DAY(0); "/"; YEAR(0), "at"; HOUR(-1); ":"; MINUTE(-1); ":";
SECOND(-1)
```

The output would look like:

```
This is 4/14/2006 at 13:15:30
```

IDLE

Syntax

IDLE (*seconds)*

Description

Delay program execution a number of seconds.

This subroutine is defined in RTC.bas It was originally called SLEEP, but the BASIC compiler now uses that to generate a WFI instruction

Example

```
#include <RTC.bas>

FOR I=0 TO 7
  IDLE(1)
  IO(I)= 1              ' set each pin HIGH one after the other every second
NEXT I
```

Differences from other BASICs

- no equivalent in Visual BASIC
- similar to SLEEP in PBASIC

See also

- **WAIT**
- **WAITMICRO**

MINUTE

The ARMmite, ARMweb, DINkit and SuperPRO have a provision to add a battery to keep these time functions running when power is removed. That battery (ML2020) has to be installed by the user or can be special ordered for larger volume customers.

The BASICchip does not have RTC hardware. The PROplus board does not have the RTC crystal installed.

Syntax

```
#include <RTC.bas>                ' source in /Program Files/Coridium/BASIClib

FUNCTION MINUTE(value)  'When called with -1 the current value for MINUTE is returned.
```

Description

Function setting or returning the day of the month.

When called with a value >= 0, the MINUTE is changed.
Range 0 to 59

Example

```
#include <RTC.bas>


MINUTE (15)

PRINT "This is ";  MONTH(0); "/"; DAY(0); "/"; YEAR(0), "at"; HOUR(-1); ":"; MINUTE(-1); ":";
SECOND(-1)
```

The output would look like:

```
This is 4/14/2006 at 13:15:30
```

MONTH

The ARMmite, ARMweb, DINkit and SuperPRO have a provision to add a battery to keep these time functions running when power is removed. That battery (ML2020) has to be installed by the user or can be special ordered for larger volume customers.

The BASICchip does not have RTC hardware. The PROplus board does not have the RTC crystal installed.

Syntax

```
#include <RTC.bas>                    ' source in /Program Files/Coridium/BASIClib

FUNCTION MONTH(value)        ' call with 0 or less to return the present MONTH, >0 will set the MONTH
```

Description

Function setting or returning the month.

When called with a non-zero value, the MONTH is changed.
Range 1 to 12.

Example

```
#include <RTC.bas>

MONTH (4)

PRINT "This is ";   MONTH(0); "/"; DAY(0); "/"; YEAR(0), "at"; HOUR(-1); ":"; MINUTE(-1); ":";
SECOND(-1)
```

The output would look like:

```
This is 4/14/2006 at 13:15:30
```

SECOND

The ARMmite, ARMweb, DINkit and SuperPRO have a provision to add a battery to keep these time functions running when power is removed. That battery (ML2020) has to be installed by the user or can be special ordered for larger volume customers.

The BASICchip does not have RTC hardware. The PROplus board does not have the RTC crystal installed.

Syntax

```
#include <RTC.bas>                    ' source in /Program Files/Coridium/BASIClib

FUNCTION SECOND(value)     'When called with -1 the current value for SECOND is returned.
```

Description

Function setting or returning the current SECOND.

When called with a value >= 0, the SECOND is changed.
Range 0 to 59

Example

```
#include <RTC.bas>

SECOND (30)

PRINT "This is "; MONTH(0); "/"; DAY(0); "/"; YEAR(0), "at"; HOUR(-1); ":"; MINUTE(-1); ":";
SECOND(-1)
```

The output would look like:

```
This is 4/14/2006 at 13:15:30
```

TIMER

Syntax

TIMER

Description

TIMER is a free running timer that increments every microsecond. Its it readable and writable using this keyword.

Interrupts that are occurring for other functions and serial input may make times using TIMER look longer than actual. Interrupts can be disabled for short periods to do fine time measurements.

For Cortex parts with firmware version after 8.27, the SysTick timer is used, which is a 24 bit counter that is set to generate an interrupt every 65.536 milliseconds. That interrupt increments a 16 counter that is combined with SysTick to generate the 32 bit TIMER value. For finer resolution a **WAITMICRO** routine has been added. And for these parts, disabling interrupts will keep the TIMER from incrementing past 65 msec.

For older LPC2103, 2106 and 2138 parts TIMER0 was used rather than Systick.

Example

```
START = TIMER
WHILE (TIMER-START < 10)  ' wait for 10 micro-seconds
LOOP
```

Differences from other BASICs

- *no equivalent in PBASIC*
- *no equivalent in Visual BASIC*

See also

- ***MINUTE***
- ***HOUR***
- ***DAY***
- ***MONTH***
- ***YEAR***
- ***WEEKDAY***

WAIT

Syntax

WAIT (*milliseconds*)

Description

Delay program execution a number of milliseconds.
1000 milliseconds is one second

Example

Print tick once per second for ever.

```
WHILE 1
  PRINT "tick"
  WAIT(1000)
LOOP
```

Differences from other BASICs

- no equivalent in Visual BASIC
- PBASIC has a similar function PAUSE that uses a CPU dependent "tick" value

See also

- **WAITMICRO**
- **IDLE**

WAITMICRO

Syntax

WAITMICRO (*microseconds*)

Description

Delay program execution a number of microseconds. This function was added to Cortex parts in version 8.27 firmware and later
It has better resolution than using TIMER. It is limited to times less than 65536 microseconds.

Example

Wiggle a line at 100 KHz.

```
IO(5)= 0
WHILE 1
  OUT(5)= 0
  WAITMICRO(50)
  OUT(5)=  1
  WAITMICRO(50)
LOOP
```

Differences from other BASICs

- no equivalent in Visual BASIC
- PBASIC has a similar function PAUSE that uses a CPU dependent "tick" value

See also

- **WAIT**
- **TIMER**

WEEKDAY

The ARMmite, ARMweb, DINkit and SuperPRO have a provision to add a battery to keep these time functions running when power is removed. That battery (ML2020) has to be installed by the user or can be special ordered for larger volume customers.

The BASICchip does not have RTC hardware. The PROplus board does not have the RTC crystal installed.

Syntax

```
#include <RTC.bas>                ' source in /Program Files/Coridium/BASIClib

FUNCTION WEEKDAY(value)     'When called with -1 the current value for WEEKDAY is returned.
```

Description

Function setting or returning the day of the week.

When called with zero or greater value, the WEEKDAY is changed.
0 corresponding to Sunday through 6 corresponding to Saturday

Example

```
#include <RTC.bas>
 '...
DIM dayname(15) as string
 '...
SECOND (30)
MINUTE (15)
HOUR (13)
DAY (14)
MONTH (4)
YEAR (2006)

SELECT WEEKDAY(-1)
CASE 0
  dayname = "Sunday"
CASE 1
  dayname = "Monday"
CASE 2
  dayname = "Tuesday"
CASE 3
  dayname = "Wednesday"
CASE 4
  dayname = "Thursday"
CASE 5
  dayname = "Friday"
CASE 6
  dayname = "Saturday"
CASE ELSE
  dayname = "not possible"
ENDSELECT

PRINT "This is "; dayname,  MONTH(0); "/"; DAY(0); "/"; YEAR(0), "at"; HOUR(-1); ":"; MINUTE(-1);
":"; SECOND(-1)
```

The output would look like:

This is Friday 4/14/2006 at 13:15:30

YEAR

The ARMmite, ARMweb, DINkit and SuperPRO have a provision to add a battery to keep these time functions running when power is removed. That battery (ML2020) has to be installed by the user or can be special ordered for larger volume customers.

The BASICchip does not have RTC hardware. The PROplus board does not have the RTC crystal installed.

Syntax

```
#include <RTC.bas>                          ' source in /Program Files/Coridium/BASIClib

FUNCTION YEAR(value)        'When called with 0 the current value for YEAR is returned.
```

Description

Function setting or returning the year.

When called with a non-zero value, the YEAR is changed.
Range 1 to 4095.

Example

```
#include <RTC.bas>

YEAR (2006)


PRINT "This is "; MONTH(0); "/"; DAY(0); "/"; YEAR(0), "at"; HOUR(-1); ":"; MINUTE(-1); ":";
SECOND(-1)
```

The output would look like:

```
This is 4/14/2006 at 13:15:30
```

Flash Access

Flash Control Functions
- FREAD
- WRITE

FREAD

Syntax

SUB FREAD (*FlashAddr*, *Destination*, *size*)

Destination = arrayname | *stringname*

size in bytes

FlashAddr an address in RAM or Flash

Description -- added version 7.13

The built-in subroutine FREAD copies data stored in the Flash memory to the Destination array, for *size* bytes. When a string is used, it is treated like a byte array, not a 0 terminated string

You can also directly access the Flash with pointers. To read a word in Flash, **FlashAddr* will return the 32 bit value starting at that address.

Example

```
'  simple example of write and read
DIM A(511) as string
DIM B(511) as string
 '...

WRITE (&H6000, A, 512)   ' this will erase the &H6000 sector, as its the first encountered
WRITE (&H6200, A, 512)   ' no erasure is required, as it was erased in the last call

FREAD (&H6200, B, 512)
 '...

WRITE (&H6000, A, 0)      ' this forces an erase of sector &H6000, needed as it was the last sector
erased
WRITE (&H6000, A, 512)

 '...

WRITE (&H6000, A, 512)    ' as the same block is being written it will automatically be erased
WRITE (&H6000, A, 512)
```

Differences from other BASICs

- Does not exist in Visual BASIC
- PBASIC has a similar function

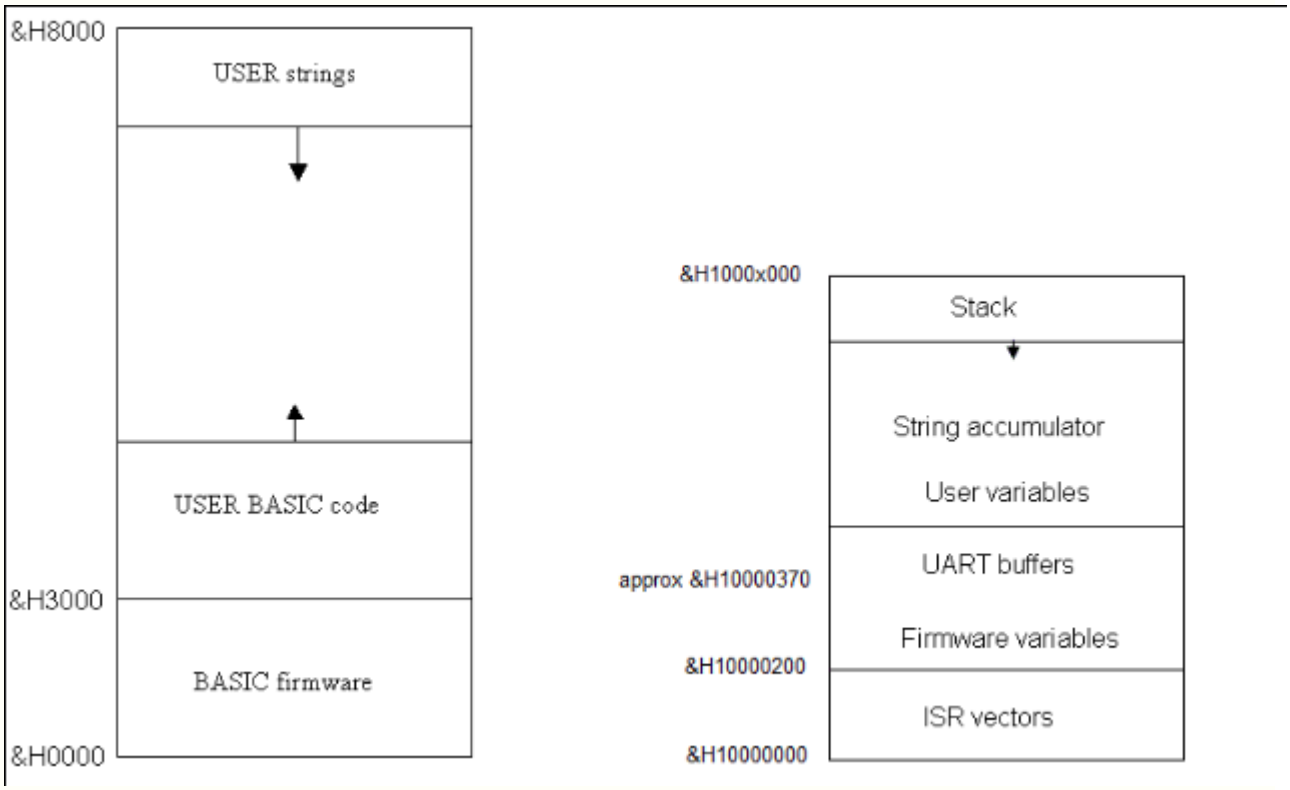
See also

- **WRITE**
- **Memory Map**
- **CPU details**

Memory Maps

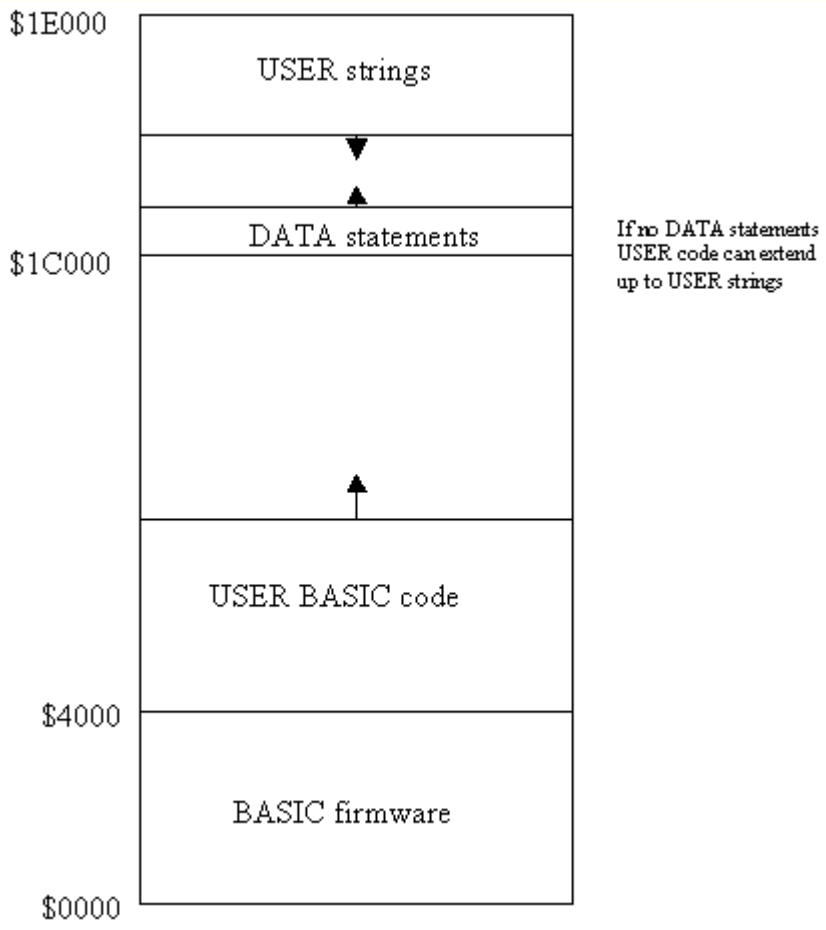
All addresses are hex values.

BASICchip, ARMmite ARMexpress LITE, ARMmite PRO, PROplus



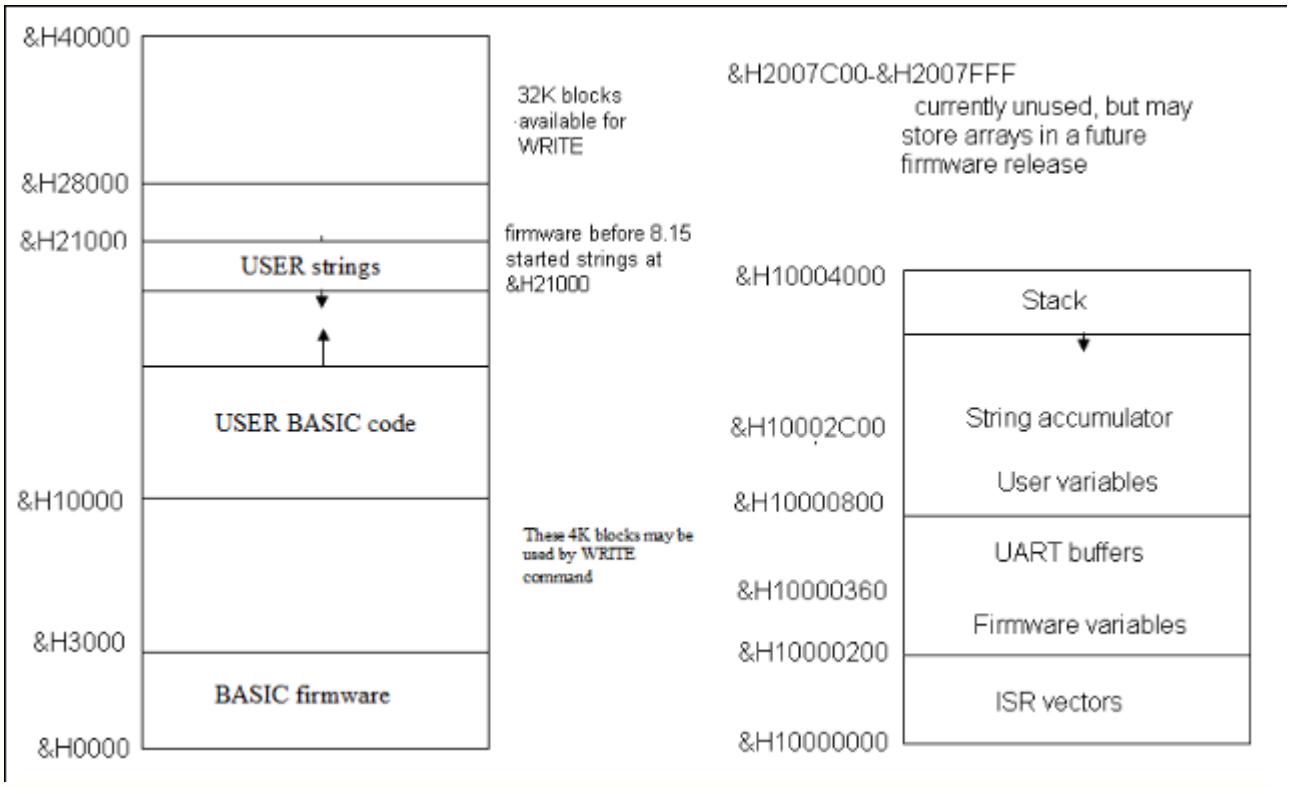
ARMmite ARMexpress LITE, ARMmite PRO, PROplus

ARMexpress



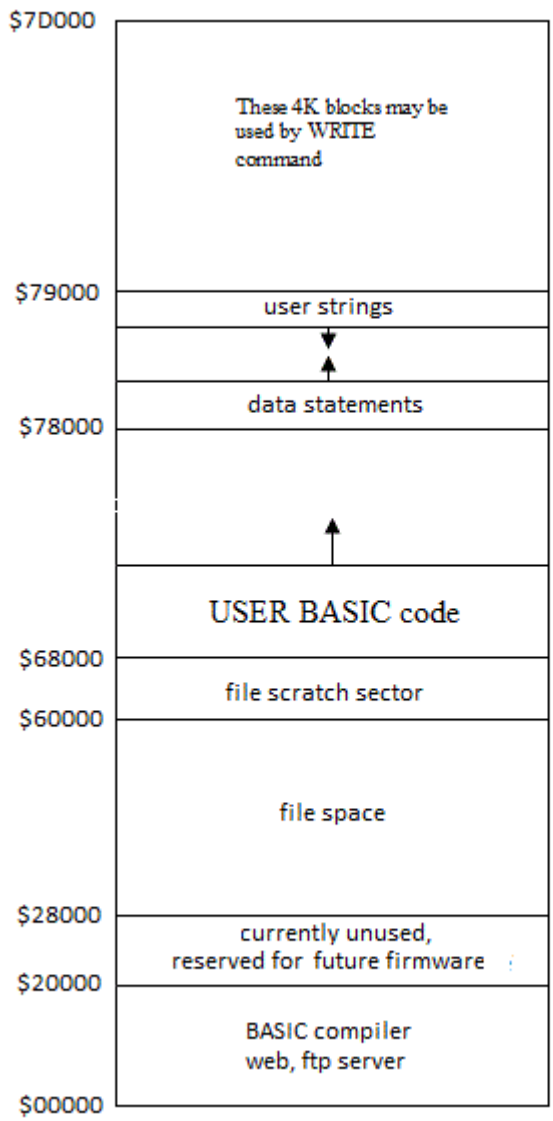
ARMexpress

SuperPRO



SuperPRO

ARMweb and DINkit/Ethernet



ARMweb and DINkit/Ethernet

DINkit (USB) and Stand-alone compiler

User code starts loading at &H3000.

Strings and DATA statements are stored in the last Flash Block, which depends on the Memory Map of the device (details in the NXP User Manuals). In the DINkit the last Flash block is from &H7C000 to &H7CFFF

LPC2103 products - ARMmite, ARMmite PRO and ARMexpress LITE

20.48K is available for code, DATA statements and string constants.

5.12K is available for data (1280 words)

LPC2106 ARMexpress

106.49K is available for code, DATA statements and string constants.

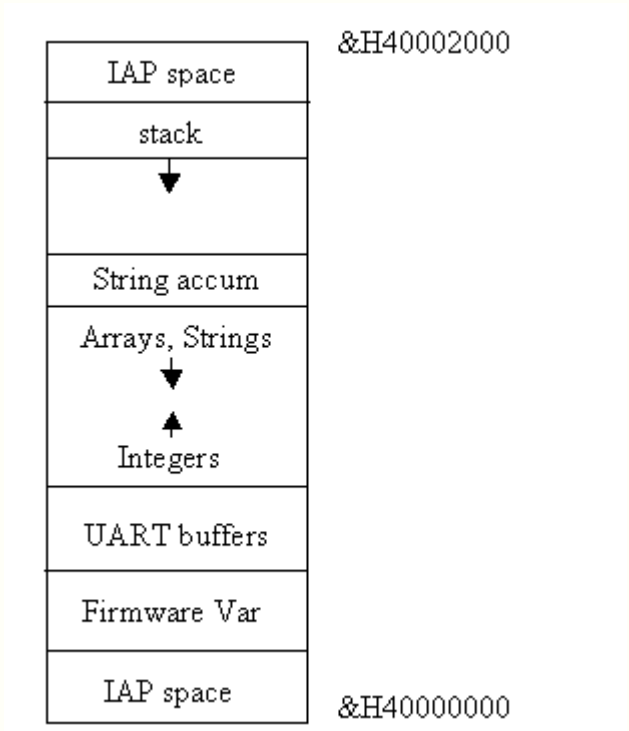
62.5K is available for data (15K words)

LPC2138 ARMweb, DINkit (Ethernet)

131K is available for code, DATA statements and string constants.

5.12K is available for data (1280 words)

DATA Memory Allocation



Local variables for FUNCTIONs and SUBs are allocated from global memory. This allows for a smaller stack size and faster calls to FUNCTIONs and SUBs. The ARMmite has only 8K total and has no stack overflow checking.

WRITE

Syntax

FUNCTION WRITE (*FlashAddr*, *Source*, *subsectorsize*)

Source = *arrayname* | *stringname*

subsectorsize = 256* | 512 | 1024 | 2048 | 4096 | 8192* in bytes

Description -- added version 7.13

WRITE copies data into the Flash memory space shared with the user code Flash space. Generally space above 0x4000 is available, but there is no protection for writing over your program. Flash is organized in sectors, 4K in ARMmite, ARMexpressLITE, 8K sectors in the ARMexpress, the ARMweb has a mix of 4K and 32K sectors. **(details in the NXP User Manual).**

For firmware versions 8.00 to 8.35, you must disable INTERRUPTs before calling WRITE and reenable them after the call.

Writing consists of erasing the whole sector and then writing a *subsector* or all.

Erases will erase the entire sector.

subsectorsize portions may be written (ARMexpress allows up to 8K but not 256). FlashAddr must be aligned to *subsectorsize* .

Data is copied from a string or array to the Flash. Only fixed *subsectorsize* sizes are allowed. This function does not look for 0 terminators when a string is the source.

The way the Flash operates, is that sectors are erased which sets all the bits to 1, or &HFFFFFFFF. When you write into a sector, bits can be changed to 0, they can not be written back to 1. Only sectors (4K, 8K or 32K) can be erased, and all bits in the sector are effected. Writes operate on *subsector* which can be as small as 256 bytes.

On reset an internal variable which indicates which sector was written to last is set to 0. Any call to WRITE will check the sector corresponding to the *FlashAddr.* If the sector is different than the last sector written, then this sector will be erased, else if it is the same then the erase does not occur.

This works for the most common use of writing to Flash, which is to start at the low end and work up.

As of firmware version 8.20 you can override this erase algorithm by setting the low 2 bits of *FlashAddr.*

- set bit 1 to force an erasure of the whole sector
- set bit 0 to disable any erase of the sector

These routines call the IAP routines for write, erase and prep commands. More details in the user manual for the corresponding CPU.

0 is returned on success, Non-zero error code when there is an error refer to **IAP section in CPU user manual for definitions** .

Example

```
' simple example of write and read
DIM A(511) as string
DIM B(511) as string
 '...
```

```
WRITE (&H6000, A, 512)   ' this will erase the &H6000 sector, as its the first encountered
WRITE (&H6200, A, 512)   ' no erasure is required, as it was erased in the last call
```

```
 '...   the following examples the user explicitly controls the erase operation with the lower 2 bits of the address

WRITE (&H6002, A, 512)       ' this forces an erase of sector &H6000, needed as it was the last sector written to

 '...
WRITE (&H6201, A, 512)    ' as the same block is being written no erase required
```

Differences from other BASICs

- Does not exist in Visual BASIC
- PBASIC has a similar function

See also

- **FREAD**
- **Memory Map**
- **CPU details**

Version 7 -- Hardware Function List

B

- BAUD

C

- COUNT

F

- FREAD

H

- HWPWM

I

- I2CIN
- I2COUT

O

- ON
- OWIN
- OWOUT

P

- PULSIN
- PULSOUT
- PWM

T

- RCTIME
- RXD

S

- SERIN
- SERINtimeout
- SEROUT
- SHIFTIN
- SHIFTOUT
- SPIBI
- SPIIN
- SPIOUT

T

- TXD

W

- WRITE

HWPWM

This function is available on ARMmite, ARMmite Wireless, ARMexpress LITE and ARMmite PRO, SuperPRO, PROplus, PROstart, BASICchip, and BASICchip-SO20

Library

```
#include <HWPWM.bas>      ' can be used for all chips, it now includes one of the following
#include <HWPWM17.bas>    ' for the PROplus and SuperPRO LPC17xx based boards.
#include <HWPWM11.bas>    ' for the BASICchip, BASICboard and PROstart
#include <HWPWM8.bas>     ' for the BASICchip-SO20
```

H

- HWPWM

Interface

```
' channels are 1-8
' cycletime and hightime are in microseconds

SUB HWPWM ( channel, cycletime, hightime )
```

Cycletime should be the same for all channels, and will be set to the last value programmed. HWPWM is used with the PWM hardware which consists of hardware registers which compare a counter that is counting from 0 to cycletime-1. If you need to change the cycletime, you should write directly to the registers, as if you write a value that is less than the current count, the cycle will count until the counter rolls over to 0, (probably not what you want, as these comparators look for equality.

BASICchip, BASICboard and PROstart version

The BASICchip supports up to 4 channels of hardware driven PWM. The IO direction of the pin will be set to output. Once programmed these will continue to generate the specified PWM until re-programmed or reset.

Cycletime is in microseconds, is the time for a single PWM cycle. *Hightimes* are also in microseconds and represent the amount of time during the cycle that the corresponding outputs are high. It is assumed, but not enforced that cycletimes for all channels will be the same.

channel1	IO(8)
channel2	IO(9)
channel3	IO(10)
channel4	IO(41)

If TIMER interrupts are used, then only 4 hardware PWM channels are available

BASICchip SO-20 version

The BASICchip supports up to 4 channels of hardware driven PWM on any of the 15 pins other than RESET, RXD0 and TXD0

channel1	IO(8) or IO(12) or IO(16)
channel2	IO(1) or IO(9) or IO(13) or IO(17)

channel3	IO(2) or IO(6) or IO(10) or IO(14)
channel4	IO(3) or IO(7) or IO(11) or IO(15)

ARMmite and Wireless ARMmite version

The ARMmite supports up to 8 channels of hardware driven PWM. The IO direction of the pin will be set to output. Once programmed these will continue to generate the specified PWM until re-programmed or reset.

Cycletime is in microseconds, is the time for a single PWM cycle. *Hightimes* are also in microseconds and represent the amount of time during the cycle that the corresponding outputs are high. It is assumed, but not enforced that cycletimes for all channels will be the same.

channel1	IO(0)
channel2	IO(1)
channel3	IO(2)
channel4	IO(3)
channel5	IO(4)
channel6	IO(9)
channel7	IO(10)
channel8	IO(11)

ARMmite PRO version

The ARMmite PRO also supports up to 8 channels of hardware driven PWM. The IO direction of the pin will be set to output. Once programmed these will continue to generate the specified PWM until re-programmed or reset.

Cycletime is in microseconds, is the time for a single PWM cycle. *Hightimes* are also in microseconds and represent the amount of time during the cycle that the corresponding outputs are high. It is assumed, but not enforced that cycletimes for all channels will be the same.

channel1	IO(0)
channel2	IO(1)
channel3	IO(8)
channel4	IO(5)
channel5	IO(14)
channel6	IO(10)
channel7	IO(11)
channel8	IO(3)

ARMexpress LITE version

The ARMexpress LITE supports up to 6 channels of hardware driven PWM. The IO direction of the pin will be set to output. Once programmed these will continue to generate the specified PWM until re-programmed or reset. 2 of the channels are not available on the pins.

Cycletime is in microseconds, is the time for a single PWM cycle. *Hightimes* are also in microseconds and represent the amount of time during the cycle that the corresponding outputs are high. It is assumed, but not enforced that cycletimes for all channels will be the same.

channel1	IO(5)
channel2	IO(6)
channel3	IO(3)

channel4	not available
channel5	IO(14)
channel6	not available
channel7	IO(13)
channel8	IO(15)

SuperPRO version

The PROplus and SuperPRO support up to 6 channels of hardware driven PWM. The IO direction of the pin will be set to output. Once programmed these will continue to generate the specified PWM until re-programmed or reset.

Cycletime is in microseconds, is the time for a single PWM cycle. *Hightimes* are also in microseconds and represent the amount of time during the cycle that the corresponding outputs are high. It is assumed, but not enforced that cycletimes for all channels will be the same.

channel1	IO(64)
channel2	IO(65)
channel3	IO(66)
channel4	IO(67)
channel5	IO(68)
channel6	IO(69)

The LPC17xx series processors also have an additional 6 channels designed to drive motors. See details in the Motor PWM Control chapter of the NXP LPC17xx User Manual. Also these pins can be re-assigned as selected by the PINSEL registers.

Example

```
#include <HWPWM.BAS>    'use HWPWM17 for SuperPRO / PROplus
 '...

'generate 1KHz with 750 and 100 uSec high signals on pins 1,2

HWPWM (2,1000,750)
HWPWM (3,1000,100)

'250 Hz with 1000, 500, 100 uSec high and LOW signals on pins 0,1,2,3

HWPWM (1,4000,1000)
HWPWM (2,4000,500)
HWPWM (3,4000,100)
HWPWM (4,4000,0)
```

I2C

Library

#include <I2C.bas>

I

- I2CIN
- I2COUT

Interface

SUB I2CIN (DATApin, CLKpin, addr, OUTcnt, BYREF OUTlist as string, INcnt, BYREF INlist as string)

FUNCTION I2COUT (DATApin, CLKpin, addr, OUTcnt, BYREF OUTlist as string)

#define I2Cspeed100 ' add this statement before the #include <I2C.bas> for 100 Kb shift rate
#define I2Cspeed50 ' for 50 Kb shift rate
#define I2CslaveCLKstretch ' trial code to support slave clock stretching (unverified on a slave that stretches clocks)

Description

These libraries are written for single master operation of the ARM talking to possible multiple slaves selected by address.

I2CIN will send *OUTcnt* bytes from *OUTlist* and then receives *INlist* bytes as i2c serial data on *CLKpin* and *DATApin* from the i2c device at *addr*. *OUTcnt* may be -1 and *OUTlist* empty. If *OUTcnt* is 0, then the string will be sent until a 0, CR or LF character is found in *OUTlist* .

INcnt bytes will be received. If *INcnt* is 0, then the string will be filled with bytes until a 0, CR or LF character is received. Note that no bounds checking is performed on the input, and if a 0, CR, or LF is never received then this routine will hang. As there is no bounds checking its possible to overwrite other variables, if less than 256 bytes have been allocated for the InputList string.

I2COUT will send *OUTcnt* bytes from *OUTlist* bytes as i2c serial data on *CLKpin* and *DATApin* to the i2c device at *addr*. If *OUTcnt* is 0, then the string will be sent until a 0, CR or LF character is found in *OUTlist*. If the i2c device does not respond 0 is returned by I2COUT, otherwise 1.

The data rate is 300Kb.

Example

```
#include <I2C.bas>
...

DIM shortMessage(20) as STRING
DIM shortResponse(20) as STRING
```

```
' test the EEPROM 24LC02 on pins 0 == SDA and 1 == SCL
shortMessage(0)= 0                ' address into EEPROM
shortMessage(1)= 11               ' data
shortMessage(2)= 22
shortMessage(3)= 33
shortMessage(5)= 44
shortMessage(6)= 55
shortMessage(7)= 66

present = I2COUT (0, 1, 0xA0, 8, shortMessage)
if present = 0 then print "NO i2c device ***"

WAIT(10) ' allow time for data to be written
I2CIN(0, 1, 0xA0, 1,shortMessage, 7, shortResponse)

' now do I2CIN as separate operations

I2COUT (0, 1, 0xA0, 1, shortMessage)   ' send just the address and offset
I2CIN(0, 1, 0xA0, -1,"", 7, shortResponse)
```

I2CIN

Syntax

```
#include <I2C.bas>                    ' source in /Program Files/Coridium/BASIClib
SUB I2CIN ( DATApin, CLKpin, addr, OUTcnt, BYREF OUTlist as string, INcnt, BYREF INlist as string)
```

Description

I2CIN will send *OUTcnt* bytes from *OUTlist* and then receives *INlist* bytes as i2c serial data on *CLKpin* and *DATApin* from the i2c device at *addr*. *OUTcnt* may be -1 and *OUTlist* empty. If *OUTcnt* is 0, then the string will be sent until a 0, CR or LF character is found in *OUTlist* .

If *INcnt* is 0, then the string will be filled with bytes until a 0, CR or LF character is received. Note that no bounds checking is performed on the input, and if a 0, CR, or LF is never received then this routine will hang. As there is no bounds checking its possible to overwrite other variables, if less than 256 bytes have been allocated for the InputList string.

Data is shifted in at 280 Kbits/sec. See the **#defines** to change this rate.

Example

```
#include <I2C.bas>
  '...
DIM shortMessage(20)  as STRING
DIM shortResponse(20)  as STRING
 '...

' test the EEPROM 24LC02 on pins 0 == SDA and 1 == SCL
shortMessage(0)= 0                ' address into EEPROM

I2CIN(0, 1, 0xA0, 1,shortMessage, 7, shortResponse)
```

Differences from other BASICs

- PBASIC output formatting not supported
- no equivalent in Visual BASIC

See also

- **I2COUT**
- **I2C Support**

I2COUT

Syntax

```
#include <I2C.bas>                    ' source in /Program Files/Coridium/BASIClib

FUNCTION I2COUT (DATApin, CLKpin, addr, OUTcnt, BYREF OUTlist as string)
```

Description

I2COUT will send *OUTcnt* bytes from *OUTlist* bytes as i2c serial data on *CLKpin* and *DATApin* to the i2c device at *addr*. If *OUTcnt* is 0, then the string will be sent until a 0, CR or LF character is found in *OUTlist*. If the i2c device does not respond 0 is returned by I2COUT, otherwise 1.

I2COUT returns a 1 if an I2C device responds, else 0.

The data rate is 280Kb. See the **#defines** to change this rate.

Example

```
#include <I2C.bas>
 '...

DIM shortMessage(20) as STRING
 '...

' test the EEPROM 24LC02 on pins 0 == SDA and 1 == SCL
shortMessage(0)= 0              ' address into EEPROM
shortMessage(1)= 11             ' data
shortMessage(2)= 22
shortMessage(3)= 33
shortMessage(5)= 44
shortMessage(6)= 55
shortMessage(7)= 66

present = I2COUT (0, 1, 0xA0, 8, shortMessage)
if present = 0 then print "NO i2c device ***"
```

Differences from other BASICs

- PBASIC output formatting not supported
- PBASIC regADDR and secondADDR are done in the OutputList
- no equivalent in Visual BASIC

See also

- **I2CIN**
- **I2C Support**

OneWire

Library

#include <ONEWIRE.bas>

O

- OWIN
- OWOUT

Interface

SUB OWIN (pin, OUTcnt, BYREF OUTlist as string, INcnt, BYREF INlist as string)

FUNCTION OWOUT (pin, OUTcnt, BYREF OUTlist as string)

Description

OWIN begins with a RESET/Presence sequence on the designated *pin*.

Then *OUTcnt* bytes from *OUTlist* will be transferred to the device to select the command. *OUTcnt* may be -1 and *OUTlist* empty. If *OUTcnt* is 0, then *OUTlist* bytes will be sent until a value of 0 is found (the 0 will not be sent). An empty *OUTlist* can be represented by "".

Following that the *INcnt* bytes will be read back from the device and saved in *INlist* .

If *INcnt* is 0, then the string will be filled with bytes until a 0, CR or LF character is received. Note that no bounds checking is performed on the input, and if a 0, CR, or LF is never received then this routine will hang. As there is no bounds checking its possible to overwrite other variables, if less than 256 bytes have been allocated for the InputList string.

OWOUT begins with a RESET/Presence sequence on the designated *Pin*.

If a one-wire device responds OWOUT will return 1, else 0.

Following that the *OUTcnt* bytes from *OUTlist* will be sent to the device. OUTlist can be a constant string.

The bit order for the 1-Wire device is assumed to be LSB (bit 0) first. The REV function can be used to change the bit order.

Example

```
#include <ONEWIRE.bas>
 '...

DIM message(20) as string
DIM response(20) as string


message = chr(&Hcc)+chr(&Hf)+chr(6)+chr(&Haa)+chr(&H55)

                ' write to the scratch pad of a DS2430
present = owout (7,5,message)
print present
```

```
message = chr(&Hcc)+chr(&Hf)+chr(6)

print present
owin (7, 3, message, 2, response)
print hex(response(0)),hex(response(1))
```

OWIN

Syntax

#include <ONEWIRE.bas> ' source in /Program Files/Coridium/BASIClib

SUB OWIN (pin, OUTcnt, BYREF OUTlist as string, INcnt, BYREF INlist as string)

Description

OWIN begins with a RESET/Presence sequence on the designated *Pin*.

Then *OUTcnt* bytes will be transferred to the device to select the command. OUTcnt may be 0, with an empty string "".

Following that the INcnt bytes *InputList* will be read back from the device. If INcnt equals 0, then the string will be filled with bytes until a 0, CR or LF character is received. Note that no bounds checking is performed on the input, and if a 0, CR, or LF is never received then this routine will hang. As there is no bounds checking its possible to overwrite other variables, if less than 256 bytes have been allocated for the *InputList* string..

The bit order for the 1-Wire device is assumed to be LSB (bit 0) first. The REV function can be used to change the bit order.

Example

```
#include <ONEWIRE.bas>

DIM outbytes(10) as string
DIM inbytes(10) as string

            ' write to the scratch pad of a DS2430
outbytes(0)=&Hcc
outbytes(1)=&Hf
outbytes(2)=&H6
outbytes(3)=&Hbe
outbytes(4)=&H41

present = owout (7 ,5, outbytes)
print present

outbytes(0)=&Hcc
outbytes(1)=&Haa
outbytes(2)=&H6

owin (7, 3, outbytes, 2, inbytes)
print hex(inbytes(0)),hex(inbytes(1))
```

Differences from other BASICs

- no equivalent in Visual BASIC
- simplified from PBASIC

See also

- OWOUT

OWOUT

Syntax

```
#include <ONEWIRE.bas>                    ' source in /Program Files/Coridium/BASIClib

FUNCTION OWOUT (pin, OUTcnt, BYREF OUTlist as string)
```

Description

OWOUT begins with a RESET/Presence sequence on the designated *Pin*.

If a one-wire device responds the FUNCTION OWOUT will return 1, else 0.

Following that OUTcnt bytes from the *OUTlist* will be sent to the device. If OUTcnt is 0, then bytes will be sent from OUTlist until a 0 is found. (the 0 is NOT sent).

The bit order for the 1-Wire device is assumed to be LSB (bit 0) first. The REV function can be used to change the bit order.

Example

```
#include <ONEWIRE.bas>

DIM outbytes(10) as string

                ' write to the scratch pad of a DS2430
outbytes(0)=&Hcc
outbytes(1)=&Hf
outbytes(2)=&H6
outbytes(3)=&Hbe
outbytes(4)=&H41

present = owout (7 ,5, outbytes)
print present
```

Differences from other BASICs

- no equivalent in Visual BASIC
- simplified than PBASIC

See also

- **OWIN**

Bit Banged Serial -- UART use is preferred

Library

#include <SERIAL.bas> ' PBASIC library, these routines were written for ARMexpress, and included for general reference,**UART functions** should be used instead of these.

Library sources are in the BASIClib directory.

This library has some initialization code that can either be copied into your program or the code can be run inline as in the following-

code without a main:

```
#include <SERIAL.bas>
... user code
```

code with a main:

```
initSerial:
#include <SERIAL.bas>
return
...
main:
   gosub initSerial
```

B

- bbBAUD

R

- bbRXD

S

- SERIN
- SERINtimeout
- SEROUT

T

- bbTXD

Interface

```
DIM bbBAUD(16)
SERINtimeout = 500000          ' timeout for bit-banged serial input in microseconds  -- this is the 0.5 second default value

FUNCTION bbRXD(pin)
SUB bbTXD(pin, ch)

FUNCTION SERIN (pin, baud, posTrue, INcnt, BYREF INlist as string)
SUB SEROUT( pin, baud, posTrue, OUTcnt, BYREF OUTlist AS STRING)
```

Description

SERIN receives *INlist* bytes as asynchronous serial data on *pin* at a *baudrate*. *PosTrue* if set to 0 then the data is inverted.

INcnt is the number of bytes that will be received. If *INcnt* is 0, then the string will be filled with bytes until a 0, CR or LF character is received. Note that no bounds checking is performed on the input, and if a 0, CR, or LF is never received then this routine will hang. As there is no bounds checking its possible to overwrite other variables, if less than 256 bytes have been allocated for the InputList string.

SERIN will timeout after 0.5 seconds and return -1 and place 255 in the next item in the *INlist* before the timeout. These routines are "bit-banged" by the processor, so the processor is consumed during these operations. Interrupts are also disabled during each byte for these operations. The hardware UART0 can be used see **RXD0 or DEBUGIN** . The timeout can be changed with SERINtimeout.

Baudrates can be up to 115.2 Kbaud for all pins on transmit. Receive rates to 57Kb

```
DIM choice(10) as STRING

SERIN(1,9600,0, choice)       ' read a UserCode CR/LF terminated

SELECT VAL (choice)
  CASE 123 ...
```

SEROUT sends a string of characters out on *pin* as an asynchronous data stream. *baud* and *posTrue* set the parameters for the transmission. *OUTcnt* is the number of bytes that will be transmitted. If OUTcnt is 0, then OUTlist will be sent until a 0 is encountered (the 0 is not sent).

```
ch = bbRXD(pin)       ' read a character from pin as an asynchronous stream (bbBAUD must have been set before use)
```

bbRXD is a bit banged routine, so that the CPU will wait up to 0.5 seconds for a character to be received. The timeout can be changed with SERINtimeout.

```
bbTXD(pin, "A")       ' send an "A" to pin as an asynchronous serial stream
```

BAUD

Syntax

```
#include <SERIAL.bas>

DIM bbBAUD( pin )              ' declared inside SERIAL.bas
```

Description

bbBAUD (*pin*) will set the baudrate for the *pin* that will be later used by either bbRXD or bbTXD functions. Baudrates can be up to 115.2 Kbaud for transmit, 57Kbaud for receive.

Example

```
#include <SERIAL.bas>

bbBAUD(2) = 19200      ' set the baud rate for serial I/O on pin 2

bbBAUD(1) = bbBAUD(2)  ' set the baud rate for pin 1 the same as that for pin 2
```

Differences from other BASICs

- no equivalent in Visual BASIC
- no equivalent in PBASIC

See also

- **bbTXD**
- **bbRXD**

RXD

Syntax

#include <SERIAL.bas>

bbRXD (*pin*)

Description

bbRXD (*pin*) will receive a single byte of data that is shifted as an asynchronous serial stream. This function is similar to SERIN, but has less overhead and only receives a single byte. The baudrate for the pin should be set before using bbRXD, that is done using the bbBAUD(pin) function.

RXD will return 0-255 if there was data present. bbRXD will timeout after 0.5 seconds and return -1 (&HFFFFFFFF) if there is no serial stream detected on *pin* .

These routines are "bit-banged" by the processor, so the processor is consumed during these operations. Interrupts are also disabled during each byte for these operations.

As of version 6.21 the 0.5 second timeout can be changed by **SERINtimeout**.

Baudrates can be up to 57 Kbaud for all pins.

Example

```
#include <SERIAL.bas>

bbBAUD(1) = 9600      ' set the baud rate for serial I/O on pin 1

' Wait for serial input on pin 1
DO
   MyByte = bbRXD(1)
UNTIL MyByte >= 0
```

Differences from other BASICs

- no equivalent in Visual BASIC
- preferred alternate to SERIN of PBASIC

See also

- **bbBAUD**
- **bbTXD**
- **SERIN**

SERIN

Syntax

```
#include <SERIAL.bas>                    ' source in /Program Files/Coridium/BASIClib

FUNCTION SERIN (pin, baud, posTrue, INcnt, BYREF INlist as string)
```

Description

SERIN receives INcnt bytes into the INlist string as asynchronous serial data on pin at a baudrate. Data is positive TRUE PosTrue if set to 1, else the data is inverted.

If INcnt is 0, then the string will be filled with bytes until a 0, CR or LF character is received. Note that no bounds checking is performed on the input, and if a 0, CR, or LF is never received then this routine will hang. As there is no bounds checking its possible to overwrite other variables, if less than 256 bytes have been allocated for the InputList string.

SERIN will timeout after 0.5 seconds and return -1 and place 255 in the next item in the INlist before the timeout. These routines are "bit-banged" by the processor, so the processor is consumed during these operations. Interrupts are also disabled during each byte for these operations. The hardware UART0 can be used see **RXD0 or DEBUGIN** . The timeout can be changed with SERINtimeout.

Baudrates can be up to 115.2 Kbaud for all pins on transmit. Receive rates to 57Kb

Example

```
#include <SERIAL.bas>

DIM Bytes(10) as STRING

' Read serial stream for 1 byte from pin 1 saving to MyByte, negative true
SERIN (1, 19200, 0, 1,Bytes)
PRINT HEX(Bytes)
```

```
#include <SERIAL.bas>

' In this case we are reading an open loop device
' that is continuously sending CR terminated strings on the serial line
' to ensure we read a complete line first sync up by looking for a CR character

DIM Astr(20) as STRING

io(15)=0 ' flag that we are sync'ing up
while 1
  serin (3,19200, 1, 1, Astr)
  if Astr(0) = 10 then exit
loop

io(15)=1 ' and that sync is complete
while 1
  serin (3,19200,1, 0, Astr)
  print Astr
loop
```

Differences from other BASICs

- *no equivalent in Visual BASIC*
- *simplified from PBASIC*

See also

- ***SEROUT***

SEROUT

Syntax

```
#include <SERIAL.bas>                    ' source in /Program Files/Coridium/BASICclib

SUB SEROUT( pin, baud, posTrue, OUTcnt, BYREF OUTlist AS STRING)
```

Description

SEROUT sends a string of characters out on *pin* as an asynchronous data stream. *baud* and *posTrue* set the parameters for the transmission. If OUTcnt is 0, then OUTlist will be sent until a 0 is encountered (the 0 is not sent).

Baudrates can be up to 115.2 Kbaud for all pins

Example

```
#include <SERIAL.bas>

DIM Astr(20) as STRING

Astr = "123"
SEROUT (3, 1200, 0, 3, Astr)     ' sends out   123 at 1.2Kbaud, negative true
```

Differences from other BASICs

- no equivalent in Visual BASIC
- simplified from PBASIC

See also

- SERIN

bbTXD

Syntax

```
#include <SERIAL.bas>                ' source in /Program Files/Coridium/BASIClib

SUB bbTXD(pin, ch)
```

Description

bbTXD (*pin, ch*) will send a single byte of data that is shifted out as an asynchronous serial stream on *pin* . This function is similar to SEROUT, but is a more efficient implementation. The baudrate for the pin should be set before using bbTXD, that is done using the bbBAUD(pin) array.

bbTXD will transmit 0-255 as a single byte of data with an added START bit and trailing STOP bit. As this function is done by the CPU (often referred to as bit-banging, the program will stay at this instruction until the shifting is completed. So the processor is consumed during these operations. Interrupts are also disabled during each byte for these operations.

Example

```
#include <SERIAL.bas>

DIM Astr(10) as STRING
bbBAUD(2) = 19200      ' set the baud rate for serial I/O on pin 2

 '...

Astr = "Hello World"
GOSUB PRINTSTR

 '...

' Send a string of characters serially out pin 2
PRINTSTR:
   I=0
   WHILE Astr(I)
      bbTXD(2,Astr(I))
      I=I+1
   LOOP

   RETURN
```

Differences from other BASICs

- no equivalent in Visual BASIC
- SEROUT in PBASIC

See also

- **bbBAUD**
- **bbRXD**
- **SEROUT**

Hardware Serial

Version 8.12 firmware adds support for all 4 UARTs in the SuperPRO and PROplus boards. Compiler switches to the array-like assignments for BAUD, RXD and TXD.

UART0 and UART1 support is built into the BASIC compiler. UART1 and BAUDx support added in 7.13 firmware.

BAUD(x)

Syntax

BAUD(*channel*) = rate ' built-in keyword

Description -- added in version 7.13

BAUD (*0*) will set the baudrate for the RXD0, TXD0 pins, that will be used by PRINT, DEBUGIN, RXD(0) or TXD(0) functions.

BAUD(*channel*) will configure the pin to the UART function and set the baudrate for the corresponding UART. On reset these pins are configured as general purpose IOs. Details on pin assignments in the Hardware Info pages for the different boards.

Baudrates for the LPC21xx and LPC23xx based boards are 15000/(n*16) in Kbaud For instance, if you call BAUD(0)=115200, the closest n is 8, and the resultant baudrate is 117.2 Kb, which is less than 2% fast, which would normally be close enough. That can be improved by using the fractional divider.

Baudrates for the LPC17xx based boards are 25000/(n*16) in Kbaud. n is an integer.

All boards except the ARMexpress support fractional baud rate generation. This is not part of the built in firmware, but can be engaged by writing directly to those registers. Details in the**Coridium Forum** or the **NXP User Manuals** .

Example

```
BAUD(1) =19200       ' set the baud rate and enable serial I/O for UART1

BAUD(0) = 9600        ' set the baud rate for RXD0 and TXD0
```

Differences from other BASICs

- no equivalent in Visual BASIC
- no equivalent in PBASIC

See also

- **TXD(x)**
- **RXD(x)**

RXD(x)

Syntax

FUNCTION RXD(channel) as INTEGER

Description

RXD(x) will return 0-255 if there was serial data present. or -1 (&HFFFFFFFF) if there is no serial character available from SIN. The hardware UARTx is used, so the CPU is not tied up, and bytes are buffered up to 256 bytes being received by an interrupt routine

SuperPRO or PROplus--

These devices support 4 UARTs, pin assignments in the **Hardware Info**

BASICchip or ARMmite--

Pin labeled RXD0 on the schematic, UART0 of the LPC2103. Data is always positive true.

Baudrates can be up to 115.2 Kbaud.

Example

```
BAUD(2)=9600     ' enable UART2 -- not all boards have UART2


' Wait for serial input on pin UART2
DO
   MyByte = RXD(2)
UNTIL MyByte >= 0
```

Differences from other BASICs

- no equivalent in Visual BASIC
- preferred alternate to SERIN of PBASIC

See also

- **TXD(x)**
- **BAUD(x)**

TXD(x)

Syntax

TXD(channel) = character ' built-in keyword

Description

The data is transmitted on the SOUT pin on the ARMexpress, ARMexpressLITE. It is the serial line connected to the USB port on the ARMmite, or the wireless serial port for the ARMmite Wireless. On the ARMweb it is serial debug port. (labeled TXD0 on the schematic, UART0 of the LPC21xx)

TX interrupts are used so characters are placed into a buffer and sent to the UART hardware when the TX FIFO or output register is empty (all charaters sent). That buffer size is 128 bytes for all but the PROplus (64 bytes). If the user sends more bytes than this buffer, the program will stall until there is room in the buffer.

SuperPRO or PROplus--

These devices support 4 UARTs, pin assignments in the **Hardware Info**. Access to UART0 and UART1 in all firmware versions, but access to all 4 UARTs requires firmware 8.14 or later.

Example

```
SUB PrintUART1 (Astr(100) as STRING)
   DIM I as INTEGER
   I=0
   WHILE Astr(I)
      TXD(1) =  Astr(I)
      I=I+1
   LOOP
END SUB
'...

main:
BAUD(1)=19200  ' enable UART1


PrintUART1 ("Hello World")       ' Send a string of characters serially out UART1
```

Differences from other BASICs

- no equivalent in Visual BASIC
- preferred alternate to SEROUT of PBASIC

See also

- **BAUD(x)**
- **RXD(x)**
- **TXFREE(x)**

TXFREE(x)

Syntax

FUNCTION TXFREE(*channel*) as INTEGER ' built-in keyword

Description -- added in version 8.30

TXFREE(*channel*) will return the amount of free space in the TX buffer of *channel* in bytes.

Because the :PC11U37 uses USB for channel 0, it appears to be always empty.

Example

```
BAUD(1)=115200 ' always need to enable UART(channel) first

print TXFREE(1)        ' show the amount of free space in TXD buffer

TXD(1)="a"
TXD(1)="b"
TXD(1)="c"
TXD(1)="d"

print TXFREE(1)        ' should be less space available for UARTs without FIFOs
```

Differences from other BASICs

- no equivalent in Visual BASIC
- no equivalent in PBASIC

See also

- **TXD(x)**
- **RXD(x)**

SHIFTIN, SHIFTOUT

Library

#include <SHIFT.bas>

S

- SHIFTIN
- SHIFTOUT

Interface

```
DIM listIN(MAXshiftARRAY)      ' values to be shifted in
DIM listOUT(MAXshiftARRAY)   ' values to be shifted out

DIM shiftCounts(MAXshiftARRAY)  ' bit counts for each value (0 assumed to be 8 bits), 1-32 allowed

' cnt is the number of elements
SUB SHIFTOUT (OUTpin, CLKpin, LSBfirst, cnt)
SUB SHIFTIN (INpin, CLKpin, LSBfirst, cnt)
```

Description

LSBfirst selects the bit order for the SHIFT routines.

A #define is used to set clock mode #define SHIFTclkNEGATIVE will invert the normally low clock. To use a normally high clock this #define must be placed before the #include <SHIFT.bas>

Another #define can be used to sample data before the clock, #define SHIFTpreSample. The default case is to sample data after each clock.

SHIFTIN can be used for devices that are not covered by SPI, I2C or 1-Wire. Data is shifted in on *INpin*, and a positive clock is sent on *CLKpin* for each bit.

While most other hardware functions use bytes, SHIFTIN is oriented for bit control. The shiftCounts defines the number of bits that will be shifted in (1 - 32) for each corresponding element of the listIN array. If a shiftCounts is 0, it is assumed to be 8. Data is shifted in at 300 Kbits/sec.

SHIFTOUT can be used for devices that are not covered by SPI, I2C or 1-Wire. Data is shifted out on *OUTpin* , and a positive clock is sent on *CLKpin* for each bit.

While most other hardware functions use bytes, SHIFTOUT is oriented for bit control. The shiftCounts array defines the number of bits that will be shifted out (1 - 32) for each corresponding element of the listOUT array. If shiftCounts is 0, it is assumed to be 8 bits.

- Mode = 0 data is shifted out MSB first
- Mode = 1 data is shifted out LSB first

NOTE* these shift modes are compatible with SHIFTIN, **BUT not the same as PBASIC**

Data is shifted out of the device at 300 Kbits/sec.

Example

```
#include <SHIFT.bas>
 '...

' use SHIFTIN/OUT to control an SPI EN28J60 connected on pins 3,4 -- 6 as CS

listOUT(0) = 2
shiftCounts(0) = 3

listOUT(1) = &H1b
shiftCounts(1) = 5

listOUT(2) = y
shiftCounts(2) = 8

io(6)=0                 ' used as CS
shiftout (3,4,1,3)      ' set reg &H1B to y
io(6)=1

listOUT(0) = reg
shiftCounts(0) = 8

io(6)=0
shiftout (3,4,1,1)                  'select the register
shiftin (5,4,0,1)                   'and read it back
x = listIN(0)
io(6)=1
```

Here is an example for a device (93LC46) which is byte oriented except for the commands. So the commands are sent with SHIFTOUT, and data transferred with SPIIN or SPIOUT. CS is manually controlled in this example (it is also positive true).

```
#include <SHIFT.bas>
#include <SPI.bas>
 '...

DIM inbuf(20) as string
DIM outbuf(20) as string

'  mixed SPI, SHIFT example for a 93LC46 connected to pins 11-14

IOI(14)=1                       ' CS to 93LC46
listOUT(0) = &H260
shiftCounts(0) = 10
SHIFTOUT(12,13,0,1)             '  write enable
IO(14)=0

listOUT(0) = &H280              ' count still 10
outbuf(0) = $41
IO(14)=1
SHIFTOUT(12,13,0,1)             ' set write to address 0
SPIOUT (-1, 13, 12, 0, 1, outbuf) ' send a byte of data
IO(14)=0

wait(20)                        ' allow for write time

IO(14)=1
```

```
listOUT(0) = &H300
SHIFTOUT(12,13,0,1)                 '   read addr 0
SPIIN (-1, 11, 13, 12, 0, -1, "", 10, inlist) ' read 10 bytes of data
IO(14)=0
```

SHIFTIN

Syntax

```
#include <SHIFT.bas>                    ' source in /Program Files/Coridium/BASIClib

SUB SHIFTIN (INpin, CLKpin, LSBfirst, cnt)
```

Description

SHIFTIN can be used for devices that are not covered by SPI, I2C or 1-Wire. Data is shifted in on *INpin*, and a positive clock is sent on *CLKpin* for each bit.

Data and shift counts are stored in 3 arrays defined in the #include file

```
DIM listIN(MAXshiftARRAY)       ' values to be shifted in
DIM listOUT(MAXshiftARRAY)    ' values to be shifted out

DIM shiftCounts(MAXshiftARRAY)
```

While most other hardware functions use bytes, SHIFTIN is oriented for bit control. The shiftCounts defines the number of bits that will be shifted in (1 - 32) for each element of the listOUT array. If a shiftCounts is 0, it is assumed to be 8.

Data is shifted in at 300 Kbits/sec.

Example

```
#include <SHIFT.bas>
 '...

' use SHIFTIN/OUT to control an SPI EN28J60 connected on pins 3,4 -- 6 as CS

listOUT(0) = 2
shiftCounts(0) = 3

listOUT(1) = &H1b
shiftCounts(1) = 5

listOUT(2) = y
shiftCounts(2) = 8

io(6)=0                 ' used asCS
shiftout (3,4,1,3)      ' set reg &H1B to y
io(6)=1

listOUT(0) = reg
shiftCounts(0) = 8

io(6)=0
shiftout (3,4,1,1)              'select the register
shiftin (5,4,0,1)               'and read it back
x = listIN(0)
io(6)=1
```

Differences from other BASICs

- no equivalent in Visual BASIC
- similar to PBASIC

See also

- **SHIFTOUT**
- **Hardware SHIFT**
- **SPIIN**

SHIFTOUT

Syntax

```
#include <SHIFT.bas>                ' source in /Program Files/Coridium/BASIClib

SUB SHIFTOUT (OUTpin, CLKpin, LSBfirst, cnt)
```

Description

SHIFTOUT can be used for devices that are not covered by SPI, I2C or 1-Wire. Data is shifted out on *OUTpin*, and a positive clock is sent on *CLKpin* for each bit.

While most other hardware functions use bytes, SHIFTOUT is oriented for bit control. The shiftCounts array defines the number of bits that will be shifted out (1 - 32) for each corresponding element of the listOUT array. If shiftCounts is 0, it is assumed to be 8 bits.

- Mode = 0 data is shifted out MSB first
- Mode = 1 data is shifted out LSB first

NOTE*** these shift modes are compatible with SHIFTIN, **BUT not the same as PBASIC**

Data is shifted out of the device at 300 Kbits/sec.

Example

```
#include <SHIFT.bas>
#include <SPI.bas>
 '...

DIM inbuf(20) as string
DIM outbuf(20) as string

'  mixed SPI, SHIFT example for a 93LC46 connected to pins 11-14

IO(14)=1                        ' CS to 93LC46
listOUT(0) = &H260
shiftCounts(0) = 10
SHIFTOUT(12,13,0,1)               '  write enable
IO(14)=0

listOUT(0) = &H280              ' count still 10
outbuf(0) = $41
IO(14)=1
SHIFTOUT(12,13,0,1)               ' set write to address 0
SPIOUT (-1, 13, 12, 0, 1, outbuf) ' send a byte of data
IO(14)=0

wait(20)                          ' allow for write time

IO(14)=1
listOUT(0) = &H300
SHIFTOUT(12,13,0,1)               '   read addr 0
SPIIN (-1, 11, 13, 12, 0, -1, "", 10, inbuf) ' read 10 bytes of data
IO(14)=0
```

Differences from other BASICs

- none from Visual BASIC
- simplified from PBASIC

See also

- **SHIFTIN**
- **Hardware SHIFT**
- **SPIIN**

SPI

Library

#include <SPI.bas>

S

- SPIBI
- SPIIN
- SPIOUT

Interface

optional #defines-
SPIclkNEGATIVE
SPIpreSample
TERMINATE_ON_0_ONLY -- ignore CR,LF as special characters

SUB SPIIN (CSpin, INpin, CLKpin, OUTpin, LSBfirst, OUTcnt, BYREF OUTlist as STRING, INcnt, BYREF INlist as STRING)

SUB SPIOUT (CSpin, CLKpin, OUTpin, LSBfirst, OUTcnt, BYREF OUTlist AS STRING)

SUB SPIBI (CSpin, INpin, CLKpin, OUTpin, LSBfirst, BIcnt, BYREF OUTlist as STRING, BYREF INlist as STRING)

Description

These libraries are written for the ARM being the master, with possible multiple slaves selected by different CS lines.

LSBfirst selects the bit order for the SPI routines.

A #define is used to set clock mode #define SPIclkNEGATIVE will invert the normally low clock. To use a normally high clock this #define must be placed before the #include <SPI.bas>

Another #define can be used to sample data before the clock, #define SPIpreSample. The default case is to sample data after each clock.

SPIIN supports the loosely defined serial protocol used by a variety of manufacturers. The desired device is selected by asserting *CSpin* LOW. If there is no *CSpin* , the value should be set to -1.

In the simplest case, *INpin* is used to input data clocked by *CLKpin*, to fill the *INlist*. (*OUTcnt* will be 0 and *OUTlist* empty)

In bi-directional cases, *OUTcnt* bytes of *OUTlist* will be output on *OUTpin* before reading the *INlist*. *OUTcnt* may be -1 and *OUTlist* empty. If *OUTcnt* is 0, then *OUTlist* bytes will be sent until a value of 0 is found (the 0 will not be sent). An empty *OUTlist* can be represented by "".

It is also allowable to have *INpin* equal to *OUTpin*, in which case that pin will be driven for the *OUTlist* and then converted to an input for *INlist*.

INlist will be filled with *INcnt* bytes. If *INcnt* is 0 then the *INlist* will be filled with bytes until a 0, CR or LF character is received. Note that no bounds checking is performed on the input, and if a 0, CR, or LF is never

received then this routine will hang.

Data is shifted in MSB first and each element of the *InputList* is filled with a byte of data. The LSBfirst can be used to change the bit order.

Data is shifted in at 330 Kbits/sec

SPIOUT supports the loosely defined serial protocol used by a variety of manufacturers. The desired device is selected by asserting *CS_pin* LOW. If there is no *CS_pin*, the value should be set to -1.

In the simplest case, *out_pin* is used to output data clocked by *clk_pin*, from the *OutputList*.

OutputList can contain a list of constants, variables, "constant-string" or *stringame$* without a *count.* The latter will send out bytes starting from *stringname$(0)* until a 0 byte is read. The 0 is not shifted out, if that is required either a *count* should be specified so as to include the 0.

Data is shifted out MSB first and each element of the *OutputList* is treated as a byte. The LSBfirst can be used to change the bit order.

Data is shifted out at 300 Kbits/sec

SPIBI supports the loosely defined serial protocol used by a variety of manufacturers. The desired device is selected by asserting *CS_pin* LOW. If there is no *CS_pin* , the value should be set to -1.

SPIBI will shift *out1, out2, out3* bytes out on *out_pin* while reading 3 or more bytes into the *InputList* from *in_pin.* For each bit the *clk_pin* will be pulsed. Data is shifted in/out MSB first. The LSBfirst can be used to change the bit order.

Data is shifted in/out at 220 Kbits/sec

Example

```
#include <SPI.bas>
,,,

DIM shortResponse(20)  as string

     ' microMega FPU uses MSB first -- positive clock

shortResponse=
chr(&HFF)+chr(&HFF)+chr(&HFF)+chr(&HFF)+chr(&HFF)+chr(&HFF)+chr(&HFF)+chr(&HFF)+chr(&HF
F)+chr(&HFF)
 SPIOUT (-1,14,15, 0, 11, shortResponse)  ' reset FPU
 WAIT (10)

 shortResponse= chr(&HF0)
 SPIOUT  (-1,14,15, 0, 1, shortResponse)         ' sync FPU
 save_time = TIMER
 while ((TIMER - save_time) < 15)  ' wait 15 uSec
 loop
 SPIIN  (-1,14,15, 0, 0,"", 1, shortResponse)             ' get 1 byte status back

 if (shortResponse(0) <> &H5C ) then
   print " No FPU found", status
   end
 endif
 print "FPU found"
 shortResponse= chr(&HF3)
```

```
SPIOUT  (-1,14,15, 0, 1, shortResponse)          ' get version
INPUT (14)                              ' allow FPU to drive this bi-directional line
while (IN(14))                          ' wait for FPU to drive that line low
loop
shortResponse= chr(&HF2)
SPIOUT (-1,14,15, 0, 1, shortResponse)          ' get string
save_time = TIMER
while ((TIMER - save_time) < 15)   ' wait 15 uSec
loop
SPIIN (-1,14,15, 0, 0,"", 0, shortResponse) ' get a 0 terminated string back

print "version = "; shortResponse;
```

For an example of an SPI device that uses non-byte oriented command see **SHIFTIN, SHIFTOUT** example.

SPIBI

Syntax

#include <SPI.bas> ' source in /Program Files/Coridium/BASICClib

SUB SPIBI (CSpin, INpin, CLKpin, OUTpin, LSBfirst, BIcnt, BYREF OUTlist as STRING, BYREF INlist as STRING)

Description

SPIBI supports the loosely defined serial protocol used by a variety of manufacturers. The desired device is selected by asserting *CS_pin* LOW. If there is no *CS_pin* , the value should be set to -1.

SPIBI will shift *out1, out2, out3* bytes out on *out_pin* while reading 3 or more bytes into the *InputList* from *in_pin.* For each bit the *clk_pin* will be pulsed. Data is shifted in/out MSB first. The LSBfirst can be used to change the bit order.

Data is shifted in/out at 220 Kbits/sec

Example

Differences from other BASICs

- no equivalent in Visual BASIC
- no equivalent in PBASIC

See also

- **SPIOUT**
- **SPI Support**

SPIIN

Syntax

```
#include <SPI.bas>                    ' source in /Program Files/Coridium/BASICIib

SUB SPIIN (CSpin, INpin, CLKpin, OUTpin, LSBfirst, OUTcnt, BYREF OUTlist as STRING, INcnt, BYREF
INlist as STRING)
```

Description

SPIIN supports the loosely defined serial protocol used by a variety of manufacturers. The desired device is selected by asserting *CSpin* LOW. If there is no *CSpin* , the value should be set to -1.

In the simplest case, *INpin* is used to input data clocked by *CLKpin*, to fill the *INlist*. (*OUTcnt* will be 0 and *OUTlist* empty)

In bi-directional cases, *OUTcnt* bytes of *OUTlist* will be output on *OUTpin* before reading the *INlist*. *OUTcnt* may be -1 and *OUTlist* empty. If *OUTcnt* is 0, then *OUTlist* bytes will be sent until a value of 0 is found (the 0 will not be sent). An empty *OUTlist* can be represented by "".

It is also allowable to have *INpin* equal to *OUTpin*, in which case that pin will be driven for the *OUTlist* and then converted to an input for *INlist*.

INlist will be filled with *INcnt* bytes. If *INcnt* is 0 then the *INlist* will be filled with bytes until a 0, CR or LF character is received. Note that no bounds checking is performed on the input, and if a 0, CR, or LF is never received then this routine will hang.

Data is shifted in MSB first and each element of the *InputList* is filled with a byte of data. The LSBfirst can be used to change the bit order.

Data is shifted in at 330 Kbits/sec

Example

```
#include <SPI.bas>

DIM Astr(20) as STRING

FUNCTION Fpu_ReadWord
 Fpu_ReadDelay
 Astr(0) = 0
 SPIIN(FpuCS, FpuIn, FpuClk, FpuOut, 0, 0, Astr, 2, Astr)
 return (Astr(0)<<8) + Astr(1)
END FUNCTION
```

Differences from other BASICs

- no equivalent in Visual BASIC
- no equivalent in PBASIC

See also

- **SPIOUT**
- **SPI Support**

SPIOUT

Syntax

```
#include <SPI.bas>                    ' source in /Program Files/Coridium/BASIClib

SUB SPIOUT (CSpin, CLKpin, OUTpin, LSBfirst, OUTcnt, BYREF OUTlist AS STRING)
```

Description

SPIOUT supports the loosely defined serial protocol used by a variety of manufacturers. The desired device is selected by asserting *CS_pin* LOW. If there is no *CS_pin*, the value should be set to -1.

In the simplest case, *out_pin* is used to output data clocked by *clk_pin*, from the *OutputList*.

OutputList can contain a list of constants, variables, "constant-string" or *stringame* without a *count.* The latter will send out bytes starting from *stringname(0)* until a 0 byte is read. The 0 is not shifted out, if that is required either a *count* should be specified so as to include the 0.

Data is shifted out MSB first and each element of the *OutputList* is treated as a byte. The LSBfirst can be used to change the bit order.

Data is shifted out at 300 Kbits/sec

Example

```
#include <SPI.bas>

DIM Astr(20) as STRING

SUB Fpu_Write(bval1)
 Astr(0) = bval1
 SPIOUT(FpuCS, FpuClk, FpuOut, 0, 1, Astr)
END SUB
```

Differences from other BASICs

- no equivalent in Visual BASIC
- no equivalent in PBASIC

See also

- **SPIIN**
- **SPI Support**

Interrupts (version 7.30 and later)

Pin Control Functions

- ADDRESSOF
- INTERRUPT
- INTERRUPT SUB

ADDRESSOF

Syntax

ADDRESSOF variable_name

or

ADDRESSOF subroutine_name

Description

ADDRESSOF will return the address of a variable or subroutine.

Example

```
#include "LPC11xx.bas"

dim e3 as integer
dim s3 as integer

INTERRUPT SUB TIMER1IRQ
  T1_IR = 1       ' Clear interrupt
  e3 = e3 + 1
ENDSUB

SUB ON_TIMER ( msec, dothis )
  TIMER1_ISR   = dothis + 1      'set vector in VIC -- the +1 for Thumb operation
  VICIntEnable = VICIntEnable or (1<<19)  'Enable interrupt
  T1_MR0 = msec-1 ' set up match number of ms
  T1_MCR = 3      ' Interrupt and Reset on MR0
  T1_IR  = 1      ' clear interrupt
  T1_TC  = 0      ' clear timer counter
  T1_TCR = 1      ' TIMER1 Enable
ENDSUB

main:
  print "TIMER1 Interrupt Test"
  print "TIMER1 will interrupt every 2 seconds and print a dot"

  ON_TIMER(2000, ADDRESSOF TIMER1IRQ)
  s3 = 0
  e3 = 0

  WHILE (1)
    if s3 <> e3 then
      s3 = e3
      print ".";
    endif
  LOOP
```

Cortex M series parts are running in Thumb code, which uses 16 bit instructions. Addresses used for the PC (program counter) indicate Thumb mode by setting the LSB of the address. That is why you see the vectors such as

```
TIMER1_ISR   = dothis + 1  ' set vector in the VIC
```

If you do not set this bit,the program will crash in the LPC11xx, LPC17xx parts. Do not set it for LPC21xx parts.

Differences from other BASICs

- similar to VB
- no equivalent in PBASIC

See also

- ADDRESSOF

INTERRUPT

Syntax

INTERRUPT (*expression)*

Description

INTERRUPT will disable interrupts if *expression* is 0. And it will enable interrupts if *expression* is non-zero. The default case is to have interrupts enabled.

Use this routine with caution, such as generating fixed time signals, or doing synchronous input. Do NOT disable interrupts around large sections of the program. Serial input will stop functioning and characters may be lost if interrupts are off for too long.

On the Cortex parts with firmware after 8.28, if interrupts are off, the TIMER will not update the upper 16 bits (meaning it can't count beyond 65 microseconds). For timing with interrupts off, use WAIT_MICRO instead. Or you can turn on just the interrupt used by TIMER with --

```
INTERRUPT(0)
ST_CTRL OR= 2   ' enable the SysTick interrupt
```

Example

```
' read a synchronous byte from a device with ready on pin 0, clock pin 1 and data on pin 2

FUNCTION ReadBit
  WHILE IN(1)=0 ' wait for clock to go high
  LOOP
 RETURN IN(2) AND 1
END FUNCTION

 '...

WHILE IN(0) ' wait for ready signal
LOOP

INTERRUPT (0)
BIT0 = ReadBit
BIT1 = ReadBit
BIT2 = ReadBit
BIT3 = ReadBit
BIT4 = ReadBit
BIT5 = ReadBit
BIT6 = ReadBit
BIT7 = ReadBit
INTERRUPT (1)

VALUE = BIT0 + (BIT1<<1) + (BIT2<<2)+ (BIT3<<3)+ (BIT4<<4)+(BIT5<<5)+ (BIT6<<6)+ (BIT7<<7)
```

Differences from other BASICs

- no equivalent in Visual BASIC
- no equivalent in PBASIC

See also

- **ON**

INTERRUPT SUB

Syntax

INTERRUPT SUB *name*

Description

INTERRUPT SUB indicates to the compiler this SUB will be used as an interrupt routine.

The address of the interrupt sub can be loaded into the interrupt hardware using the ADDRESSOF operator.

This requires firmware 7.30 or later and compiler version 7.44 or later.

This will be the way interrupts will be supported on Cortex M0,M3 parts, the ON construct will be maintained for backward compatibility, but will not be expanded.

There are example programs for interrupts in the \Program Files (x86)\Coridium\Examples directory . These examples cover TIMER, GPIO, RITIMER and EINT interrupts.

PRINT statements are useful for debugging. However PRINT statements should NOT be used inside INTERRUPT routines, as the PRINT routine is not re-entrant, in other words a PRINT inside an interrupt will interfere with a PRINT or many string expressions outside the interrupt. You can use it just to see if the INTERRUPT occurs, but beyond that expect it to cause problems in your program.

Example

```
'                                                ARM7 -- LPC21xx of ARMmite, PRO, ARMweb
' Test EINT0 on PWM02
' For ARMmite connect PWM02 to P17
' The program will poll for a "0" or "1" on RXD0
' Receiving a "0" will clear output P17, a "1" will set the output
' triggering an EINT0 interrupt

#define LPC2103

#include "LPC21xx.bas"

dim e0 as integer
dim s0 as integer
dim rx as integer


INTERRUPT SUB EINT0IRQ
   SCB_EXTINT = 1 ' Clear interrupt
   VICVectAddr = 0 ' Acknowledge Interrupt
   e0 = e0 + 1
ENDSUB


SUB ON_EINT0(rise_edge, dothis)
   ' Setup MUST be done before enabling the interrupt
   PCB_PINSEL1 = PCB_PINSEL1 or psfEINT0 ' select pin function
   SCB_EXTINT = 1 ' clear interrupt
   SCB_EXTMODE = SCB_EXTMODE or 1 ' enable edge mode

   if rise_edge
```

```
    SCB_EXTPOLAR = SCB_EXTPOLAR or 1 ' trigger on rise edge
  else
    SCB_EXTPOLAR = SCB_EXTPOLAR & &HFFFFFFFE ' trigger on fall edge (default)
  endif

  VICVectAddr4 = dothis ' set function of VIC 4
  VICVectCntl4 = &H2e ' use it for EINT0 Interrupt:
  VICIntEnable = &H4000 ' enable EINT0 Interrupt:

  VICVectAddr = 0 ' Acknowledge all Interrupts
ENDSUB


main:

print "EINT0 Interrupt Test"
print "Enter 0 to clear EINT0 input, 1 to set input"

ON_EINT0(1, ADDRESSOF EINT0IRQ) 'set up for rising edge

e0 = 0
s0 = 0
rx = 0
OUTPUT (12)
OUT(12) = 0

WHILE (1)
  rx = RXD0
  if rx > 0 then
    TXD0 = rx
    if rx = "0" then OUT(12) = 0
    if rx = "1" then OUT(12) = 1
  endif

  if s0 <> e0 then
    s0 = e0
    print "Received EINT0 "
  endif
LOOP
```

```
'                                   Cortex   M3 example -- PROplus SuperPRO

' Test EINT0 on C10 (P2.10)
' For ARMmite connect C10 to P18
' The program will poll for a "0" or "1" on RXD0
' Receiving a "0" will clear output P18, a "1" will set the output
' triggering an EINT0 interrupt

#include "LPC17xx.bas"

dim e0 as integer
dim s0 as integer
dim rx as integer

INTERRUPT SUB EINT0IRQ
 SCB_EXTINT  = 1 ' Clear interrupt
 e0 = e0 + 1
```

```
ENDSUB

SUB ON_EINT0(rise_edge, dothis)
  PCB_PINSEL4 = &H00100000        ' EINT0 on P2.10
  SCB_EXTMODE = SCB_EXTMODE or 1 ' Enable edge mode
  SCB_EXTINT  = 1                 ' Clear interrupt
  if rise_edge
    SCB_EXTPOLAR = SCB_EXTPOLAR or 1         ' trigger on rise edge
  else
    SCB_EXTPOLAR = SCB_EXTPOLAR & &HFFFFFFFE ' trigger on fall edge (default)
  endif
  EINT0_ISR    = dothis or 1              'set vector in the VIC
  VICIntEnable = VICIntEnable or (1<<18) '&H00040000 'Enable interrupt
ENDSUB


main:
  print "EINT0 Interrupt Test"
  print "Enter 0 to clear EINT0 input, 1 to set input"

  ON_EINT0(0, ADDRESSOF EINT0IRQ) 'set up for rising edge

  e0 = 0
  s0 = 0
  rx = 0
  OUTPUT(18)
  OUT(18) = 0

  WHILE (1)
    rx = RXD0
    if rx > 0 then
      TXD0 = rx
      if rx = "0" then OUT(18) = 0
      if rx = "1" then OUT(18) = 1
    endif

    if s0 <> e0 then
      s0 = e0
      print "Received EINT0 "
    endif

  LOOP
```

Cortex M series parts are running in Thumb code, which uses 16 bit instructions. Addresses used for the PC (program counter) indicate Thumb mode by setting the LSB of the address. That is why you see the vectors such as

```
  EINT0_ISR    = dothis or 1              'set vector in the VIC
```

If you do not set this bit,the program will crash in the LPC11xx, LPC17xx parts. Do not set it for LPC21xx parts.

Differences from other BASICs

- no equivalent in Visual BASIC
- no equivalent in PBASIC

See also

- **ON**

SLEEP

Syntax

SLEEP

Description

Execute a WFI instruction (Wait For Interrupt) which puts the CPU into a low power state. That state requires an interupt to end it.

Added to version 8.20 of the ARMbasic compiler. The RTC.bas library had a SUBroutine named SLEEP that has been renamed IDLE.

This function is not available on LPC2103 or LPC2138 parts. (ARMmite, ARMweb, ARMexpress).

Example

```
#include "LPC17xx.bas"

INTERRUPT SUB TIMER1IRQ
  T1_IR = 1 ' Clear interrupt
ENDSUB

SUB ON_TIMER ( msec, dothis )
  TIMER1_ISR = dothis + 1 'set function of VIC -- need the +1 for Thumb operation
  VICIntEnable or= (1<<2) 'Enable interrupt
  T1_MR0 = msec-1 ' set up match number of ms
  T1_MCR = 3 ' Interrupt and Reset on MR0
  T1_IR = 1 ' clear interrupt
  T1_TC = 0 ' clear timer counter
  T1_TCR = 1 ' TIMER1 Enable
ENDSUB


main:
  ON_TIMER(5000, ADDRESSOF TIMER1IRQ)

  print "going to sleep"
  wait(500) ' if we don't the messages don't ever come out until we are wake up

  SLEEP

  print "we woke up"
```

Differences from other BASICs

- no equivalent in Visual BASIC
- no equivalent in PBASIC

See also

- WAIT

Pin Control Functions

IO(x) is the simplest way to access the digital pins. Read IO(x) and the pin becomes or stays an input and the value is returned. Write to IO(x) and the pin becomes or stays and output and is set low when 0, or high for all other values.

For firmware after 8.11, IO(x) accesses the upper ports (P1 through P4), in groups of 32, IO(32) through IO(63) access P1(0) through P1(31) and so on.

Pin Control Functions

- **AD**
- **BYTEBUS** -- obsolete
- **DIR**
- **HIGH**
- **IN**
- **INPUT**
- **IO**
- **LOW**
- **OUT**
- **OUTPUT**
- **Port P0..P4**

Overview

Description

The original BASIC used the IO(x) keyword to control pins. It was simple and handled both reading and writing pins. If the IO is on the left hand side of an = sign, then the pin is made into an output and the pin is set high when the expression on the right hand side of the = is not 0, or set low when the right hand side is 0. When the IO is on the right hand side of an = sign, then the pin is converted to an input and the value of the pin is read. IO(x) also controls pin direction (input vs. output). Reading the pin turns it to an input, and you can always ignore the data.

When PBASIC support was added for the ARMexpress, the IN, OUT, INPUT, OUTPUT, HIGH, LOW and DIR keywords were added. Yes a lot of complexity, but the intention was to be compatible with PBASIC. These keywords are still available and will remain so. IN will always read the state of the pin, whether the pin is configured as an input or output. OUT will set the pin only if the pin has been configured as an output. Both IN and OUT are faster than IO, HIGH and LOW keywords that also configure the pin direction.

It's unfortunate that INPUT was used, as for most early BASICs this was used to get user input for a program, a function now handled by DEBUGIN. For those who'd like to use INPUT for that you could always do

```
#define INPUT   DEBUGIN
```

To match the BASIC Stamp, pins were mapped to port bits that did not correspond to the NXP bit assignments. This was also done for the ARMmite and original PRO board. But with all future boards, the NXP bit assignment will be used.

With the use by NXP of multiple 32 bit ports for IO pins, it was necessary to expand controls. P0(x) through P5(x) keywords have been added. The compiler optimizes code for these keywords, and generates inline code to increase performance. Because this code is inline it will be faster, but also larger.

With version 7.50 for ARM7 and 8.12 for ARM Cortex parts, the original keywords IO, IN OUT, INPUT, OUTPUT, DIR, HIGH and LOW have been expanded to handle multiple ports, by defining bits 32 through 63 as port P1, 64 through 95 as port P2... These keywords call built-in functions. So the code for these will be smaller, but also slower.

If you want to control or read multiple pins, you can always **access the FIOxPIN register** directly using pointers. Details on how the pins work are in the **NXP User Manual** for the appropriate part.

See also

- **OUT**
- **IN**
- **Port P0..P4**
- **@ ' dump memory**

Pin Mapping

Description

On the LPC2103 (ARMmite, ARMexpress LITE and ARMmite PRO there is only one 32 bit IO port. The first product was the ARMexpress which also has a single 32 bit IO port. As the ARMexpress is footprint compatible with the Parallax BASIC stamp, the assignment of IOs was done to match that BASIC stamp. As some ports were used for special functions, like the debug serial port, IOs 0 through 15 were not available. So that assignment does not match the bit assignment by NXP. To accomplish this the pins were remapped using an array in firmware, so that a call like IO(4) on the ARMmite will lookup 4 in an array which corresponds to P0(20)

On the LPC2138 in the ARMweb and DINkit, all pins correspond to the NXP assigned bit mapping. , There are 10 additional lines in a second port, which can be accessed using the ARMbasic P1(x) keyword. They may also be accessed with IO, DIR, IN, OUT... with firmware version 7.50 and later as IO(x) where x= 48 to 55 corresponding to P1(16) to P1(23)

On the LPC175x of the SuperPRO and PROplus there are 4 ports that have a subset of the 32 pins connected to the outside. These pins are accessed by P0(x) through P4(x) to read or write the state. As of firmware version 8.12, the IO, IN, OUT, DIR keywords are also used, with indexes 0-31 for P0, 32-63 for P1, 64-95 for P2 and 128-159 for P4.

ARMbasic provides mechanisms to control the direction and state of these pins with IO, the direction with DIR, and the state with OUT, or read with IO or IN.

These correspond to the low level registers defined by NXP FIOxDIR register, FIOxSET and FIOxCLR registers. The state of the pin can be read at anytime with the FIOxPIN register. Each of these registers consists of 32 bits each bit corresponding to a pin. When a bit in the FIOxDIR register is 1, then that pin is an output. The state of that output can be controlled by writing a 1 to that bit of the FIOxSET register to make it high or write 1 to that pin of the FIOxCLR register to make it low. Writing 0s to the FIOxSET and FIOxCLR registers do not affect those pins.

Example

On the BASICchip, BASICboard, PROstart, PROplus, SuperPRO there is no re-mapping of pins, so the bit positions match the NXP assignments.

On the ARMmite, the pin re-mapping can be viewed as code like the following:

```
#include <LPC21xx.bas>

CONST ReMap={9,8,30,21,20,29,4,5,6,7,13,19,18,17,16,15}

#define OUTPUT(x) FIO0DIR = FIO0DIR or (1 << ReMap(x))
#define INPUT(x)   FIO0DIR  = FIO0DIR and not (1 << ReMap(x))
#define IN(x)         P0(ReMap(x))
#define OUT(x)      P0(ReMap(x))
#define HIGH(x)    P0(ReMap(x))= 1
#define LOW(x)     P0(ReMap(x)) = 0
```

On the PRO, the pin re-mapping can be viewed as code like the following:

```
#include <LPC21xx.bas>

CONST ReMap={9,8,27,19,28,21,5,29,30,16,7,13,4,6,20,15}
```

```
#define OUTPUT(x) FIO0DIR = FIO0DIR or (1 << ReMap(x))
#define INPUT(x)    FIO0DIR = FIO0DIR and not (1 << ReMap(x))
#define IN(x)          P0(ReMap(x))
#define OUT(x)       P0(ReMap(x))
#define HIGH(x)     P0(ReMap(x))= 1
#define LOW(x)      P0(ReMap(x)) = 0
```

See also

- **OUT**
- **IN**
- **Port P0..P4**
- **@ ' dump memory**

AD

Syntax

FUNCTION AD (*expression*) AS INTEGER

Description

AD will return 0..65472 that corresponds to the voltage on the pin corresponding to *expression* . The value returned will have the top 10 bits of significance followed by bits 5..0 will be 0. 0 would be read for 0V and 65472 for 3.3V.

An analog conversion on pin *expression* is performed when this built-in FUNCTION is called. This process takes less than 6 usec.

Dual Use AD pins
On reset or power up the AD pins are configured as AD inputs. To change those to digital IOs, the user must individually specify a control direction using INPUT(x), OUTPUT(x), DIR(x), or IO(x) commands. After that they will remain digital IOs until the next reset or power up.

BASICchip

AD pins are configured as ADs. To make them to IOs, the IOCON register needs to be written which is done when you access the corresponding IO(x)

For BASICchip firmware versions before 8.28, to convert from AD to digital IO the following code must be used

```
#include <LPC11xx.bas>

IOCON_R_PIO0_11 = &H91          ' convert AD(0)  to digital input or output
IOCON_R_PIO1_0 = &H91           'AD(1)
IOCON_R_PIO1_1 = &H91           'AD(2)
IOCON_R_PIO1_2 = &H91           'AD(3)
IOCON_SWDIO_PIO1_3 = &H90       'AD(4)
IOCON_PIO1_4 = &H90             'AD(5)
```

for firmware versions after 8.28 you can convert to digital IOs by access using IO for either input or output

```
x = IO(11)          ' make AD(0) an IO
x = IO(32)          ' make AD(1) an IO
x = IO(33)          ' make AD(2) an IO
x = IO(34)          ' make AD(3) an IO
x = IO(35)          ' make AD(4) an IO
x = IO(36)          ' make AD(5) an IO
```

SuperPRO and PROplus version

The LPC17xx series have a 12 bit converter, so the top 12 bits of the 16 bit value are returned by AD. Version 8.14 of the firmware allows the AD pins to be changed to digital pins using IO(x), INPUT(x), DIR(x) and OUTPUT(x) commands. AD(6) and AD(7) are on shared pins with RXD(0) and TXD(0), and to use those the PINSEL registers have to be re-programmed.

ARMmite and PRO version

Eight 10-bit ADs are available.

ARMweb

AD pins are configured as ADs. To return them to IOs, the PINSEL register needs to be written (details in the NXP LPC2138 User Manual)

```
PINSEL1 AND= &HC003FFFF    ' return ADs to IOs
```

ARMexpress LITE version

The ARMexpress LITE supports up to 6 channels of AD converters.

On the ARMexpress LITE and ARMweb these pins are configured as digital IOs at reset, but will be switched to AD operation when AD(x) is read.

AD(0)	IO(7)
AD(1)	IO(10)
AD(2)	IO(8)
AD(3)	not available
AD(4)	not available
AD(5)	IO(9)
AD(6)	IO(11)
AD(7)	IO(12)

Example

```
voltage = AD (0)      ' this will read the voltage on pin 0
```

Differences from other BASICs

- no equivalent in Visual BASIC
- no equivalent in PBASIC

See also

- **IO**
- **DIR**
- **OUTPUT**

BYTEBUS (ARMweb only) -- OBSOLETE, use P1(pin)

To read more than 1 bit at a time, the FIO1PIN register can be read directly.

```
#include <LPC21xx>          ' or <LPC17xx> for SuperPRO / PROplus

x = FIO0PIN   ' read the 32 bits of port 0

y = FIO1PIN   ' read the 32 bits of port 1 (only 10 are connected externally on the ARMweb)
```

DAC

Syntax

DACsetup()

DACout(expression)

Description

Control of the DAC is done by writing directly to the registers. Details can be found in the User manual of the appropriate part, links in the **Hardware Section** .

Rather than having built in functions in BASIC, this will be done by **subroutines** . Samples of those subroutines are below

Example

On the SuperPRO:

```
#define PCB_PINSEL1       *(&H4002C004)
#define PCB_PINMODE1      *(&H4002C044)

#define DACR              *(&H4008C000)     ' or use #include <LPC17xx.bas>

sub DACsetup
  PCB_PINSEL1 = PCB_PINSEL1 and (not (3<<20)) or (2<<20)   ' enable DAC output
  PCB_PINMODE1 = PCB_PINMODE1 or (2<<20)                   ' disable pull-ups
endsub

sub DACout(value)
  DACR = value << 6
endsub

main:
DACsetup

for i= 0 to 1023
  DACout(i)
  wait(10)
 next i
```

On the ARMweb or DINkit:

```
#define PCB_PINSEL1       *(&HE002C004)

#define DACR              *(&HE006C000)     ' or use #include <LPC21xx.bas>

sub DACsetup
  PCB_PINSEL1 = PCB_PINSEL1 and (not (3<<18)) or (2<<18)   ' enable DAC output
endsub

sub DACout(value)
  DACR = value << 6
endsub
```

```
main:
DACsetup

for i= 0 to 1023
  DACout(i)
  wait(10)
 next i
```

Differences from other BASICs

- no equivalent in Visual BASIC
- no equivalent in PBASIC

See also

- **OUT**
- **IN**
- **@ ' dump memory**

DIR

Syntax

DIR (*expression*)

Description

The DIR keyword is for compatibility with PBASIC, and to support multiple ports on newer ARM devices.

DIR (expression) can be used to set or read the direction of the pins. If DIR (expression) value is 0 then that pin is an input, otherwise for other values the pin is an output.

The ARMmite allows control of 24 pins (0..23), with pins 16..23 shared with the AD pins. On reset or power up the AD pins are configured as AD inputs. To change those to digital IOs, the user must individually specify a control direction using INPUT(x), OUTPUT(x), DIR(x), or IO(x) commands. After that they will remain digital IOs until the next reset or power up.

For the ARMmite, PRO, ARMexpress and ARMexpress LITE these pin numbers correspond to the pin numbers shown in the **Hardware Section**. The numbering was assigned by physical location on the board. So DIR, HIGH, IN, INPUT, IO, LOW, OUT and OUTPUT use these physical pin assignments. But P0(pin) will use the bit assigned by NXP. Going forward new board designs will maintain the bit assignment from NXP for all keywords.

For the ARMweb, DINkit, SuperPRO, PROplus and PROstart these pin numbers correspond only to the Port 0 assigned by NXP, for instance DIR(11) corresponds to P0(11)

With version 8.11 of the firmware, ports beyond port 0, can be accessed as 32-63 for port 1, 64-95 for port 2, and so on.

As of version 8.25-SuperPRO and 8.26-BASICchip of the firmware, using INPUT will also convert AD pins to digital function. To switch back you would need to program the PINSEL register manually (see the NXP User Manual).

Example

```
DIR(4) = 0  ' Set pin 4 as an input

DIR(12) = 1   ' Set pin 12 as an output
```

On the extended ports for the SuperPRO DIR BASIC style use the following

```
DIR(64+10) = 1      ' makes P2(10) an output

while 1
  x=x+1
  P2(10) = x and 1          ' blinky for the SuperPRO and PROplus
  wait(500)
loop
```

Differences from other BASICs

- no equivalent in Visual BASIC
- equivalent to DIR0..15 in PBASIC

See also

- **INPUT**
- **OUTPUT**

HIGH

Syntax

HIGH (*expression*)

Description

**** kept for PBASIC compatibility, IO(x)=1 is preferred *****

HIGH will set the pin corresponding to *expression* to a positive value (3.3V) and then set it to an output.

HIGH and LOW have been added for PBASIC compatibility.

For the ARMmite, PRO, ARMexpress and ARMexpress LITE these pin numbers correspond to the pin numbers shown in the **Hardware Section**. The numbering was assigned by physical location on the board. So DIR, HIGH, IN, INPUT, IO, LOW, OUT and OUTPUT use these physical pin assignments. But P0(pin) will use the bit assigned by NXP. Going forward new board designs will maintain the bit assignment from NXP for all keywords.

For the ARMweb, DINkit, SuperPRO, PROplus and PROstart these pin numbers correspond only to the Port 0 assigned by NXP, for instance DIR (3) corresponds to P0(3)

With version 8.11 of the firmware, ports beyond port 0, can be accessed as 32-63 for port 1, 64-95 for port 2, and so on. Or you may use the **P1 .. P4 commands** .

Example

```
SUB DIRS (x)          ' similar to PBASIC keyword
  DIM i AS INTEGER

  FOR i = 0 to 15
    if x and (1 << i) then OUTPUT(i) else INPUT(i)
  NEXT i
END SUB

main:

DIRS (&H00FF)      '  set pins 0 to 7 to output

FOR I=0 TO 7
  WAIT (1000)
  HIGH(I)              ' set each pin HIGH one after the other every second
NEXT I
```

Differences from other BASICs

- no equivalent in Visual BASIC
- none from PBASIC

See also

- **LOW**

IN

Syntax

IN (*expression*)

Description

When reading from IN (*expression*), -1 or 0 will be returned corresponding to the voltage level on the pin numbered *expression.* Why -1 and 0? The main reason is that operations of operators like NOT are assumed to be bit-wise until there is a Boolean operation in the expression, and NOT 0 is equal to -1.

This directive does not change the input/output configuration of the pin.

The ARMmite allows control of 24 pins (0..23), with pins 16..23 shared with the AD pins. On reset or power up the AD pins are configured as AD inputs. To change those to digital IOs, the user must individually specify a control direction using INPUT(x), OUTPUT(x), DIR(x), or IO(x) commands. After that they will remain digital IOs until the next reset or power up.

For the ARMmite, PRO, ARMexpress and ARMexpress LITE these pin numbers correspond to the pin numbers shown in the **Hardware Section**. The numbering was assigned by physical location on the board. So DIR, HIGH, IN, INPUT, IO, LOW, OUT and OUTPUT use these physical pin assignments. But P0(pin) will use the bit assigned by NXP. Going forward new board designs will maintain the bit assignment from NXP for all keywords.

For the ARMweb, DINkit, SuperPRO, PROplus and PROstart these pin numbers correspond only to the Port 0 assigned by NXP, for instance DIR(3) corresponds to P0(3)

With version 8.11 of the firmware, ports beyond port 0, can be accessed as 32-63 for port 1, 64-95 for port 2, and so on. Or you may use the **P1 .. P4 commands** .

Example

```
' Set pin 9 as an input
INPUT (9)

' Assume an external device has driven pin 9 high

PRINT "The current value of Input pin 9 is "; IN(9) AND 1

The current value of Input pins is 1
```

Differences from other BASICs

- no equivalent in Visual BASIC
- equivalent to IN0..15 PBASIC

See also

- **OUT**
- **IO**

INPUT

Syntax

INPUT (*expression*)

Description

INPUT will set the pin corresponding to *expression* to an input.

INPUT was added for PBASIC compatibility, same function as DIR(x)= 0.

The ARMmite allows control of 24 pins (0..23), with pins 16..23 shared with the AD pins. On reset or power up the AD pins are configured as AD inputs. To change those to digital IOs, the user must individually specify a control direction using INPUT(x), OUTPUT(x), DIR(x), or IO(x) commands. After that they will remain digital IOs until the next reset or power up.

Making a pin an INPUT will also tri-state that pin.

For the ARMmite, PRO, ARMexpress and ARMexpress LITE these pin numbers correspond to the pin numbers shown in the **Hardware Section**. The numbering was assigned by physical location on the board. So DIR, HIGH, IN, INPUT, IO, LOW, OUT and OUTPUT use these physical pin assignments. But P0(pin) will use the bit assigned by NXP. Going forward new board designs will maintain the bit assignment from NXP for all keywords.

For the ARMweb, DINkit, SuperPRO, PROplus and PROstart these pin numbers correspond only to the Port 0 assigned by NXP, for instance DIR(7) corresponds to P0(7)

With version 8.11 of the firmware, ports beyond port 0, can be accessed as 32-63 for port 1, 64-95 for port 2, and so on. Or you may access the FIOxDIR register directly.

As of version 8.25-SuperPRO and 8.26-BASICchip of the firmware, using INPUT will also convert AD pins to digital function.

Example

```
INPUT (0)     ' this will make pin 0 an input
```

On the extended ports for the SuperPRO INPUT/OUTPUT BASIC style use the following

```
OUTPUT(64+10)     ' makes P2(10) an output

while 1
  x=x+1
  P2(10) = x and 1          ' blinky for the SuperPRO and PROplus
  wait(500)
loop
```

Differences from other BASICs

- INPUT gets a value from the user in some BASICs, in ARMbasic get a value from the debug serial port with **DEBUGIN**
- none from PBASIC

See also

- **DIR**
- **OUTPUT**

- DEBUGIN

IO

Syntax

IO (*expression*)

Description

IO is the simple way to access or control the pins. When IO (*expression*) is read or on the right hand side of the = sign , the pin corresponding to *expression* is converted to an input and the value on that pin is read.

When assigning a value to IO(*expression*), or on the left hand side of the = sign, then pin *expression* is converted to an output and the logic value is written to the pin, 0 writes a low level any other value sets the pin high. When read IO returns a 0 or -1. Why -1 and 0? The main reason is that operations of operators like NOT are assumed to be bit-wise until there is a Boolean operation in the expression, and NOT 0 is equal to -1. When setting a pin state with IO(x) = 0 then the pin becomes low, any other value and the pin becomes high, so IO(x) =1 and IO(x) = -1 both set the pin high.

Using IO simplifies pins that are being used as both inputs and outputs. As it also sets direction it will be slower than IN, OUT, or Px(y)

The ARMmite allows control of 24 pins (0..23), with pins 16..23 shared with the AD pins. On reset or power up the AD pins are configured as AD inputs. To change those to digital IOs, the user must individually specify a control direction using INPUT(x), OUTPUT(x), DIR(x), or IO(x) commands. After that they will remain digital IOs until the next reset or power up.

For the ARMmite, PRO, ARMexpress and ARMexpress LITE these pin numbers correspond to the pin numbers shown in the **Hardware Section**. The numbering was assigned by physical location on the board. So DIR, HIGH, IN, INPUT, IO, LOW, OUT and OUTPUT use these physical pin assignments. But P0(pin) will use the bit assigned by NXP. Going forward new board designs will maintain the bit assignment from NXP for all keywords.

For the ARMweb, DINkit, SuperPRO, PROplus and PROstart these pin numbers correspond only to the Port 0 assigned by NXP, for instance DIR(12) corresponds to P0(12)

With version 7.52/8.12 of the firmware, ports beyond port 0, can be accessed as 32-63 for port 1, 64-95 for port 2, and so on. Or you may use the **P1 .. P4 commands** .

Example

```
' Set pin 9 as an output and drive it high
IO(9) = 1

IO(9) = NOT IN(9)   ' invert pin DO NOT USE IO(9) as that would be ambiguous for controlling the direction of the pin

' Set pin 8 as an input and reads its value
x = IO(8)
```

Differences from other BASICs

- no equivalent in Visual BASIC
- no equivalent in PBASIC

See also

- **OUT**
- **IN**

LOW

Syntax

LOW (*expression*)

Description

**** kept for PBASIC compatibility IO(x) = 0 is preferred *****

LOW will set the pin corresponding to *expression* to a low value (0V) and then set it to an output.

HIGH and LOW have been added for PBASIC compatibility.

For the ARMmite, PRO, ARMexpress and ARMexpress LITE these pin numbers correspond to the pin numbers shown in the **Hardware Section**. The numbering was assigned by physical location on the board. So DIR, HIGH, IN, INPUT, IO, LOW, OUT and OUTPUT use these physical pin assignments. But P0(pin) will use the bit assigned by NXP. Going forward new board designs will maintain the bit assignment from NXP for all keywords.

For the ARMweb, DINkit, SuperPRO, PROplus and PROstart these pin numbers correspond only to the Port 0 assigned by NXP, for instance DIR(13) corresponds to P0(13)

With version 8.11 of the firmware, ports beyond port 0, can be accessed as 32-63 for port 1, 64-95 for port 2, and so on. Or you may use the **P1 .. P4 commands** .

Example

```
SUB OUTS (x)          ' similar to PBASIC keyword
  DIM i AS INTEGER

  FOR i = 0 to 15
    OUT(i) = x and (1 << i)
  NEXT i
END SUB

SUB DIRS (x)          ' similar to PBASIC keyword
  DIM i AS INTEGER

  FOR i = 0 to 15
    DIR(i) = x and (1 << i)
  NEXT i
END SUB


main:

DIRS ( &H00FF)      '  set pins 0 to 7 to output
OUTS (255)          '  and then set them high or to 3.3 V

FOR I=0 TO 7
  WAIT (1000)
  LOW (I)             ' set each pin LOW one after the other every second
NEXT I
```

Differences from other BASICs

- no equivalent in Visual BASIC

- none from PBASIC

See also

- **HIGH**
- **IO**

OUT

Syntax

OUT (*expression*)

Description

When writing to OUT (*expression*), the pin corresponding to *expression* will be set a voltage level corresponding to TRUE or FALSE, non-zero or 0. When setting a pin state with OUT(x) = 0 then the pin becomes low, any other value and the pin becomes high, so OUT(x) =1 and OUT(x) = -1 both set the pin high.

The OUT directive does not change the input/output configuration of the pin. Following reset all pins are inputs, before an OUT () will have an effect on a pin, that pin must be made an output using an OUTPUT command. The reason for this is to make OUT faster, if the pin direction were changed each OUT, then the speed of one OUT to the next would be slower.

The ARMmite allows control of 24 pins (0..23), with pins 16..23 shared with the AD pins. On reset or power up the AD pins are configured as AD inputs. To change those to digital IOs, the user must individually specify a control direction using INPUT(x), OUTPUT(x), DIR(x), or IO(x) commands. After that they will remain digital IOs until the next reset or power up.

For the ARMmite, PRO, ARMexpress and ARMexpress LITE these pin numbers correspond to the pin numbers shown in the **Hardware Section**. The numbering was assigned by physical location on the board. So DIR, HIGH, IN, INPUT, IO, LOW, OUT and OUTPUT use these physical pin assignments. But P0(pin) will use the bit assigned by NXP. Going forward new board designs will maintain the bit assignment from NXP for all keywords.

For the ARMweb, DINkit, SuperPRO, PROplus and PROstart these pin numbers correspond only to the Port 0 assigned by NXP, for instance DIR(10) corresponds to P0(10)

With version 8.11 of the firmware, ports beyond port 0, can be accessed as 32-63 for port 1, 64-95 for port 2, and so on. Or you may use the **P1 .. P4 commands** .

Example

```
' Set pin 9 as an output
OUTPUT (9)

' Drive pin 9 high
OUT(9) = 1

PRINT "The current value of Output pin 9 is "; OUT(9)

The current value of Output pins is 1
```

Differences from other BASICs

- no equivalent in Visual BASIC
- equivalent to OUT0..15 in PBASIC

See also

- **IN**
- **IO**

OUTPUT

Syntax

OUTPUT (*expression*)

Description

OUTPUT will set the pin corresponding to *expression* to an output.

OUTPUT was added for PBASIC compatibility, same function as DIR(x)= 1.

The ARMmite allows control of 24 pins (0..23), with pins 16..23 shared with the AD pins. On reset or power up the AD pins are configured as AD inputs. To change those to digital IOs, the user must individually specify a control direction using INPUT(x), OUTPUT(x), DIR(x), or IO(x) commands. After that they will remain digital IOs until the next reset or power up.

For the ARMmite, PRO, ARMexpress and ARMexpress LITE these pin numbers correspond to the pin numbers shown in the **Hardware Section**. The numbering was assigned by physical location on the board. So DIR, HIGH, IN, INPUT, IO, LOW, OUT and OUTPUT use these physical pin assignments. But P0(pin) will use the bit assigned by NXP. Going forward new board designs will maintain the bit assignment from NXP for all keywords.

For the ARMweb, DINkit, SuperPRO, PROplus and PROstart these pin numbers correspond only to the Port 0 assigned by NXP, for instance DIR(3) corresponds to P0(3)

With version 8.11 of the firmware, ports beyond port 0, can be accessed as 32-63 for port 1, 64-95 for port 2, and so on. Or you may access the FIOxDIR register directly.

As of version 8.25-SuperPRO and 8.26-BASICchip of the firmware, using OUTPUT will also convert AD pins to digital function.

Example

```
' Set pin 9 as an output
OUTPUT (9)
```

```
As of firmware version 8.12 for the SuperPRO, access to port 1,2 and 4 is
done by accessing bits 32-63, 64-95, 128-159 respectively
```

```
#include <LPC17xx.bas>

OUTPUT(10+64)   ' make P2.10 and output

while 1
  x=x+1
  OUT(74) = x and 1          ' blinky for the SuperPRO and PROplus
  wait(500)
loop
```

Differences from other BASICs

- no equivalent in Visual BASIC
- none from PBASIC

See also

- **DIR**
- **INPUT**

PORT P0..P4

Syntax

Pn (*expression*) ' where n is 0 through 4

Description

Px allows you to read or write individual pins using the NXP assigned port and pin number. When Pn (*expression*) is read, the logic state of the pin corresponding to *expression* is returned.

When assigning a value to Pn(*expression*), then pin *expression* is set to that value if that pin has been assigned to be an output by writing to FIOxDIR.

When read Pn(x) returns a 0 or -1. Why -1 and 0? The main reason is that operations of operators like NOT are assumed to be bit-wise until there is a Boolean operation in the expression, and NOT 0 is equal to -1. When setting a pin state with Pn(x) = 0 then the pin becomes low, any other value and the pin becomes high, so Pn(x) =1 and Pn(x) = -1 both set the pin high.

These pin numbers correspond to the port pin assignments from NXP.

This feature is part of the compiler and requires version 8.04c or later. It has not been added to the on-chip compiler of the ARMweb. Access to IO pins using Px commands is optimized by the compiler for the SuperPRO and PROplus. These do work for the BASICchip, but have not been optimized yet.

Example

On the SuperPRO and PROplus:

```
#include <LPC17xx.bas>

' Set pin 9 as an output and drive it high
FIO1DIR = FIO1DIR OR (1<<9)
P1(9) = 1

P1(9) = NOT (P1(9) and (1 <<9))   ' invert pin P1(9) -- works as you can always read the state of a pin

' read value of P1(8)
x = P1(8)

' change bit 9 back to an input
FIO1DIR = FIO1DIR AND NOT(1<<9)
```

Differences from other BASICs

- no equivalent in Visual BASIC
- no equivalent in PBASIC

See also

- **OUT**
- **IN**
- **@ ' dump memory**

Register Port Access

Description

On the LPC2103 (ARMmite, ARMexpress LITE and PRO there is only one 32 bit IO port. On the LPC2138 in the ARMweb and DINkit, there are 10 additional lines in a second port. On the LPC17xx of the SuperPRO and PROplus there are 4 ports that have a subset of the 32 pins connected to the outside.

Those are controlled by a DIR register, SET and CLR registers. The state of the pin can be read at anytime with the PIN register. Each of these registers consists of 32 bits each bit corresponding to a pin. When a bit in the DIR register is 1, then that pin is an output. The state of that output can be controlled by writing a 1 to that bit of the SET register to make it high or 1 to that pin of the CLR register to make it low. Writing 0s to the SET and CLR registers do not affect those pins.

Example

On the SuperPRO and PROplus:

```
#include <LPC17xx.bas>

' Set port 2 pins 1 through 9 as an output and drive it every other bit high
FIO2DIR = FIO2DIR or ( &H3FE)

FIO2SET = &H2AA
FIO2CLR = &H154

' read value of P2.8 and 9
x = FIO2PIN and &H300

' change bit 9 back to an input
FIO2DIR = FIO2DIR and NOT(1<<9)
```

On the ARMweb or DINkit:

```
#include <LPC21xx.bas>

SCB_SCS  = 3          ' required to enable port1 for firmware before 7.47

' Set port 1 pins 1 through 9 as an output and drive it every other bit high
FIO1DIR = FIO1DIR or ( &H3FE)

FIO1SET = &H2AA
FIO1CLR = &H154

' read value of P1.8 and 9
x = FIO1PIN and &H300

' change bit 9 back to an input
FIO1DIR = FIO1DIR and NOT(1<<9)
```

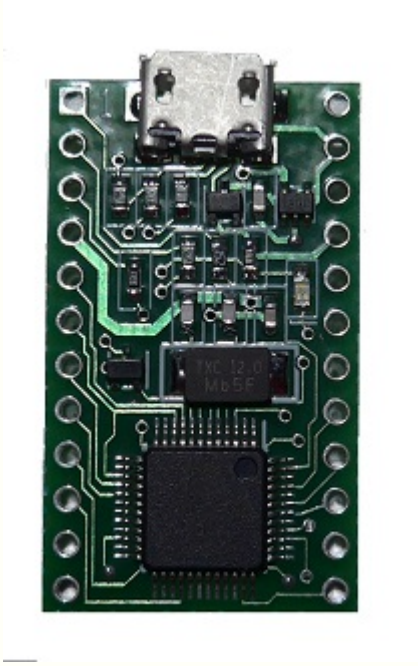
Differences from other BASICs

- no equivalent in Visual BASIC
- no equivalent in PBASIC

See also

- OUT
- IN
- Port P0..P4
- @ ' dump memory

Hardware Specs



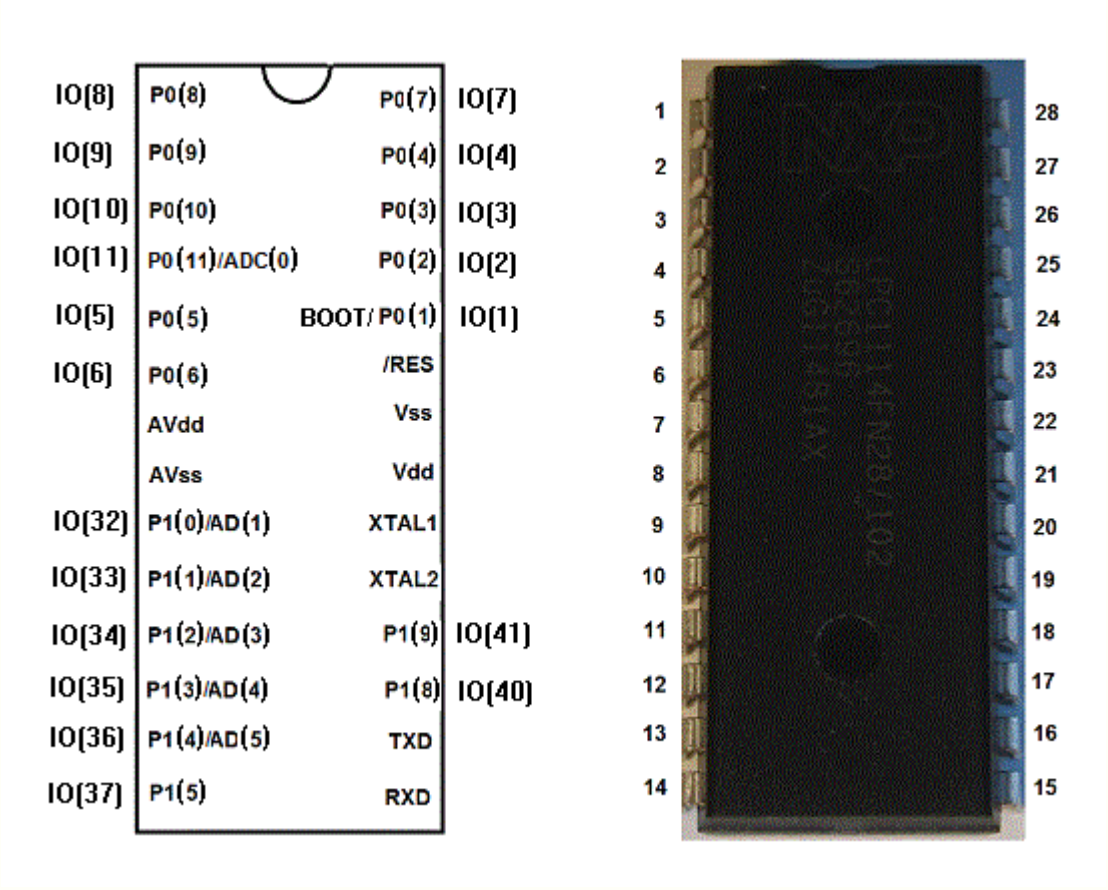
Hardware Specs

- ARMmite Pin Diagram
- ARMmite PRO Pin Diagram
- PROplus SuperPRO Pin Diagram
- ARMweb Pin Diagram
- DIN rail Pin Diagram
- ARMexpress LITE Pin Diagram
- ARMexpress Pin Diagram

- Schematics
- Suggested RS232 connection

- Power On behavior
- USB use
- USB with MatLab or legacy Serial Programs
- TTL and other interfacing
- Power
- Timing
- SPI,Microwire
- Using the I2C Bus
- ARM Peripheral Use

BASICchip Pin Diagram



The BASICchip is a complete System on a Chip, all that is required is 1.8 through 3.3V power and GND. Then just wire the available IOs into your application. No extra crystals, external memories, or second supplies required.

There are a number of ways to program the BASICchip

- **BASICboard** -- all you need for programming from a PC USB port -- includes a BASICchip in a socket
- **USBdongle** -- used by Arduino-shield compatible Coridium products -- but you will need to wire it up (pictured below)
- **FTDI cable** -- used by many people to program Arduino boards -- again pictured below, many clones of this are available
- PICaxe cable + inverter + reset switch -- these cables use inverted serial signals so a 74HC04 or equivalent is needed*

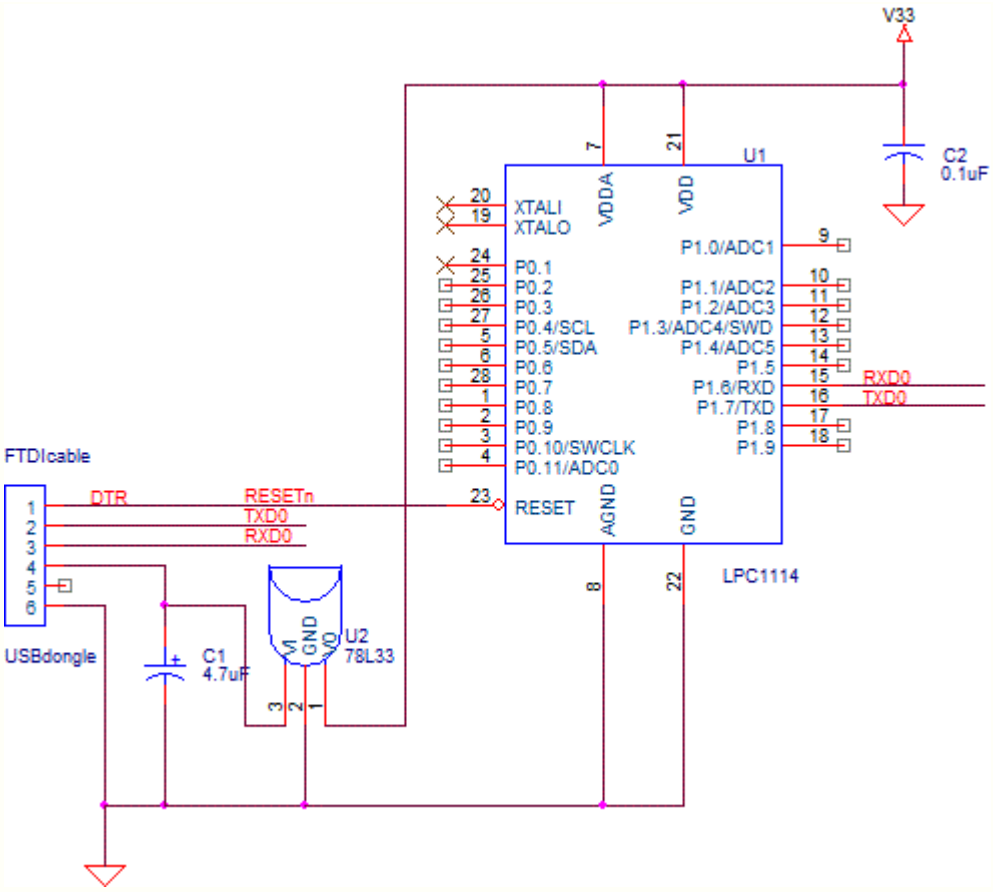
BASIC	function	pin #	alt	notes
IO(39)	TXD(0)	16	P1(7)	Serial Output, TTL compatible (active high) -- debug connection
IO(38)	RXD(0)	15	P1(6)	Serial Input, TTL compatible (active high) -- debug connection
	/RES	23	P0(0)	RESET (internal pull-up) (active low)

IO(1)	BOOT	24	P0(1)	when LOW during reset, ISP is started which disables BASIC, (internal pull-up)
IO(4) IO(5)	SCL SDA	27 5	P0(4) P0(5)	open drain outputs, can only pull down, require a pull-up resistor to drive high can be connected to internal i2c peripheral
IO(2) IO(3) IO(6) IO(7) IO(8) IO(9) IO(10) IO(11)	P0(2) P0(3) P0(6) P0(7) P0(8) P0(9) P0(10) P0(11)	25 26 6 28 1 2 3 4	AD(0)	Input/Outputs -- user controlled - 0-3.3V level 4mA drive when configured as Outputs P0.7 has a 20 mA driver 5V tolerant - use limiting resistor when connecting to a 5V supply
IO(32) IO(33) IO(34) IO(35) IO(36) IO(37) IO(40) IO(41)	P1(0) P1(1) P1(2) P1(3) P1(4) P1(5) P1(8) P1(9)	9 10 11 12 13 14 17 18	AD(1) AD(2) AD(3) AD(4) AD(5)	Input/Outputs -- user controlled 0-3.3V level 4mA drive when configured as Outputs 5V tolerant - use limiting resistor when connecting to a 5V supply
	XTAL	19 20		optional crystal connection -- do not exceed 1.8V
	VDD	21		Power 2.5-3.3V input power --do not exceed 3.3V
	GND	22		Ground (0V)
	AVDD	7		Analog power, must be equal to or less than VDD
	AGND	8		Analog Ground (0V)

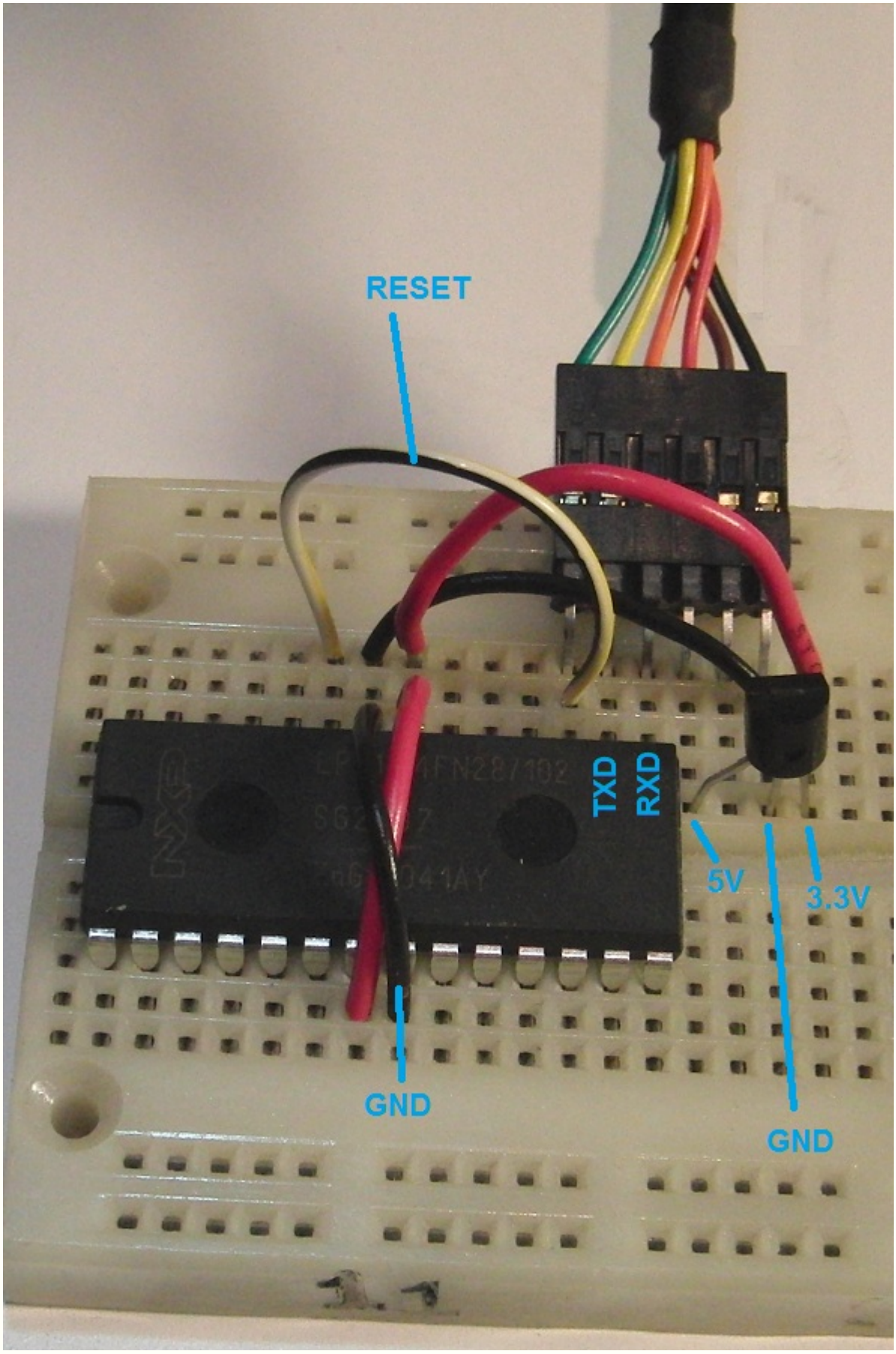
[1]These pins P0(4) and P0(5) are open-drain, when configured as outputs they can only pull down.
Port P1(x) pins can be accessed using the P1(x) keyword. They can also be accessed using IO, IN, OUT, and DIR with indexes 32-41.

Minimal requirements to program a BASICchip

For reference the minimal connections for the BASIC chip are power (supplied from a 78L33 or similar or a 2V Zener diode, with adequate bypass caps) and connections for RESET, TXD0 and RXD0. These are all that are needed to program the part from BASICtools.

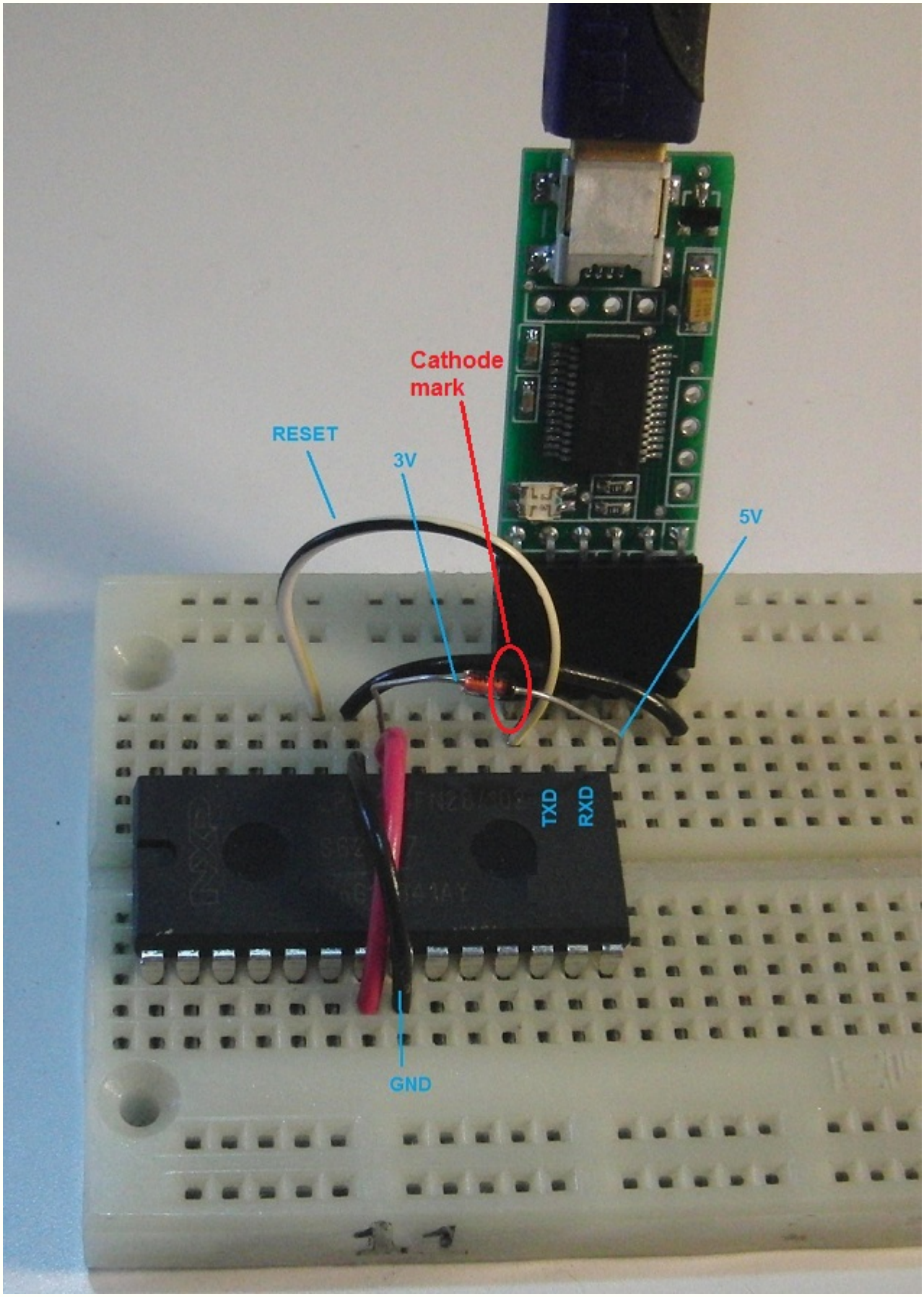


Normally you can use the internal oscillator of the LPC1114, which is set to 12MHz plus or minus 1%, This is good enough accuracy for serial communication and for most applications. In this case XTALI and XTALO are not connected. P0.1 is the BOOT line, you can use it as an output, but if you use it as an input, it must be high when RESET is asserted low, otherwise your BASIC program will not start up. Below an illustration using the **FTDI serial cable**. This setup uses the 5V from the cable and a 78L33 to generate 3.3V. You should add bypass capacitors between power and ground.



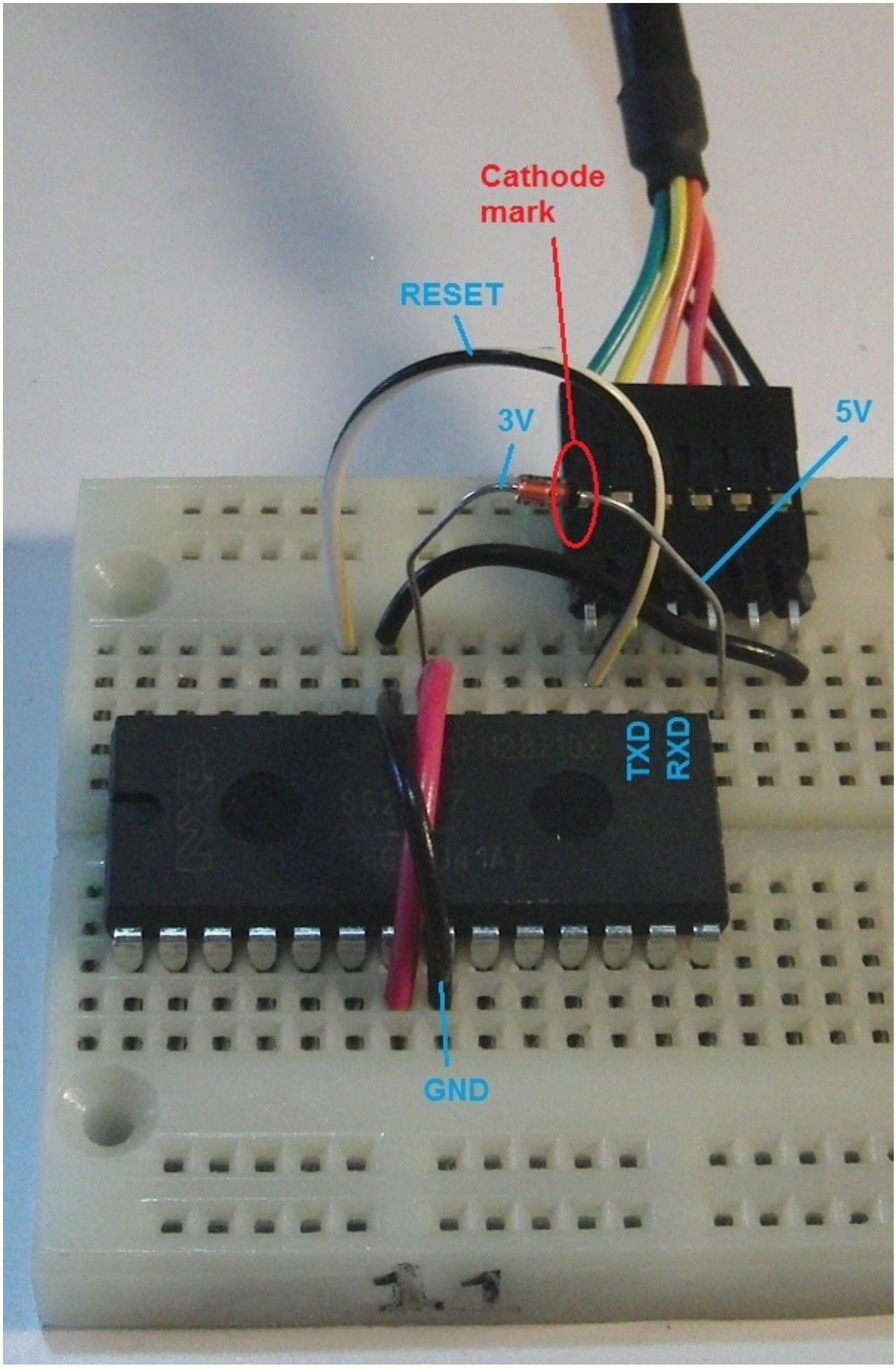
USB dongle

And a minimal hookup with a **Coridium USB dongle** below. This time using a 2V Zener diode to generate 3V from 5. Notice the diode cathode mark, remember Zener diodes are reverse breakdown devices. You should also add bypass capacitors between power and ground.



Using Zener Diode for power

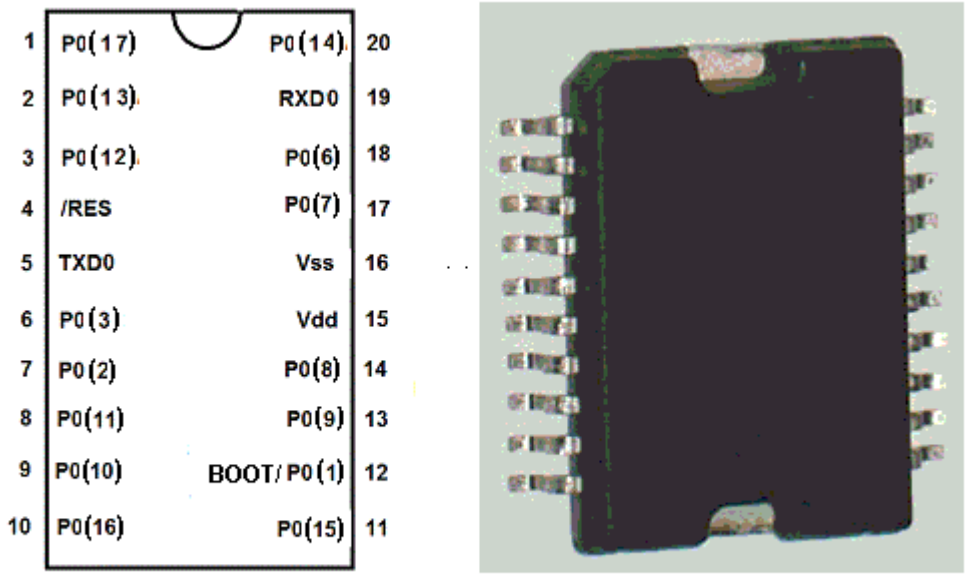
Below another illustration using the **FTDI serial cable**. This setup uses the 5V from the cable and a 2V Zener diode to keep the supply at 3V, make sure the cathode of the diode connects to the 5V, remember Zener diodes are reverse breakdown devices. You should add bypass capacitors between power and ground.



notes --

***PICaxe cable** -- You can use FTDI's utility to un-invert the signals of the PICaxe USB cable, but that is a cumbersome process.

BASICchip SO-20 Pin Diagram



The BASICchip is a complete System on a Chip, all that is required is 1.8 through 3.3V power and GND. Then just wire the available IOs into your application. No extra crystals, external memories, or second supplies required.

There are a number of ways to program the BASICchip. The first 2 methods connect directly to the breakout board.

- **USBdongle** -- used by Arduino-shield compatible Coridium products
- **FTDI cable, SparkFun FTDI breakout** -- used by many people to program Arduino boards
- PICaxe cable + inverter + reset switch -- these cables use inverted serial signals so a 74HC04 or equivalent is needed*

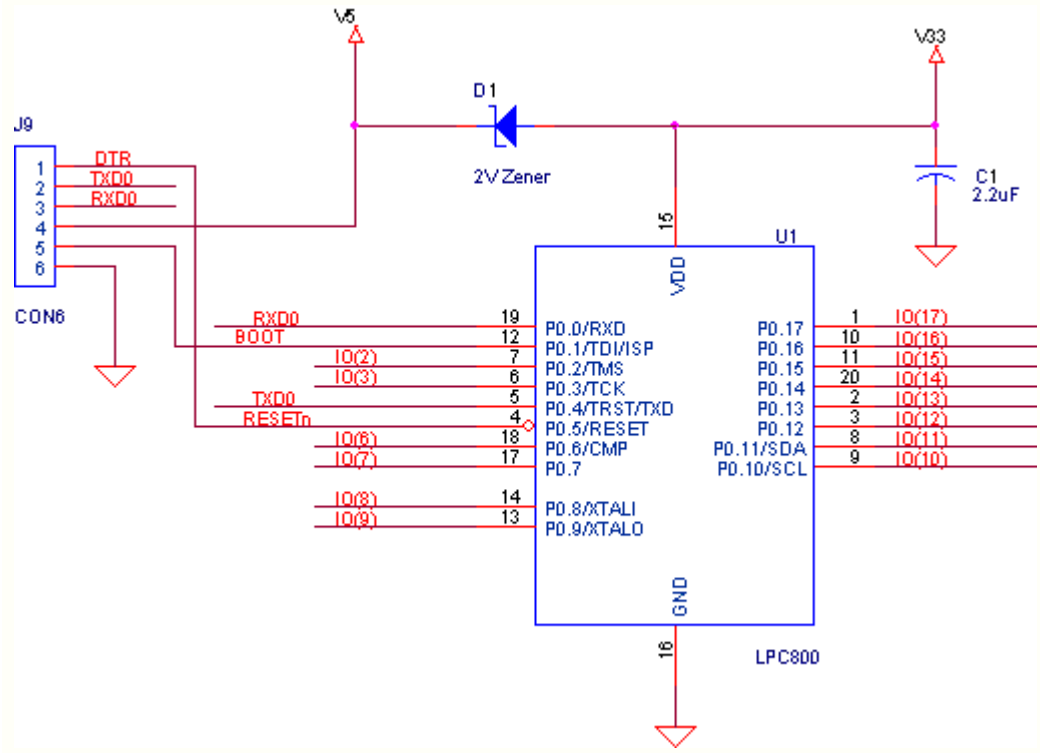
BASIC	function	pin #	alt	notes
	TXD(0)	5	P0(4)	Serial Output, TTL compatible (active high) -- debug connection
	RXD(0)	19	P0(0)	Serial Input, TTL compatible (active high) -- debug connection
	/RES	4	P0(5)	RESET (internal pull-up) (active low)
IO(1)	BOOT	12	P0(1)	when LOW during reset, ISP is started which disables BASIC, (internal pull-up)
IO(2)	P0(2)	7		Input/Outputs -- user controlled - 0-3.3V level

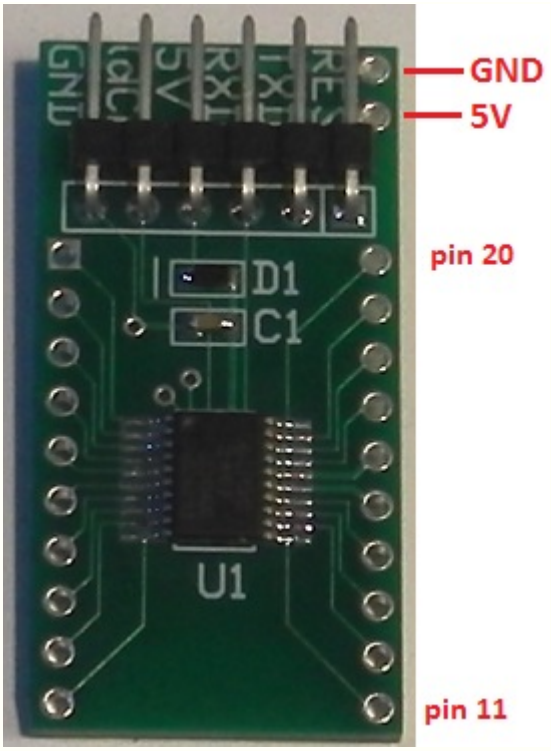
IO(3) IO(6) IO(7) IO(12) IO(13) IO(14) IO(15) IO(16) IO(17)	P0(3) P0(6) P0(7) P0(12) P0(13) P0(14) P0(15) P0(16) P0(17)	6 18 17 3 2 20 11 10 1		4mA drive when configured as Outputs P0.2, P0.3, P0.7, P0.12, P0.13 have 20 mA drivers 5V tolerant - use limiting resistor when connecting to a 5V supply
IO(10) IO(11)	I2C[1]	9 8		open drain outputs (these IOs only pulldown)
IO(8) IO(9)	XTAL	14 13		optional crystal connection -- normally IOs in BASIC
	VDD	15		Power 1.8-3.3V input power --do not exceed 3.3V
	GND	16		Ground (0V)

[1]These pins P0(10) and P0(11) are open-drain, when configured as outputs they can only pull down.

Breakout Board version -- BASICchip SO-20

For reference the minimal connections for the BASIC chip are power supplied through a 2V Zener diode, with adequate bypass caps and connections for RESET, TXD0 and RXD0. These are all that are needed to program the part from BASICtools.





Normally you can use the internal oscillator of the LPC812, which is set to 12MHz plus or minus 1%, This is good enough accuracy for serial communication and for most applications. In this case XTALI and XTALO are used as IO(8) and IO(9). IO(1) is the BOOT line, you can use it as an output, but if you use it as an input, it must be high when RESET is asserted low, otherwise your BASIC program will not start up.

CPU details

These are links to detailed documentation for the CPUs used in the Coridium products. These files are at the NXP website, when in doubt check **www.nxp.com** for the latest.

LPC1114 used in the BASICchip, PROstart and BASICboard

LPC1114 data sheet

LPC1114 user manual

LPC1756 used in the Super PRO and LPC1751 used in the PROplus

LPC1756 data sheet

LPC1756 user manual

LPC2103 used in the ARMmite, ARMexpress LITE and ARMmite PRO

LPC2103 data sheet

LPC2103 User manual

LPC2106 used in the ARMexpress

LPC2106 data sheet

LPC2106 User manual

LPC2138 used in the ARMweb

LPC2138 data sheet

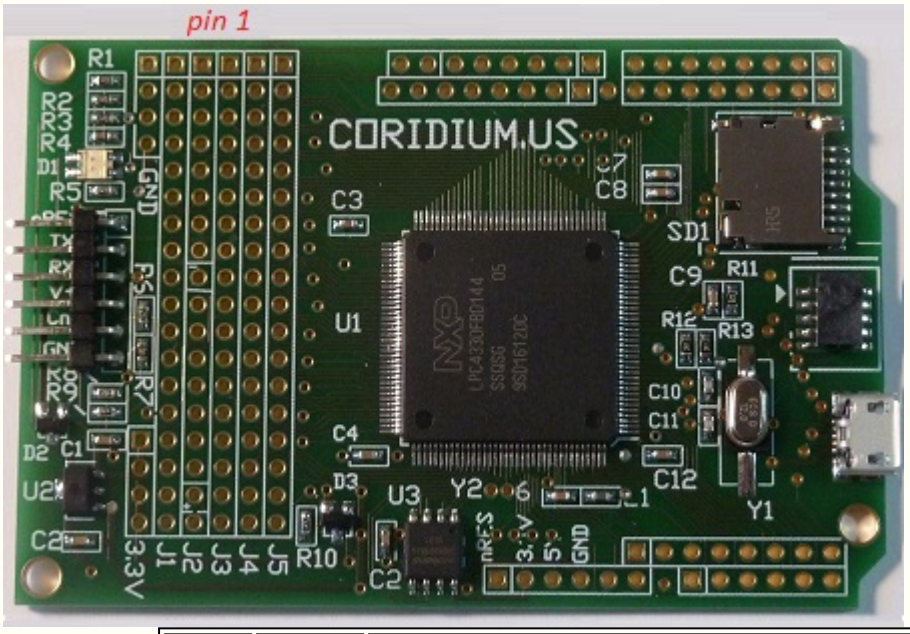
LPC2138 user manual

DATA Logger Pin Description

Pins in the Array area

The DATAlogger is the first of the multi-core CPU boards from Coridium. BASICtools supports **multi-core programming** .

Pins in the Array area



J1.1 J1.2 J1.3 J1.4 J1.5 J1.6 J1.7 J1.8 J1.9 J1.10 J1.11 J1.12 J1.13 J1.14 J1.15 J1.16 J1.17 J1.18	IO(4) IO(1) IO(0) IO(38) IO(10) IO(2) IO(11) IO(40) IO(139) IO(178) IO(5) IO(175) IO(163) IO(164) IO(165) IO(104)	P1_0 -- SGPIO7 P0_1 -- SGPIO1 P0_0 -- SGPIO0 P1_13 -- SGPIO9 P1_3 --SGPIO10 P1_15 -- SGPIO2 P1_4 -- SGPIO11 P1_5 -- SGPIO15 P9_6 -- SGPIO8 P9_5 -- SGPIO3 P6_6 -- SGPIO5 P6_7 -- SGPIO6 P2_2 -- SGPIO12 P2_4 -- SGPIO13 P2_5 -- SGPIO14 P7_0 -- SGPIO4 WAKEUP PF_4 -- CLKIN
J2.1 J2.2 J2.3 J2.4 J2.5 J2.6 J2.7 J2.8	IO(8) IO(9) .. IO(35) IO(39) IO(173) NC IO(41)	P1_1 -- pulled up during RESET (required for correct boot) P1_2 -- pulled down during RESET (required for correct boot) CLK0 P1_10 P1_14 P4_9 .. P1_6 ..

J2.9	GND	P6_2
J2.10	IO(97)	P2_7 -- pulled up during RESET (required for correct boot) --
J2.11	IO(7)	connected to LED and ISP BOOT pin
J2.12	IO(167)	P2_8 -- pulled down during RESET (required for correct boot)
J2.13	IO(102)	P6_10
J2.14	IO(14)	P2_10
J2.15	IO(70)	P4_6
J2.16	..	P3_0
J2.17	Vbat	..
J2.18	..	RTC_ALARM
J3.1	IO(174)	P4_10
J3.2	IO(34)	P1_9
J3.3	IO(37)	P1_12
J3.4	IO(75)	P5_2
J3.5	IO(71)	P5_7
J3.6	IO(13)	P1_18
J3.7	IO(96)	P6_1
J3.8	TXD0	P2_0
J3.9	RXD0	P2_1
J3.10	..	P4_7
J3.11	..	I2C0C
J3.12	IO(101)	P6_9
J3.13	IO(42)	P2_9 -- pulled down during RESET (required for correct boot)
J3.14	IO(44)	P2_12
J3.15	IO(105)	P7_1
J3.16	IO(107)	P7_3
J3.17	IO(66)	P4_2
J3.18	..	RESETn
J4.1	IO(73)	P5_0
J4.2	IO(33)	P1_8
J4.3	IO(36)	P1_11
J4.4	IO(77)	P5_4
J4.5	IO(3)	P1_16
J4.6	..	P1_19
J4.7	..	P6_0
J4.8	IO(99)	P6_4
J4.9	IO(162)	P2_2
J4.10	IO(172)	P4_8
J4.11	..	USB1P
J4.12	..	I2C0D
J4.13	IO(103)	P6_11
J4.14	IO(43)	P2_11
J4.15	IO(168)	P3_1
J4.16	IO(106)	P7_2
J4.17	IO(68)	P4_4
J4.18	IO(110)	P7_6
J5.1	IO(74)	P5_1
J5.2	IO(32)	P1_7
J5.3	IO(76)	P5_3
J5.4	IO(78)	P5_5
J5.5	IO(79)	P5_6
J5.6	IO(12)	P1_17
J5.7	IO(15)	P1_20
J5.8	IO(98)	P6_3
J5.9	IO(100)	P6_5
		P6_8

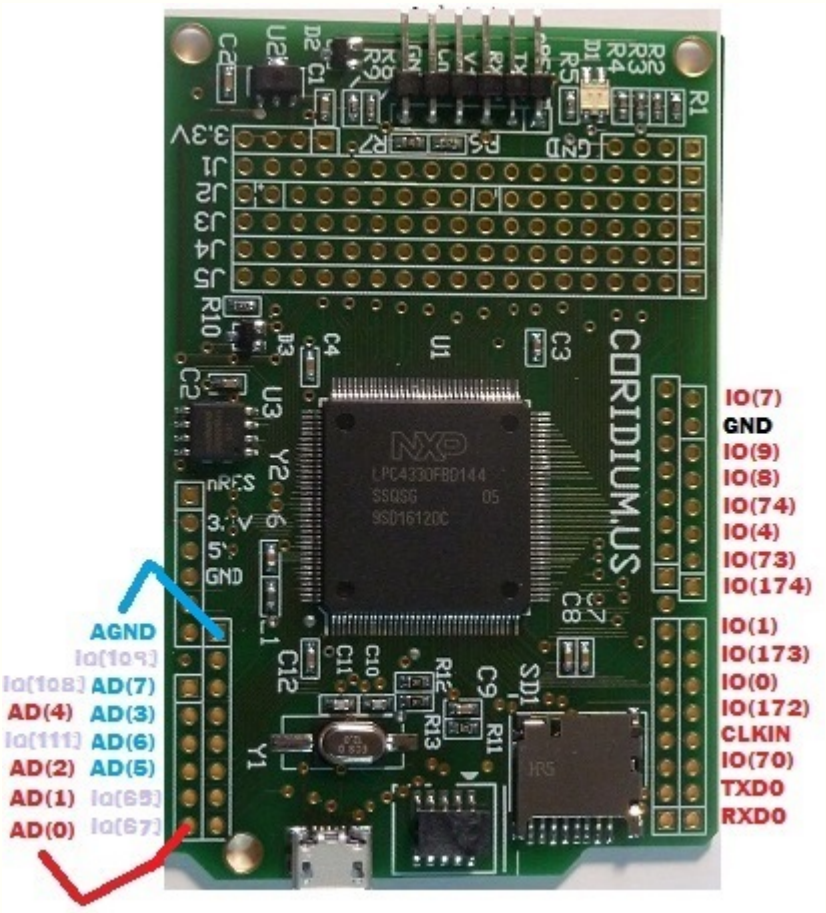
J5.10	IO(176)	USB1M
J5.11	..	P2_6
J5.12	IO(166)	CLK2
J5.13	..	P6_12
J5.14	IO(72)	P2_13
J5.15	IO(45)	P3_2
J5.16	IO(169)	P4_0
J5.17	IO(64)	
J5.18	IO(69)	P4_5

Pins that are pulled up or down during RESET can be used as inputs if during RESET your circuitry does not drive them or if it drives them to the required state for boot. They can all be used as outputs after RESET. P2_7 should be driven low during RESET to enter ISP mode which is used for loading C programs, in that case the other BOOT select pins are ignorred.

Some of these digital pins are also connected to the Arduino digital pins.

In most NXP parts the port designation corresponds to the GPIO port, this is **NOT** the case with the LPC4330. See the NXP user manual for detailed pin assignments.

Analog connections and Arduino pin connections



AD pins

With the DATAlogger, the AD converters are connected to 8 dedicated analog pins. ADC0 can be used as an analog input, or the output of the DAC.

Alternate Pin functions

The LPC4330 supports a number of dedicated functions. Those include 4 UARTs, USB, 2 SSPs, 1 SPI, 2 CAN, 2 I2C, I2S, 2 multi-channel PWMs, Quadrature Encoder, dedicated motor control PWM, interrupts, timer counter capture and match.

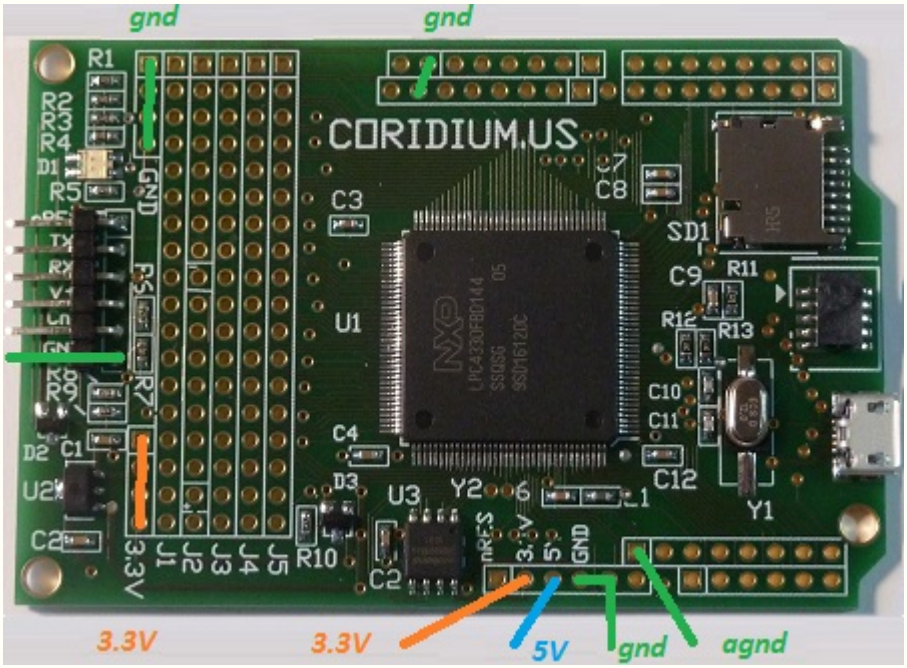
In addition most can be configured with pull-ups and default to pull-ups following reset.

Details can be found in NXP's User manual.

UARTs are enabled by calling BAUD(x) for x=0 to 3. UART0 is enabled by default as the programming debug connection. The pin assignment BASIC uses is in the following table (you can change the settings by changing the PINSEL registers, details in the NXP User Manual)

UART	BASIC	NXP	UART
RXD(0)	IO(161)	P2_1	UART0
TXD(0)	IO(160)	P2_0	
RXD(1)	IO(39)	P1_14	UART1
TXD(1)	IO(38)	P1_13	
RXD(2)	IO(3)	P1_16	UART2
TXD(2)	IO(2)	P1_15	
RXD(3)	IO(164)	P2_4	UART3
TXD(3)	IO(163)	P2_3	

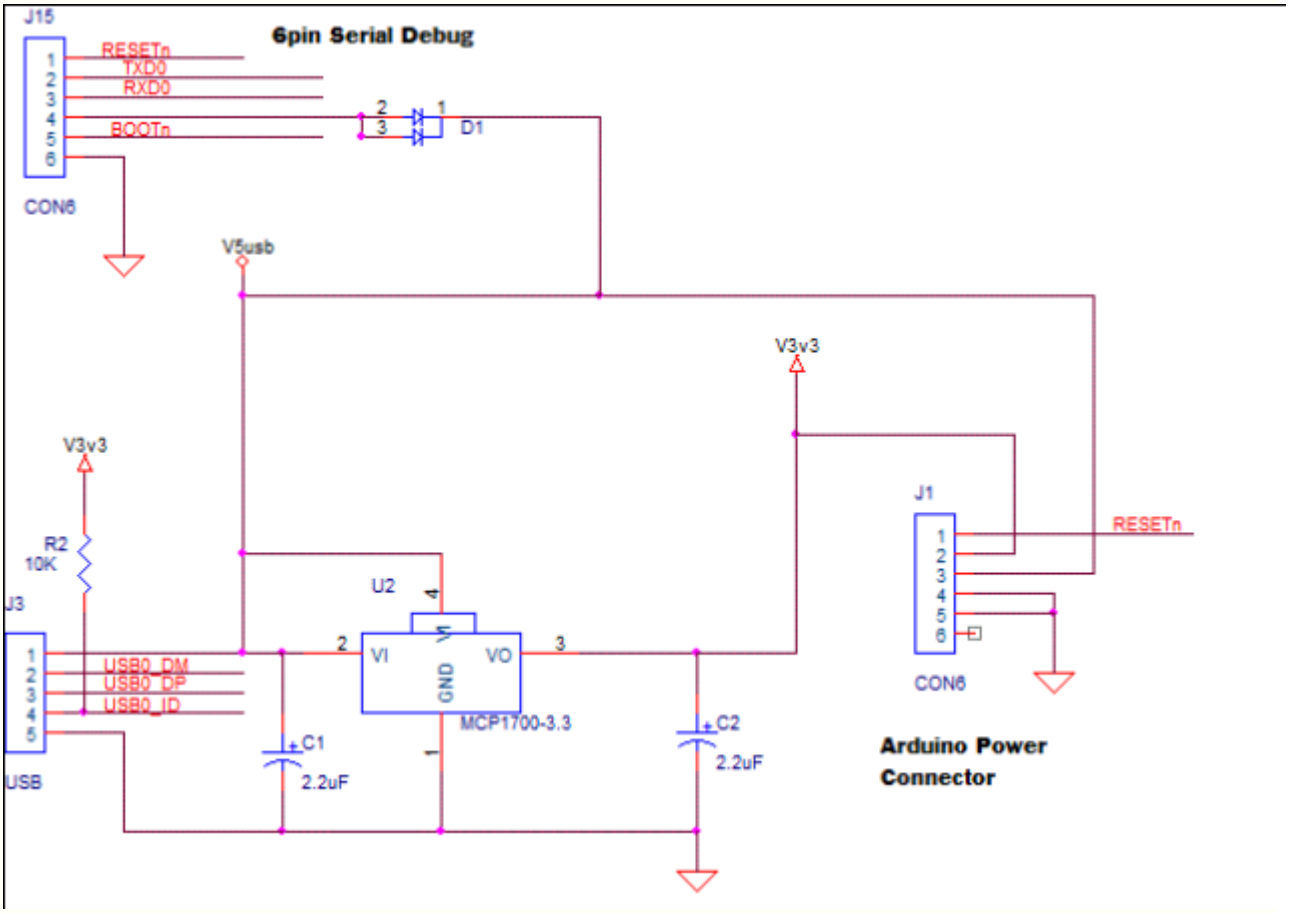
Power connections



Diode steering allows 5V power to be supplied from either the USB connector or the 6 pin Serial Debug connector.

If you intend to supply 5V power from the Arduino 6 pin power connector, you should not connect the USB connector.

The schematic describes this circuit

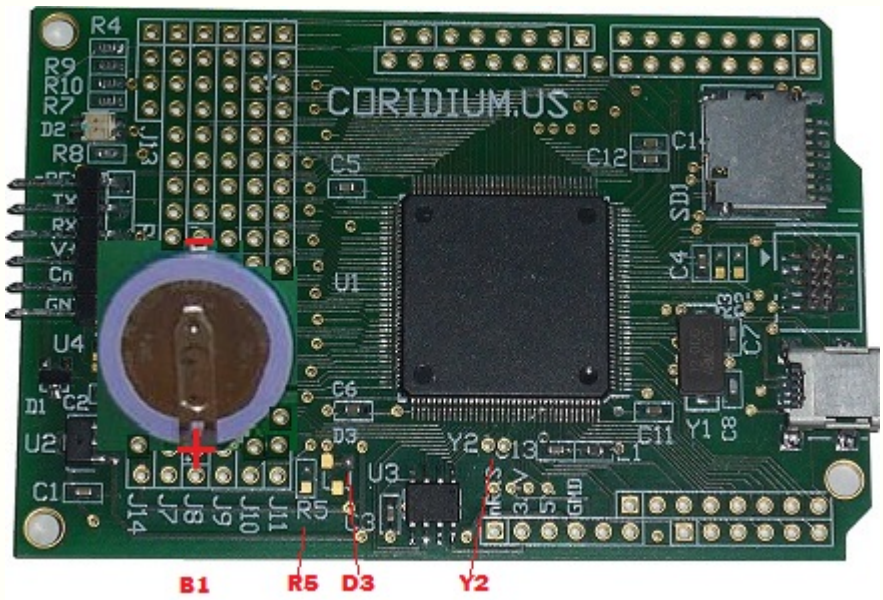


The **full schematic can be seen here**

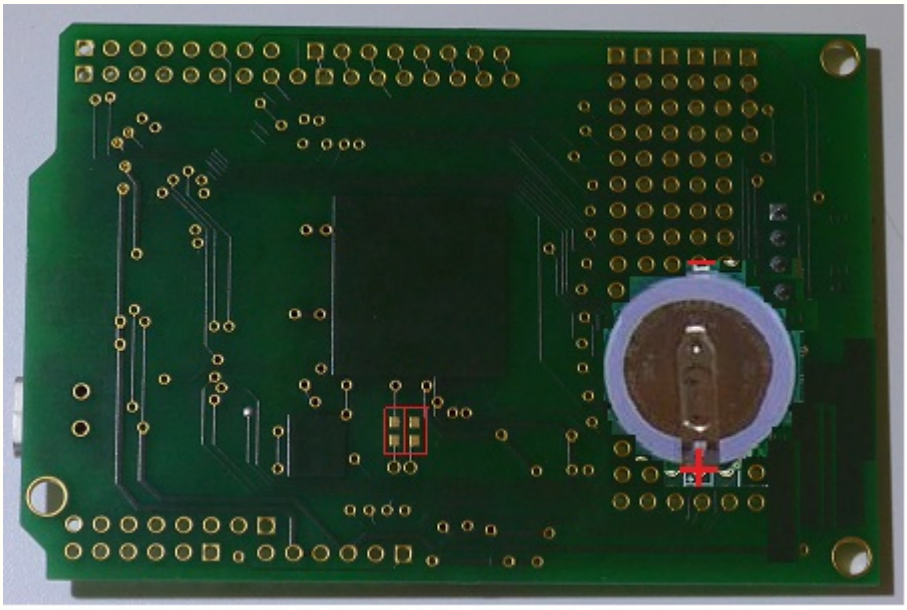
Real Time Clock Oscillator

The DATAlogger has a provision to load a 32 KHz crystal (Y2) and 2 20pf caps (C9 and C10) to use that for the Real Time Clock. It can also add battery backup by loading the ML2020 battery (on either top or bottom of board), 330 ohm resistor (R5) and a BAT54C diode (D3).

The crystal should be a 32.768 KHz can type with an 18pF rating the capacitors are 0603 size 20pF.


CORIDIUM.US
B1
R5
D3
Y2

..


back side
C9 C10

LPC11U37 Details

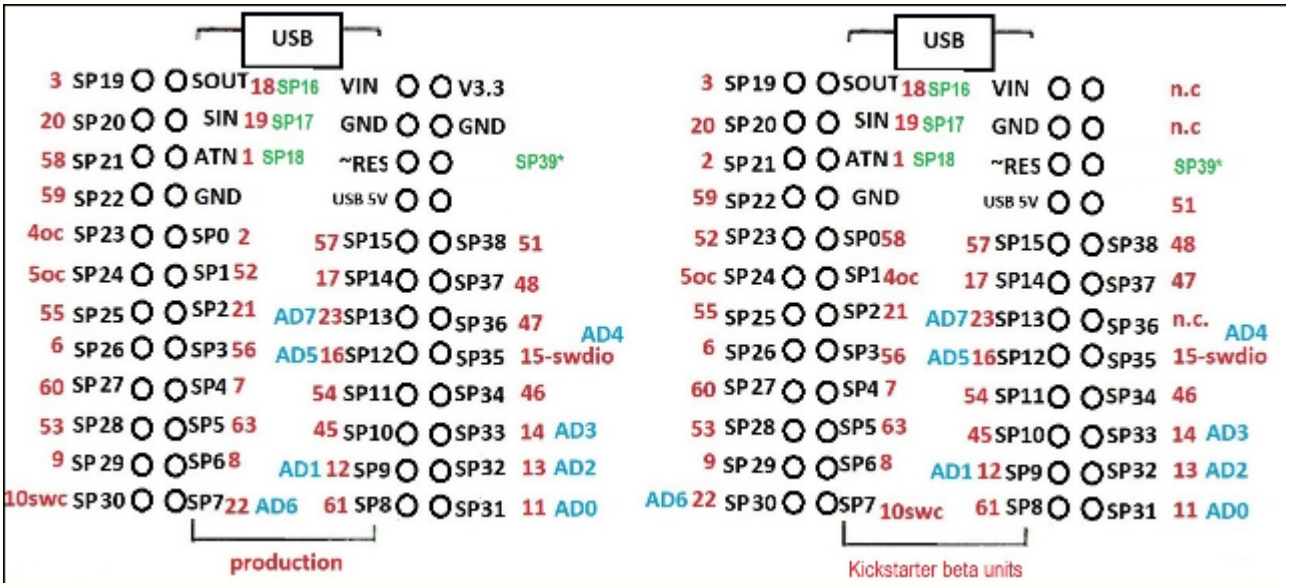
The LPC11U37 is footprint and pin superset of the old Parallax BASIC stamp ®. The board can be used with 5V TTL signals. It has a total of 40 IO pins available.

BASIC or C programs can be downloaded using the builtin USB connector. MakeItC and BASICtools use the USB serial connection of the board. You can load mBed programs by shorting the ATN and GND pins (3 and 4 of the stamp arrangement) which then bring the device up as a disk drive that you can overlay the firmware.bin file. It can be ordered with an optional JTAG/SW connector so that it can use Keil, IAR, MCUxpresso or other tools. The board can also be ordered as a JTAG/SW debugger that can connect to other ARM CPUs.

Digital IO connections

Below is a diagram of the pins, with the USB connector shown at the top. The diagram on the right ONLY applies to the handful of alpha test boards shipped via eBay or Kickstarter beta testers.

The NXP pins P0.1-31 shown as 1-31 and P1.0-31 as 32-63 below in red. For convenience the pins can be refered to as SP0-39 as stamp pins by using the definitions in STAMP_PINS.bas and STAMP_PINS.h (shown in black and green below).



Analog pins

There is an A/D converter with an 8 input analog multiplexor. After power on these pins are inputs, when a program calls for an AD input they are converted to analog.

Special purpose pins

RESET pin starts the ARM program if the ATN/SP18 (P0.1) pin is high. If you use ATN/SP18 P0.1 as an input you MUST make sure it is in the high state when RESET is asserted, otherwise your program

will NOT start. RESET can also be defined as a IO, though you must enable it with a specific call. On power on the RESET pin has an internal pullup and must be high during this time or your program will not start.

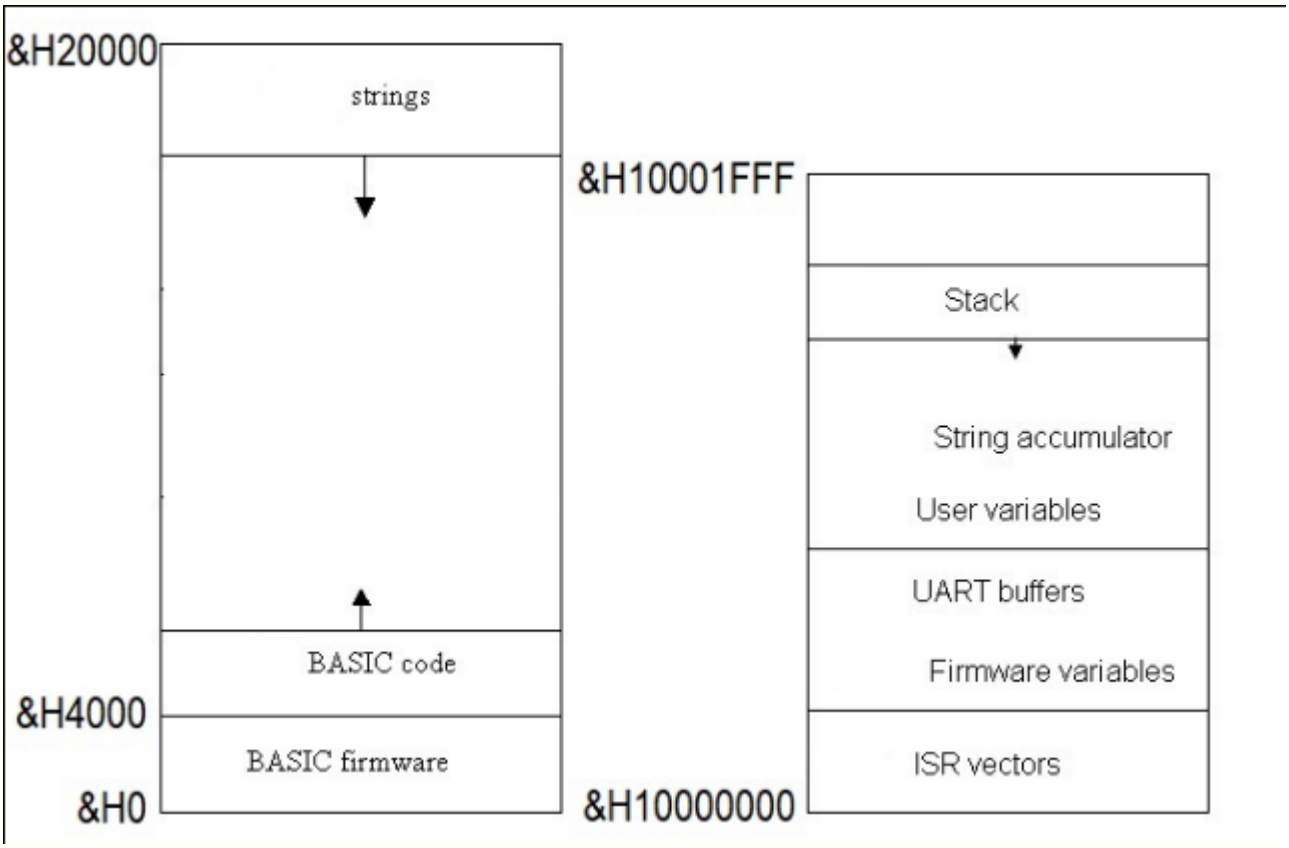
SOUT and SIN (SP17-18) are configured as inputs on reset. They are converted to TTL serial lines (positive true) when **BAUD(1)** is set. The USB serial connection is TXD(0), RXD(0) and configured by default on reset.

Pins are configured as digital inputs on power on, SP23 and SP24 are open collector pins, which can be used as IOs or dedicated I2C pins. The Serial Wire debug pins (SP30 SP35) are converted to inputs by the initial firmware, but can be used by SW/JTAG pins by other tools such as Keil, IAR and MCUxpresso.

The LPC11U37 supports a number of dedicated functions. Those include 1 UARTs, 2 SSP/SPI, 1 I2C, I2S, 2 multi-channel PWMs,timer counter capture and match.

Details can be found in **NXP's User manual** .

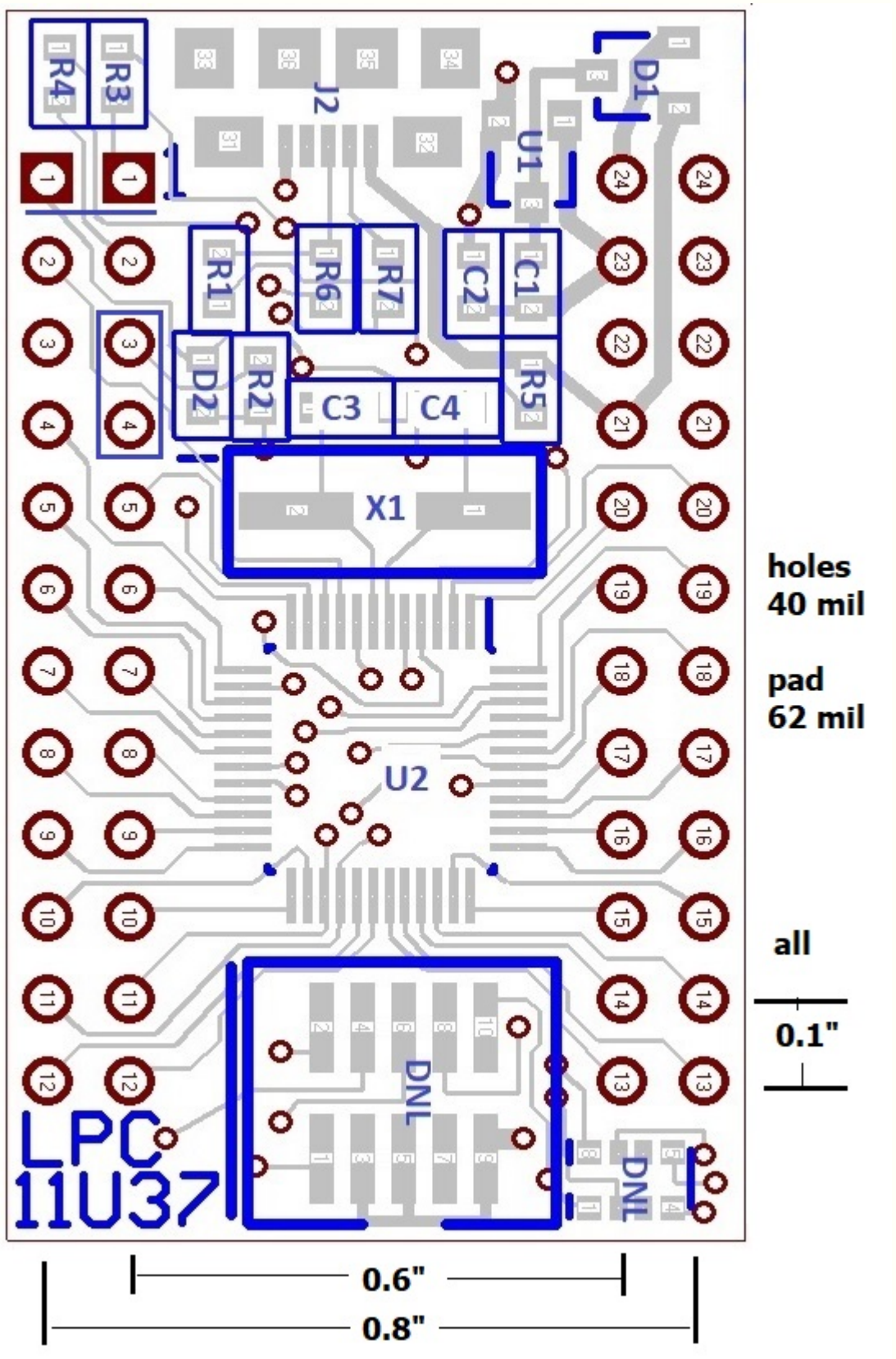
Memory Map



The **full schematic can be seen here**

The internal 24 stamp pins are on a 0.1" DIP spacing with a 0.6" width. An additional 24 pins are also 0.1" space with a 0.8" width.

Part placement



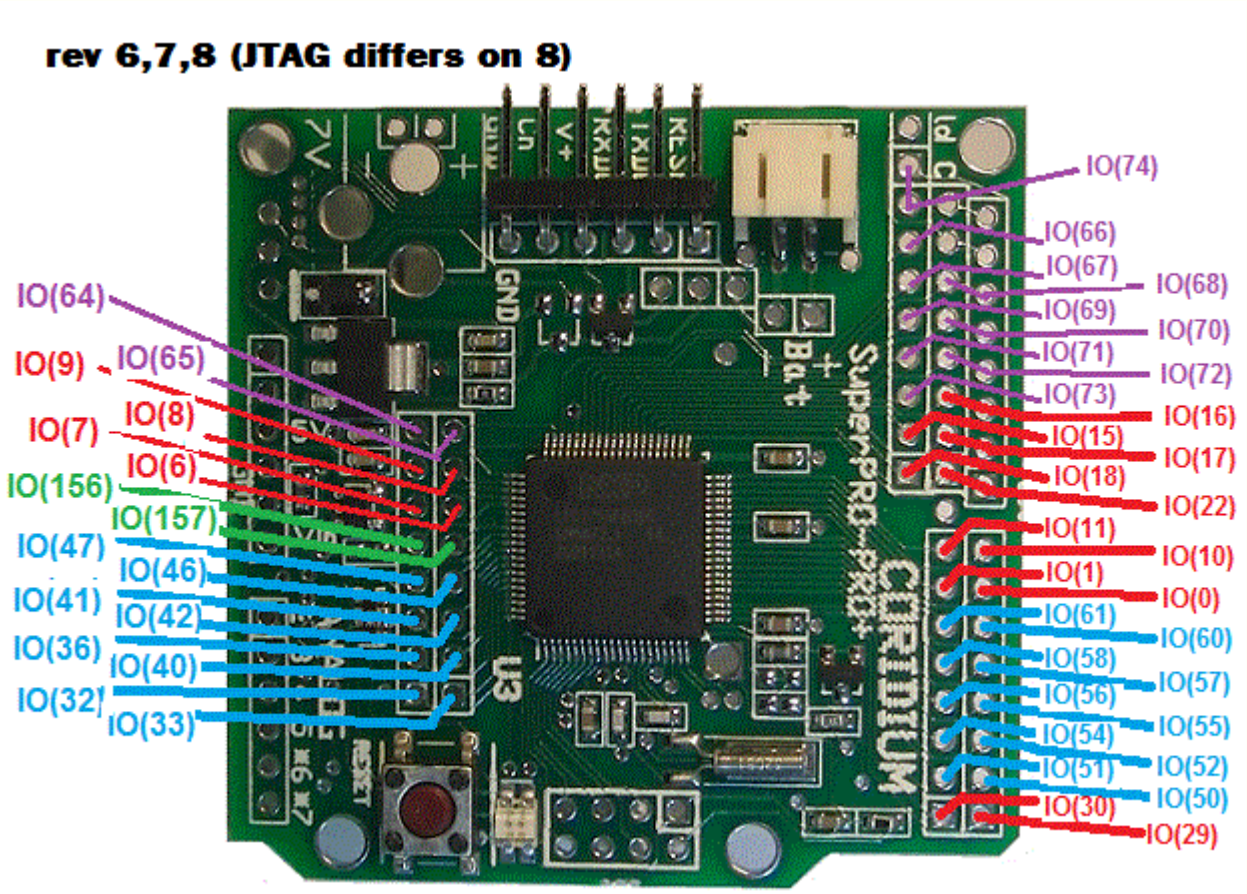
LPC4078 Details

The LPC4078 is footprint and pin compatible with the Arduino PRO, as well as Coridium's SuperPRO and PROplus. The board can be used with 5V TTL signals.

BASIC or C programs can be downloaded using the installed test connector using the USB dongle contained in Coridium's evaluation kit or using the **SparkFun USB Basic Breakout board** or FTDI cable from **Digikey**. More details on **these connections here**.

Digital IO connections

Below is a diagram of the pins, note it has been rotated 90 degrees to make it easier to read,



IO(32) is the equivalent of P1(0) and can be accessed either way.

Special purpose pins

RESET pin starts the ARM program if the BOOT(P2.10) pin is high. If you use P2.10 as an input you MUST make sure it is in the high state when RESET is asserted, otherwise your program will NOT

start.

UARTs are enabled by calling BAUD(x) for x=0 to 3. UART0 is enabled by default as the programming debug connection. The pin assignment BASIC uses is in the following table (you can change the settings by changing the PINSEL registers, details in the NXP User Manual)

UART	BASIC	NXP	UART
RXD(0)	IO(3)	P0(3) / AD(6)	UART0
TXD(0)	IO(2)	P0(2) / AD(7)	
RXD(1)	IO(65)	P2(1)	UART1
TXD(1)	IO(64)	P2(0)	
RXD(2)	IO(73)	P2(9)	UART2
TXD(2)	IO(72)	P2(8)	
RXD(3)	IO(157)	P4(29)	UART3
TXD(3)	IO(156)	P4(28)	

Analog connections

4 A/D converters are readily available, Analog 2-4. 2 more are available, but share the pins with UART0 -- what was NXP thinking, I have no idea.

1 10 bit DAC is available shared with AD(3) available on the SuperPRO (not on PROplus)

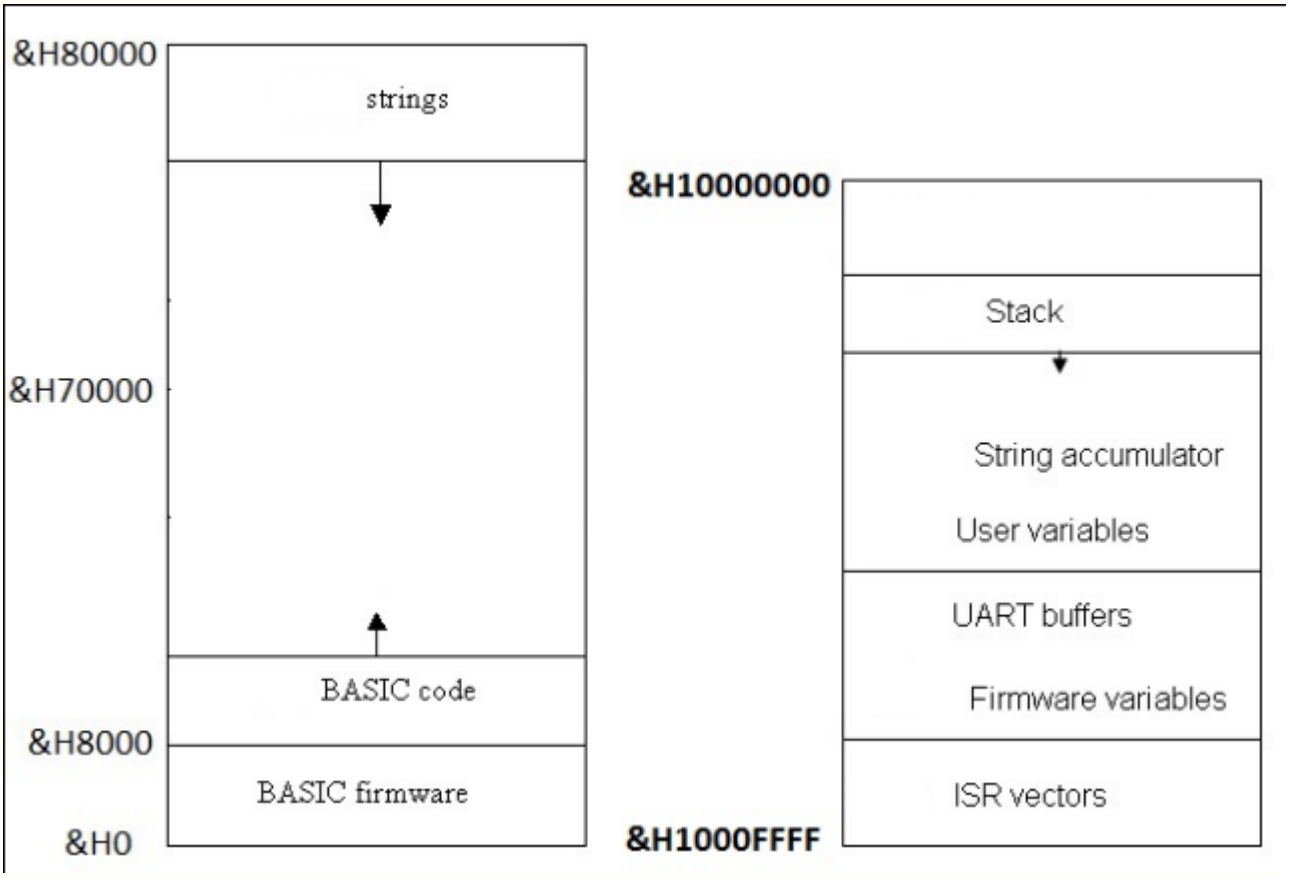
On reset or power up the AD pins are configured by software as AD inputs. To change those to digital IOs, the user must write to the appropriate PINSEL register, or with version 8.11 firmware or later you can change it to an IO by accessing the corresponding IO pin in the following table.

AD	BASIC	NXP
AD(2)	IO(25)	P0(25),DACOUT
AD(3)	IO(26)	P0(26)
AD(4)	IO(62)	P1(30)
AD(5)	IO(63)	P1(31)
AD(6)	IO(3)	RXD(0)/P0(3) / AD(6)
AD(7)	IO(2)	TXD(0)/P0(2) / AD(7)

The LPC4078 does support an external reference for the A/D converters, but to use the Arduino AREF pin a jumper is required (details on the schematic).

Details can be found in NXP's User manual.

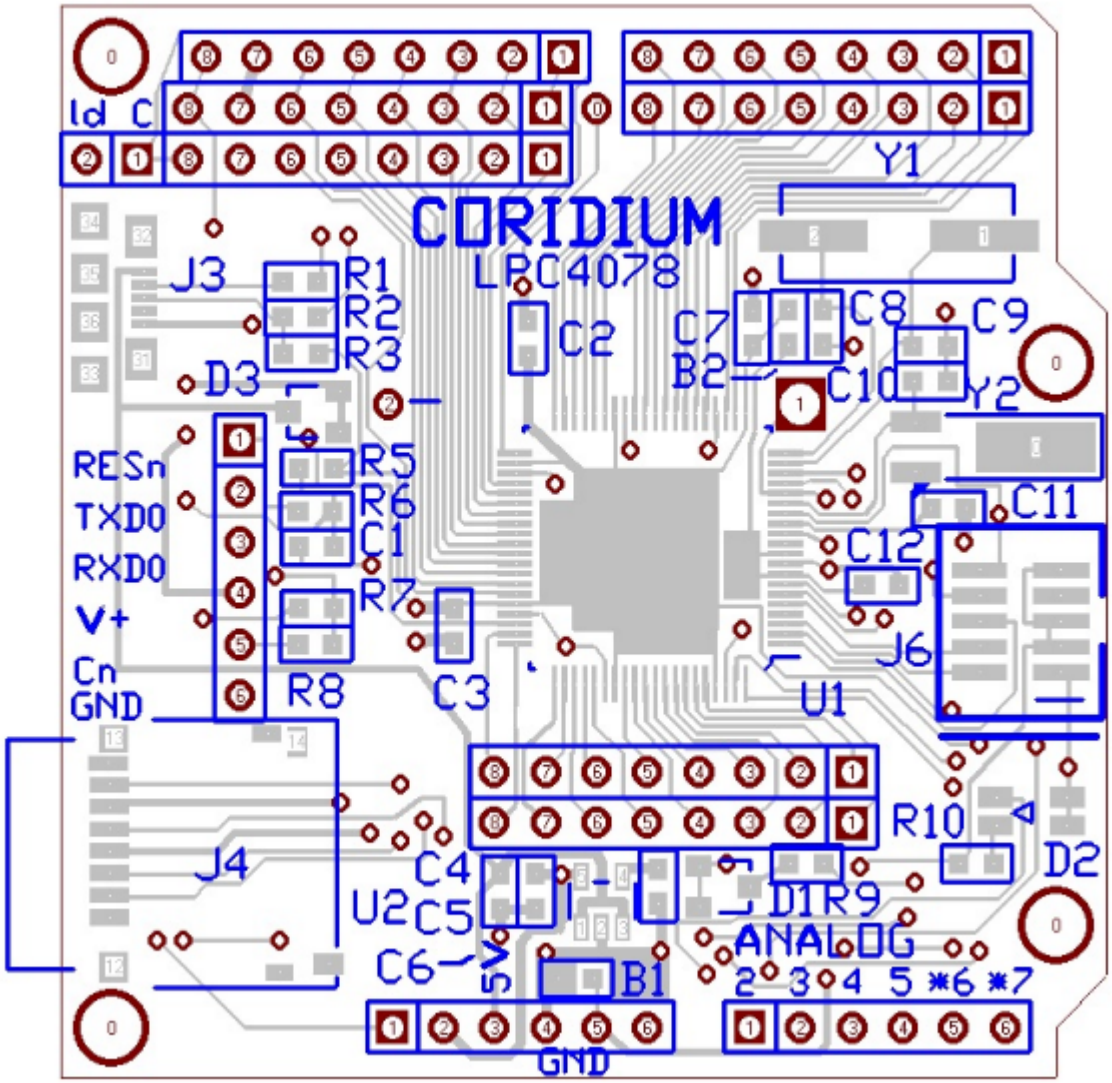
Memory Map



The **full schematic can be seen here**

And the **DXF** file

Part placement



SDcard sockets and USB sockets are not normally loaded, but they can be loaded by special order (minimum charges apply).

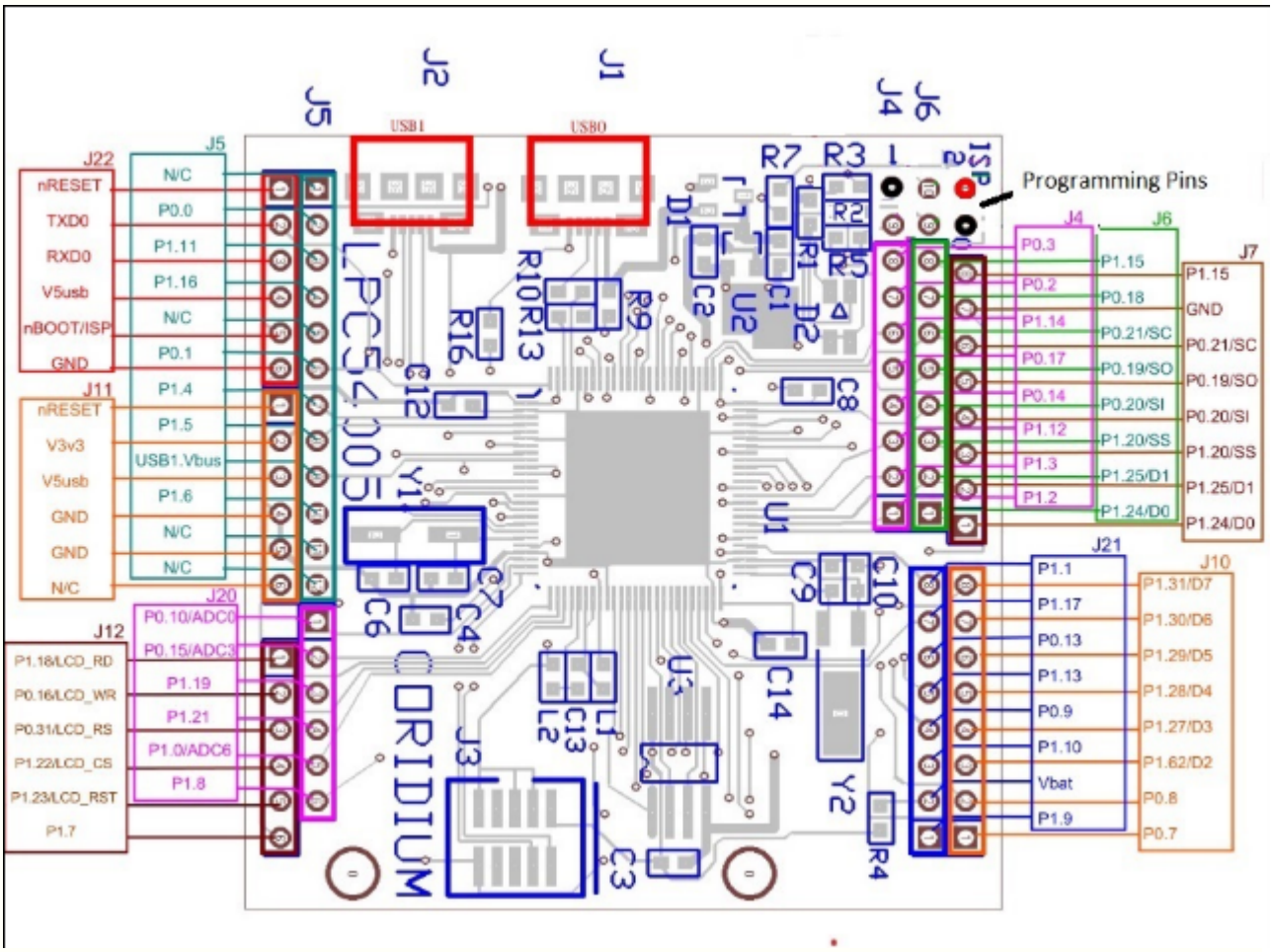
LPC54005 Details

The LPC54005 is footprint and pin compatible with the Arduino PRO. The board can be used with 5V TTL signals.

BASIC or C programs can be downloaded using the installed using the USB0 connector.

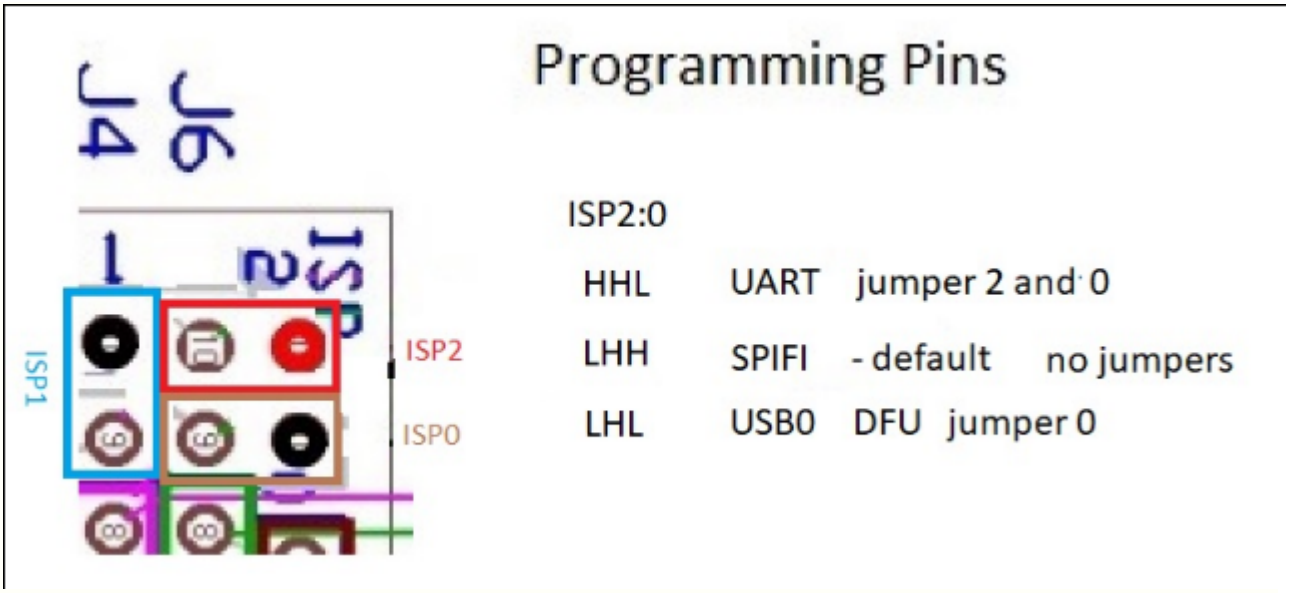
Digital IO connections

Below is a diagram of the pins, note it has been rotated 90 degrees to make it easier to read,



Special purpose pins

RESET pin starts the ARM program depending on the state of the ISP pins. With no jumpers the ARM program in QSPI (SPIFI) Flash is copied to RAM and then executed.

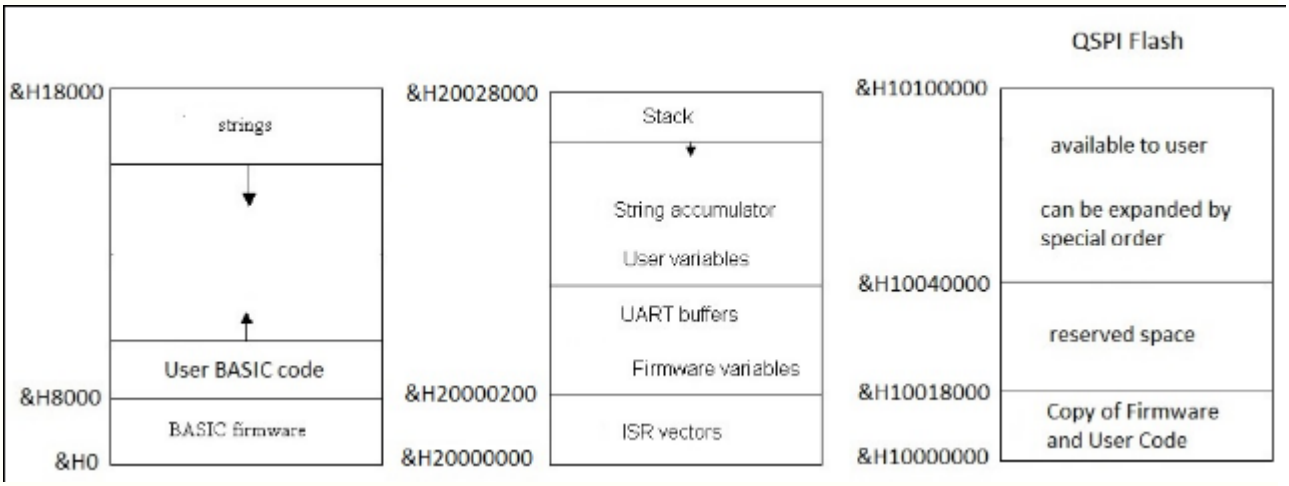


The LPC54005 supports a number of dedicated functions. Those upto 11 Flexcoms that can be USARTs, SPI, or I2C, 2 Flexcomms support I2S, 5 timer/counters with PWM, and 1 State Controller Timer.

In addition most can be configured with pull-ups and default to pull-ups following reset.

Details can be found in NXP's User manual.

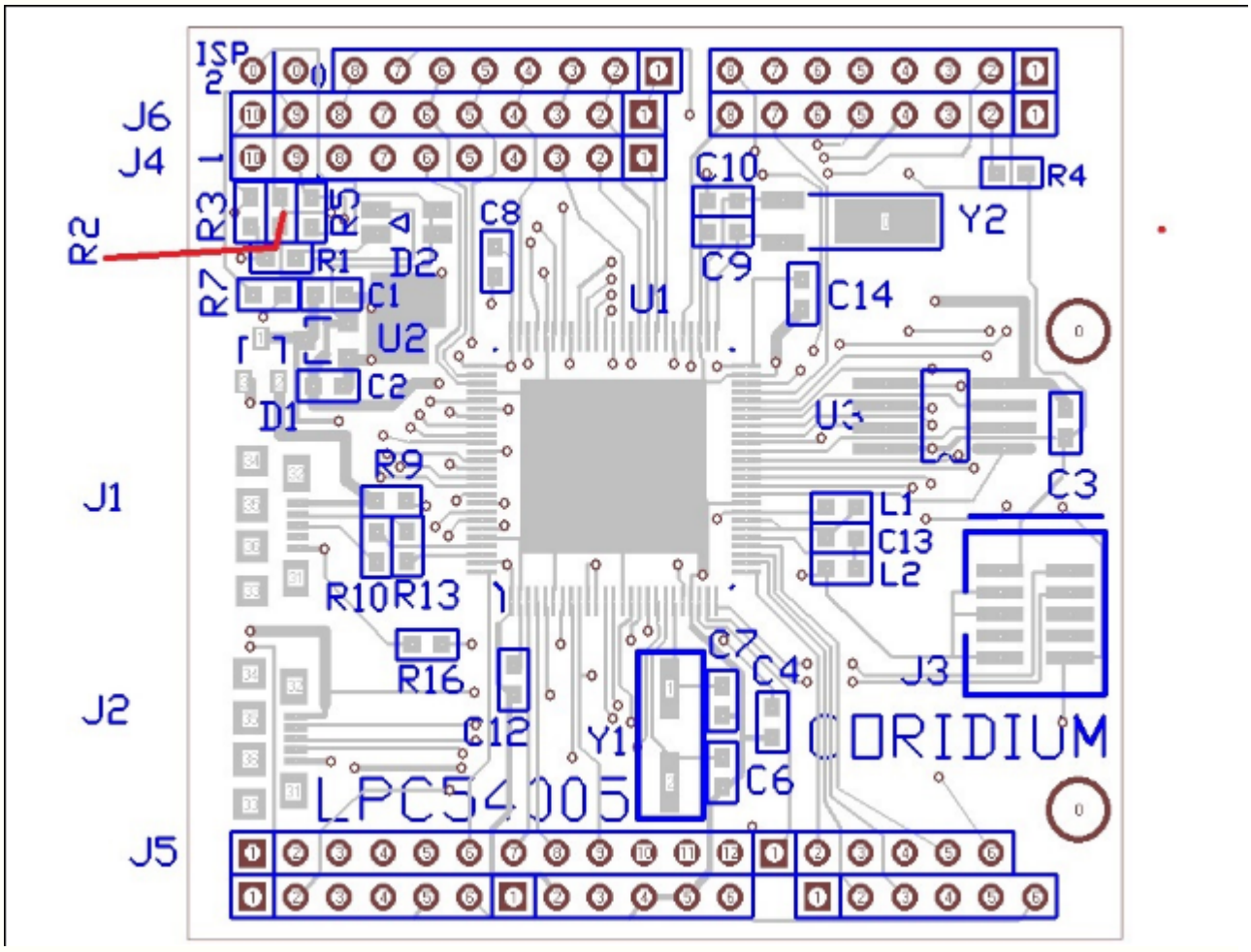
Memory Map



The **full schematic can be seen here**

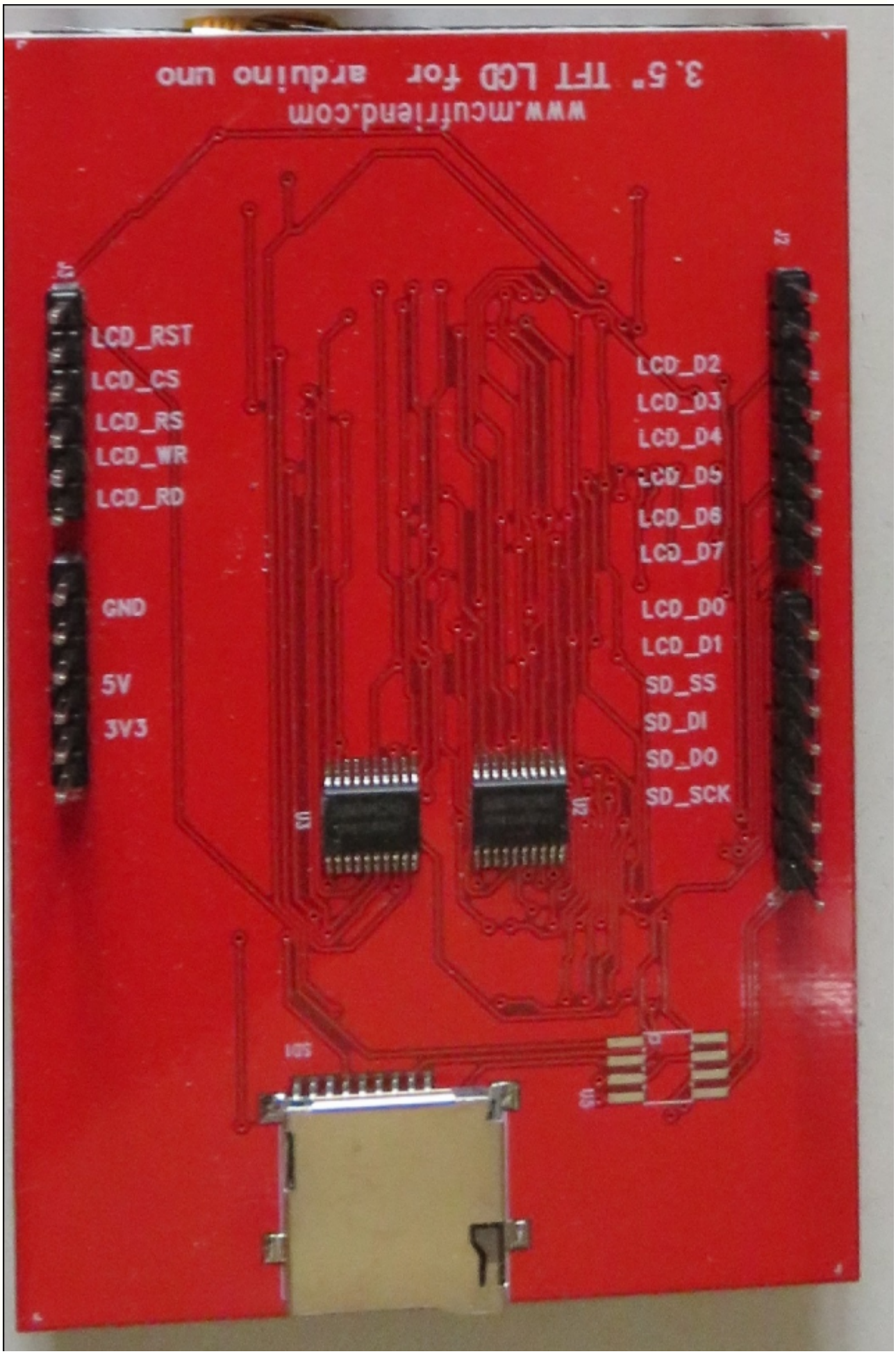
And the **DXF** file

Part placement



LCD pin out

This board has been designed to mate to LCD boards for the Arduino Uno (pin out picture below). Many of these boards also have an SD card and touch screen, that we have published example files for.

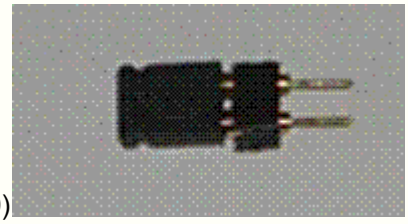

3.5" TFT LCD for arduino uno
www.mcufriend.com
LCD_RST
LCD_CS
LCD_RS
LCD_WR
LCD_RD
GND
5V
3V3
LCD_D2
LCD_D3
LCD_D4
LCD_D5
LCD_D6
LCD_D7
LCD_D0
LCD_D1
SD_SS
SD_DI
SD_DO
SD_SCK

Firmware Installation

First download the NXP **LPCScrypt installer** . Newer version may be available at the NXP site. Run that installer.

Open a command prompt from Windows, and go to the LPCScrypt install directory -- typically C:\NXP\LPCScrypt\bin

Add jumper 0 to the ISP lines and connect USB0. (we typically use 2 pin header with a shorting block



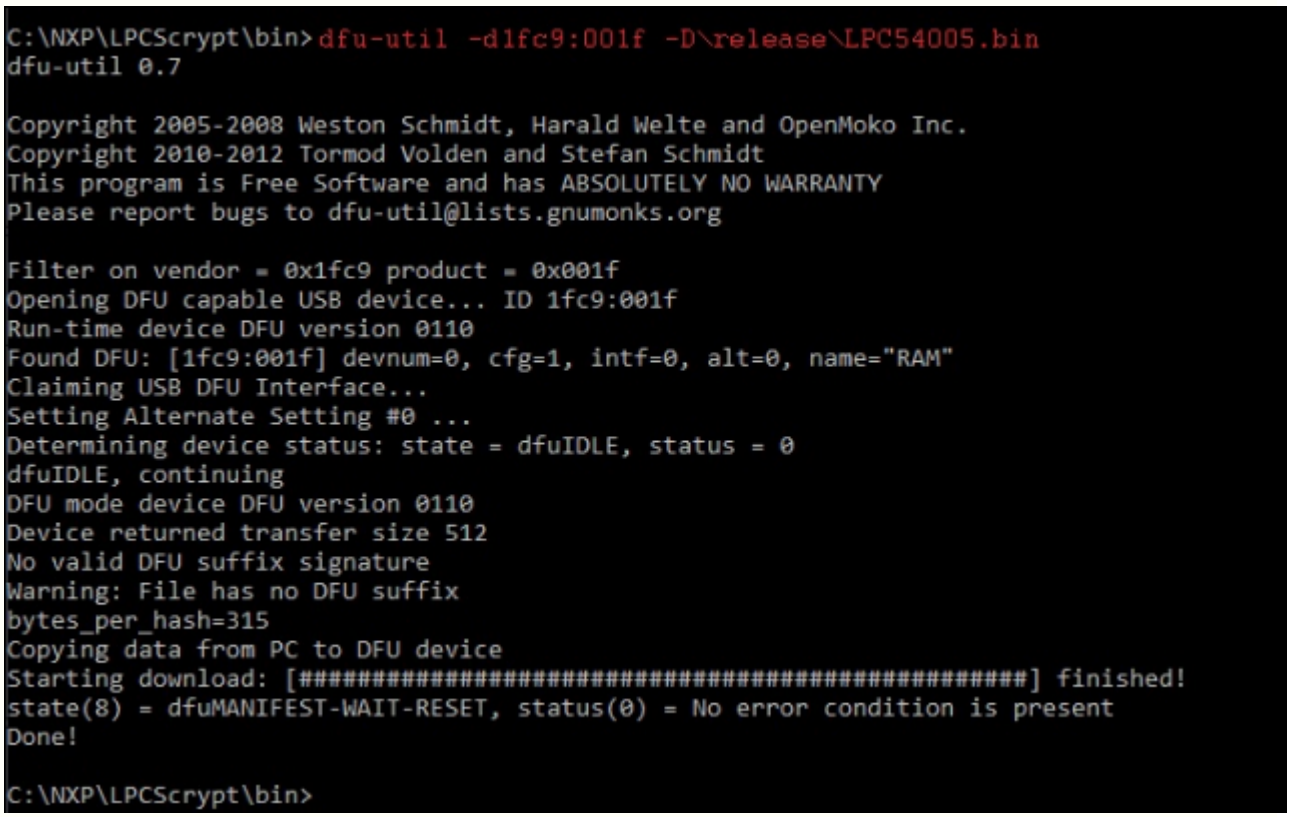
that we hold in place while connecting the USB0)

Then check that the LPC54005 is in DFU mode by typing dfu-util -l. If you don't see any devices repeat the powerup sequence with the jumper installed.

```
C:\NXP\LPCScrypt\bin>dfu-util -l
dfu-util 0.7

Copyright 2005-2008 Weston Schmidt, Harald Welte and OpenMoko Inc.
Copyright 2010-2012 Tormod Volden and Stefan Schmidt
This program is Free Software and has ABSOLUTELY NO WARRANTY
Please report bugs to dfu-util@lists.gnumonks.org

Found DFU: [1fc9:001f] devnum=0, cfg=1, intf=0, alt=0, name="RAM"

C:\NXP\LPCScrypt\bin>
```

In this case 1 DFU device was found [1fc9:001f] . You can remove jumper 0 at this point

Next load the firmware using that DFU number and firmware binary.



When the LED is flashing the firmware binary has been copied to QSPI (SPIFI) Flash.

Reboot (by disconnecting USB0 and reconnecting) and open BASICtools.

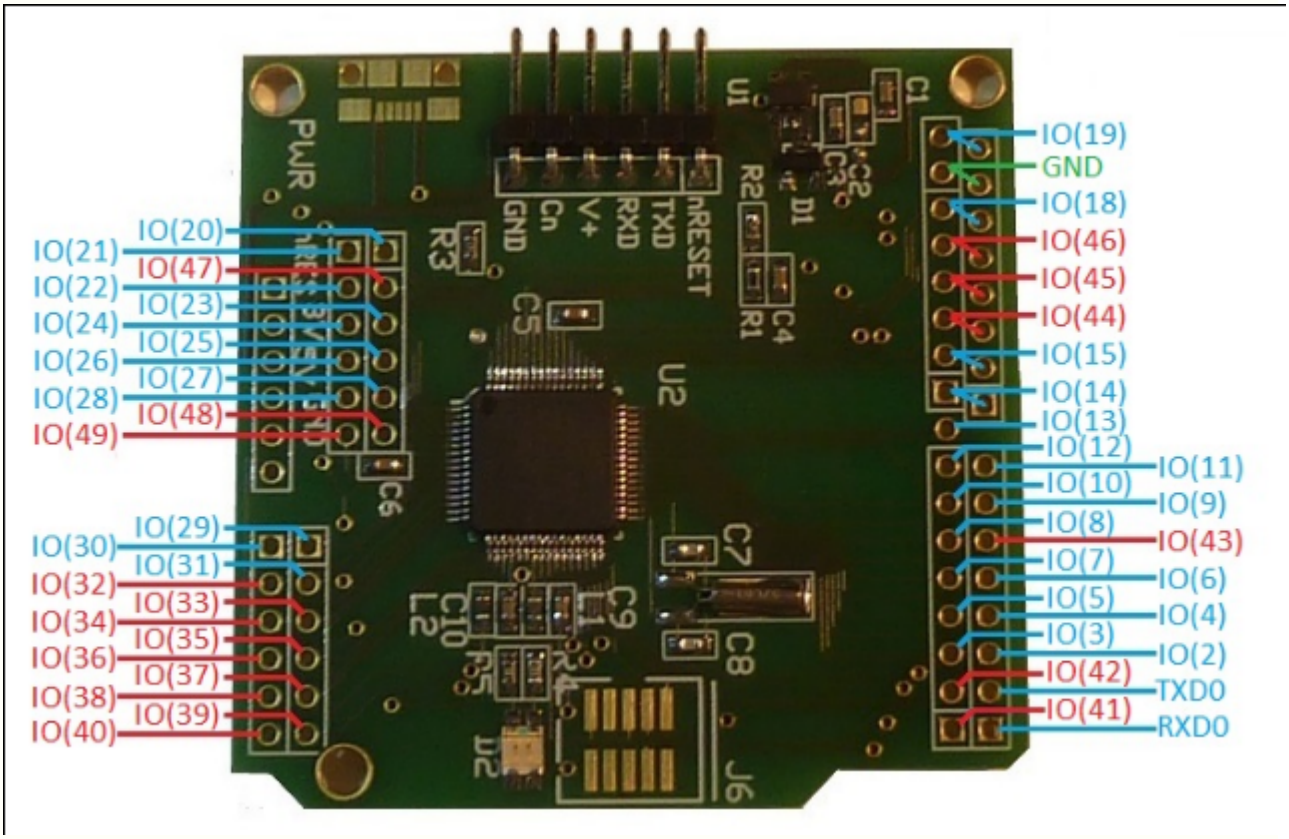
LPC54102 Details

The LPC54102 is footprint and pin compatible with the Arduino PRO. The board can be used with 5V TTL signals.

BASIC or C programs can be downloaded using the installed test connector using the USB dongle contained in Coridium's evaluation kit or using the **SparkFun USB Basic Breakout board** or FTDI cable from **Digikey**. More details on **these connections here**.

Digital IO connections

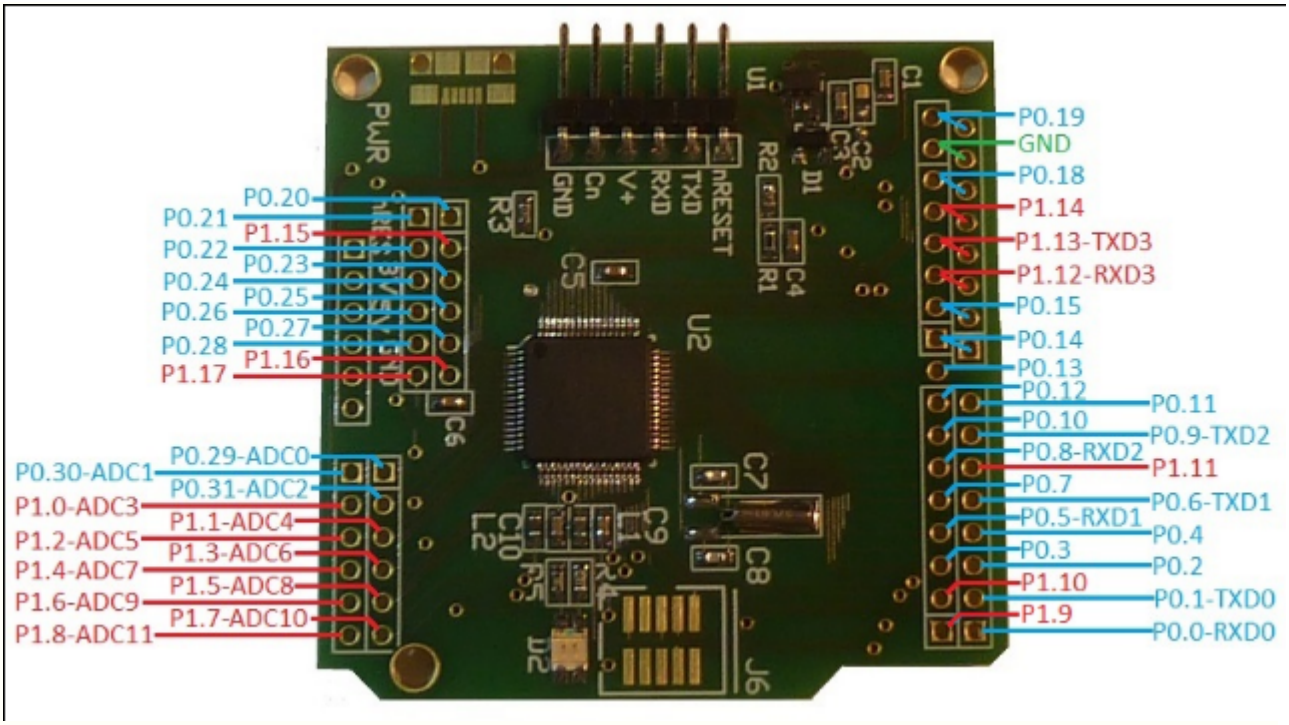
Below is a diagram of the pins, note it has been rotated 90 degrees to make it easier to read,



IO(32) is the equivalent of P1(0) and can be accessed either way.

Special purpose pins

RESET pin starts the ARM program if the BOOT(P0.31) pin is high. If you use P0.31 as an input you MUST make sure it is in the high state when RESET is asserted, otherwise your program will NOT start.

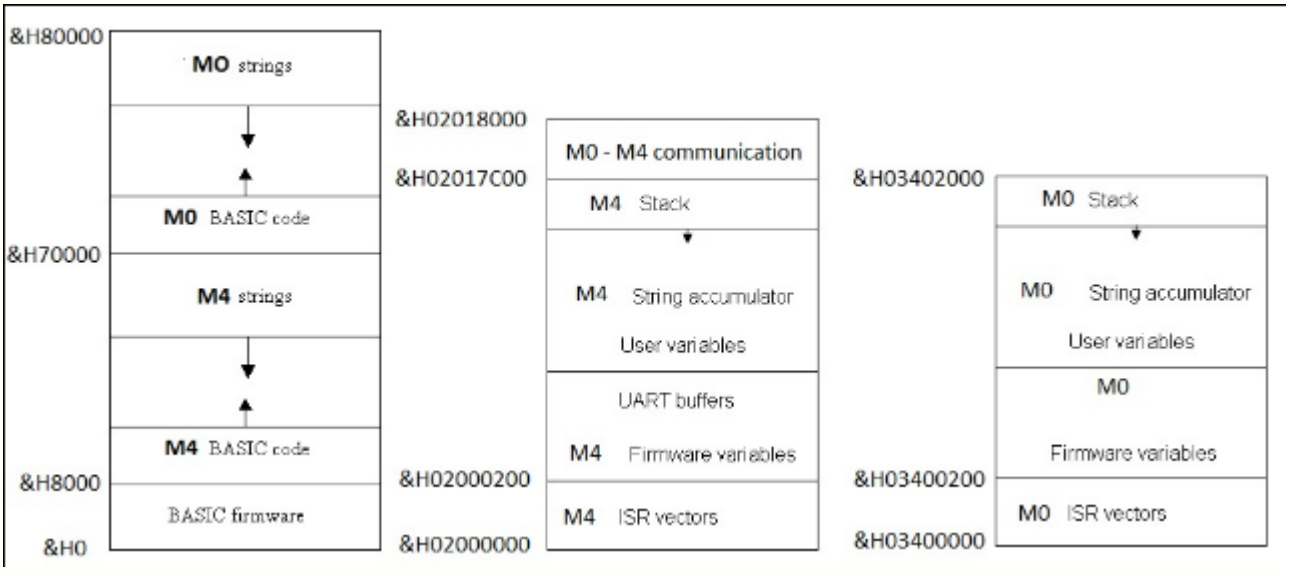


The LPC54102 supports a number of dedicated functions. Those include 4 UARTs, 2 SSPs, 1 SPI, 2 CAN, 2 I2C, I2S, 2 multi-channel PWMs, Quadrature Encoder, dedicated motor control PWM, interrupts, timer counter capture and match.

In addition most can be configured with pull-ups and default to pull-ups following reset.

Details can be found in NXP's User manual.

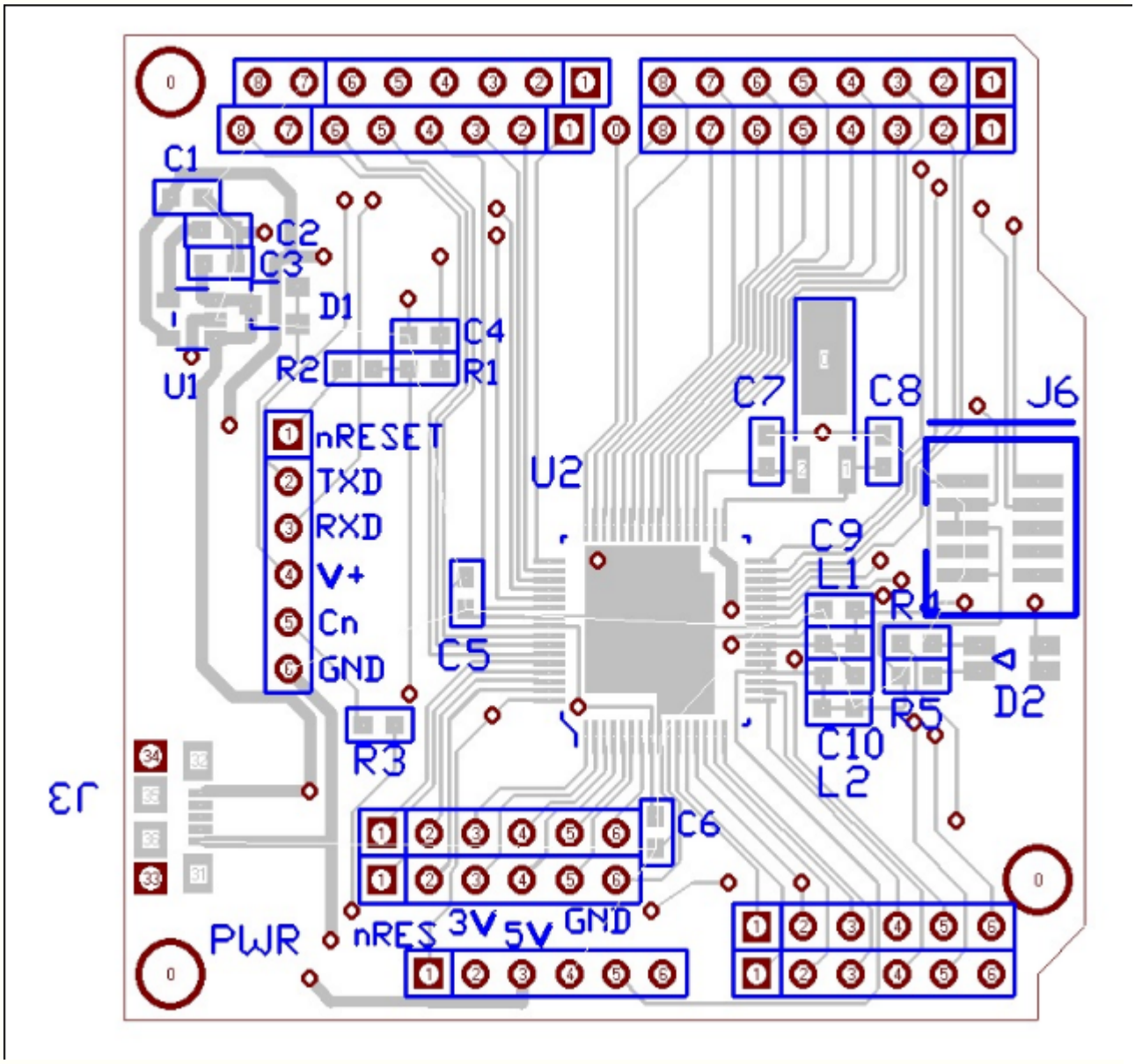
Memory Map



The **full schematic can be seen here**

And the **DXF** file

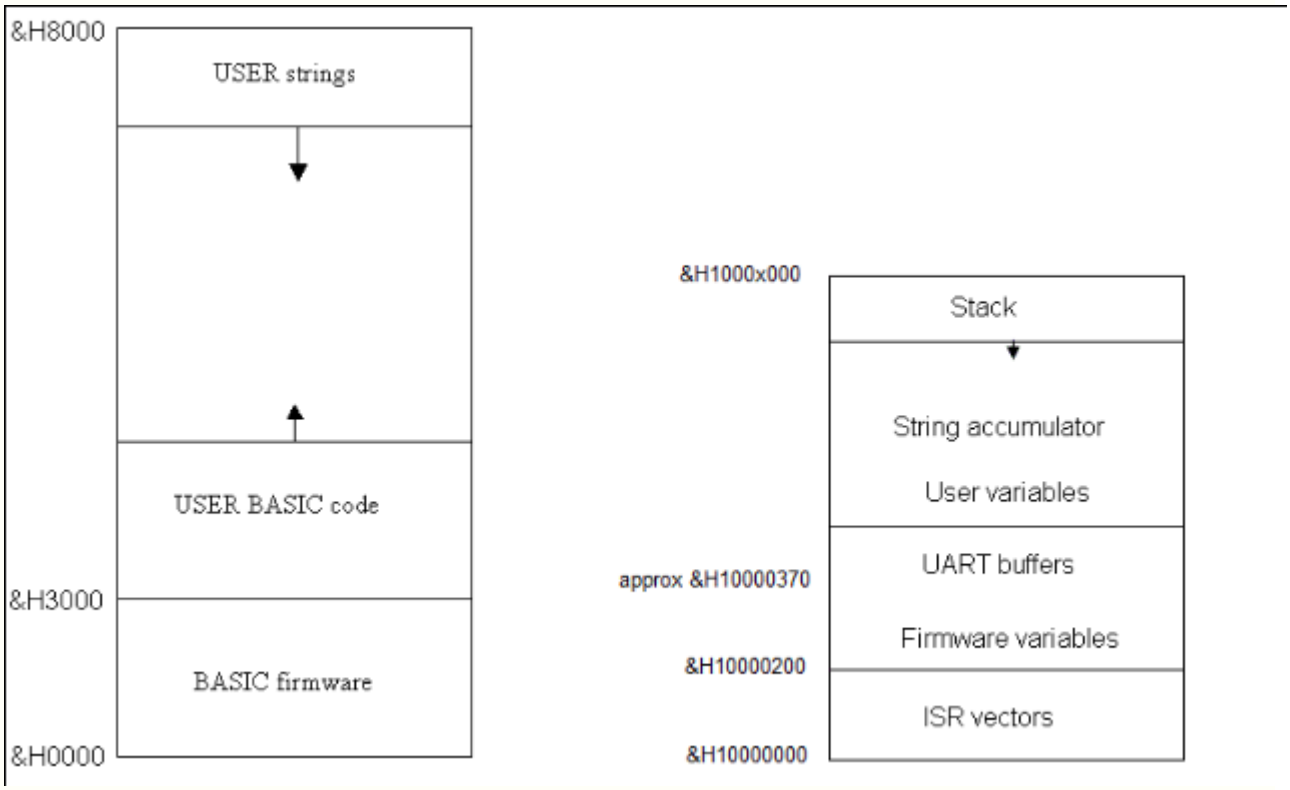
Part placement



Memory Maps

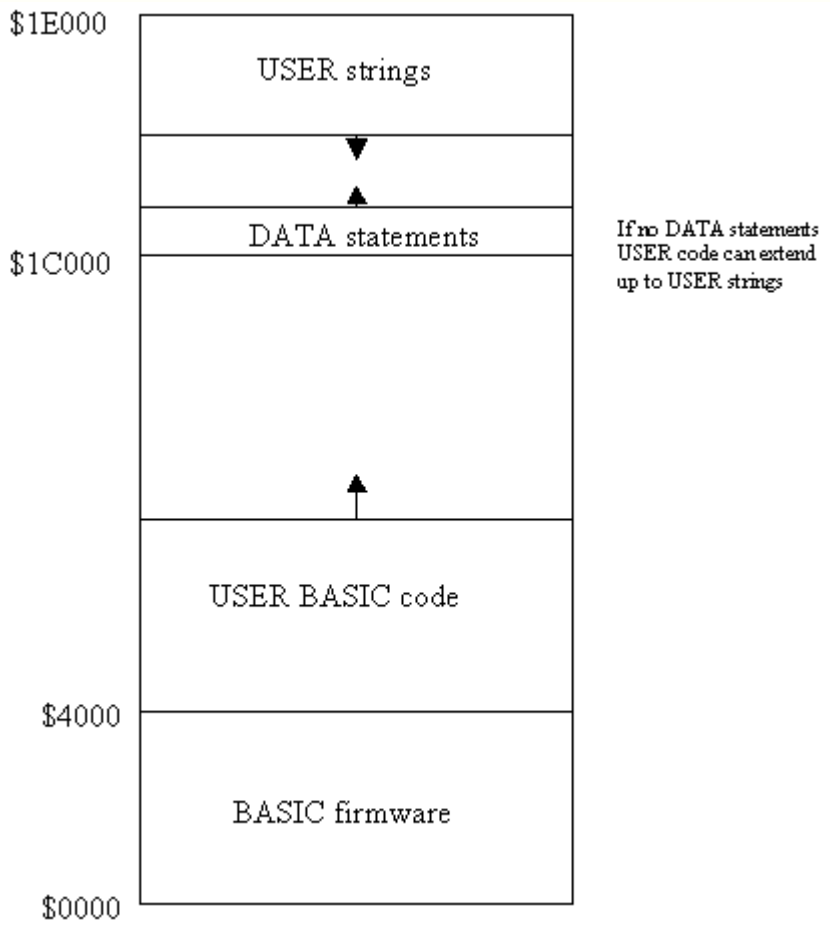
All addresses are hex values.

BASICchip, ARMmite ARMexpress LITE, ARMmite PRO, PROplus



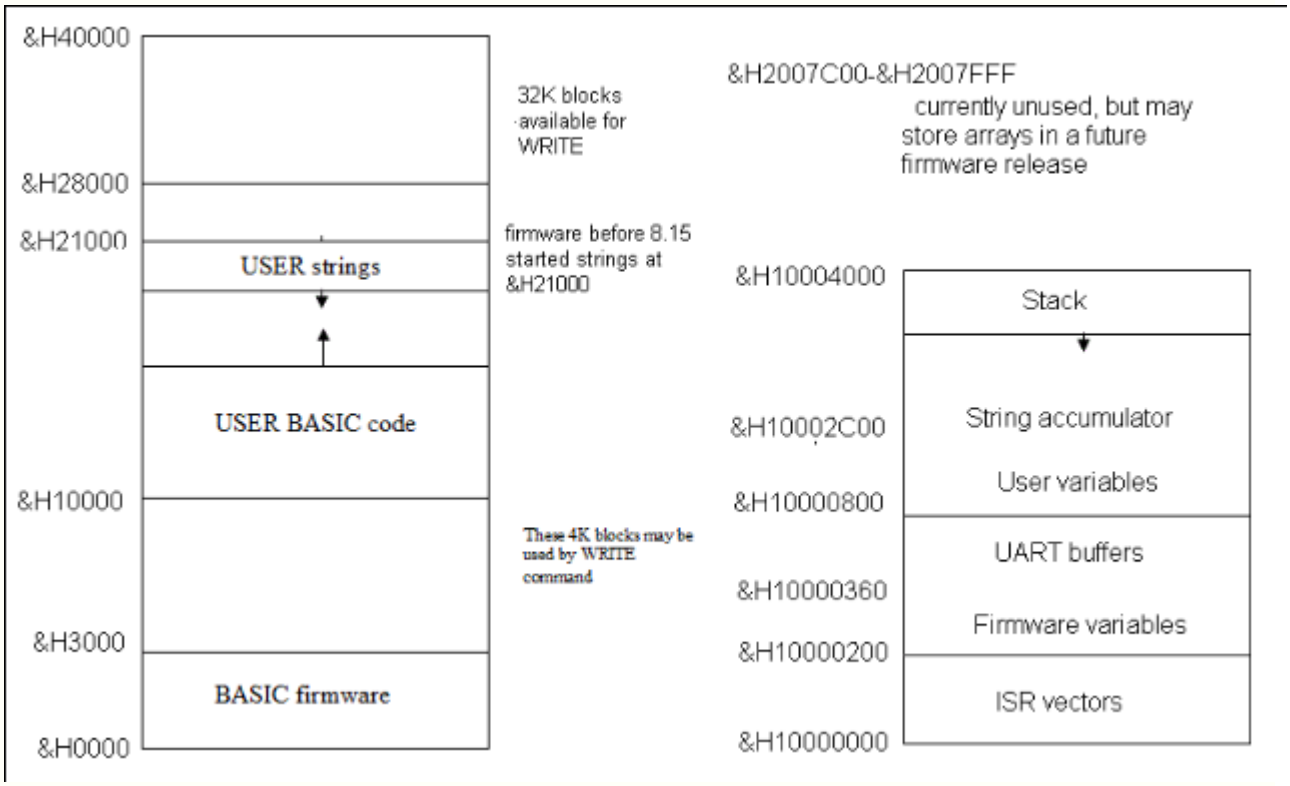
ARMmite ARMexpress LITE, ARMmite PRO, PROplus

ARMexpress



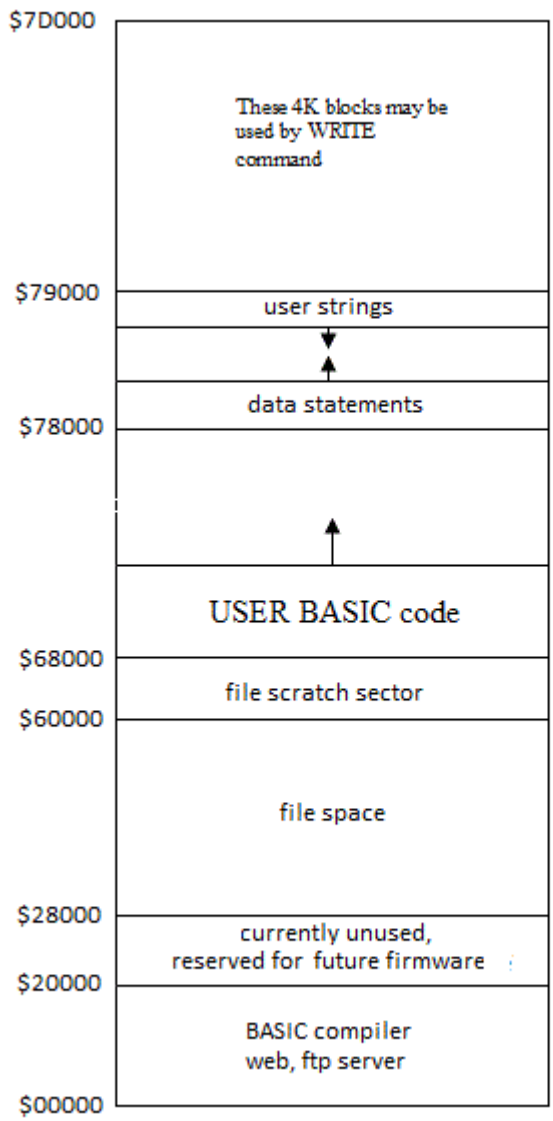
ARMexpress

SuperPRO



SuperPRO

ARMweb and DINkit/Ethernet



ARMweb and DINkit/Ethernet

DINkit (USB) and Stand-alone compiler

User code starts loading at &H3000.

Strings and DATA statements are stored in the last Flash Block, which depends on the Memory Map of the device (details in the NXP User Manuals). In the DINkit the last Flash block is from &H7C000 to &H7CFFF

LPC2103 products - ARMmite, ARMmite PRO and ARMexpress LITE

20.48K is available for code, DATA statements and string constants.

5.12K is available for data (1280 words)

LPC2106 ARMexpress

106.49K is available for code, DATA statements and string constants.

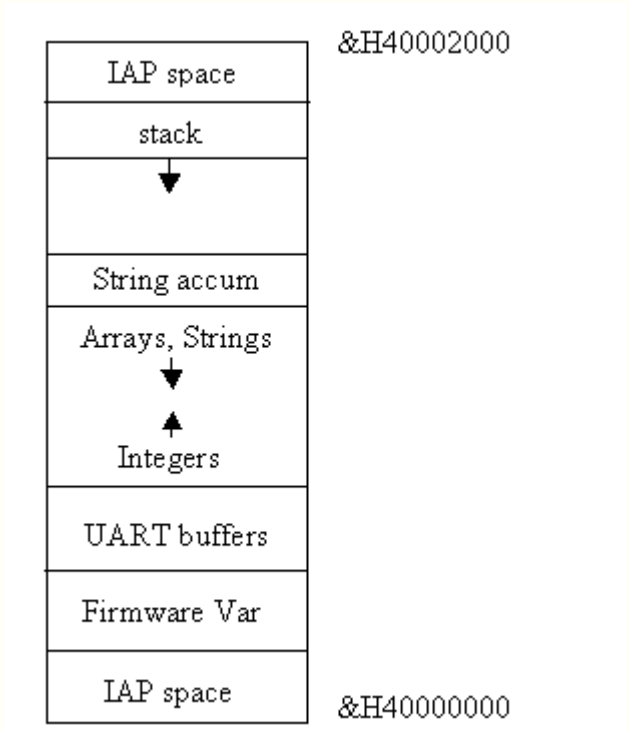
62.5K is available for data (15K words)

LPC2138 ARMweb, DINkit (Ethernet)

131K is available for code, DATA statements and string constants.

5.12K is available for data (1280 words)

DATA Memory Allocation



Local variables for FUNCTIONs and SUBs are allocated from global memory. This allows for a smaller stack size and faster calls to FUNCTIONs and SUBs. The ARMmite has only 8K total and has no stack overflow checking.

ARM Memory Peripheral Use

Memory Maps

Can be found here.

The ARM peripheral bus

Timer0 free running micro-second counter (TIMER command)
Timer1 used on ARMweb or with ON TIMER
Timer1 setup as 1msec timer, may be reprogrammed
Timer1 , Timer2 and Timer3 used for HWPWM on ARMmite or ARMexpress LITE
Uart0 UART for debug/download
Uart1-3 Not Used unless requested by user with BAUD(1) .. BAUD(3)
PWM used when HWPWM is engaged on PROplus, SuperPRO, used in the web versions of LPC1768
I2C Not Used
SPI reserved
RTC used for time-keeping

Interrupt use -- 21xx

```
FIQ             not used
ISR0            UART0
ISR2            PWM -- only used by ARMweb
ISR3            UART1 if RXD1, TXD1 used
ISR4            EINT0 if ON EINT0 used
ISR5            EINT1 if ON EINT1 used

ISR6            EINT2 if ON EINT2 used
ISR7            TIMER1 if ON TIMER used
ARMweb has  EINT0 connected to ENC28J60,
but it is not used and available to the user.

ARMweb firmware also uses EINT2 for remote debugging.
```

Interrupt use -- 175x, 1768

```
ISR21           UART0
ISR22           UART1

web
version           --
ISR25           PWM1
```

```
If a function is not included in the BASIC code the interrupt is available,
for instance RXD(1) uses the UART1 interrupt.
```

Background Tasks

Except for the ARMweb, the only background tasks are interrupt handlers for UART0 and UART1. UART1 is not active until the BAUD(1) is assigned.

The DATAlogger is the first of the multicore products. BASICtools allows you to compile programs for both CPUs.

The M4 CPU is the master CPU and is the first to start up. The M4 initializes the system, and checks for valid BASIC code, and if there is valid code it will launch both M4 and M0

Peripheral Sharing

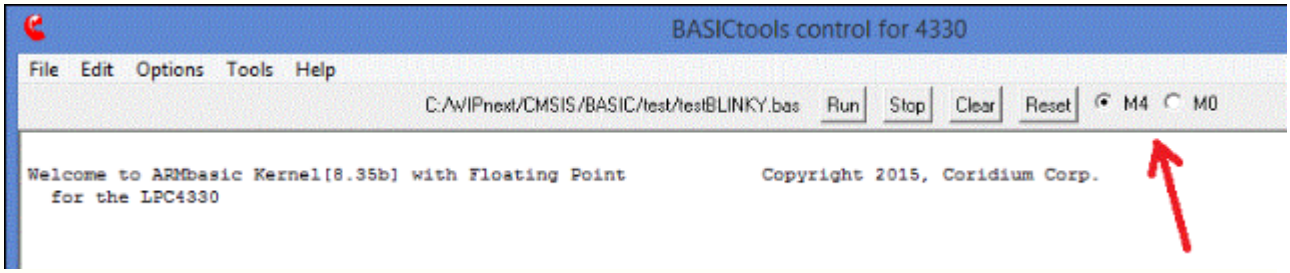
The M4 CPU controls writes to Flash, interrupts for UART0, and SDcard access. The M4 has a floating point unit so floating point math will be much faster on the M4 CPU.

The M0 CPU can send debug information to UART0, but when it does so it uses polling so it can potentially wait until UART0 is idle. The M0 can NOT receive characters on UART0, as the M4 interrupt preempts them.

Other peripherals can be used by either CPU, but is up to the user to decide which CPU has control. While in some cases you can share peripherals, it is better to dedicate them to one CPU or the other. This is especially true for interrupts, the M0 can setup interrupts, but if that interrupt is setup by the M4, results are not predictable.

BASICtools multicore support

When BASICtools begins, it does not check to see which type of board it is connected to. But when you load the first program it determines that this is a multicore chip and adds the ability to select the core in the upper right hand corner.



By default the M4 (master CPU) is the first to be loaded with a program. Note that any program loaded into the M0 will be executed in parallel with the M4 program.

When you switch to load programs on the M0, the M4 will be loaded with a short program that starts the M0 and then the M4 terminates.

Both CPUs can access RAM at &H2000C000, it can be used for mailbox, shared data or other communication between the CPUs. Pointers can be used access there. A future version of the compiler will allow sharing of some global variables between the 2 CPUs.

The intended method of debugging is to debug programs separately, then once the program is operating correctly load the M0 CPU program, then load the M4 program. After that when the program runs, both CPU programs will be executed.

When the M4 starts up it looks for a valid program in both M4 and M0 memory, and if found they will be started. The easiest way to "erase" programs in both memories is to select the **M0** CPU and type **PRINT** into the command then push the **RUN** button. That starts you off with a clean slate.

Power

Common to all boards

Initial Power on conditions
On power up all pins are tri-stated on the NXP LPC based boards Some have weak pullup or pulldown resistors enabled depending on model That information is containted in the pin description of the User Manual or Data Sheet.

Restarting the program
If the user has programmed the ARMexpress/ARMmite, that program will be started when the power is applied, or restarted when RESET is asserted either low on the open-collector pin 22, or positive true on the ATN pin.

If the user program ends by getting to the last statement of the program or executing an END instruction, the ARMexpress will power down and await either input on the debug serial port, or a RESET.

Break operation or STOP
If the user code is running, it can be stopped by a RESET condition. This will normally restart the user code, but there is a short window (500 msec) where the ARMexpress will wait to see if there is input on the serial debug port. If the character received on the serial port is ESCAPE (27) or CTL-C (3) then the user program is prevented from running and the ARMexpress is ready to be reprogrammed. Or the user can restart the program by typing RUN or using the RUN button in BASICtools.

USB Power
The USB specification allows for up to 500 mA at 5V to be supplied to external devices. In many cases this is limited to 100 mA by the manufacturer of the PC or hub.

ARMexpress and its eval PCB uses approximately 50 mA when running and 10 mA when idle. So it can be powered from the USB port for programming, without the need for the alternate power supply. The same is true for the ARMmite.

Once the programming is completed, the ARMexpress may be run without a connection to a PC. In this case an alternate power supply connection has been provided. This input goes to a regulator to supply 5V which is connected to pin 24 on the ARMexpress. Onboard the ARMexpress this will be regulated to 3.3V and 1.8V for use by the ARM CPU. The ARMmite takes this same unregulated input to generate either 5V or 3.3V on the rev2/rev3 versions respectively.

Smart Power

The USB evaluation board can be powered from either the USB, an external supply or BOTH. Power from the USB is controlled such that it is turned on by the USB controller. Power to the ARMexpress can also come from the external power supply and these are controlled to allow both USB and the power supply to be connected to the device at the same time.

The power connector is a 2.1mm, which is compatible with the Cui PP-002B part.

Battery backup

The ARMmite has a provision for adding a battery to keep its real time clock alive when power is removed. The circuit is designed to use a Panasonic ML2020 rechargeable Li battery.

Parallax STAMP compatibility

The Parallax STAMP products operate from a 5V supply. This can come from an unregulated input on pin 24, or from a regulated 5V supply on pin 21. The ARMexpress is backward compatible with both these connections, but for new designs it is recommended that power be supplied on pin 24. The voltage required is 4.5V or greater on pin 24, or 5V on pin 21. Also for C programming, pin 21 should not be connected to power. The maximum voltage that may be applied to either pin 24 is 16V, but this is not a recommended

continuous voltage, as it will cause extra heat to be generated by the ARMexpress onboard voltage regulators. For this reason the recommended maximum is 9V. When using an unregulated supply not supplied by Coridium, care should be exercised, as the current draw of the ARMexpress is low and the voltage will often be much higher than the rated voltage. The user should ensure that this voltage does not exceed the limit of 16V.

Power On Behavior

Initial Power on conditions

On power up all pins are tri-stated on the ARMexpress, ARMweb, PRO or ARMmite. On the SuperPRO and PROplus, pins are also tri-stated, but all have a weak pull-up resistor.

Following reset, the board waits at least 0.5 seconds for an ESC character (or upto 3 seconds for direct USB connected boards like ARMstamp or LPC54005), which if received stops the user program from running. If no ESC is received the user program starts.

Restarting the program

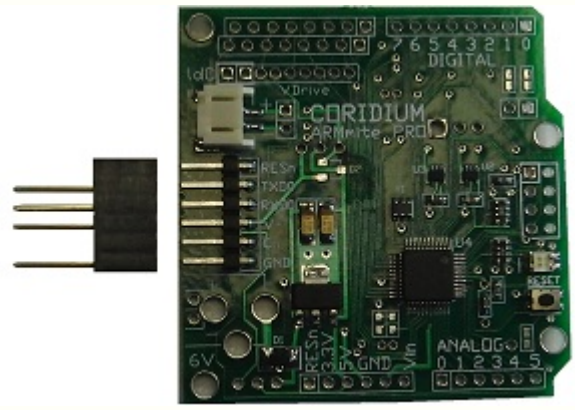
If the user has entered a BASIC program into the ARM, that program will be started when the power is applied, or restarted when RESET is asserted either with the pushbutton, or from the BASICtools program via asserting the DTR line (low on ARMmite, high on ARMexpress).

The BASIC firmware sends out a &H01 to indicate a user BASIC program has started. This tells BASICtools that the program is running (signified by turning the RUN button to red and showing STOP. If your program changes the baudrate and you are using BASICtools as a terminal window, you should also send out this character after you have switched baud rates, otherwise BASICtools will not think you program is running and it will try to compile any input from the enter line rather than passing it on to your program.

If the user program ends by getting to the last statement of the program or executing an END instruction, the ARM will await either input on the debug serial port, or a RESET. The BASIC firmware sends out a &H04 which indicates to BASICtools that the program has ended.

Reset and Boot for PRO boards

For the PRO, PROplus and SuperPRO boards when connecting a PC to a board that is running, the reset and boot control signals will be toggled by the PC. This is a function of Windows and the Drivers. This will reset the board or possibly put it into a load program state. To avoid this you can disconnect the Reset or Boot signals from the USB dongle, either by cutting pins or making an adapter using a 6 pin female header with long pins**(available from SparkFun**).



The above shows both RESET and BOOT signals disconnected.

Break operation or STOP

If the user code is running, it can be stopped by a RESET condition. This will normally restart the user code, but there is a short window (500 msec) where the ARM will wait to see if there is input on the serial debug port. If the character received on the serial port is ESCAPE (27) or CTL-C (3) then the user program is prevented from running and the is ready to be reprogrammed.

BASIC Boot Loader serial commands

When the user program is not running or not at a STOP, the BASIC bootloader is functioning.

There are 2 versions of this bootloader, the one on the ARMweb, and then all the others. The ARMweb has a full compiler ready to compile BASIC programs line by line. This can be used with the TclTerm terminal emulator or the web interface of the ARMweb. when running BASICtools programs are compiled on the PC and downloaded to the ARM. The ARMweb also supports the commands used by all the others, and these are used to load and control BASIC programs-

- **:**20.... Coridium hex format line, copy this data into the code buffer
- **:**00000001FF write the code buffer into the appropriate Flash space
- ARM responds by sending **+**
- **?** get vectors for ARMbasic compiler running on the PC
- **^** launch any user program contained in the Flash space
- **@**HHHH dump memory starting at HHHH which is a hex value without a preceding &H or $
- **"**message echo message back
- **!HHHH VVVV** write to hex address HHHH the hex value VVVV
- ctl-C or ESC on reset run the BASIC bootloader rather than the User program

At a STOP the ARM will respond to **^** run or **@** dump-memory commands which are used in the BASICtools variables page.

PROstart Pin Description

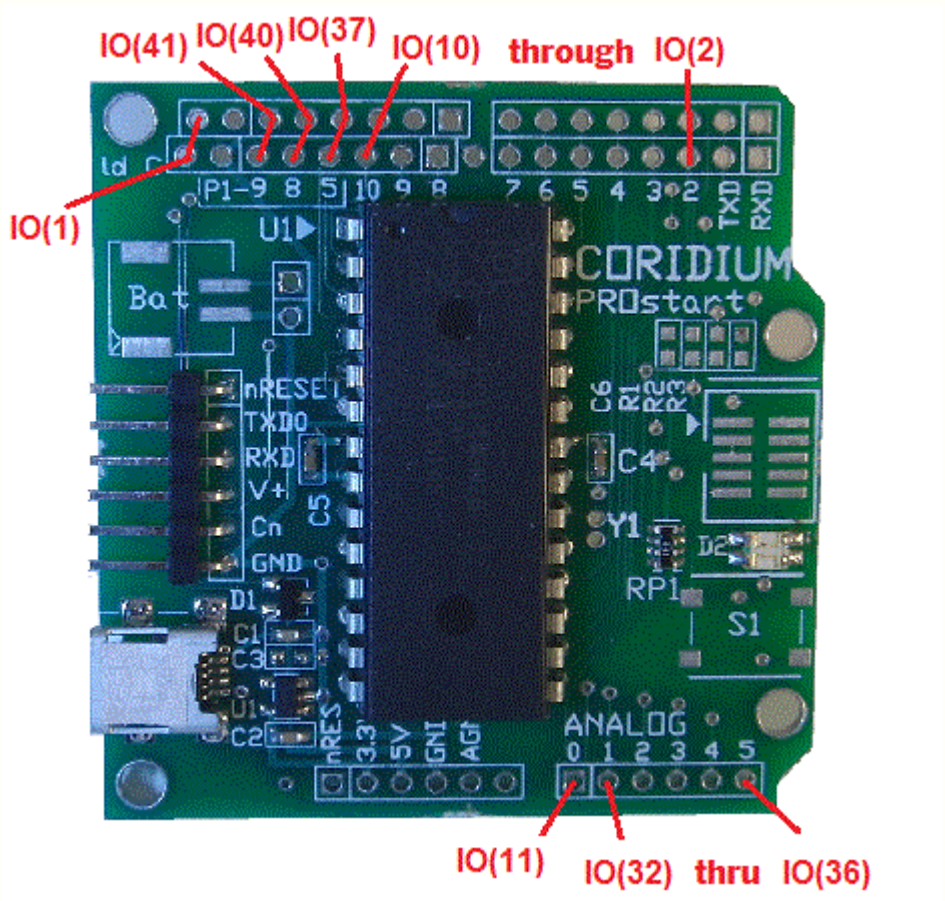
The PROstart is footprint and pin compatible with the Arduino PRO.

BASIC or C programs can be downloaded using the installed test connector using the USB dongle contained in Coridium's evaluation kit or using the **SparkFun USB Basic Breakout board** or FTDI cable from **Digikey**. More details on **these connections here** .

Digital IO connections

Port pins can be controlled with the**IN, OUT... keywords** . For port 1 pins are accessed with IO(32) through IO(41).

IO(4) and IO(5) are open-drain outputs, so for those to go high, you need to have a pull-up resistor (1K to 10K are good values, depending on the speed required).



Special purpose pins

The LPC1114 supports a number of dedicated functions. Those include 4 timers, 2 SPIs, and I2C.

In addition most can be configured with pull-ups or pull-downs and default to pull-ups following reset.

Details can be found in NXP's User manual.

Analog connections

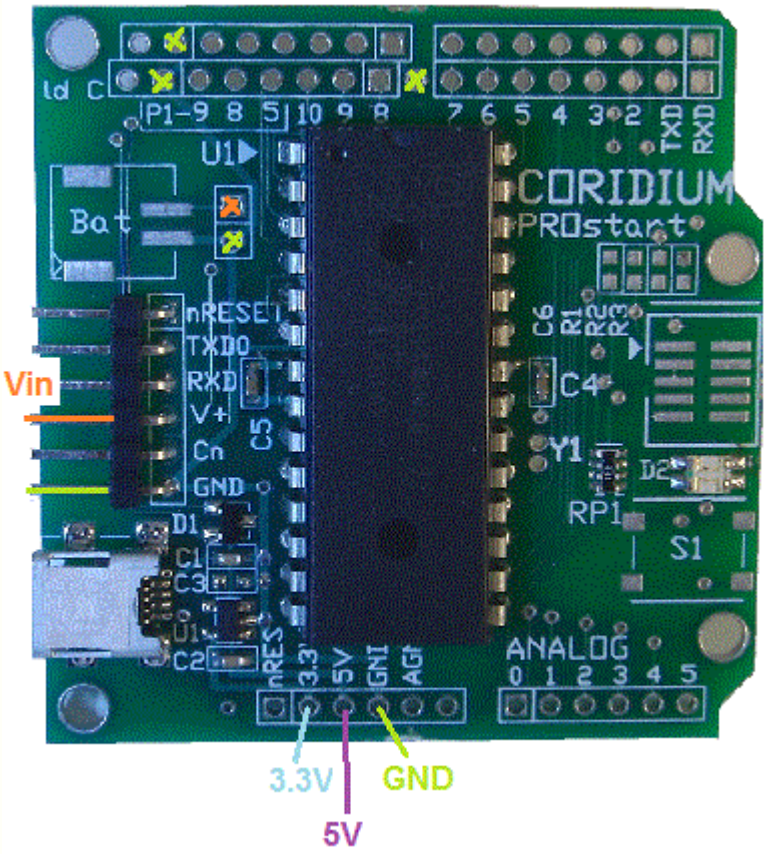
6 A/D converters are readily available, Analog 0-5.

On reset or power up the AD pins are configured by software as AD inputs. To change those to digital IOs, use IO(11), IO(32) through IO(36). Those pins will remain digital IOs until the next reset or can be changed back to A/D by writing to the PINSEL register.

The LPC1114 does not support an external reference for the A/D converters.

The A/D input requires a drive impedance of 7.5K or less (see NXP LPC175x spec sheet).

Power connections -- SuperPRO

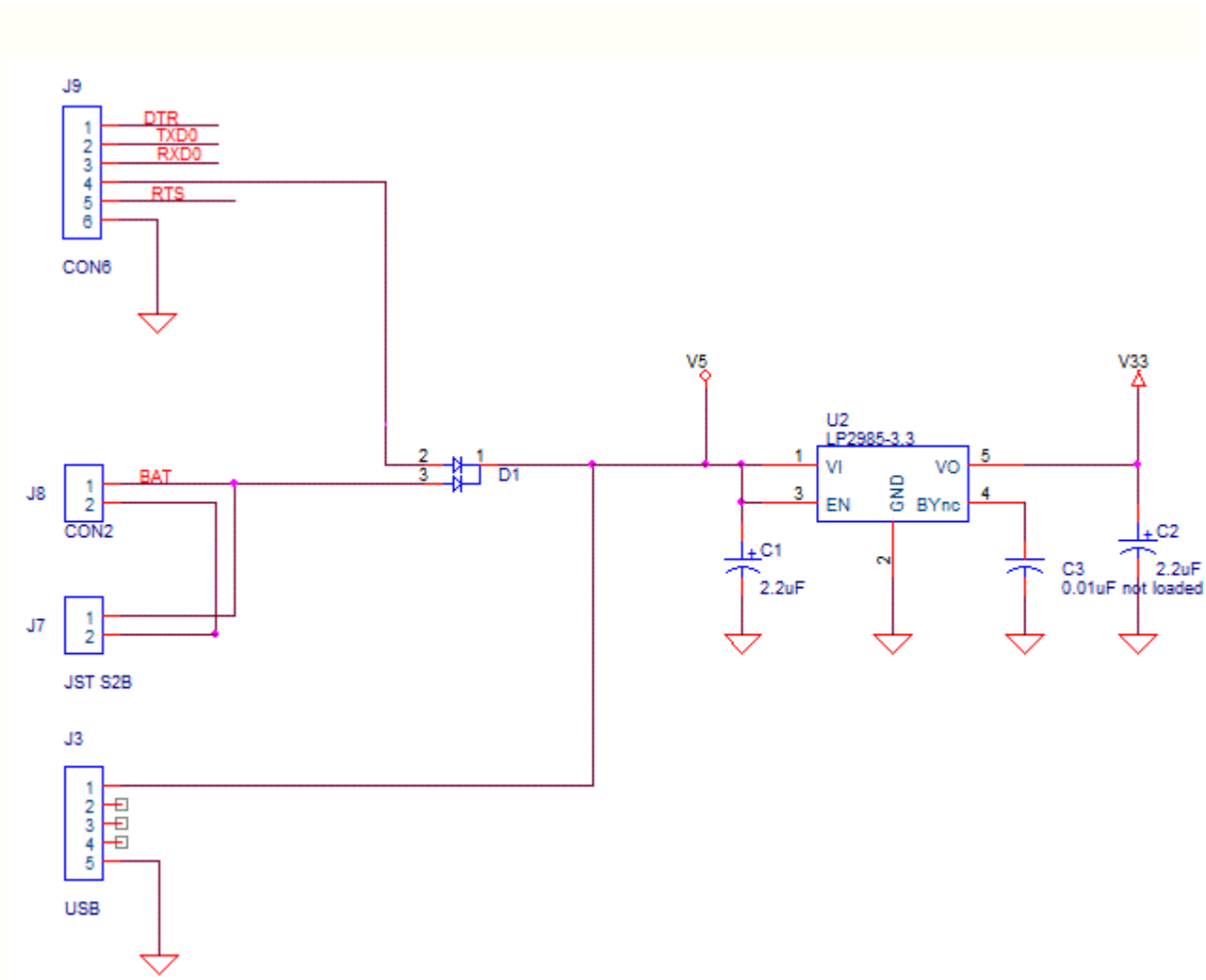


The board is shipped with a USB mini B type female connector for power, it is not a USB port. With the wide availability of USB chargers, this is becoming the most cost effective method of powering small electronics.

Pads for a a **2mm power jack compatible** with a JST PHR/S2B or **SparkFun PRT8671** for various battery packs from SparkFun are available, but not loaded in the basic configuration.

Diode steering allows power to be supplied from 5V USB, the USB test connector, or the battery connector. Because of the Schottky diodes, all 3 power sources can be connected simultaneously.

The schematic below describes this circuit on the PROstart



The **full schematic can be seen here**

Power connections details

The 3.3V regulator can supply 150 mA, with 10 mA being used by the LPC1114.

The analog GND should be used to connect to the GND of analog inputs. Digital and Analog GNDs are connected together with a small trace, but to minimize noise you should use the analog GND only for analog signals.

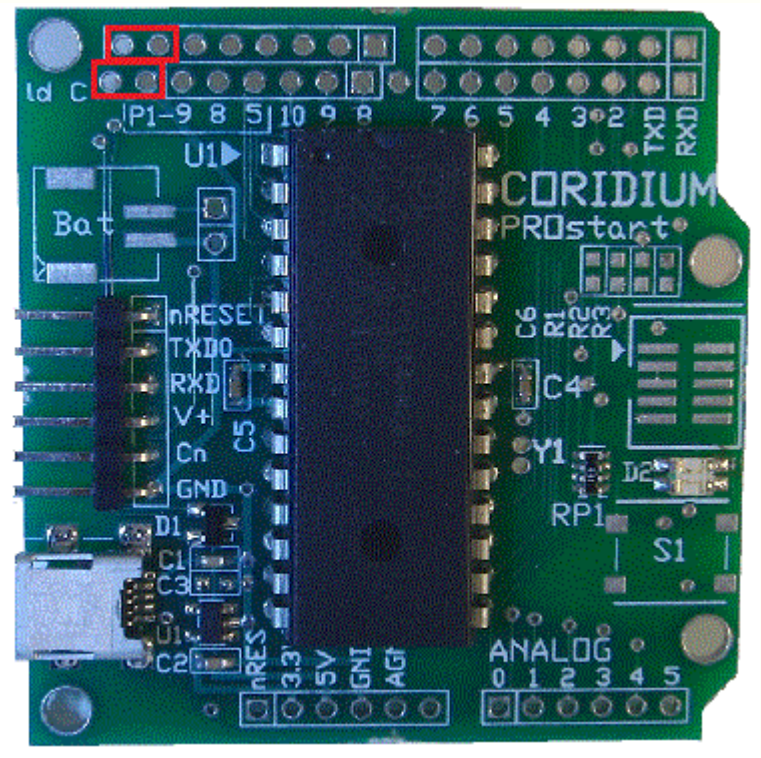
Jumpers and test connector for Program Download

The USB Dongle from Coridium or FTDI cable will supply 5V from the USB to power the PROstart. The Coridium USB dongle also controls the RESET and BOOT signals to automatically load C or BASIC programs using MakeItC or BASICtools. Remember, if you load a C program, it will erase the BASIC firmware and you will not be able to load BASIC programs after that.

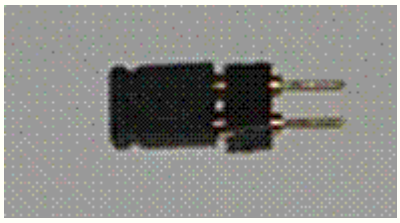
When using the SparkFun FTDI Basic Breakout Board, a limited amount of power can be supplied from the BBB, but this is limited to 50 mA and after diode drops, its about 2.8V to the LPC1114. In practice this will run, but it is outside the part specifications, so it should be limited in use.

Also with the SparkFun FTDI Basic Breakout Board or FTDI cable to load a C program, the LOAD C jumper needs to be installed, then removed to run the program. The Coridium USB dongle controls the BOOT signal so C programs can be loaded without the jumper.

BASIC programs can be loaded and controlled using the Cordium USB dongle, FTDI cable, or SparkFun board, with no additional steps/jumpers.

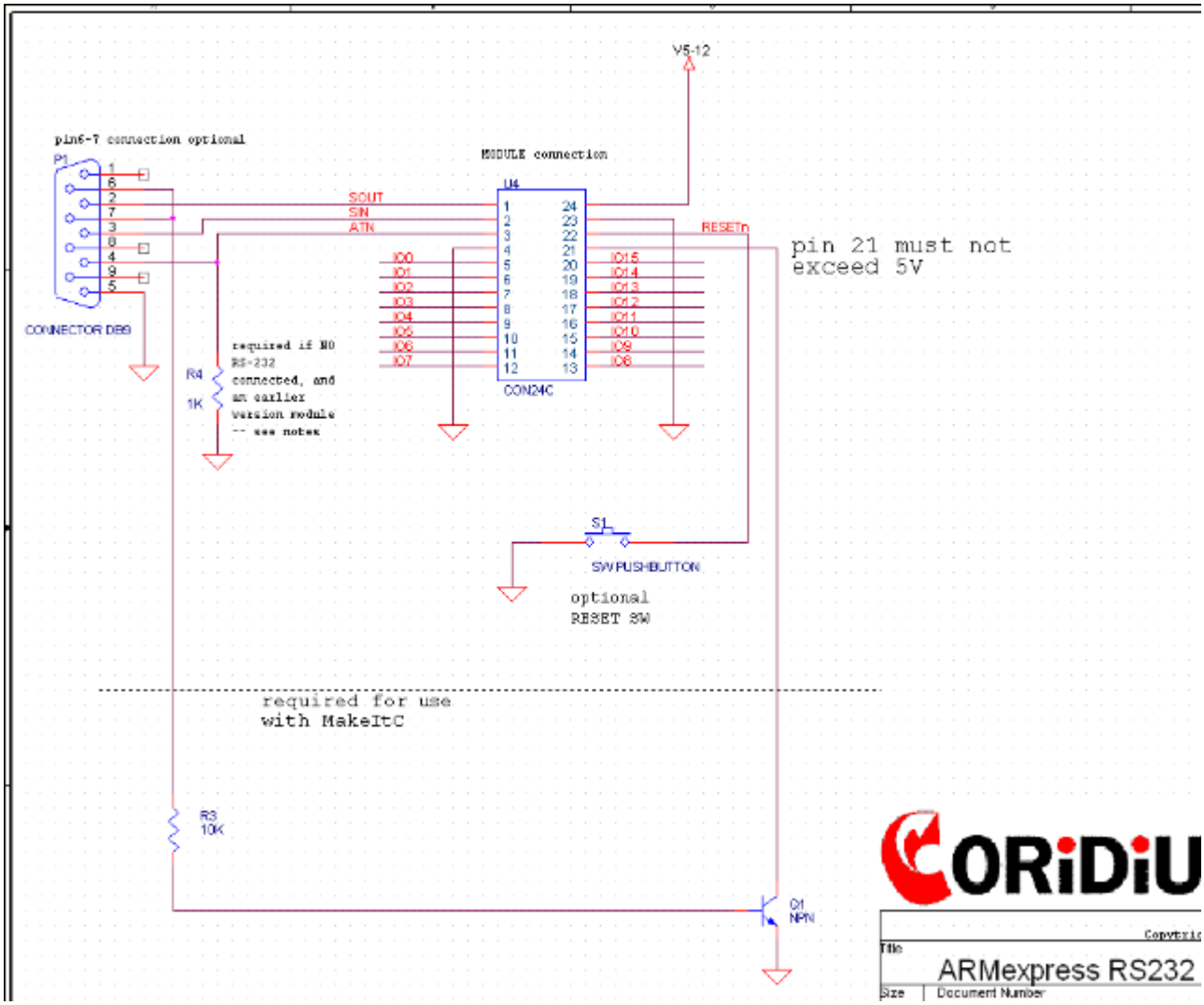


An alternative is to use a 2 pin header with a shorting block (pictured below)



ARMexpress Suggested RS232 / USB connection

For a finer image see ARMexpRS.pdf in your install directory (C:\Program Files\Coridium\Schematics).



Pin 21

On most Parallax boards this line is connected to a regulated 5V supply.

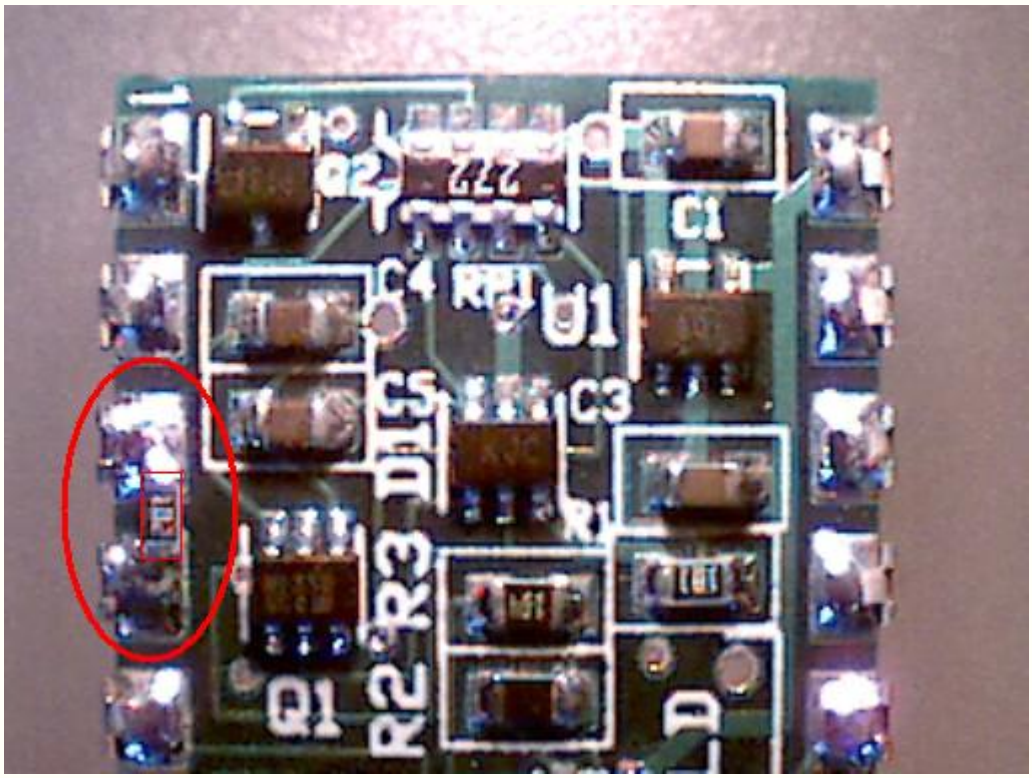
Do not connect a power source greater than 5V directly to pin 21.

When not connected this pin is pulled up to 3.3V by RP1 on the module.

When using MakeItC, this line is pulled low to download a C program, which can be done automatically by connecting to an NPN transistor with the RTS line on the serial port.

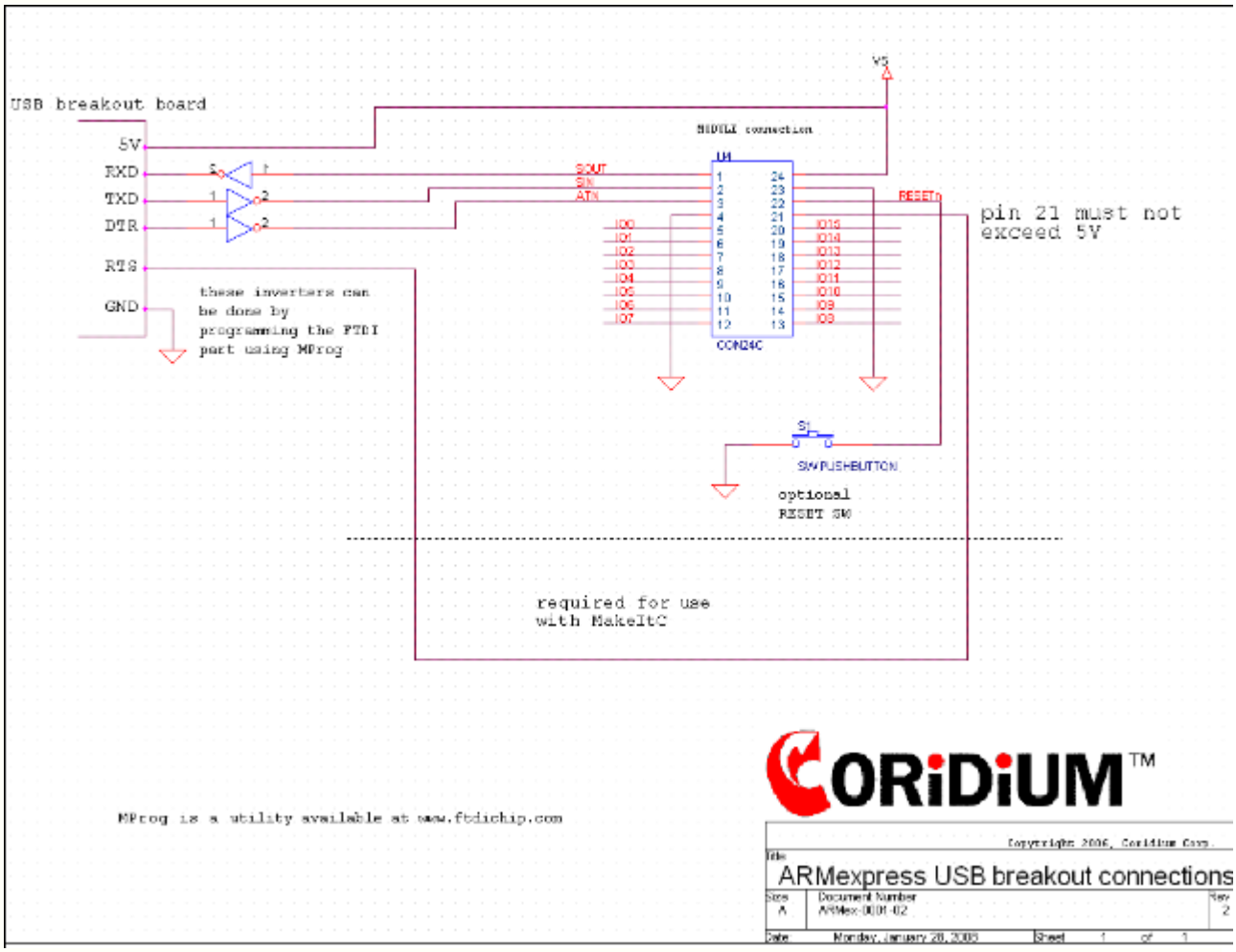
Pin 3

On later revision ARMexpress and ARMexpress LITE a 1K pull-down has been added on the module between pins 3 and 4 (as pictured below. If your unit does not have this, then a 1K pull-down resistor is required, when there is no signal on pin 3.



USB connection

A serial connection can be made with a USB breakout board. The suggested wiring should be done as follows. The inversion of RXD, TXD and DTR can be done by the FTDI chip using their **MProg utility**. Mprog programs ALL FTDI parts connected to the PC, so make sure only the one you want to change is connected. Also changes do not occur until the FTDI chip is powered up (so you must disconnect it and reconnect it).



Hints for debugging

Make sure you have both Power and GND connected.

When running BASICtools, the idle condition is

PIN 1 low

PIN 2 low

PIN 3 low

PIN 22 high

PIN 21 high

When RESET, either by pulling 3 high or 22 low, there will be some activity on pin 1 as the ARMexpress sends the Welcome message.

Reset and Boot for PRO boards

For the PRO, PROplus and SuperPRO boards when connecting a PC to a board that is running, the reset and boot control signals will be toggled by the PC. This is a function of Windows and the Drivers. This will reset the board or possibly put it into a load program state. To avoid this you can disconnect the Reset and Boot signals from the USB dongle, either by cutting pins or making an adapter using a 6 pin female header with long pins(**available from SparkFun**).


7 6 5 4 3 2 1 0
DIGITAL
VDrive
CORIDIUM
ARMmite PRO
RESET
ANALOG
0 1 2 3 4 5
6V
3.3V
5V
GND

Schematics

PDF copies of the schematics are copied into the Program Files/Coridium/Schematics directory when you install either the BASIC or C tools.

Or you can follow these links to PDF schematics on the Coridium website.

- BASICboard schematic
- ARMmite schematic
  - ARMmite rev 2 schematic
- PROstart schematic
- PRO schematic
- PROplus schematic
- Super PRO schematic
  - PROplus / SuperPRO rev 4/5 schematic
  - USB dongle schematic
- ARMweb schematic
  - ARMweb rev 3 schematic
- DATA Logger schematic
- DINkit schematic
  - DINkit USB board
  - DINkit Ethernet board
- XB Sensor
- Wireless ARMmite

DXF files are mechanical drawings of the boards, they are also available from these links or in the Schematics directory.

- ARMmite mechanical
- DATA Logger mechanical
- PROstart mechanical
- PRO mechanical
- ARMweb mechanical
- SuperPRO PROplus mechanicals
- XB sensor mechanicals

For a DXF viewer, try **http://www.abviewer.com/viewer.html,** not free but reasonable and has a 30 day trial.

Serial Configuration

Though we recommend using BASICtools or TclTerm to talk to Coridium ARM products, here are settings for other terminal programs.

Any program on the PC that can communicate with a serial port can send or receive data to the ARM. This would include MSCOMM and Visual BASIC. Also various C's including GCC. Other options include Perl or Tcl scripts.

However these programs must be able to control the DTR and RTS lines under user control. If they cannot refer to the next **section**. Programs that cannot include Matlab, Hyperterm and Teraterm.

is the source for a Tcl program that operates as a terminal emulator for Coridium products. You can use it if you have access to any of the GPL Tcl interpreters, or a compiled version is available on the Coridium Support page. The sources are also installed with BASICtools.

Baudrate

19.2 Kbaud, 8 bit, No Parity, 1 stop bit. These settings are controlled by BASICtools , any settings in the Device Manager are IGNORRED.

End of Line

expects a LF (line feed),

CR is currently ignored.

Voltage Levels -- ARMexpress ONLY

/SOUT, /SIN and ATN (pins 1,2,3) will accept either TTL or RS-232 levels. ATN when high resets the ARM, and ATN should not be allowed to float. It should either be connected directly to DTR, or some TTL signal that is LOW or Ground. The /SOUT driver relies on either /SIN or ATN being low to generate the low going voltage. This allows for full-duplex serial operation.

When BASICtools appears to be deaf

There are cases where the USB driver and BASICtools get out of sync. This includes when the board is disconnected from the USB port, and sometimes when the serial configuration is changed. In these cases it may be necessary to exit BASICtools and then restart it.

TclTerm configuration settings

The configuration of BASICtools is saved in a file BASICtools.ini. It is written when either it does not exist (when first installed) or when the configuration is changed by the user. This file is a Tcl source which may be edited by the user. If it becomes corrupt, delete the file and the default configuration will be restored.

The file can be found in %appdata%\coridium directory

SPI,Microwire

The **Serial Peripheral Interface Bus** or **SPI** bus is a very loose standard for controlling almost any digital electronics that accepts a clocked serial stream of bits. A nearly identical standard called "**Microwire**" is a restricted subset of SPI.

SPI is cheap, in that it does not take up much space on an **integrated circuit**, and effectively multiplies the pins, the expensive part of the IC. It can also be implemented in software with a few standard IO pins of a microcontroller.

Many real digital systems have peripherals that need to exist, but need not be fast. The advantage of a **serial bus** is that it minimizes the number of conductors, pins, and the size of the package of an integrated circuit. This reduces the cost of making, assembling and testing the electronics.

A serial peripheral bus is the most flexible choice when many different types of serial peripherals must be present, and there is a single controller. It operates in full duplex (sending and receiving at the same time), making it an excellent choice for some data transmission systems.

In operation, there is a **clock**, a "data in", a "data out", and a "chip select" for each integrated circuit that is to be controlled. Almost any serial digital device can be controlled with this combination of signals.

SPI signals are named as follows:

- SCLK - serial clock
- MISO - master input, slave output
- MOSI - master output, slave input
- CS - chip select (optional, usually inverted polarity)

Most often, data goes into an SPI peripheral when the clock goes low, and comes out when the clock goes high. Usually, a peripheral is selected when chip select is low. Most devices have outputs that become high **impedance** (switched-off) when the device is not selected. This arrangement permits several devices to talk to a single input. Clock speeds range from several thousand clocks per second (usually for software-based implementations), to several million per second.

Most SPI implementations clock data out of the device as data is clocked in. Some devices use that trait to implement an efficient, high-speed full-duplex data stream for applications such as digital audio, digital signal processing, or full-duplex telecommunications channels.

On many devices, the "clocked-out" data is the data last used to program the device. Read-back is a helpful built-in-self-test, often used for high-reliability systems such as avionics or medical systems.

In practice, many devices have exceptions. Some read data as the clock goes up (**leading edge**), others read as it goes down (**falling edge**). Writing is almost always on clock movement that goes the opposite direction of reading. Some devices have two clocks, one to "capture" or "display" data, and another to clock it into the device. In practice, many of these "capture clocks" can be run from the chip select. Chip selects can be either selected high, or selected low. Many devices are designed to be daisy-chained into long chains of identical devices.

SPI looks at first like a non-standard. However, many programmers that develop **embedded systems** have a software module somewhere in their past that drives such a bus from a few general-purpose I/O pins, often with the ability to run different clock polarities, select polarities and clock edges for different devices.

The interface is also easy to implement for bench test equipment. For example, the classic way to implement an SPI interface from a personal computer to custom electronics is via a custom cable to the PC's parallel printer port. The parallel port generates and reads standard **TTL** logic voltages; +5V is high, ground is low. A number of helpful people have developed drivers to give access to this port in the most restrictive operating systems, such as Windows NT (see below), from the least likely environments, such as Visual Basic.

SuperPRO Pin Description

PROplus Pin Description

The SuperPRO is footprint and pin compatible with the Arduino PRO. In addition it has an onboard 5V regulator so it is compatible with 5V shield boards. The SuperPRO also has an RTC crystal and provision for a battery.

The PROplus has only a 3.3V power supply and does not include the RTC crystal. Both boards can be used with 5V TTL signals.

BASIC or C programs can be downloaded using the installed test connector using the USB dongle contained in Coridium's evaluation kit or using the **SparkFun USB Basic Breakout board** or FTDI cable from **Digikey**. More details on **these connections here**.

Digital IO connections -- rev 5 or later

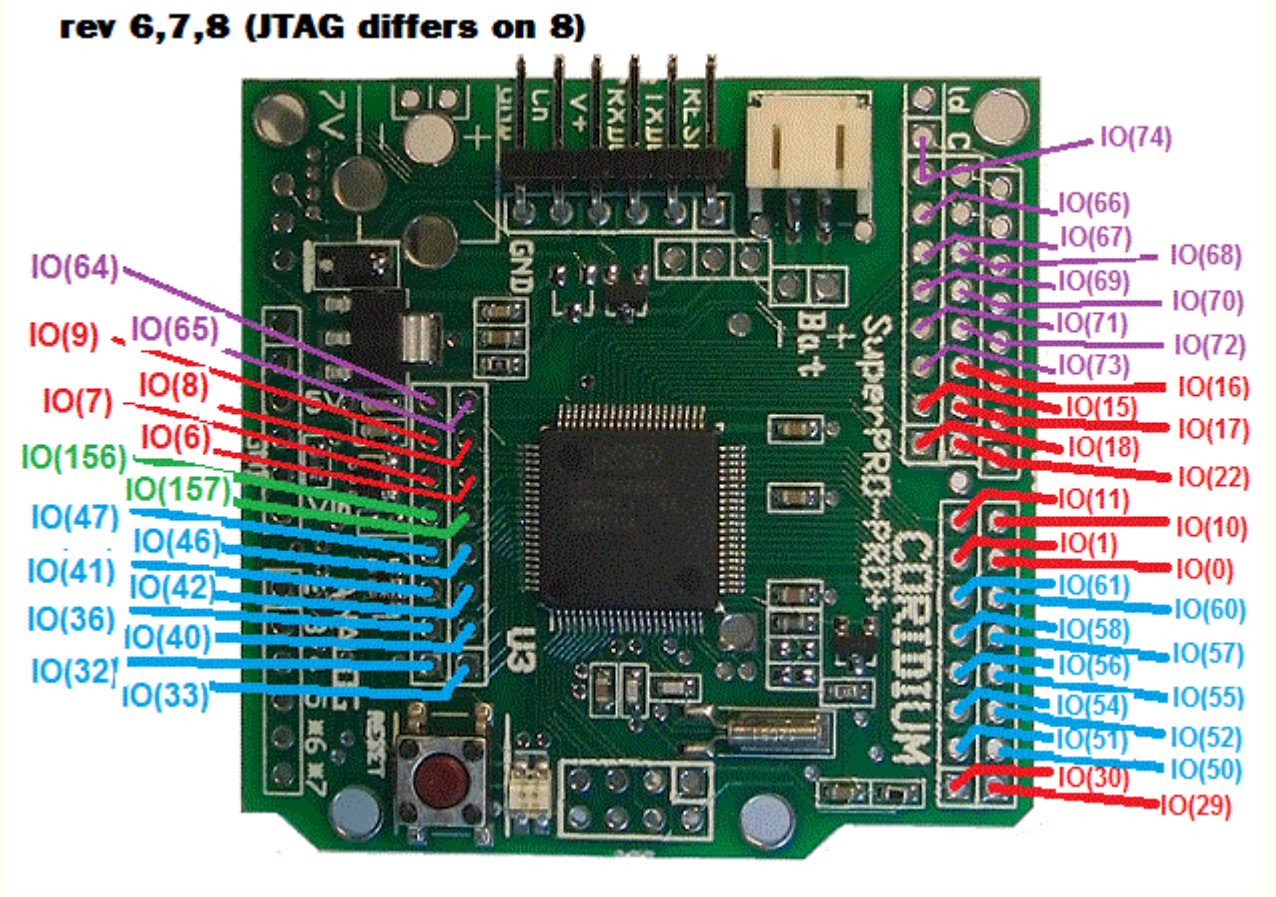
The rev 5 adds a parallel connection for pins that are on 0.1" centers. This artwork is also shared with the PROplus version of the board.

The SuperPRO uses an LPC1756 and has 5V and 3.3V supplies.

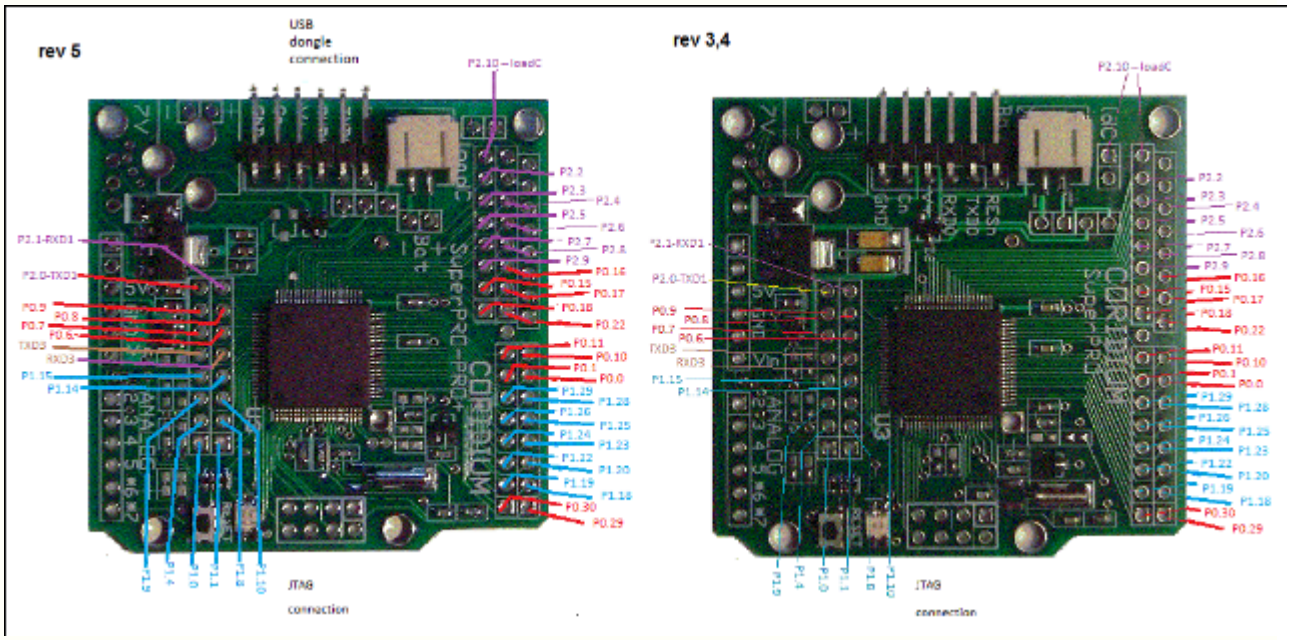
The simpler PROplus uses an LPC1751 and has only the 3.3V supply.

Port pins can be controlled with the **P0..P4 keywords**. and with firmware 8.12 or later the original **IO, IN, OUT.. keywords**. More details on the GPIOs can be found in the NXP User Manual.

Below is a diagram of the pins, note it has been rotated 90 degrees to make it easier to read,



All boards rev 5 and later share the SAME pinouts, except for the location of the load-C jumper. To use IO(x) with x greater than 32, firmware 8.11 or later has to be used. IO(32) is the equivalent of P1(0) and can be accessed either way in the latest firmware, in earlier versions P1(0) was the way to access that pin.



Special purpose pins

RESET pin starts the ARM program if the BOOT(P2.10) pin is high. If you use P2.10 as an input you MUST make sure it is in the high state when RESET is asserted, otherwise your program will NOT start.

The LPC1756 supports a number of dedicated functions. Those include 4 UARTs, USB, 2 SSPs, 1 SPI, 2 CAN, 2 I2C, I2S, 2 multi-channel PWMs, Quadrature Encoder, dedicated motor control PWM, interrupts, timer counter capture and match.

In addition most can be configured with pull-ups and default to pull-ups following reset.

Details can be found in NXP's User manual.

UARTs are enabled by calling BAUD(x) for x=0 to 3. UART0 is enabled by default as the programming debug connection. The pin assignment BASIC uses is in the following table (you can change the settings by changing the PINSEL registers, details in the NXP User Manual)

UART	BASIC	NXP	UART
RXD(0)	IO(3)	P0(3) / AD(6)	UART0
TXD(0)	IO(2)	P0(2) / AD(7)	
RXD(1)	IO(65)	P2(1)	UART1
TXD(1)	IO(64)	P2(0)	
RXD(2)	IO(73)	P2(9)	UART2
TXD(2)	IO(72)	P2(8)	
RXD(3)	IO(157)	P4(29)	UART3
TXD(3)	IO(156)	P4(28)	

Analog connections

4 A/D converters are readily available, Analog 2-4. 2 more are available, but share the pins with UART0 -- what was NXP thinking, I have no idea.

1 10 bit DAC is available shared with AD(3) available on the SuperPRO (not on PROplus)

On reset or power up the AD pins are configured by software as AD inputs. To change those to digital IOs, the user must write to the appropriate PINSEL register, or with version 8.11 firmware or later you can change it to an IO by accessing the corresponding IO pin in the following table.

AD	BASIC	NXP
AD(2)	IO(25)	P0(25),DACOUT
AD(3)	IO(26)	P0(26)
AD(4)	IO(62)	P1(30)
AD(5)	IO(63)	P1(31)
AD(6)	IO(3)	RXD(0)/P0(3) / AD(6)
AD(7)	IO(2)	TXD(0)/P0(2) / AD(7)

The LPC1756 does support an external reference for the A/D converters, but to use the Arduino

AREF pin a jumper is required (details on the schematic)

The A/D input requires a drive impedance of 7.5K or less (see NXP LPC175x spec sheet). We've also found a 100 to 1000 pF cap from AD input to GND can remove high frequency noise affecting high order bits in the converter.

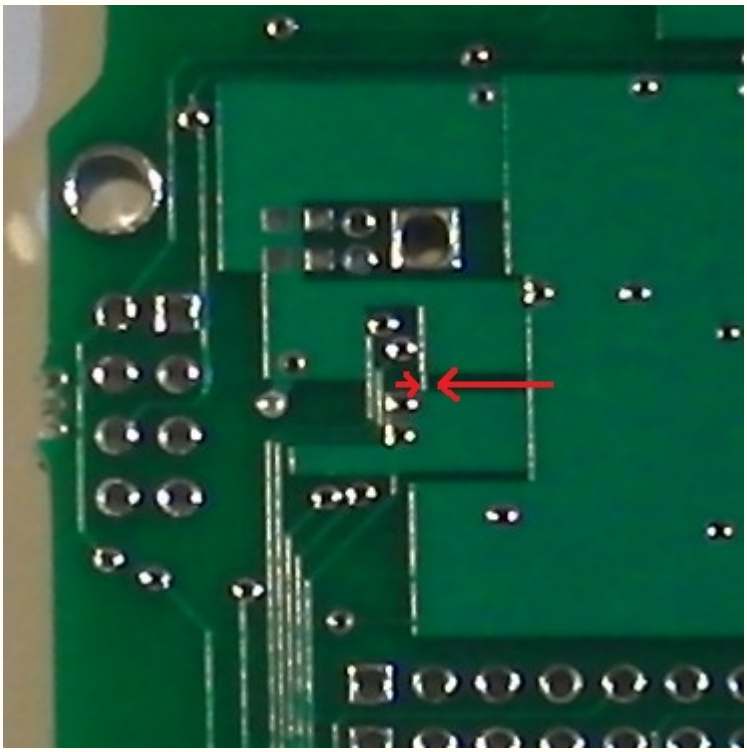
Analog Isolation

The rev 6 and 7 boards isolate both GND and power for the analog section using ferrite beads.

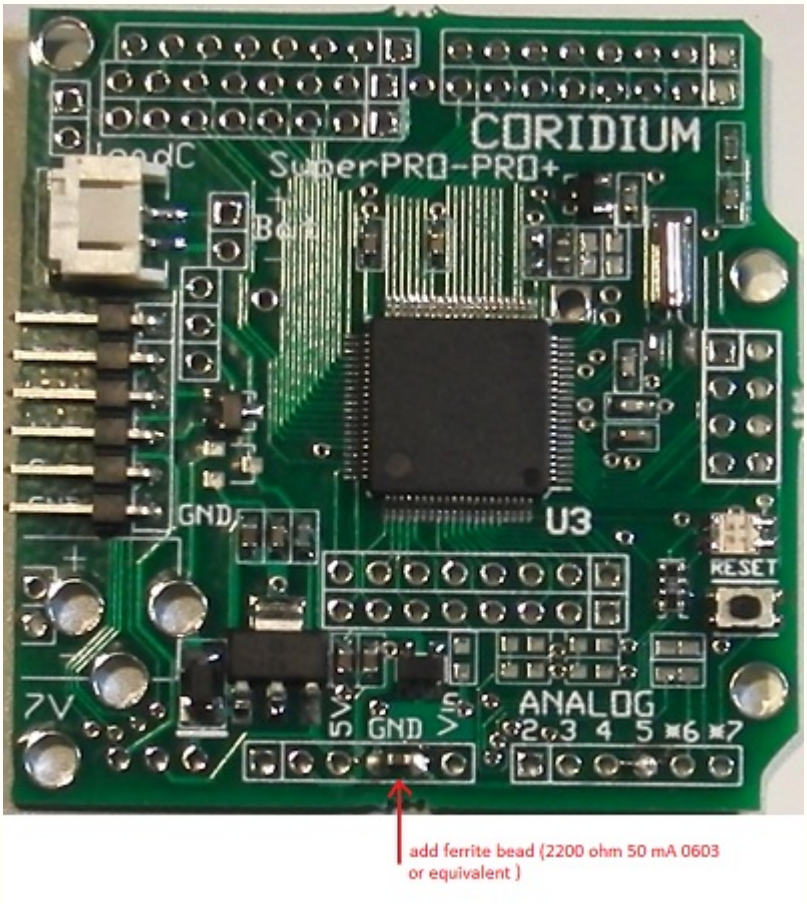
to add isolation to rev 4/5 boards -

The LPC17xx series chips AD converter are sensitive to high frequency noise on the analog GND (Vssa) or on the AD inputs themselves. A symptom that will show up is bits in any bit position turned on/off when the conversion is done. This makes it hard to average out, but conversion can be voted on, choosing 2/3 conversions that agree within a few bits. The occurrence of these errors is in less than 1% of the conversions, unless your setup is very noisy.

Another option is to change the analog GND connection on the board. Do this by cutting the trace on the back side between GND under the crystal and the GND connected to Vssa (shown on the picture below)



Then connect digital GND to analog GND using a ferrite bead, a convenient place to do this is on the front side as shown below.



Pin limitations

P0.29 and P0.30 direction control must be done in parallel, they can be both outputs or both inputs, but not mixed. With firmware 8.11 or later, changing the direction of either P0.29 or P0.30 will change the other pin.

Power connections -- SuperPRO

The board is shipped with a **2mm power jack compatible** with a JST PHR/S2B or **SparkFun PRT8671** or various battery packs from SparkFun.

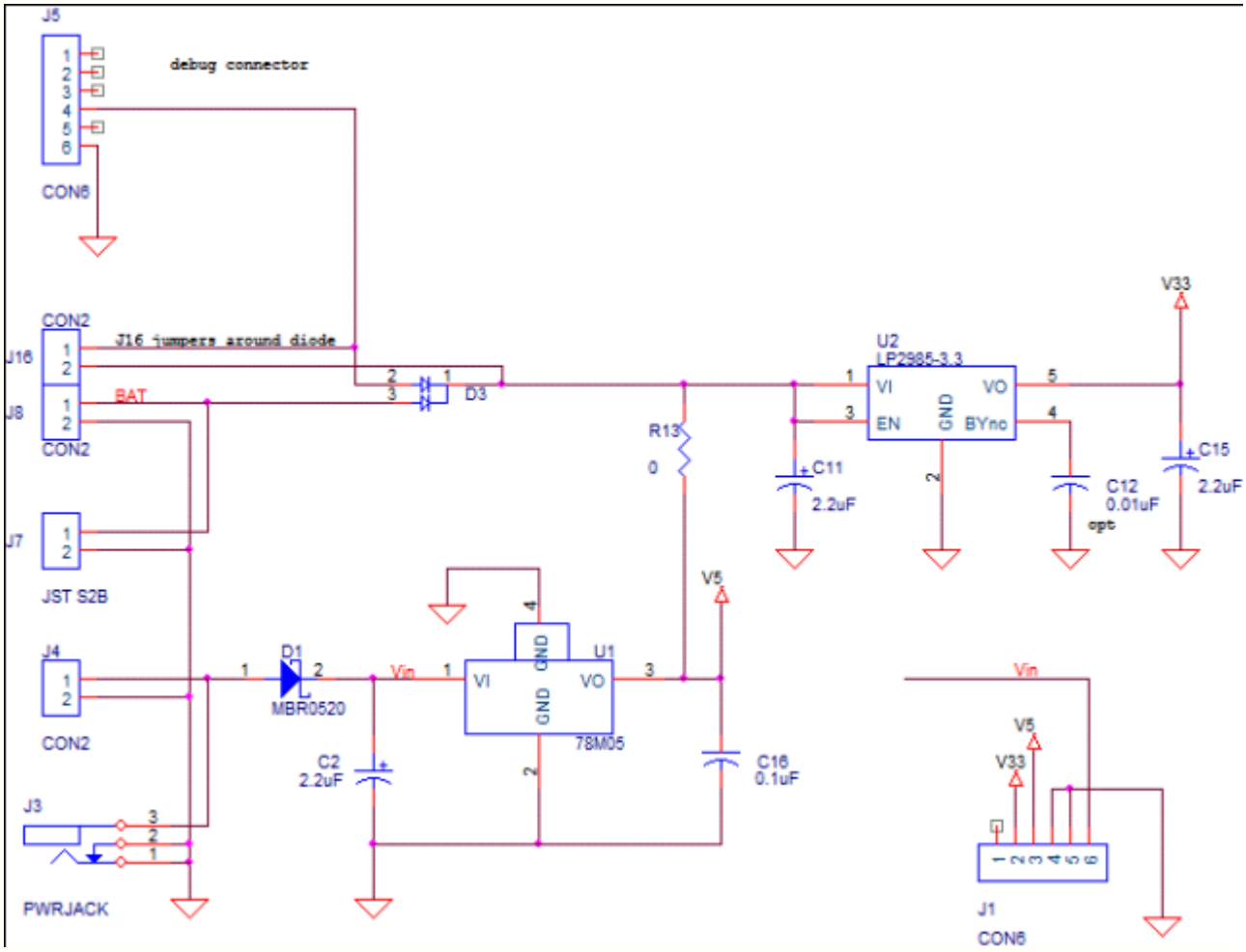
Pads for a Cui PJ-002A or **SparkFun PRT-119** power connector are available in the lower left hand corner.

For both battery and 6V input, 2 pin 0.1" spaced holes are available for wires or headers. When using the battery connector, total current draw for the board must be limited to 200mA. If you want to use more current, you should install a jumper around the D2 diode (holes are available above D2).

Diode steering allows power to be supplied from a barrel connector from a 6V unregulated source, 5V USB test connector, or the battery connector. Because of the Schottky diodes, all 3 power sources can be connected simultaneously. If you are using an unregulated wall transformer, you must check the open circuit voltage and it MUST be less than 12V.

When the 6V source is used, 5V Arduino shields can be powered from the SuperPRO.

The schematic below describes this circuit on the SuperPRO



Power connections -- PROplus

The board is shipped with a **2mm power jack compatible** with a JST PHR/S2B or **SparkFun PRT8671** or various battery packs from SparkFun.

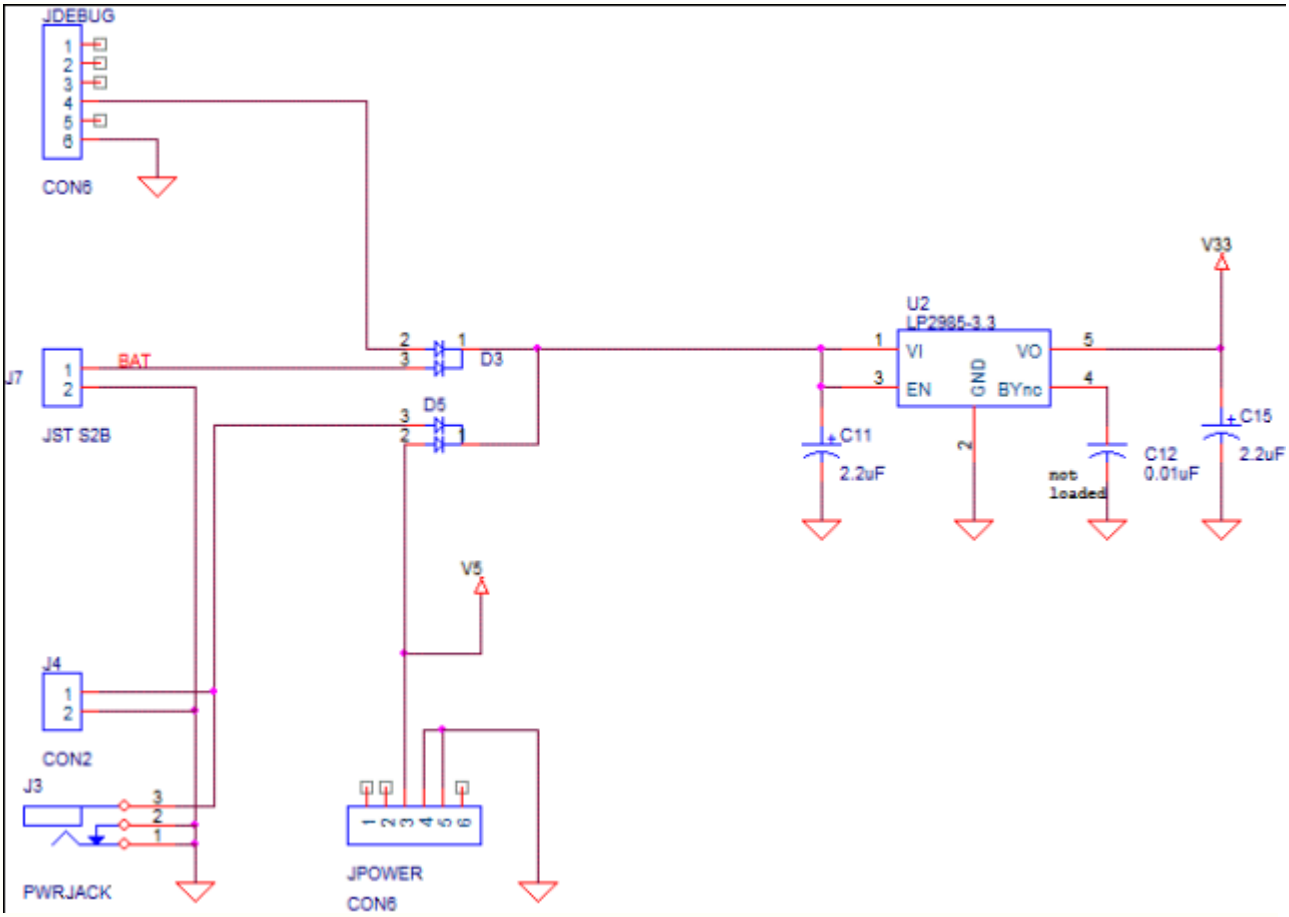
Pads for a Cui PJ-002A or **SparkFun PRT-119** power connector are available in the lower left hand corner.

For both battery and 6V input, 2 pin 0.1" spaced holes are available for wires or headers. When using the battery connector, total current draw for the board must be limited to 200mA. If you want to use more current, you should install a jumper around the D2 diode (holes are available above D2).

Diode steering allows power to be supplied from a barrel connector from a 6V unregulated source, 5V USB test connector, 5V from a shield or the battery connector. Because of the Schottky diodes, all 3 power sources can be connected simultaneously. If you are using an unregulated wall transformer, you must check the open circuit voltage and it MUST be less than 12V.

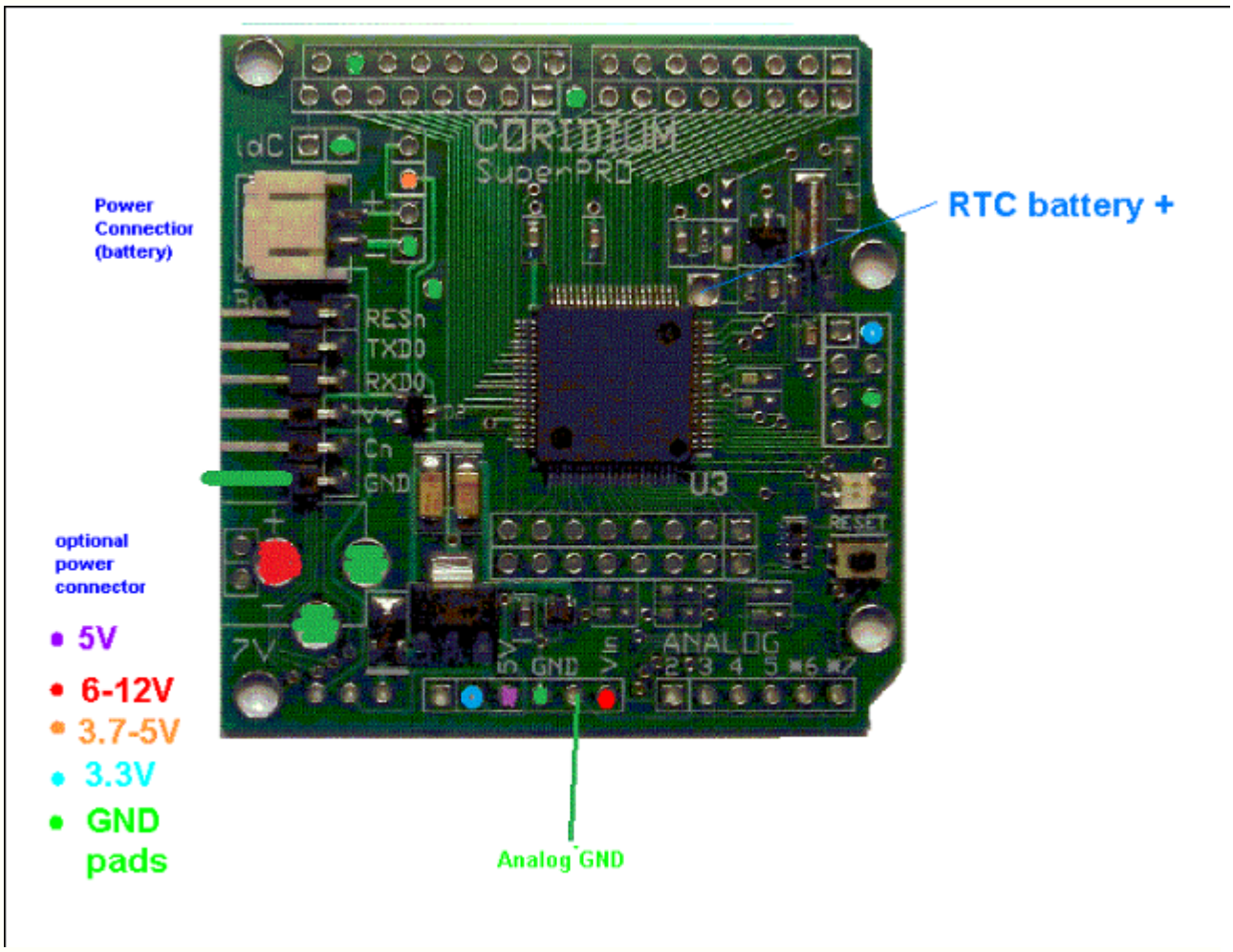
The PROplus only has the 3.3V regulator, so it cannot supply power to a 5V shields.

The schematic below describes this circuit on the PROplus



The **full schematic can be seen here** and for **rev 8**

Power connections details



The 3.3V regulator can supply 50 mA, with most being used by the LPC2103. The 3.3V connection next to RESn on the lower power connector is only connected if the shorting pads are shorted (NOT the factory default).

The analog GND should be used to connect to the GND of analog inputs. Digital and Analog GNDs are connected together with a small trace, but to minimize noise you should use the analog GND only for analog signals.

Jumpers and test connector for Program Download

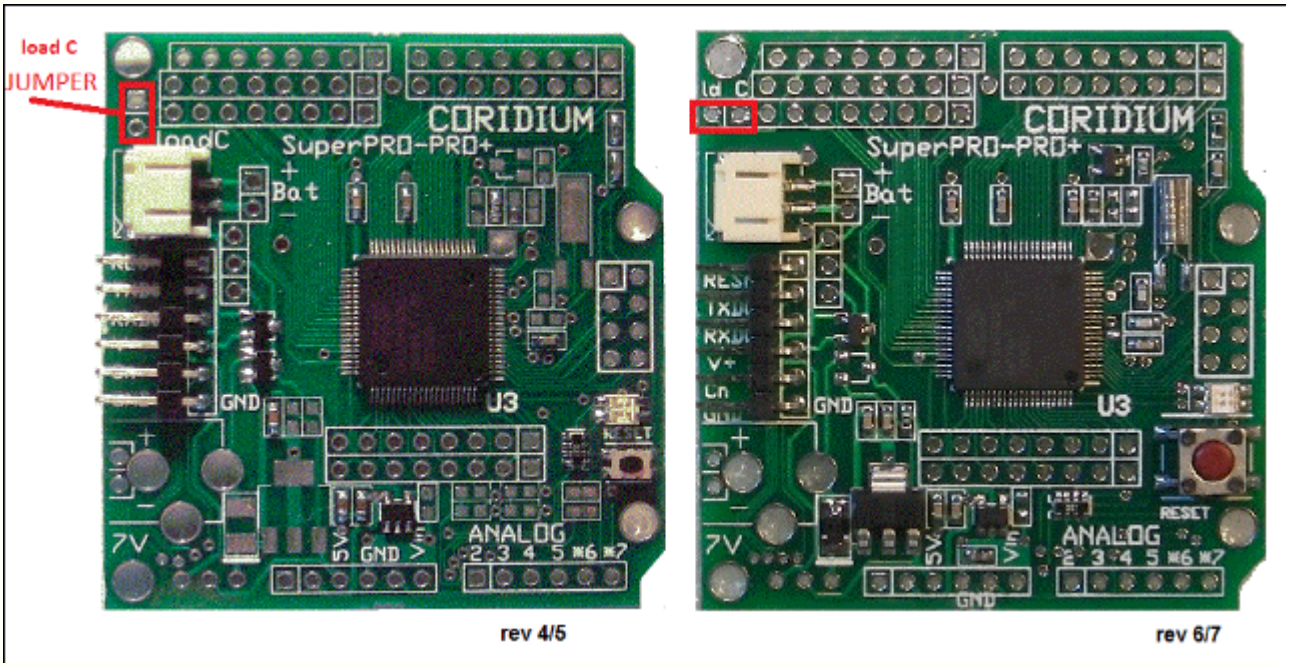
The USB Dongle from Coridium will supply 5V from the USB to power the ARMmite PRO. It also controls the RESET and BOOT(P2.10) signals to automatically load C or BASIC programs using MakeItC or BASICtools. Remember, if you load a C program, it will erase the BASIC firmware and you will not be able to load BASIC programs after that.

When using the SparkFun FTDI Basic Breakout Board, a limited amount of power can be supplied from the BBB, but this is limited to 50 mA and after diode drops, its about 2.8V to the LPC2103. In practice this will run, but it is outside the part specifications, so it should be limited in use.

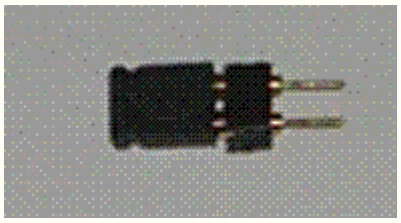
Also with the SparkFun FTDI Basic Breakout Board to load a C program, the LOAD C jumper needs to be installed, then removed to run the program. BASIC programs can be loaded and controlled using the SparkFun board, with no additional steps/jumpers.

PROplus

SuperPRO



An alternative is to use a 2 pin header with a shorting block (pictured below)

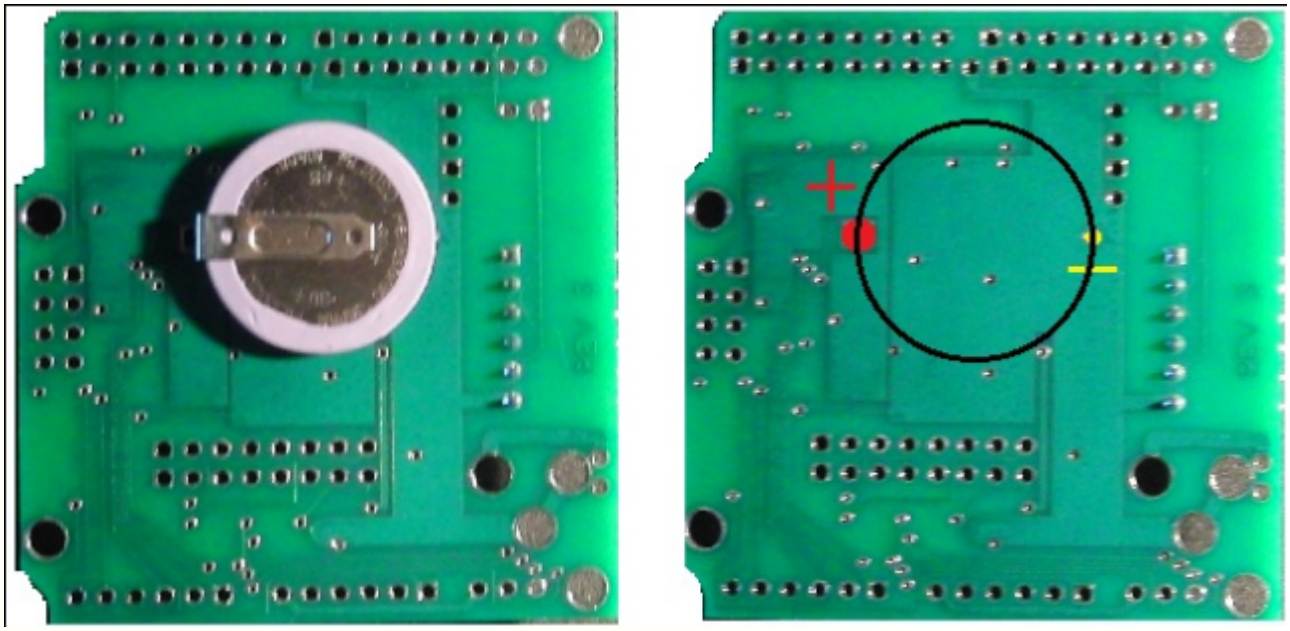


Real Time Clock Oscillator

The RTC oscillator of the LPC17xx parts has been resolved. The first generation parts which were shipped in early 2011 had an unreliable oscillator and this has been corrected by NXP.

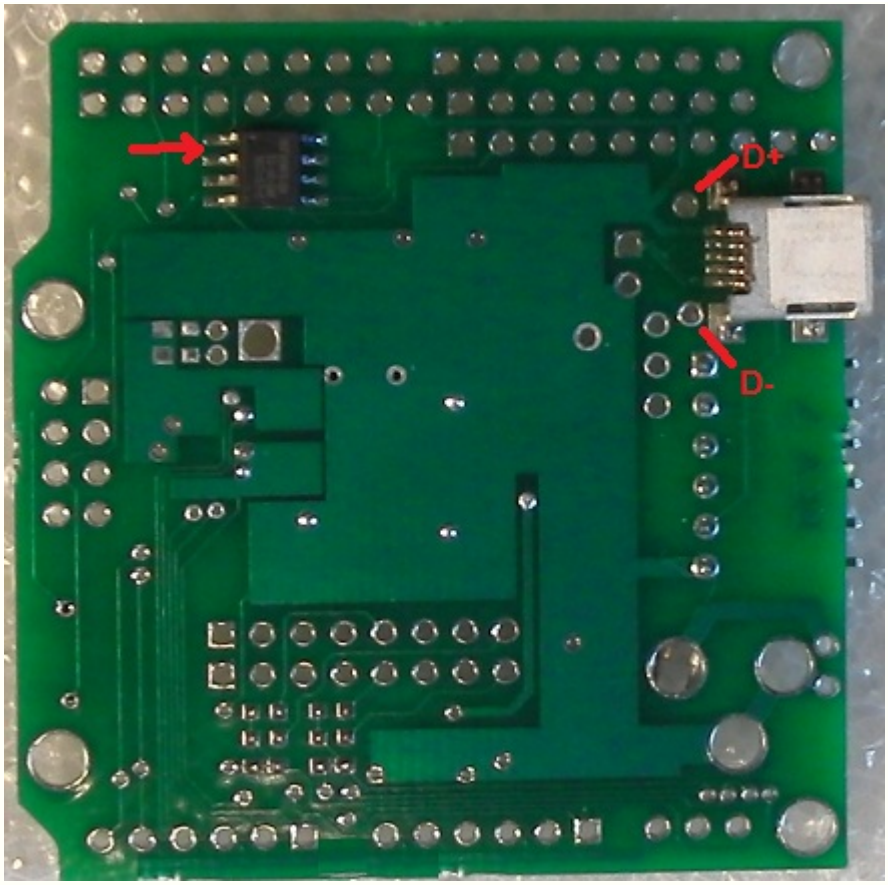
A 32 KHz crystal and diode for battery backup with an optional ML2020 rechargeable Li battery.

A Panasonic ML2020H rechargeable battery may be added to keep the real time clock running when power is removed. The battery is mounted on the back of the board as shown below. The VL2020/HFN will also work, though it is more expensive and has less power.



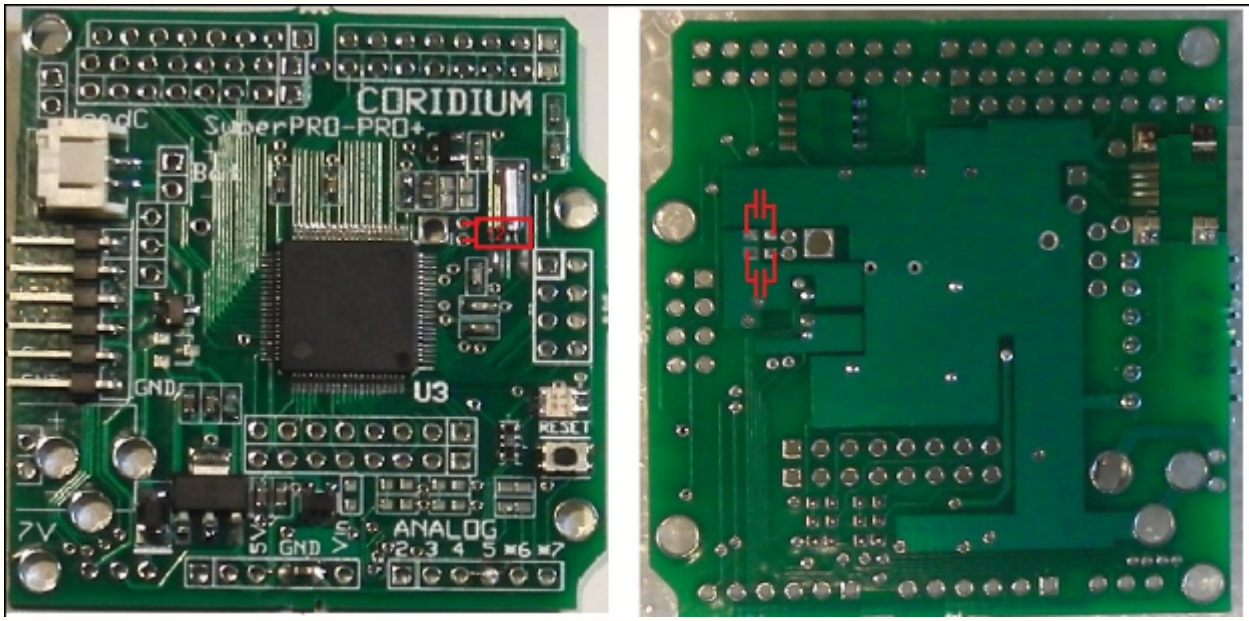
USB connector option for power and SPI Flash option

The rev 6/7 boards add pads for an optional SPI Serial Flash (note pin 1 location). Also pads for a USB mini-B connector have been added, this is intended primarily to supply power, and the data lines are connected to pads. These options can be installed at Coridium for orders of 10 boards or more. Contact **sales@coridiumcorp.com** for details.



Main Clock Crystal option

You can add a **12 MHz crystal** with 39pf 0603 load caps, for use as a more accurate clock source. Locations marked below



Timing

The oscillator
The ARMexpress uses a ceramic resonator for the timing element. It is accurate for 1%. It is used for timing of operations of SERIN, SEROUT, OWIN, OWOUT, PULSEIN, PULSEOUT, and COUNT.

Other operations such as I2CIN, I2COUT, SPIIN, SPIOUT, SHIFTIN, SHIFTOUT, and PWM are "bit-banged" loops that are calibrated to the speed of the CPU.

The real time clock
The ARMexpress, ARMexpress LITE, or ARMmite wireless use the CPU clock based on the ceramic resonator for the timing element. It is accurate for 1%.

The ARMmite and ARMweb use a 32KHz crystal which is much more accurate for timing of SECONDS, MINUTES, HOURS, DAYS, MONTH and YEAR. It is accurate to 100ppm. And on the ARMmite or ARMweb it can be kept running with a battery.

Interrupts

The serial port connection through the USB uses interrupts for all products. The service routines for these actions have been minimized so that the user program is only interrupted for TBD microseconds. The ARMweb also uses a 10 msec timer interrupt. With version 7.09 firmware and later interrupts on 2 pins or timer are available to the user BASIC program.

Operations that require accurate timing will disable the interrupts during that critical period. These operations include OWIN, OWOUT, SERIN and SEROUT. Other operations that would be negatively impacted by an interrupt also disable the interrupt for a period of time. Those include PULSIN, PULSOUT, PWM, and RCTIME.

Interrupts and User code
When the ARMexpress receives serial input it will interrupt to copy data into its buffer. This will cause a small delay in the users program. In most cases this is not noticeable, but may be where user is timing with TIMER.

User code can cause the serial port to be deaf when running long operations such as PWM. In normal operation this should not be a problem.

AD timing (ARMmite, ARMmite Wireless, ARMexpress LITE, and ARMweb)

The analog inputs can do a conversion in 11 uSec.

General Interfacing

Both the ARMexpress and the ARMmite can be directly connected to 5V TTL devices. The output voltage for these ARM devices ranges from 0.4V to 2.9V when driving up-to 4mA of current. Most TTL devices will recognize these as valid logic levels (normally defined to be 0.8 and 2.0V)

Inputs

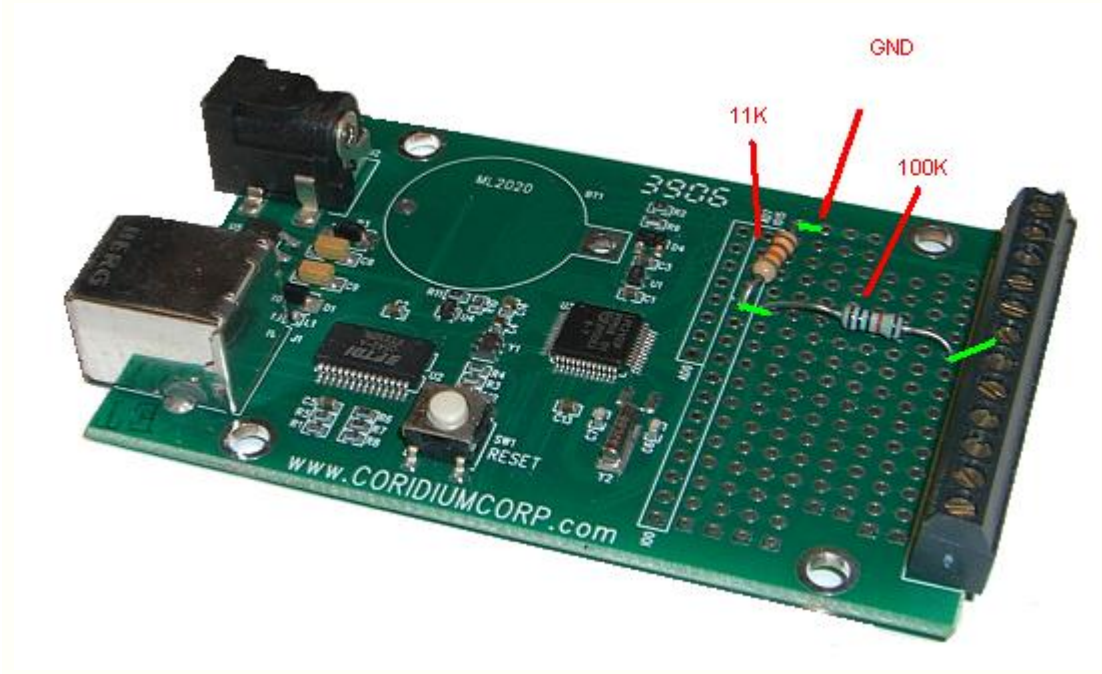
The ARMexpress and ARMmite may also be directly connected to 5V TTL outputs. If they are TTL compatible the voltage levels of the TTL output would normally be (0.4 and 3.4V), though they may go higher. The inputs for these ARM devices are 5V compatible.

Tying to Supply lines

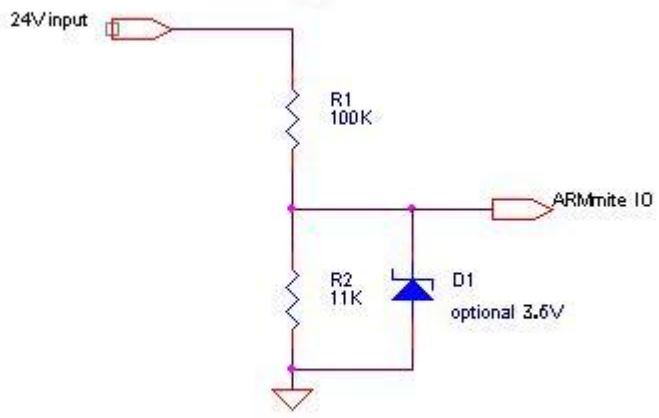
The ARMexpress and ARMmite inputs may be connected directly to a GND pin, but if connecting to a fixed voltage supply, then a 1K or greater resistor in series is recommended. This is the same recommendation for any TTL compatible device. The reason being is that the 5V supply may exceed the 5V at times, or if that voltage is available before the power supply to the CPU, large currents may flow through the protection diodes in the CPU.

Interfacing to higher voltages

A resistor divider may be used to connect the ARMexpress and ARMmite to voltages that go higher than 5V. The picture below shows a connection appropriate for a 24V signal. A 100K resistor is connected from the input to IO(11) and then an 11K resistor connects IO(11) to GND. This will divide that 24V input to vary between 0 and 2.4V.



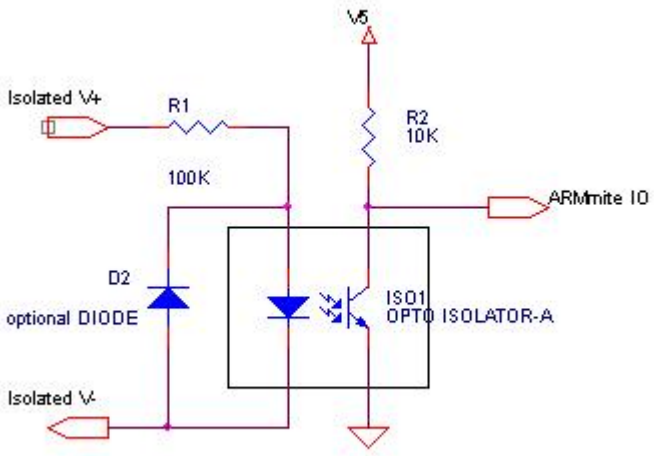
This resistor divider divides the 24V by 10 and also limits the current if that 24V goes higher. The circuit below shows schematically the connection that was made.



The resistors can be varied to handle different voltages. If the voltage to be sensed is susceptible to large spikes a 3V Zener diode can be connected in parallel with R2 to further protect the ARMmite IO.

Opto-Isolator

Another way to sense large voltages and to isolate the ARMmite from those voltages is to use an opto-isolator. These devices consist of an LED and a photo-transistor in a single package. They can provide isolation of 1000s of Volts. Below is a sample circuit. The D2 optional diode should be used if the isolated voltage to be sensed is an AC voltage. The value of R1 should be chosen depending on the Opto-isolator spec, with the current through the opto-isolator diode typically being 10 mA.



Driving Transistors

The ARMexpress outputs are rated for 4mA, when more is required a common 2N3904 transistor can be used for 100-200 mA. The base of the transistor is driven from an IO with a series resistor. When the IO is high the transistor is turned on.

Driving Relays

When higher currents or voltage are involved a relay can be used. For mechanical relays a driving transistor with a catch diode are required. The circuit starts as the above transistor circuit, which when on can either close or open the relay contacts. When it turns off, current continues to flow in the coil of the relay as the

magnetic field collapses, this current needs to go somewhere, that's what the catch diode provides is a path for that current to flow back into the supply of the relay.

USB use

During programming BASICtools is used to load the users ARMbasic program. But once the user's ARMbasic program is running the USB port may be used to communicate data back to the PC.

General Info

The USB port is configured as a USB slave device and emulates a serial port for the PC. Drivers are also available from FTDI for the Mac or Linux (FTDI 232RL running in serial emulation mode, normally VCP type driver).

PC side programs

Any program on the PC that can communicate with a serial port can send or receive data to the ARMexpress eval PCB or the ARMmite. This would include MSCOMM and Visual BASIC. Also various C's including GCC. Other options include Perl or Tcl scripts.

However these programs must be able to control the DTR and RTS lines under user control. If they cannot refer to the next **section**. Programs that cannot include **Teraterm, Hyperterm** and **MatLab** .

The TclTerm.tcl is the source for a Tcl program that operates as a terminal emulator for Coridium products. You can use it if you have access to any of the GPL Tcl interpreters, or a compiled version is available on the Coridium Support page. The sources are also installed with MakeItC and BASICtools.

Baudrate

Baudrate will remain at 19.2Kb, unless changed by the user program which can be done with

```
#include <SERIAL.bas>
BAUD0 (newrate)
```

Output of Data to PC

The ARMbasic program can use PRINT, or TXD0

Input of Data from PC

An ARMbasic program should use RXD0. These routines will return -1 if no data is available. This allow the users program to continue doing other tasks, or the user program can loop waiting for input on RXD0.

DEBUGIN in a user program will wait for data, even if that is for ever. It is not a good practice to use this function for sending data back to the PC. Its operation is recommended for user interaction with programs during the development stage, while using BASICtools.

USB use with Linux, Hyperterm, TeraTerm

General Info

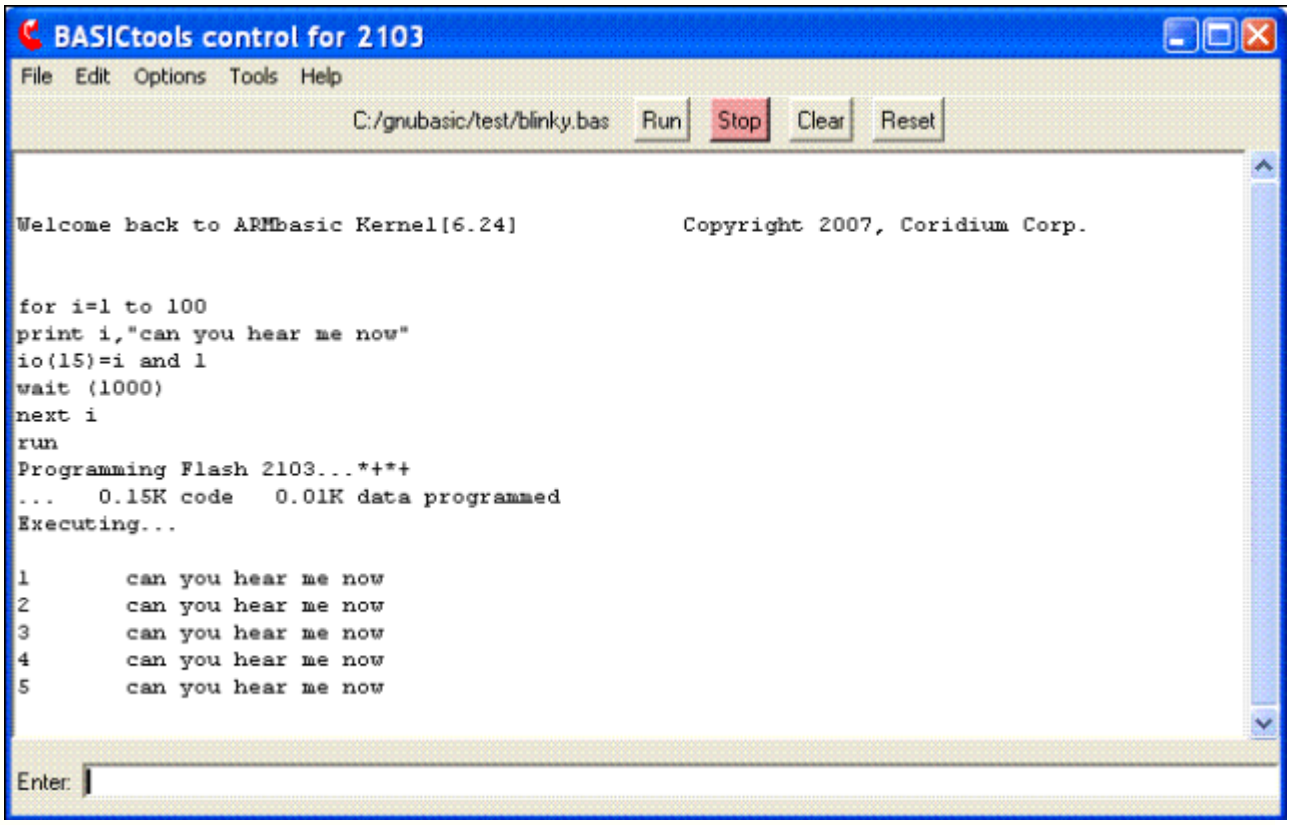
The ARMmite and ARMexpress use the DTR and RTS serial control lines to control programming and reset for the device. The state chosen allows the ARMmite/express to run and be reset by the push button while idle (i.e. no serial program running).

PC side programs

Programs on the PC such as Tcl, MSCOMM and GCC allow the control lines to be controlled by the user. But some pre-compiled programs do not allow this control, such as HyperTerminal, TeraTerm, and some Linux apps. This page describes the steps to allow these programs to operate.

Useful debugging tool

Before starting its useful to load a program into the ARMmite/express that will pulse the LED and also continuously send some data out the serial port. Here is one that works well...



Download the latest BASICtools and TclIterm

In order to be able to communicate with the ARMmite/express after the control lines have been changed, make sure you are running the latest TclTerm. Versions 1.6 and later have this support.

http://www.coridiumcorp.com/files/setupBASIC.exe

Next, the driver must be changed for the USB serial device. The FTDI D2XX driver must be used. Download it from the FTDI website.

http://www.ftdichip.com/Drivers/D2XX.htm

Choose the proper version for your operating system, and download and install the driver. The installation executable may be used, and there are instructions in the Installation Guides on that page.

Configuration Utility

Next the settings of the serial control lines need to be changed, this requires the MProg utility from FTDI. Download and install this program.

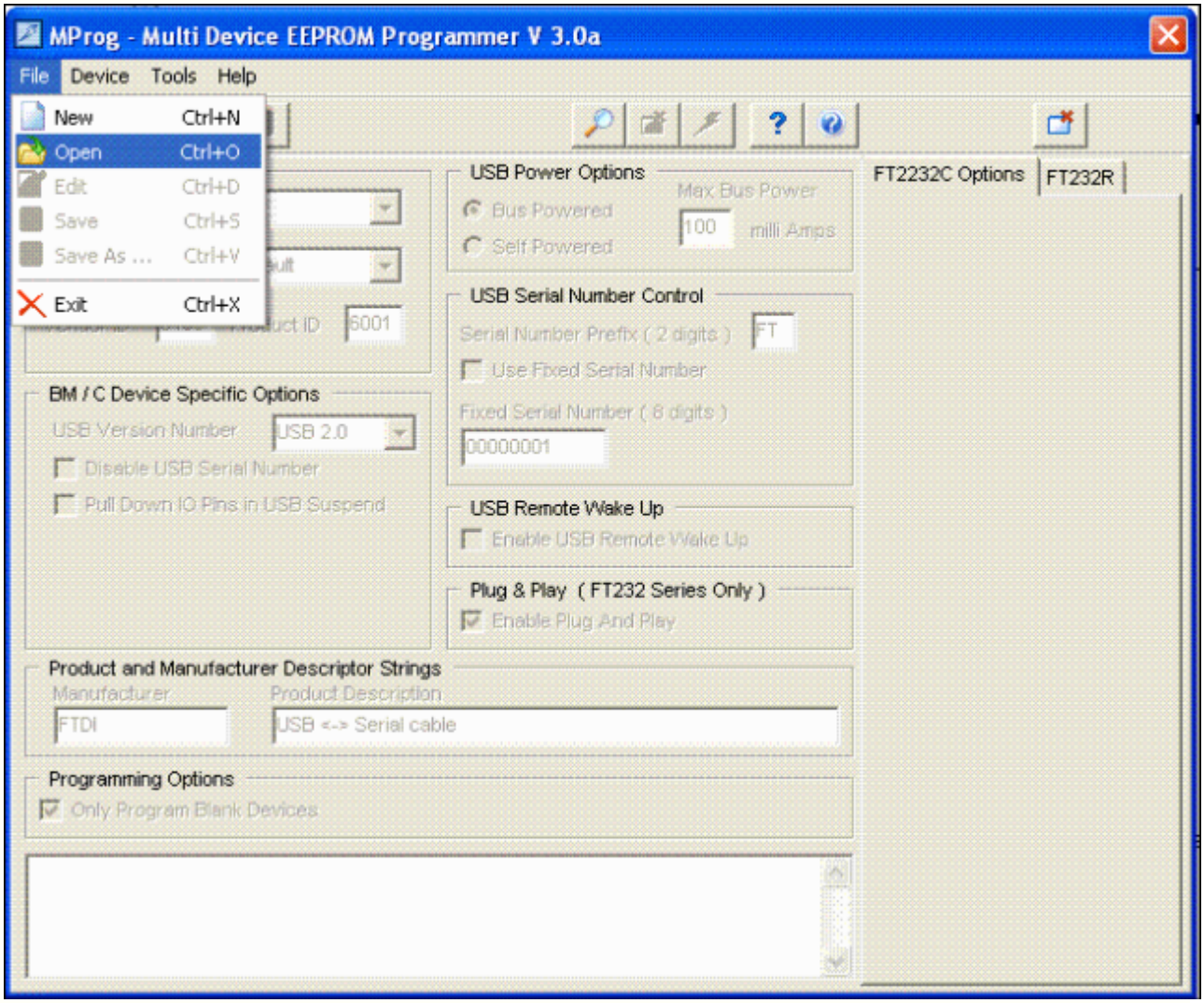
http://www.ftdichip.com/Resources/Utilities/MProg3.0_Setup.exe

Next download the data files for configuration of the ARMmite or ARMexpress eval PCBs. Unzip these files and store in a convenient directory (such as C:/Program Files/MProg 3.0a/Templates)

http://www.coridiumcorp.com/files/USBconfig.zip

Setup ARMmite/ARMexpress for MatLab, HyperTerminal, or TeraTerm

Run the MProg utility. Load the ***serial or legacy*** File version. And then reprogram the FTDI chip. **ONLY have 1 ARMmite or ARMexpress plugged in at time when you perform this operation.**

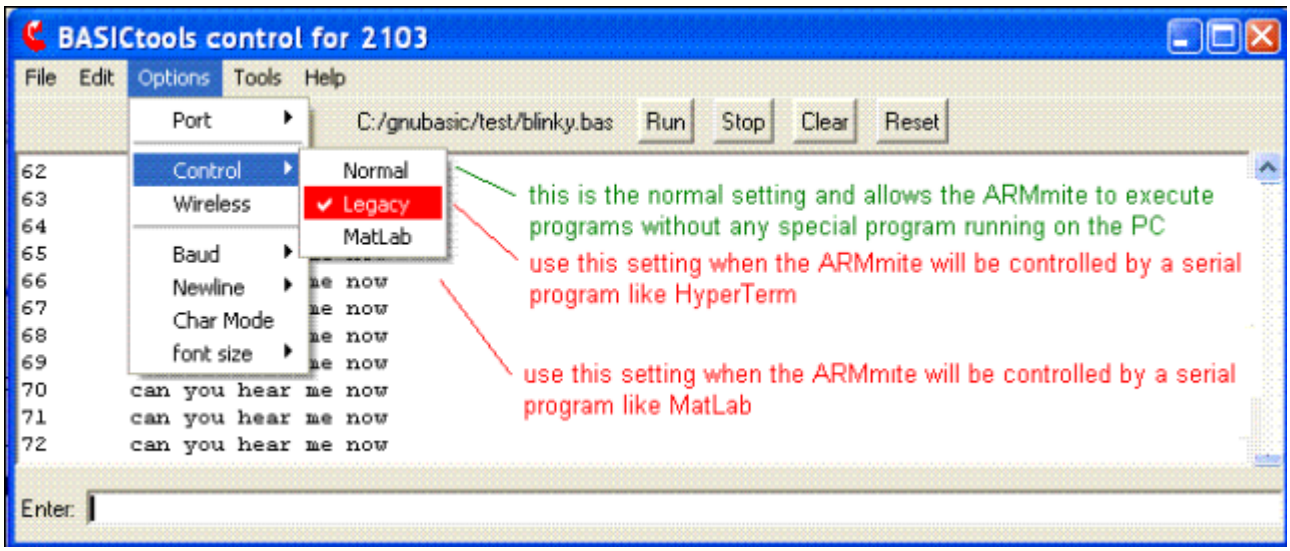


Exit this program and close any serial programs such as BASICtools. For this change to take effect, the ARMmite/express must be disconnected from the PC and reconnected.

Now the ARMmite/express will be idle until the serial port is open, when Hyperterminal, or TeraTerm is run. Then after those programs are run, to start your BASIC or C program press the RESET pushbutton on the ARMmite/express.

Change the BASICtools settings for the reconfigured ARMmite/ARMexpress

In order to be able to change the BASIC program, you will still need to use BASICtools, but it will have to be configured to use the new control line configuration (DTR and RTS inverted).



USB use with MatLab

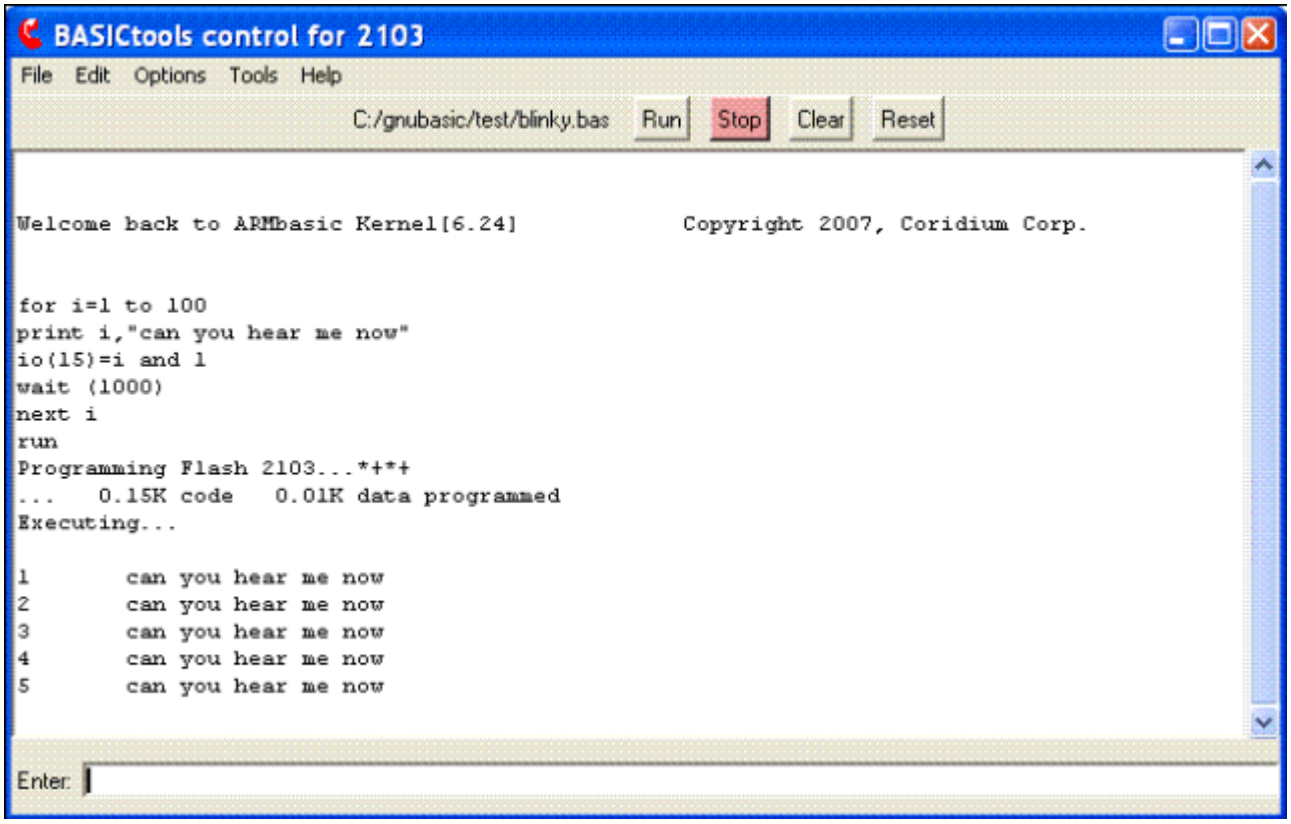
General Info

The ARMmite and ARMexpress use the DTR and RTS serial control lines to control programming and reset for the device. The state chosen allows the ARMmite/express to run and be reset by the push button while idle (i.e. no serial program running).

MatLab holds DTR high, but RTS low when it opens a serial port.

Useful debugging tool

Before starting its useful to load a program into the ARMmite/express that will pulse the LED and also continuously send some data out the serial port. Here is one that works well...



Download the latest BASICtools and TcIterm

In order to be able to communicate with the ARMmite/express after the control lines have been changed, make sure you are running the latest BASICtools. Versions 4.1 and later have support for MatLab.

http://www.coridiumcorp.com/files/setupBASIC.exe

Next, the driver must be changed for the USB serial device. The FTDI D2XX driver must be used. Download it from the FTDI website.

http://www.ftdichip.com/Drivers/D2XX.htm

Choose the proper version for your operating system, and download and install the driver. The installation executable may be used, and there are instructions in the FTDI Installation Guides on that page.

Configuration Utility

Next the settings of the serial control lines need to be changed, this requires the MProg utility from FTDI. Download and install this program.

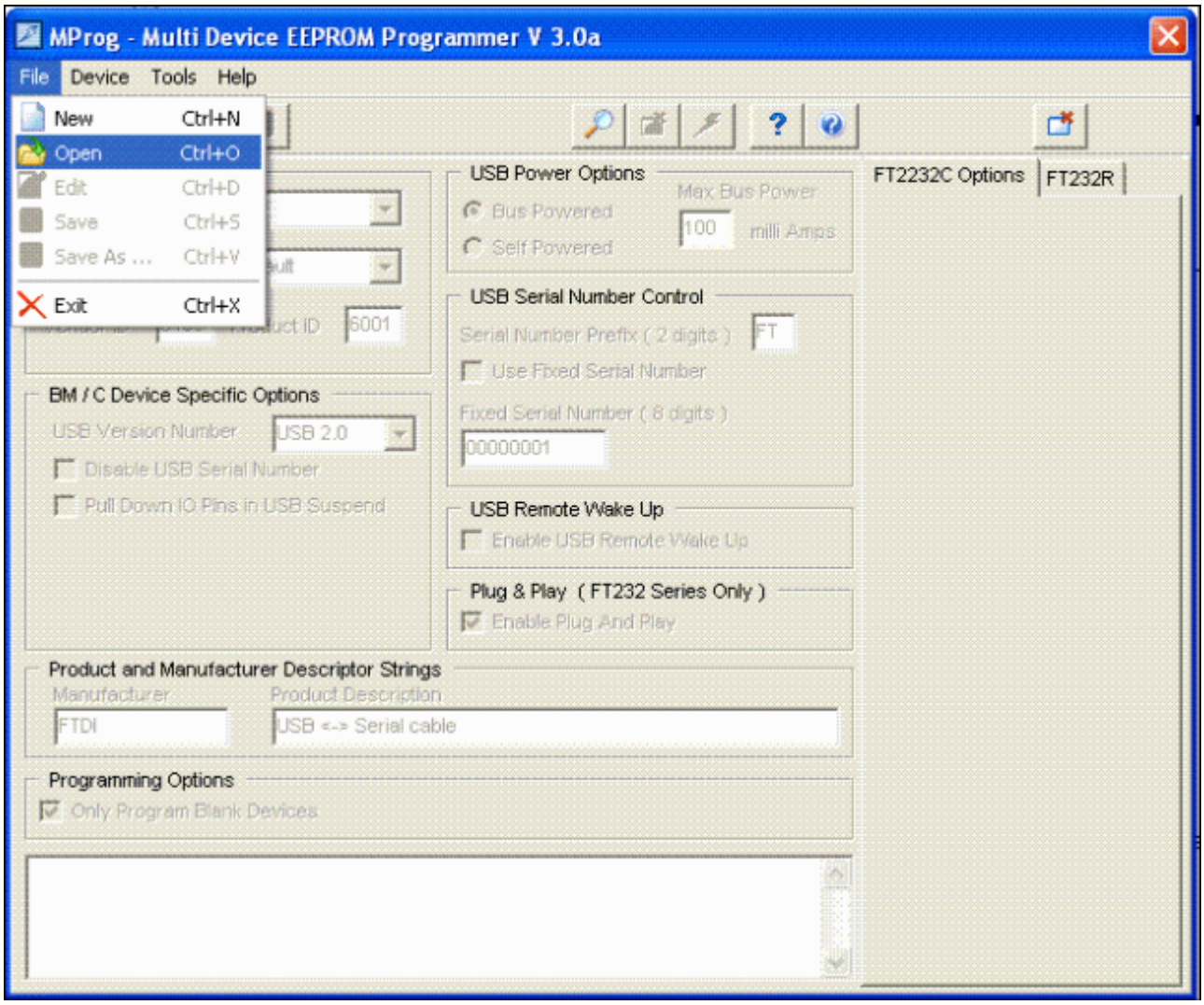
http://www.ftdichip.com/Resources/Utilities/MProg3.0_Setup.exe

Next download the data files for configuration of the ARMmite or ARMexpress eval PCBs. Unzip these files and store in a convenient directory (such as C:/Program Files/MProg 3.0a/Templates)

http://www.coridiumcorp.com/files/USBconfig.zip

Setup ARMmite/ARMexpress for MatLab

Run the MProg utility. Load the ***matlab*** File version in. And then reprogram the FTDI chip. **ONLY have 1 ARMmite or ARMexpress plugged in at time when you perform this operation.**

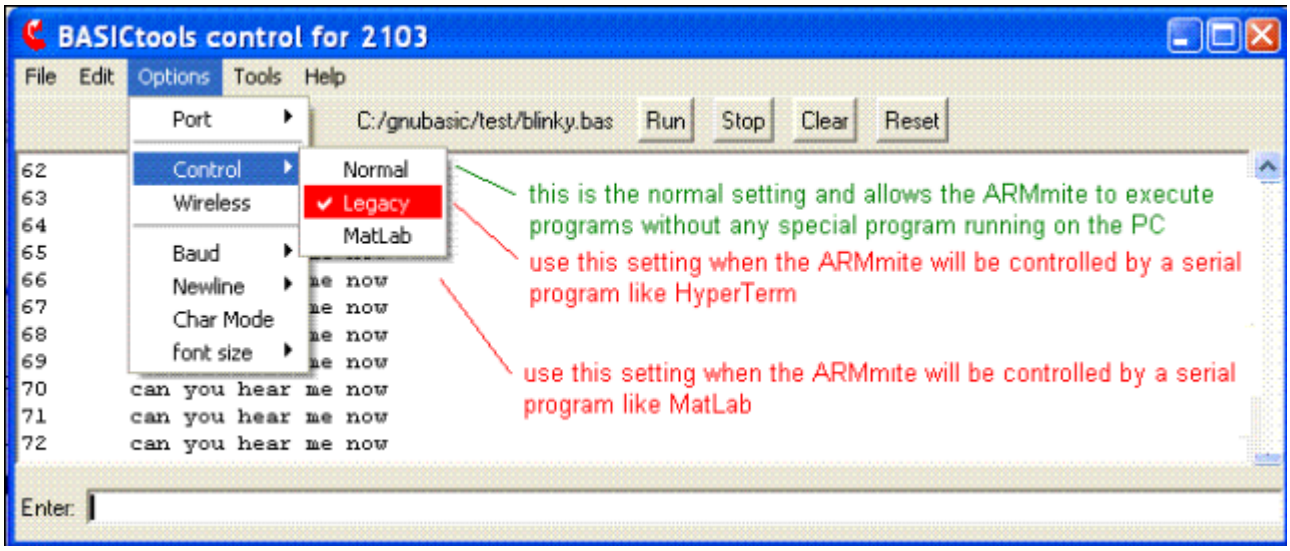


Exit this program and close any serial programs such as BASICtools. For this change to take effect, the ARMmite/express must be disconnected from the PC and reconnected.

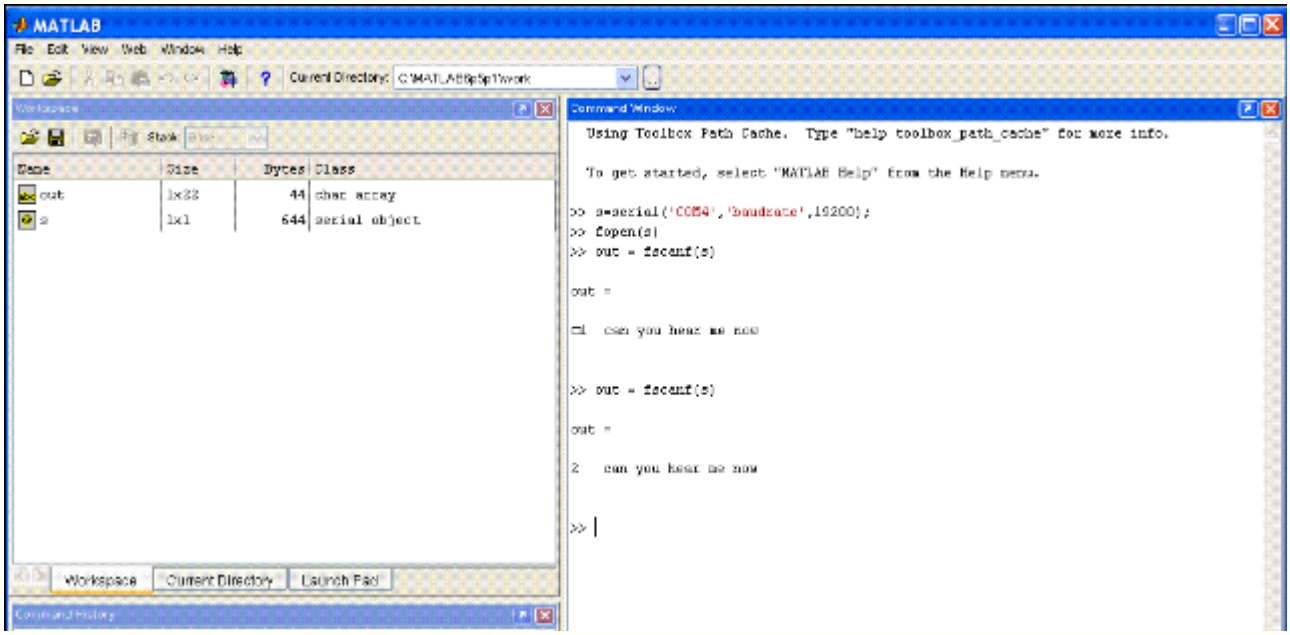
Now the ARMmite/express will be idle until the MatLab serial port is open. Then after those programs are run, to start your BASIC or C program press the RESET pushbutton on the ARMmite/express.

Change the BASICtools settings for the reconfigured ARMmite/ARMexpress

In order to be able to change the BASIC program, you will still need to use BASICtools, but it will have to be configured to use the new control line configuration (DTR and RTS inverted).

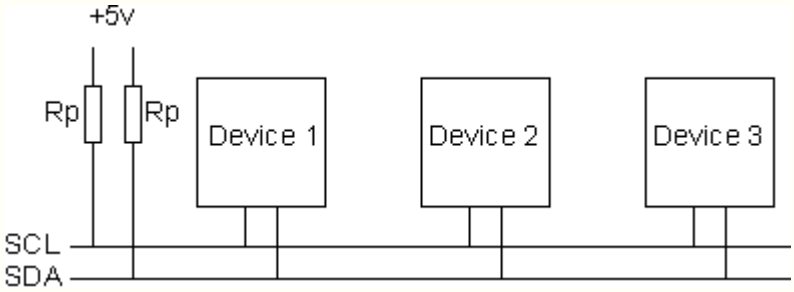


Check operation with MatLab



Using the I2C Bus

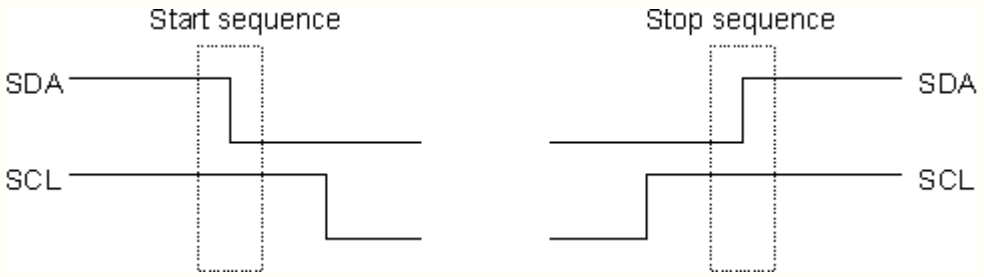
The physical I2C bus
This is just two wires, called SCL and SDA. SCL is the clock line. It is used to synchronize all data transfers over the I2C bus. SDA is the data line. The SCL & SDA lines are connected to all devices on the I2C bus. There needs to be a third wire which is just the ground or 0 volts. There may also be a 5volt wire is power is being distributed to the devices. Both SCL and SDA lines are "open drain" drivers. What this means is that the chip can drive its output low, but it cannot drive it high. For the line to be able to go high you must provide pull-up resistors to the 5v supply. There should be a resistor from the SCL line to the 5v line and another from the SDA line to the 5v line. You only need one set of pull-up resistors for the whole I2C bus, not for each device, as illustrated below:



The value of the resistors should be from 1.8K (1800 ohms) to 4.7k (4700 ohms). It depends on the length of the I2C bus, the longer the bus, the smaller value should be used. If the value is too large, the rise time of the signals will be too slow and the bus may not work properly. If the resistors are missing, the SCL and SDA lines will always be low - nearly 0 volts - and the I2C bus will not work.

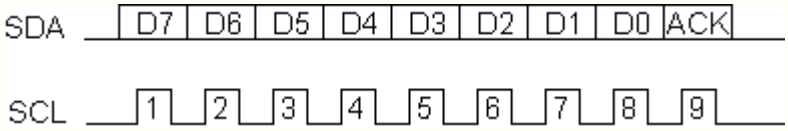
Masters and Slaves
The devices on the I2C bus are either masters or slaves. The ARMexpress as a master is always the device that drives the SCL clock line. The slaves are the devices that respond to the master. A slave cannot initiate a transfer over the I2C bus, only a master can do that. There can be, and usually are, multiple slaves on the I2C bus, however there is normally only one master. ARMexpress does not support multiple masters. Slaves will never initiate a transfer. Both master and slave can transfer data over the I2C bus, but that transfer is always controlled by the master.

The I2C Physical Protocol
When the ARMexpress wishes to talk to a slave it begins by issuing a start sequence on the I2C bus. A start sequence is one of two special sequences defined for the I2C bus, the other being the stop sequence. The start sequence and stop sequence are special in that these are the only places where the SDA (data line) is allowed to change while the SCL (clock line) is high. When data is being transferred, SDA must remain stable and not change whilst SCL is high. The start and stop sequences mark the beginning and end of a transaction with the slave device.



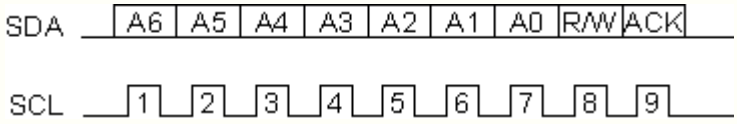
Data is transferred in sequences of 8 bits. The bits are placed on the SDA line starting with the MSB (Most Significant Bit). The SCL line is then pulsed high, then low. Remember that the chip cannot really drive the line high, it simply "lets go" of it and the resistor actually pulls it high. For every 8 bits transferred, the device receiving the data sends back an acknowledge bit, so there are actually 9 SCL clock pulses to transfer each 8 bit byte of data. If the receiving device sends back a low ACK bit, then it has received the data and is ready to accept another byte. If it sends back a high then it is indicating it cannot accept any further data and the

master should terminate the transfer by sending a stop sequence.



How fast?
ARMexpress runs in Fast mode at approximately 380 KHz.

I2C Device Addressing
All I2C addresses are either 7 bits or 10 bits. The use of 10 bit addresses is rare and is not covered here. All of our modules and the common chips you will use will have 7 bit addresses. This means that you can have up to 128 devices on the I2C bus, since a 7bit number can be from 0 to 127. When sending out the 7 bit address, we still always send 8 bits. The extra bit is used to inform the slave if the master is writing to it or reading from it. If the bit is zero are master is writing to the slave. If the bit is 1 the master is reading from the slave. The 7 bit address is placed in the upper 7 bits of the byte and the Read/Write (R/W) bit is in the LSB (Least Significant Bit).



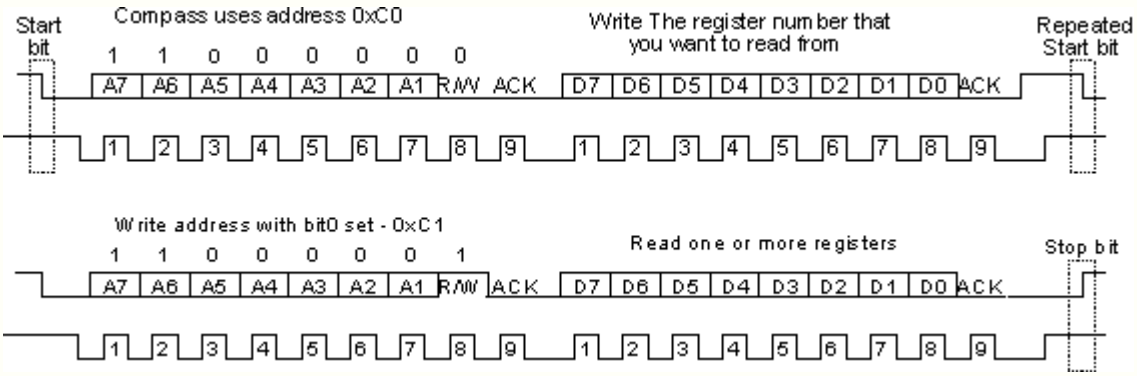
The placement of the 7 bit address in the upper 7 bits of the byte is a source of confusion for the newcomer. It means that to write to address 21, you must actually send out 42 which is 21 moved over by 1 bit. It is probably easier to think of the I2C bus addresses as 8 bit addresses, with even addresses as write only, and the odd addresses as the read address for the same device.

The I2C Software Protocol
The first thing that will happen is that the master will send out a start sequence. This will alert all the slave devices on the bus that a transaction is starting and they should listen in incase it is for them. Next the master will send out the device address. The slave that matches this address will continue with the transaction, any others will ignore the rest of this transaction and wait for the next. Having addressed the slave device the master must now send out the internal location or register number inside the slave that it wishes to write to or read from. This number is obviously dependant on what the slave actually is and how many internal registers it has. Some very simple devices do not have any, but most do. Having sent the I2C address and the internal register address the master can now send the data byte (or bytes, it doesn't have to be just one). The master can continue to send data bytes to the slave and these will normally be placed in the following registers because the slave will automatically increment the internal register address after each byte. When the master has finished writing all data to the slave, it sends a stop sequence which completes the transaction. So to write to a slave device:
1. Send a start sequence
2. Send the I2C address of the slave with the R/W bit low (even address)
3. Send the internal register number you want to write to
4. Send the data byte
5. [Optionally, send any further data bytes]
6. Send the stop sequence.

Reading from the Slave
This is a little more complicated - but not too much more. Before reading data from the slave device, you must tell it which of its internal addresses you want to read. So a read of the slave actually starts off by writing to it. This is the same as when you want to write to it: You send the start sequence, the I2C address of the slave with the R/W bit low (even address) and the internal register number you want to write to. Now you send another start sequence (sometimes called a restart) and the I2C address again - this time with the read bit set. You then read as many data bytes as you wish and terminate the transaction with a stop sequence. So to read the compass bearing as a byte from the CMPS03 module:
1. Send a start sequence
2. Send the I2C address of the slave with the R/W bit low (even address)
3. Send the internal register number you want to read from.
4. Send a start sequence again (repeated start)

2. Send the I2C address of the slave with the R/W bit high (odd address)
6. Read data byte from the slave device. (may be repeated depending on the slave capabilities)
7. Send the stop sequence.

The bit sequence will look like this:



Wait a moment
The ARMexpress does not support slaves that use clock stretching. The result is that erroneous data is read from the slave. Beware! Luckily this function is relatively rare these days.

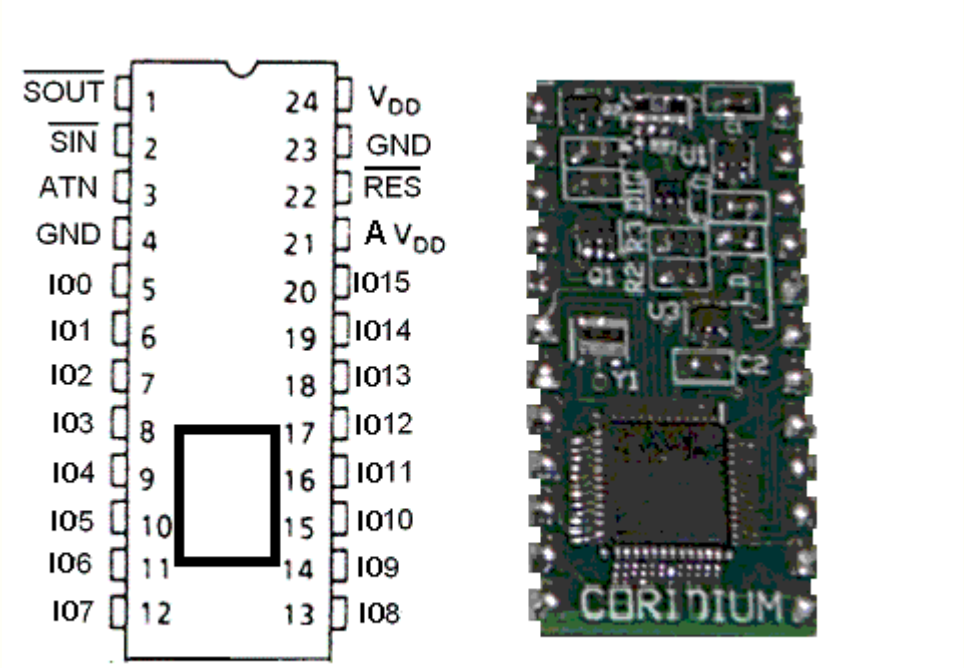
Example Master Code

```
#include <I2C.bas>
 '...

' test the EEPROM 24LC02 on pins 0 == SDA and 1 == SCL
shortMessage(0)= 0 ' address into EEPROM

present = I2COUT (0, 1, 0xA0, 8, shortMessage)
if present = 0 then print "NO i2c device ***"

WAIT(10) ' allow time for data to be written
I2CIN(0, 1, 0xA0, 1,shortMessage, 7, shortResponse)

' now do I2CIN as separate operations

I2COUT (0, 1, 0xA0, 1, shortMessage)   ' send just the address and offset
I2CIN(0, 1, 0xA0, -1,"", 7, shortResponse)
```

Easy isn't it?

The definitive specs on the I2C bus can be found on the Philips website. Its currently **here** but if its moved you'll find it easily be googleing on "i2c bus specification".

ARMexpress LITE Pin Diagram



The ARMexpress LITE is pin compatible with the Parallax BASIC Stamp. BASIC Stamp is a registered trademark of Parallax Inc.

/SOUT	1		Serial Output, RS-232 compatible (active low)
/SIN	2		Serial Input, RS-232 compatible (active low)
ATN	3		connect to DTR with RS-232, when HIGH reset the Node (active high)
/RES	22		TTL level RESET (open collector with 2.7K pull-up) (active low)
IO0 IO1 IO2 IO3 IO4 IO5 IO6 IO7 IO8 IO9 IO10 IO11 IO12 IO13	5 6 7 8 9 10 11 12 13 14 15 16 17 18	 PWM3 PWM1, RXD1 PWM2, TXD1 AD0 AD2 AD5 AD1 AD6 AD7 PWM7	 Input/Outputs -- user controlled 0-3.3V level 4mA drive when configured as Outputs 5V tolerant - use limiting resistor when connecting to a 5V supply EINT2

IO14 IO15	19 20	PWM5 PWM8	EINT0
GND	4,23		Ground (0V)
VDD	24		Power 5-12V input power
Alt-VDD	21		Alternate 5V input power (for Parallax compatibility) DO NOT exceed 5V on this pin connection to pin 24 is preferred this pin is pulled low during download of a C program

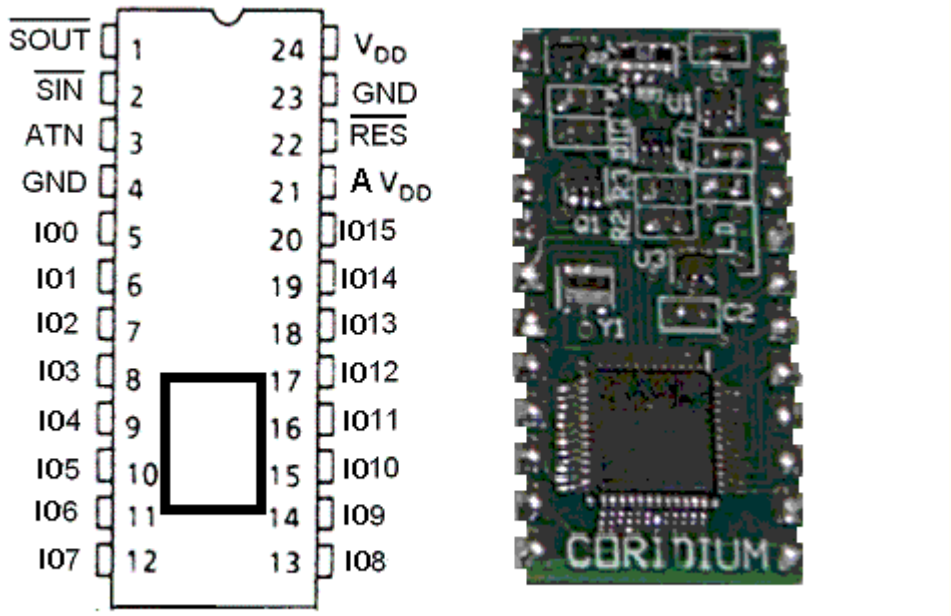
Dual Use AD pins

On reset or power up the AD pins are configured as digital IOs on the ARMexpress LITE. When the BASIC accesses these pins they are changed to analog inputs. After that they will remain analog inputs until the next reset or power up.

PWM pins

All pins can be used for the software PWM function, and 6 pins can be used for the hardware driven HWPWM function (HWPWM channels 4 and 6 are not connected).

ARMexpress Pin Diagram



The ARMexpress is pin compatible with the Parallax BASIC Stamp. BASIC Stamp is a registered trademark of Parallax Inc.

/SOUT	1		Serial Output, RS-232 compatible (active low)
/SIN	2		Serial Input, RS-232 compatible (active low)
ATN	3		connect to DTR with RS-232, when HIGH reset the Node (active high)
/RES	22		TTL level RESET (open collector with 2.7K pull-up) (active low)
IO0 IO1 IO2 IO3 IO4 IO5[1] IO6[1] IO7 IO8 IO9 IO10 IO11 IO12 IO13 IO14 IO15	5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20	 note 1 note 1 EINT2 EINT0	Input/Outputs -- user controlled 0-3.3V level 4mA drive when configured as Outputs 5V tolerant - use limiting resistor when connecting to a 5V supply

GND	4,23		Ground (0V)
VDD	24		Power 5-12V input power
Alt-VDD	21		Alternate 5V input power (for Parallax compatibility) DO NOT exceed 5V on this pin connection to pin 24 is preferred this pin is pulled low during download of a C program

[1]These pins (IO5 an IO6) are open-drain, when configured as outputs can only pull down.

ARMmite Pin Description

24 pins available to the user, 8 of which can be analog inputs

IO(0) IO(1) IO(2) IO(3) IO(4) IO(5) IO(6) IO(7) IO(8) IO(9) IO(10) IO(11) IO(12) IO(13) IO(14) IO(15)	P0(9) P0(8) P0(30) P0(21) P0(20) P0(29) P0(4) P0(5) P0(6) P0(7) P0(13) P0(19) P0(18) P0(17) P0(16) P0(15)	RXD(1) TXD(1) EINT0 EINT2	PWM1 PWM2 PWM3 PWM4 PWM5 PWM6 PWM7 PWM8	Input/Outputs -- user controlled 0-3.3V level 4mA drive when configured as Outputs 5V tolerant - use limiting resistor when connecting to a 5V supply IO(15) connected to LED
AD(0) AD(1) AD(2) AD(3) AD(4) AD(5) AD(6) AD(7)	P0(22) P0(23) P0(24) P0(10) P0(11) P0(12) P0(25) P0(26)	IO(16) IO(17) IO(18) IO(19) IO(20) IO(21) IO(22) IO(23)		10 bit A/D inputs may also be used as digital Input/Outputs IO(16-23) -- user controlled when used as analog lines, voltage levels should not exceed 3.3V

Dual Use AD pins

On reset or power up the AD pins are configured as AD inputs. To change those to digital IOs, the user must individually specify a control direction using INPUT(x), OUTPUT(x), DIR(x), or IO(x) commands. After that they will remain digital IOs until the next reset or power up.

PWM pins

All pins can be used for the software PWM function, and 8 pins can be used for the hardware driven HWPWM function.

Battery Real Time Clock

The ARMmite board is designed to accept a Panasonic ML2020/H1C rechargeable Lithium battery at

position BT1. This battery powers the real time clock of the LPC2103. The contents of RAM is not kept alive while running on battery, and the CPU restarts the user program in Flash when power is restored. This battery is designed to maintain power for a few days without power, and will recharge fully in about 1 day.

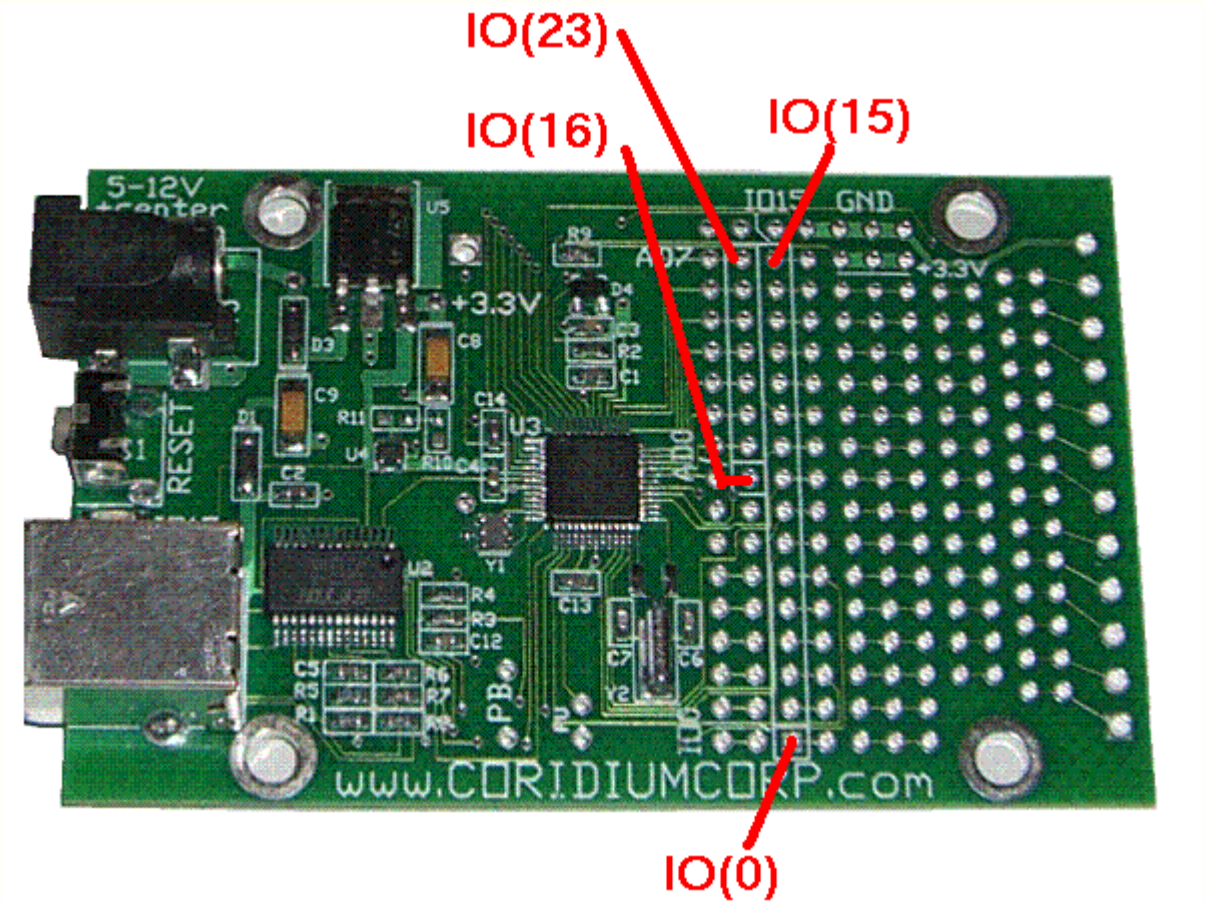
Power connection

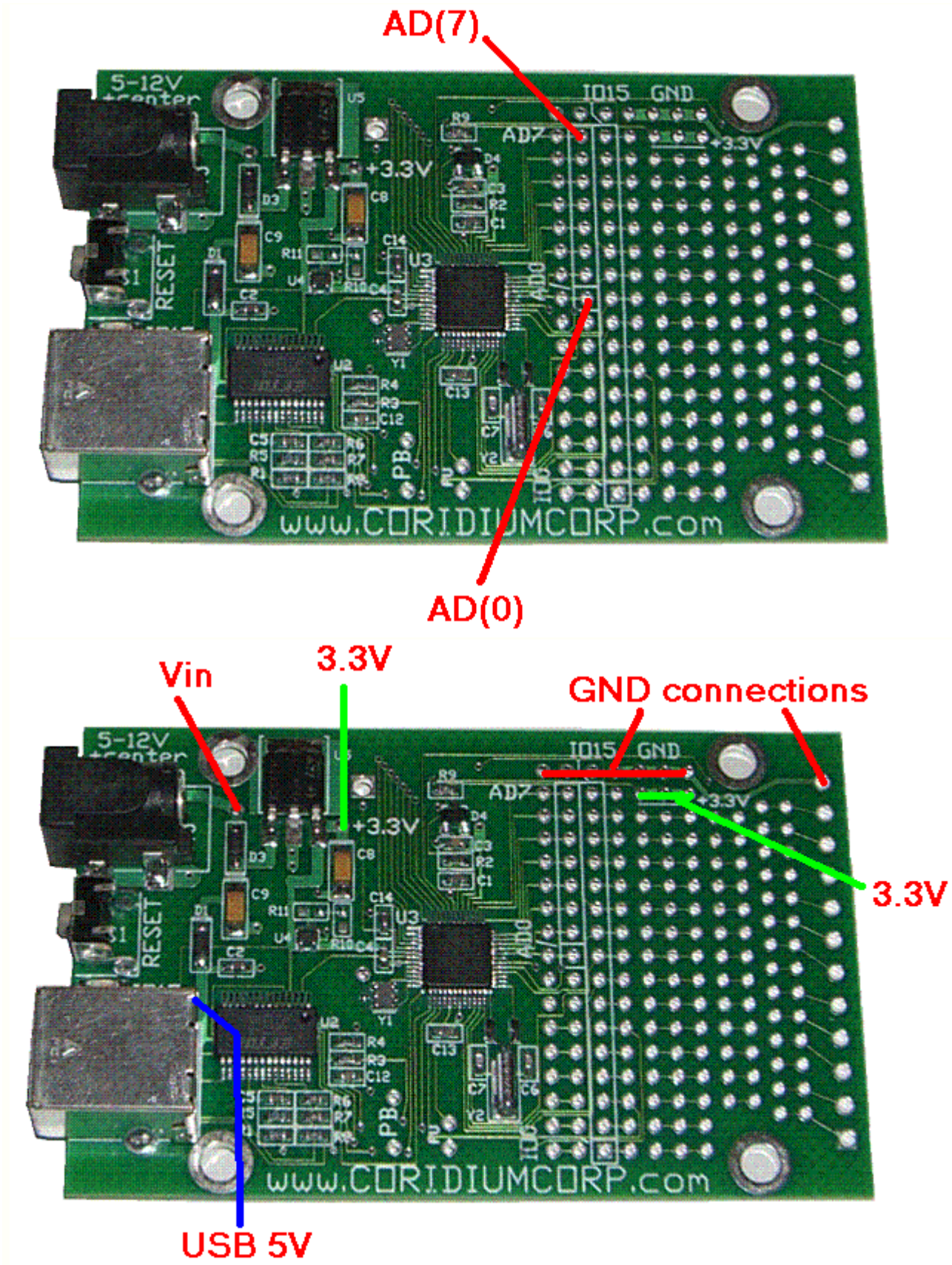
Power when not being supplied by a USB connection uses a 2.1mm barrel connector (Cui PJ-002A). Diodes allow both USB and separate power to be connected simultaneously. If you are using an unregulated wall transformer, you must check the open circuit voltage and it MUST be less than 12V.

Pin spacing

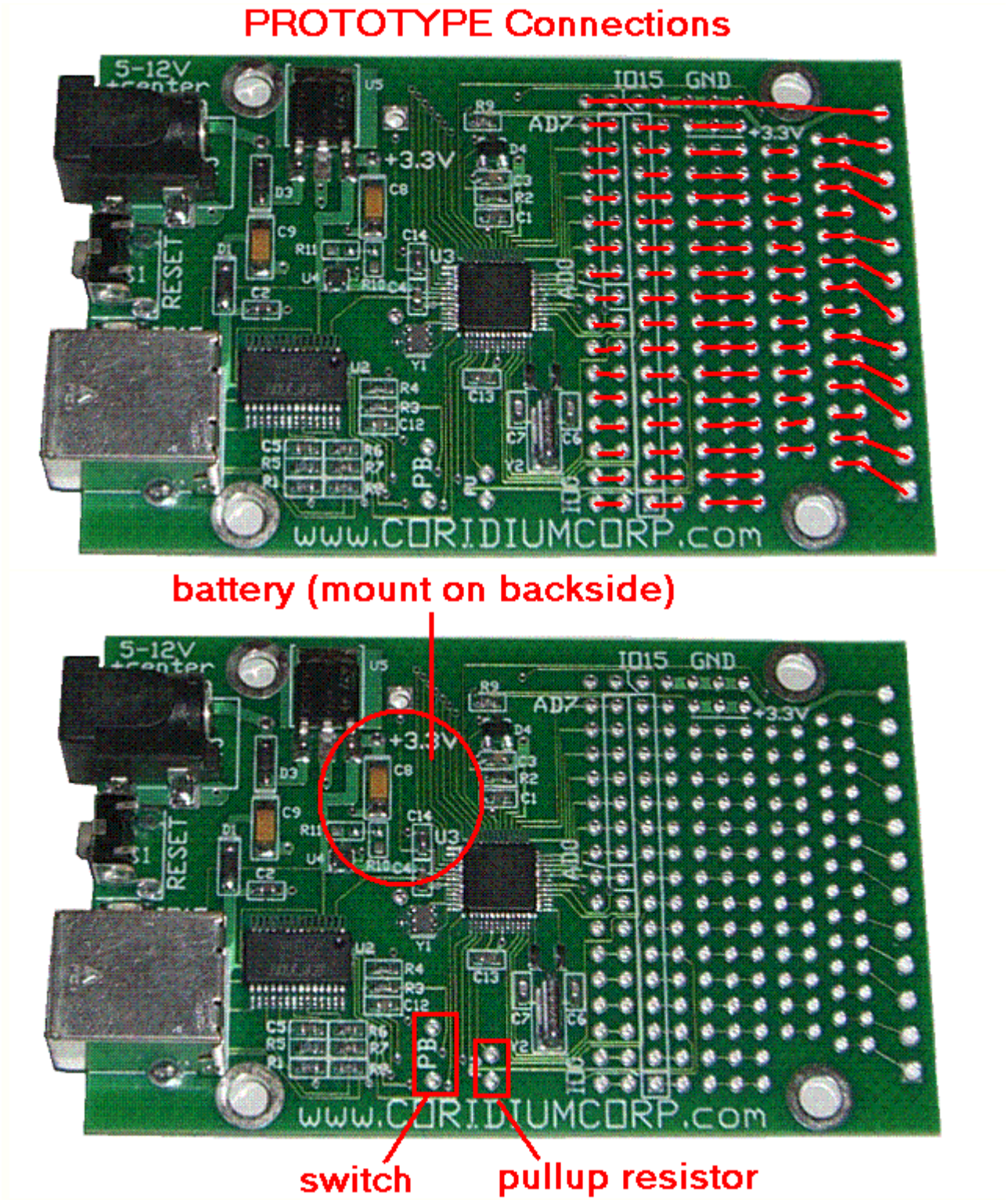
The spacing in the prototype area is 0.1" and the terminal strip row on the right side is designed for 3.5mm terminal strips.

REV 3

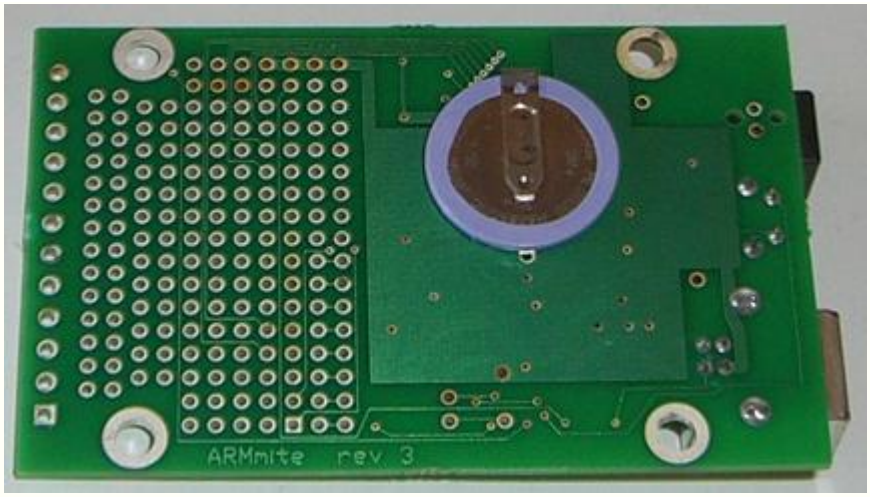




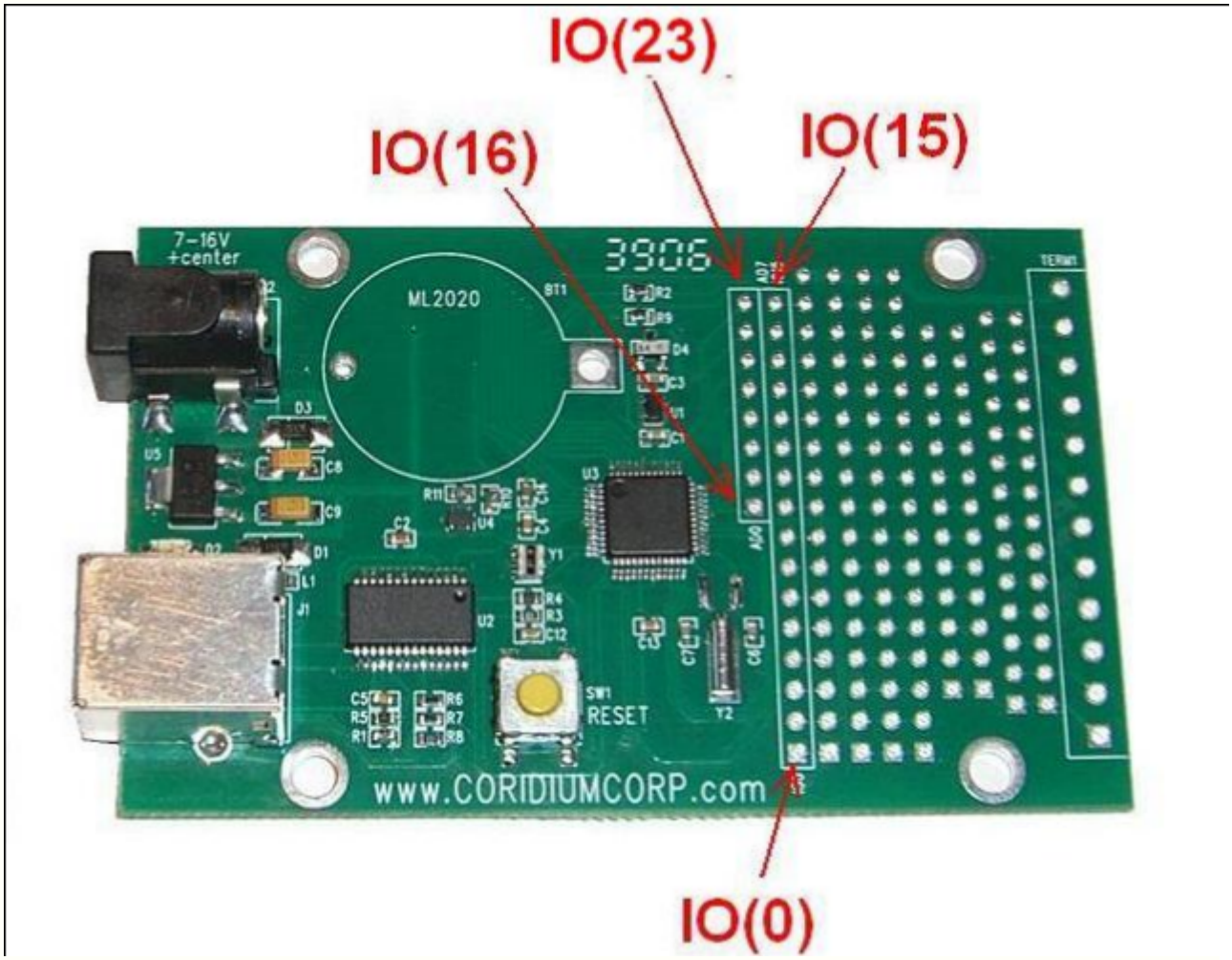
When USB power is not used, a 5-12V supply is required. If 5V is required for some portion of your circuit, it is suggested that a regulated 5V supply be used for input power. These are available from **SparkFun**.

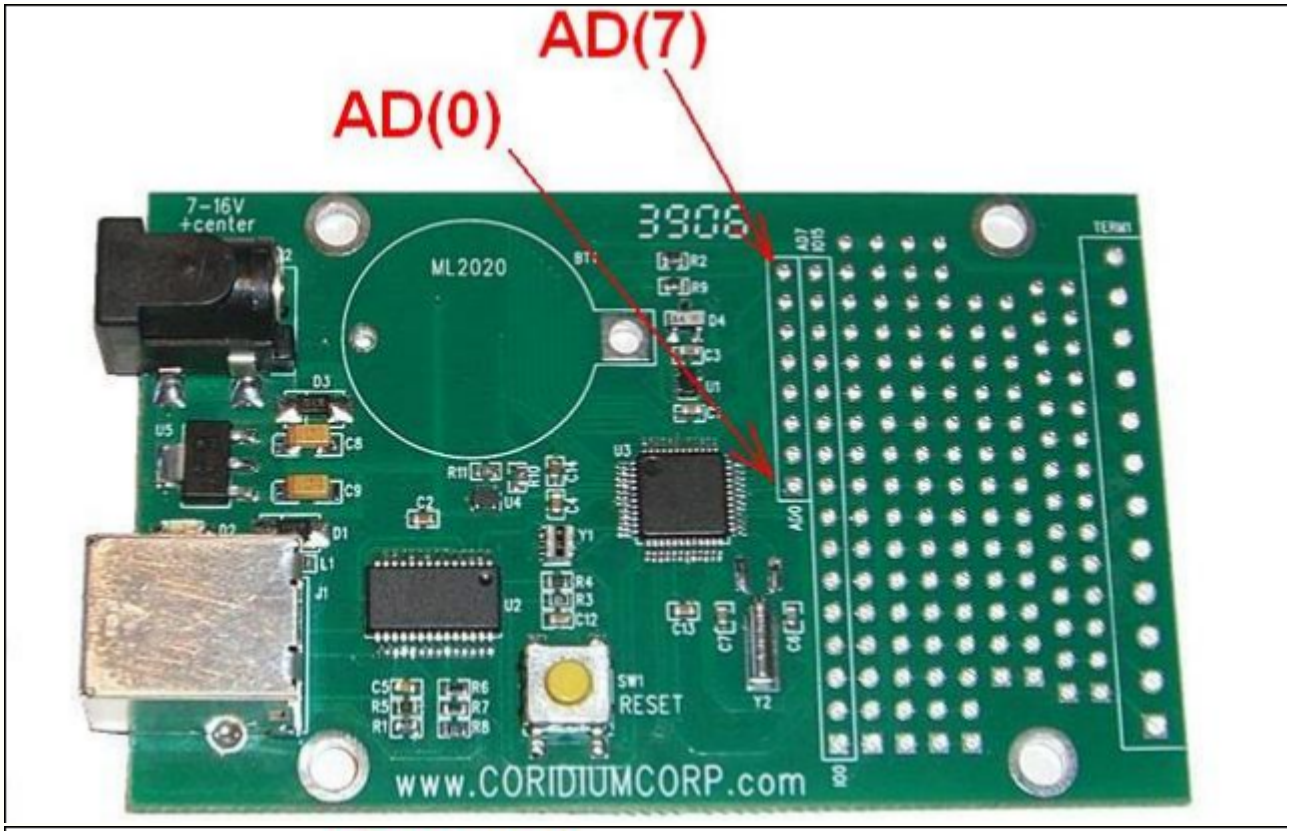


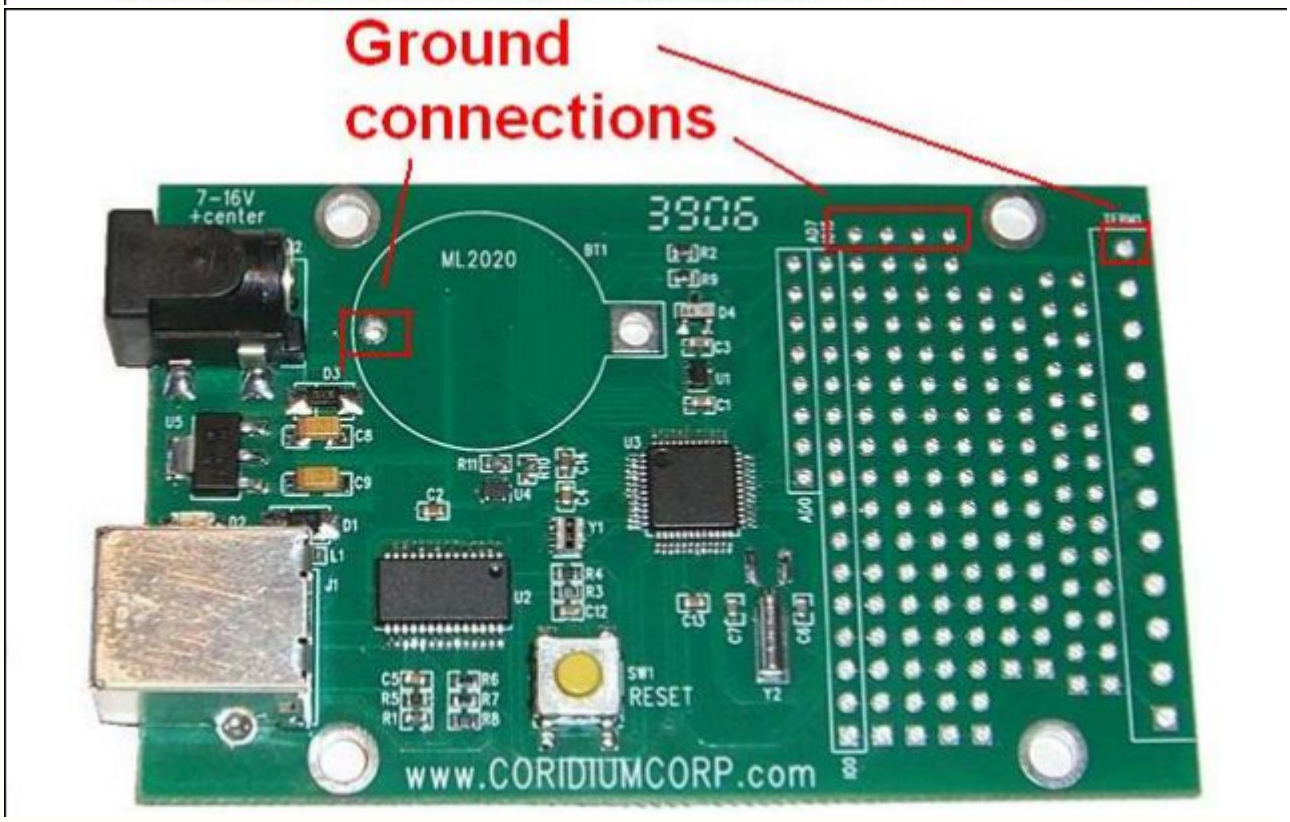
A push button switch and pull-up resistor can also be mounted (connected to IO(2)). The optional battery for the real time clock (Panasonic ML2020) can be mounted on the back of the PCB. The VL2020/HFN will also work, though it is more expensive and has less power.

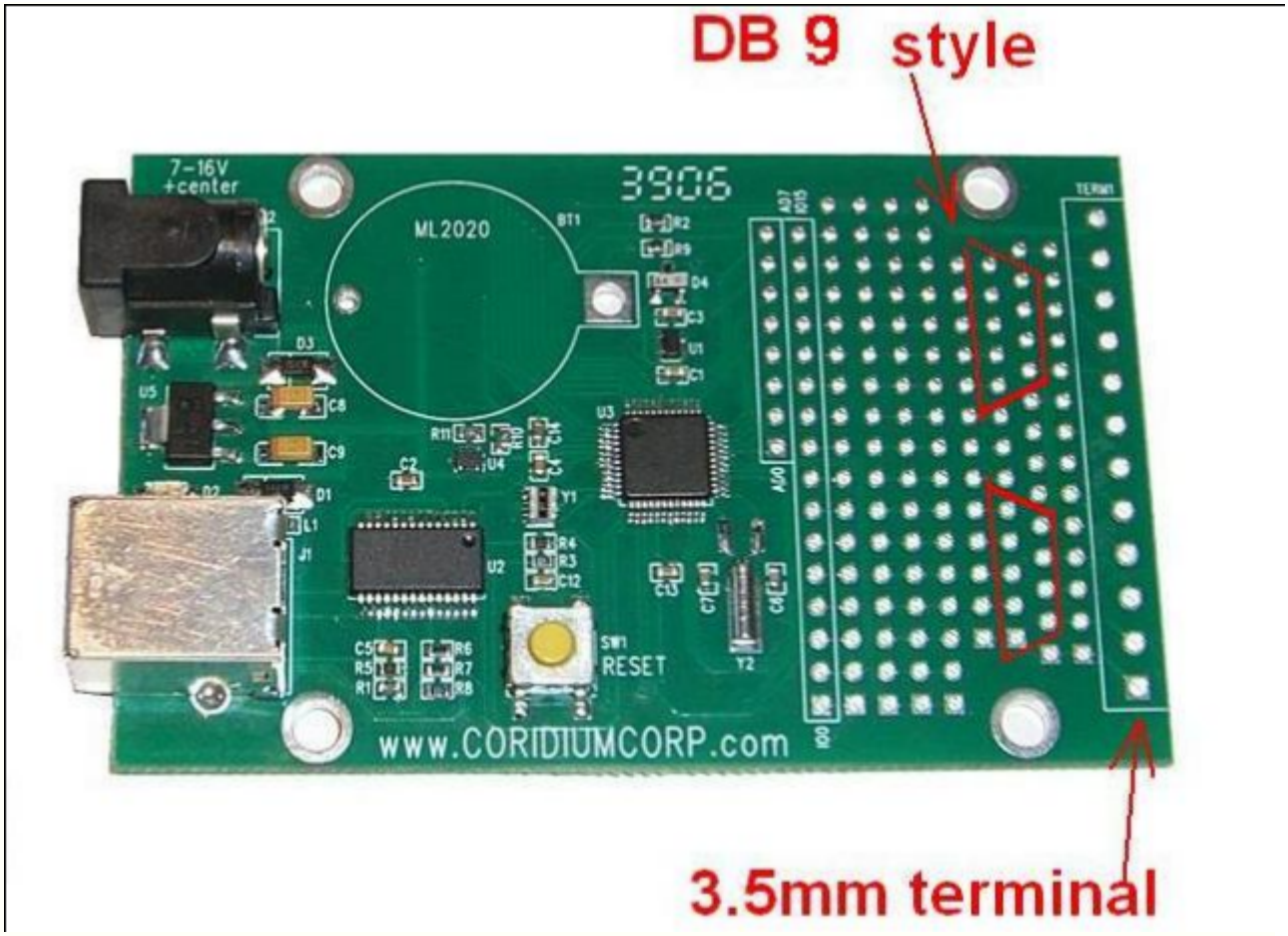

ARMmite rev 3

REV 2

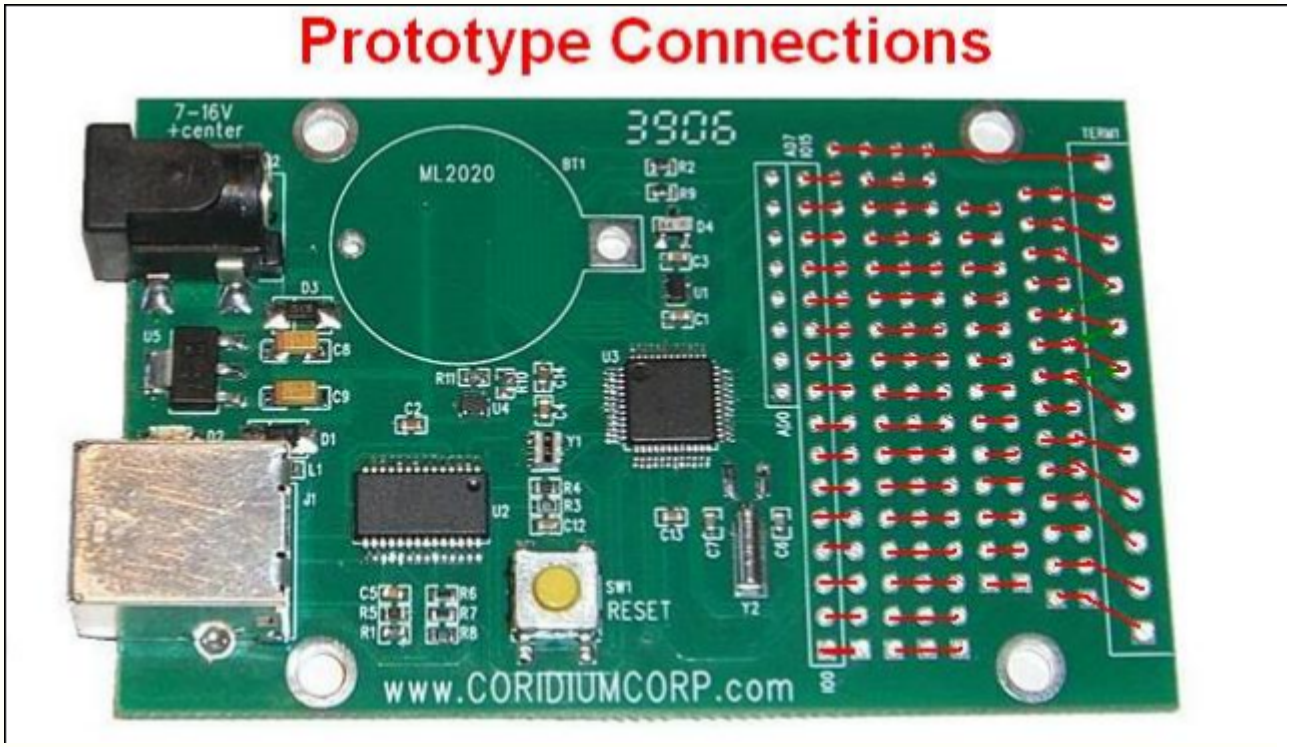

IO(23)
IO(16)
IO(15)
7-16V
+center
ML2020
3906
RESET
www.CORIDIUMCORP.com
IO(0)


AD(7)
AD(0)
7-16V
+center
ML2020
3906
RESET
www.CORIDIUMCORP.com


Ground
connections
7-16V
+center
ML2020
3906
RESET
www.CORIDIUMCORP.com



suggested terminal strip On Shore Tech ED550/12DS or equivalent 3.5mm pitch connector (available at Digikey)



ARMweb Pin Description

Rev 4 and 5

32 pins available to the user, 5 of which can be analog inputs, and one dedicated analog input

With Rev 4 the pin numbering for the ARMweb will reflect the assignment native to the LPC2138. The revision of the board is etched on the backside of the board.

P0(7) P0(8) P0(9) P0(10) P0(11)** P0(12) P0(13) P0(15) P0(17) P0(18) P0(19) P0(20) P0(21) P0(22) P0(23) P0(25) -- P0(27) P0(28) P0(29) P0(30) P0(31)++	IO(7) IO(8) IO(9) IO(10) IO(11)** IO(12) IO(13) IO(15) IO(17) IO(18) IO(19) IO(20) IO(21) IO(22) IO(23) IO(25) -- IO(27) IO(28) IO(29) IO(30) IO(31)++	TXD(1) RXD(1) AD(4), DAout AD(5) AD(0) AD(1) AD(2) AD(3)	IO(7) is connected to LED and PUSHBTTON as an input the push button is 0 when pressed **IO(11) is open drain when an output (i.e. can not pull up) IO(15) also controls LED (when low, the LED will be lit) Input/Outputs -- user controlled 0-3.3V level, 4mA drive when configured as Outputs 5V tolerant - use limiting resistor when connecting to a 5V supply AD(5) is always an analog input, IO(26) does not exist 10 bit A/D inputs when used as analog lines, voltage levels should not exceed 3.3V ++IO(31) is always an output
P1(16) P1(17) P1(18) P1(19) P1(20) P1(21) P1(22) P1(23) P1(24) P1(25)	IO(48) IO(49) IO(50) IO(51) IO(52) IO(53) IO(54) IO(55) IO(56) IO(57)		Input/Outputs -- user controlled 0-3.3V level, 5 volt tolerant, 4ma drive when outputs

-

Port pins can be controlled with the **P0..P4 keywords**. Port 0 pins can be accessed with the original **IN, OUT... keywords** . More details on the FIOs can be found in the NXP User Manual.

Dual Use AD pins

On reset or power up the AD pins are configured as IO inputs. To change those to analog IOs, the user must individually read them as AD(x) commands. After that they will remain analog inputs until the next reset or power up.

PWM pins -- not yet implemented

All pins can be used for the software PWM function, and <TBD> pins can be used for the hardware driven HWPWM function.

Battery Real Time Clock

The ARMweb board is designed to accept a Panasonic ML2020/H1C rechargeable Lithium battery at position BT1. This battery powers the real time clock of the LPC2138. The contents of RAM is not kept alive while running on battery, and the CPU restarts the user program in Flash when power is restored. This battery is designed to maintain power for a few days without power, and will recharge fully in about 1 day.

LED

On the beta units, this is connected to IO(16) not 15. On later units while the LED is connected to IO(7), the silkscreen shows it as connected to IO(15), and the example programs for the ARMmite and ARMexpress use IO(15). So firmware on the board allows IO(15) to also control the LED.

U6 has duplicate connections for IO(17)-IO(20). U6 is designed to accept a ULN2803.

The bottom proto area connects neighboring pairs of pins. In the top proto area near C23, neighboring triplets of pins are connected horizontally.

In addition the ARMweb can be ordered in larger quantities with a switching power supply, which replaces U4, C1 and C9 with U1, D2, L4, C1 and C9

Pin spacing

The spacing in the prototype area is 0.1" and the terminal strip row on the right side is designed for 3.5mm terminal strips.

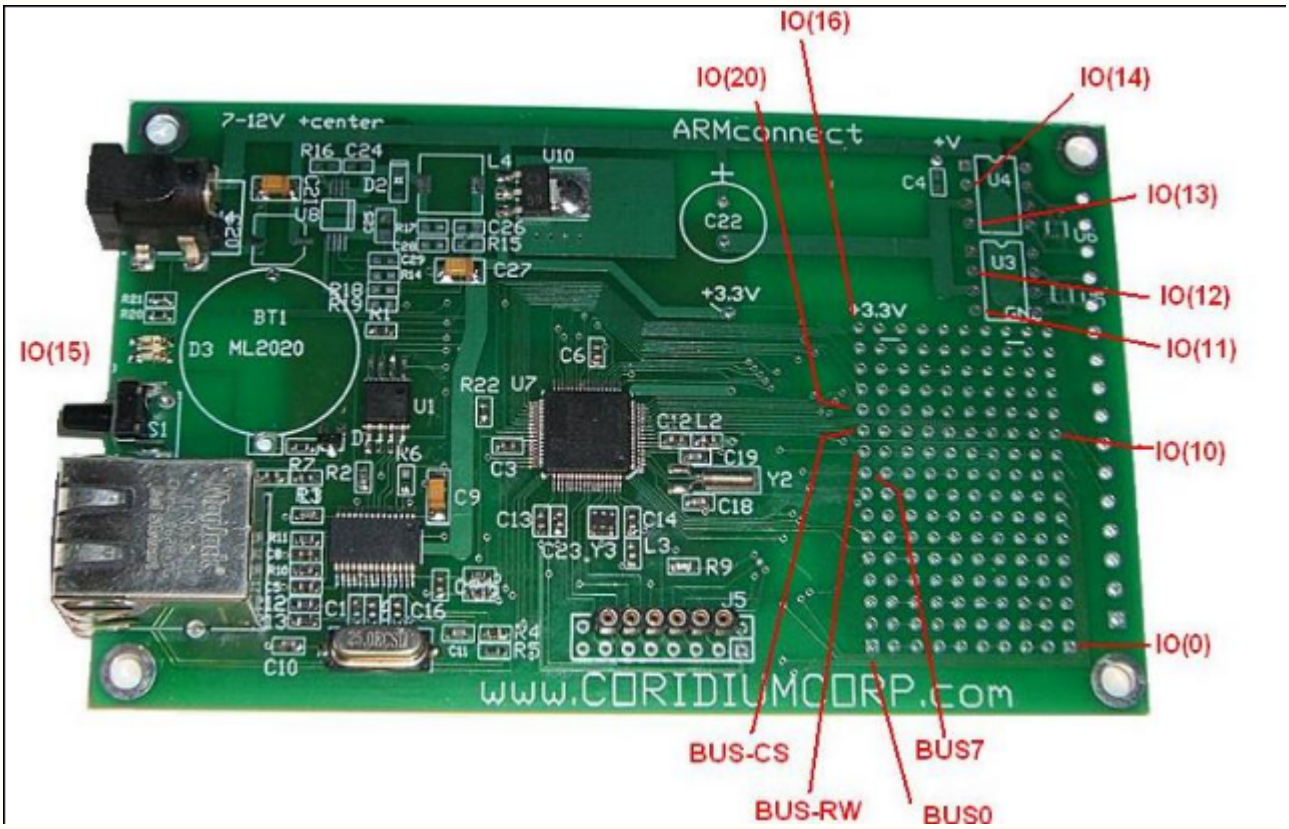
Rev 2,3

31 pins available to the user, 6 of which can be analog inputs

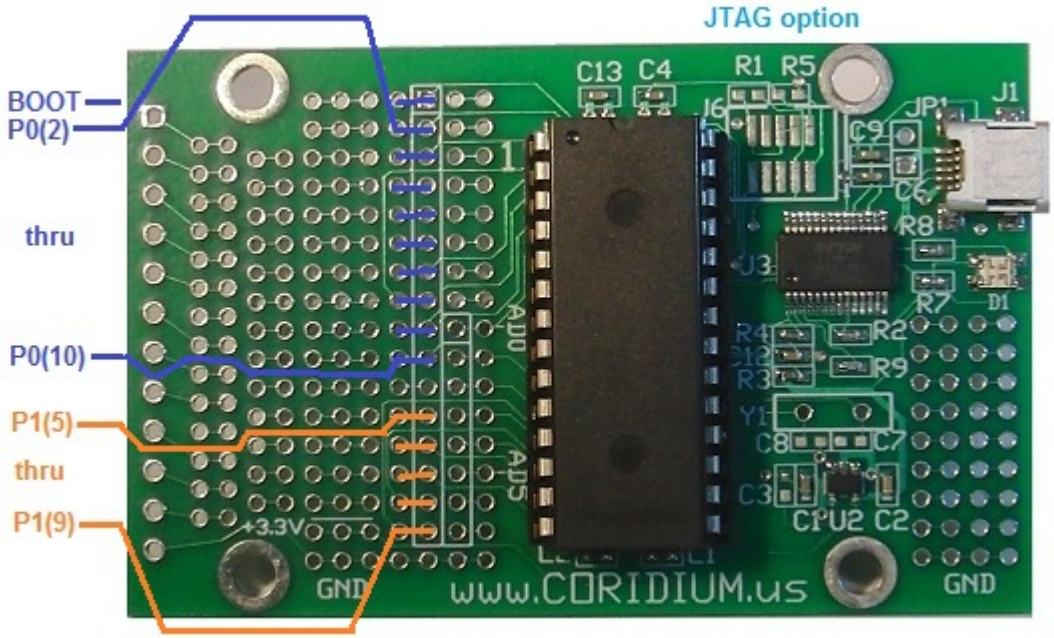
The revision of the board is etched on the backside of the board.

IO0 IO1 IO2 IO3 IO4 --- IO6 IO7 IO8 IO9 IO10 IO11 IO12 IO13 IO14++	AD0 AD1 AD2 AD3 AD4 AD5		Input/Outputs -- user controlled 0-3.3V level 4mA drive when configured as Outputs 5V tolerant - use limiting resistor when connecting to a 5V supply 10 bit A/D inputs when used as analog lines, voltage levels should not exceed 3.3V ++IO14 is always an output AD5 is always an analog input, IO5 does not exist
IO15			Input/Outputs -- user controlled controls LED (when low, the LED will be lit) as an input also connects to the push button (0 when pressed)
IO16 IO17 IO18** IO19 IO20			Input/Outputs -- user controlled 0-3.3V level, 5 volt tolerant, 4mA drive when output **IO18 is open drain when an output (i.e. can not pull up)
BUS0 BUS1 BUS2			

BUS3 BUS4 BUS5 BUS6 BUS7 BUS-RW BUS-CS			Input/Outputs -- user controlled 0-3.3V level, 5 volt tolerant, 4ma drive when outputs this functions as a byte-wide bus with control of RW and CS



BASICboard Pin Diagram



The BASICchip is a complete System on a Chip, all that is required is 1.8 through 3.3V power and GND. Then just wire the available IOs into your application. No extra crystals, external memories, or second supplies required.

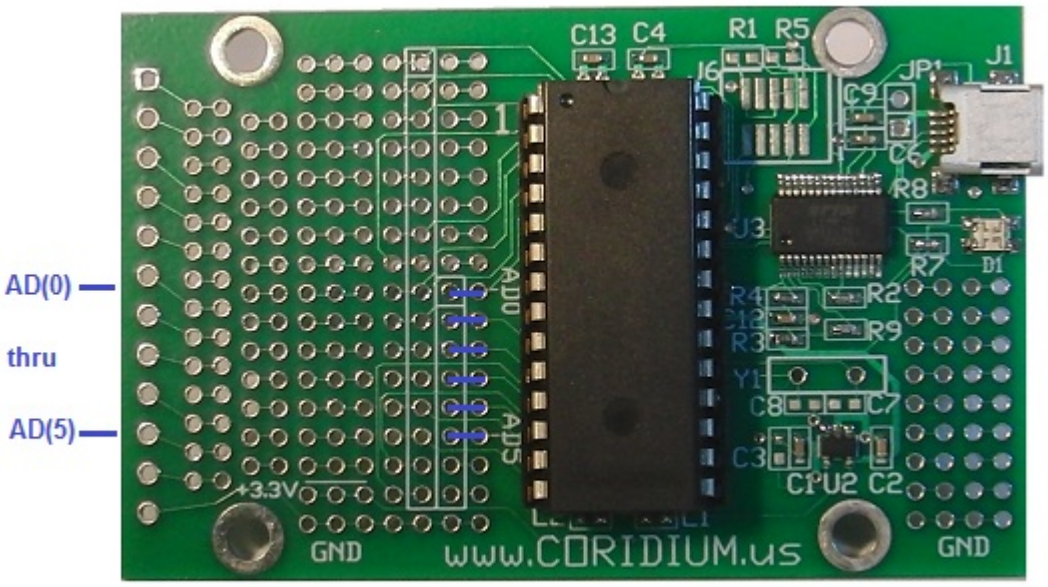
BASIC	function	pin #	alt	notes
IO(39)	TXD(0)	16	P1(7)	Serial Output, TTL compatible (active high) -- debug connection
IO(38)	RXD(0)	15	P1(6)	Serial Input, TTL compatible (active high) -- debug connection
	/RES	23	P0(0)	RESET (internal pull-up) (active low)
IO(1)	BOOT	24	P0(1)	when LOW during reset, ISP is started which disables BASIC (connects to LED and resistors on board)
IO(4) IO(5)	SCL SDA	27 5		open drain outputs, can only pull down, require a pull-up resistor to drive high
IO(2) IO(3) IO(6) IO(7) IO(8) IO(9) IO(10) IO(11)	P0(2) P0(3) P0(6) P0(7) P0(8) P0(9) P0(10) P0(11)	25 26 6 28 1 2 3 4	AD(0)	Input/Outputs -- user controlled - 0-3.3V level 4mA drive when configured as Outputs P0.7 has a 20 mA driver 5V tolerant - use limiting resistor when connecting to a 5V supply
IO(32)	P1(0)	9	AD(1)	Input/Outputs -- user controlled

IO(33) IO(34) IO(35) IO(36) IO(37) IO(40) IO(41)	P1(1) P1(2) P1(3) P1(4) P1(5) P1(8) P1(9)	10 11 12 13 14 17 18	AD(2) AD(3) AD(4) AD(5)	0-3.3V level 4mA drive when configured as Outputs 5V tolerant - use limiting resistor when connecting to a 5V supply
	XTAL	19 20		optional crystal connection -- do not exceed 1.8V
	VDD	21		Power 2.5-3.3V input power -- do not exceed 3.3V
	GND	22		Ground (0V)
	AVDD	7		Analog power, must be equal to or less than VDD
	AGND	8		Analog Ground (0V)

[1]These pins P0(4) and P0(5) are open-drain, when configured as outputs they can only pull down.

Port P1(x) pins can be accessed using the P1(x) keyword. They can also be accessed using IO, IN, OUT, and DIR with indexes 32-41.

Analog Inputs



Power Connections

Optional Crystal

The LPC1114 has an internal 12 MHz oscillator that is trimmed to 1% accuracy. This is good enough for most operations, including serial communication. If more accuracy is desired, then add an optional 12 MHz crystal at Y1 and C7 and C8 with 18pF capacitors.

JP1 option

JP1 in the upper right can have a 2 pin header installed by the user, When this jumper is installed (using a 2 pin shorting block), the control lines from the USB are disabled, so that the board can be plugged into a PC and no matter what state the PC is in, as long as the power is on the BASIC program will start up and run on the ARM.

DIN rail Pin Description

USB connection shown. Details on the enclosure at **OKW enclosures** .

The Ethernet version is software compatible with the **ARMweb, refer to those pages** for more information.

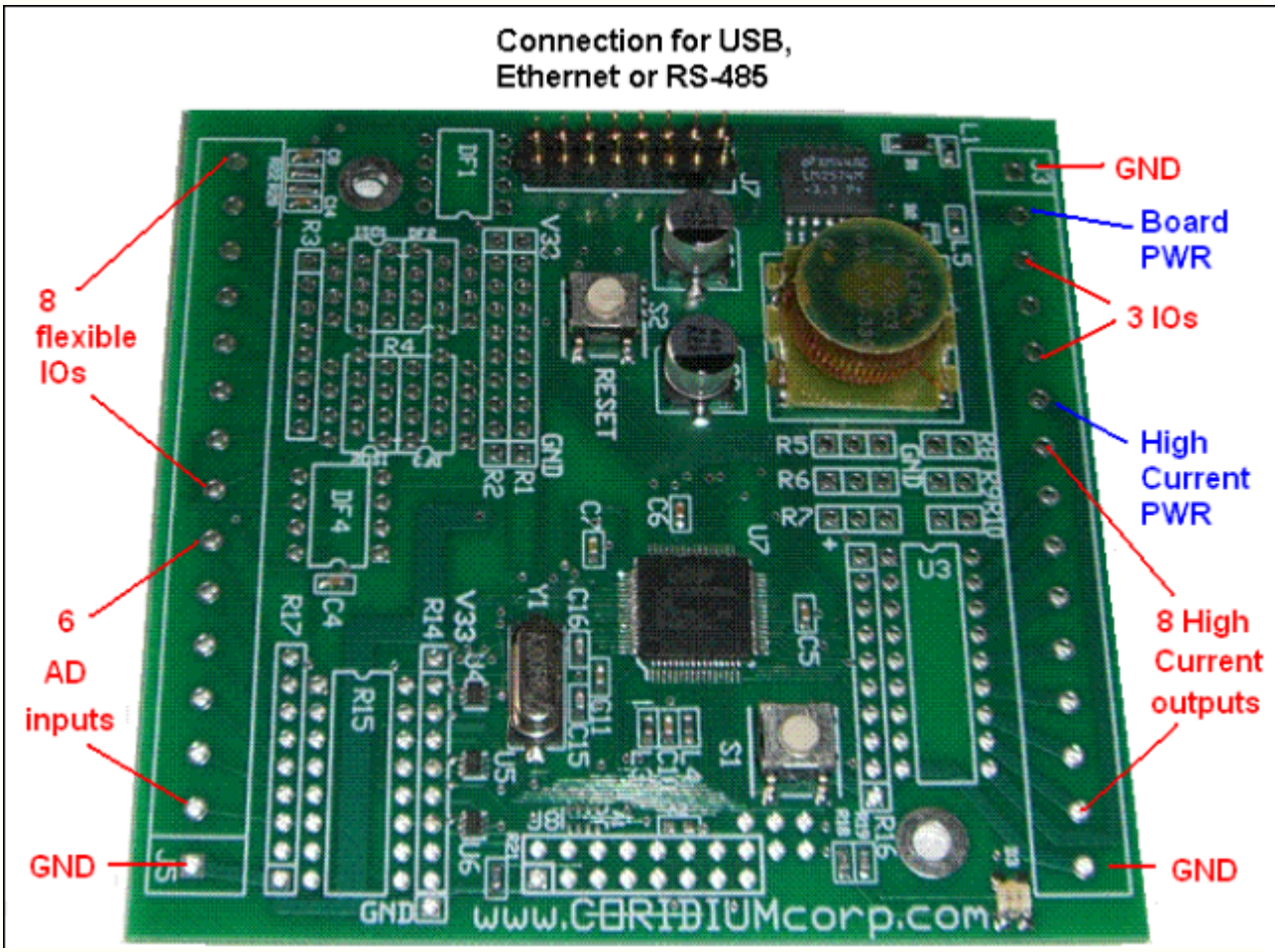
The USB version uses the standalone ARMbasic compiler on the PC.

Rev 1

25 pins available to the user, 6 of which can be analog inputs, 8 high current drivers, 3 digital IOs, and 8 flexible IOs

The LPC2138 is used with 512K Flash and 32K of SRAM.

Optional connections to USB, 10Mb Ethernet, or RS-485 (with optional isolation)



picture shown without screw terminals for clarity

Power Inputs

Board 7-40V DC. This voltage is reduced with a switching regulator for the 3.3V internal board supply.

High Current Driver (**ULN2803**) 5-50V. This can be a separate supply from the Board input power, or can be the same supply. It is a required connection for relay drivers to provide a path for current when the relay coil is turned off, it does not have to be the power supply for the board in this case, but it can be.

For volume customers the power supply can be stuffed to accept a regulated 3.3V supply directly, this is done by omitting the switching power supply and adding an appropriate ferrite bead at L5.

Schematic

The schematic is too large to include on this page, but is downloaded into the /Program files/Coridium/Schematic directory. **Is also available here.**.

Enclosure

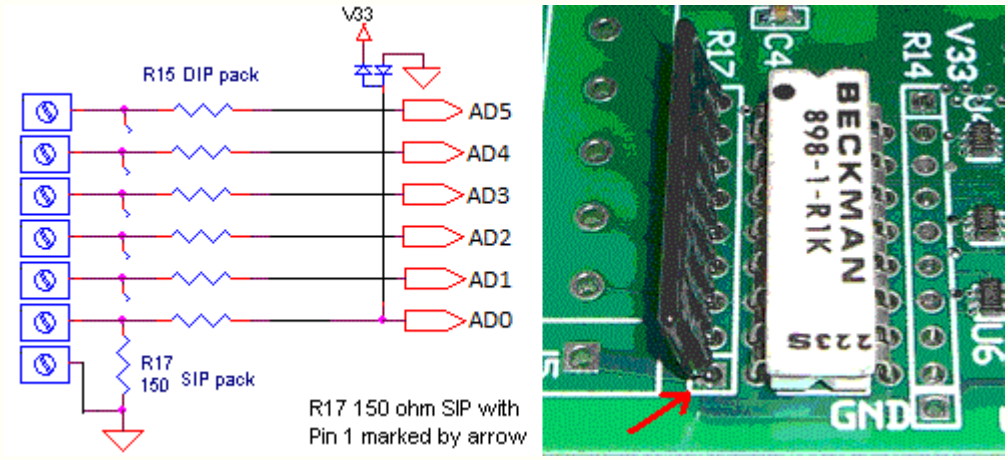
OKW B6704100 The kits include custom cutouts for either Ethernet or USB connections. Mechanical drawing for the **enclosure is here** ,

All the following options can be configured by the user, by optionally stuffing the through-hole components in the DIN rail kit. Coridium will configure boards when 10 or more are ordered.

6 AD pins

These may configured for 4-20 mA sensors, with resistor dividers, or as digital inputs. These inputs have diode clamps to 3.3V and GND.

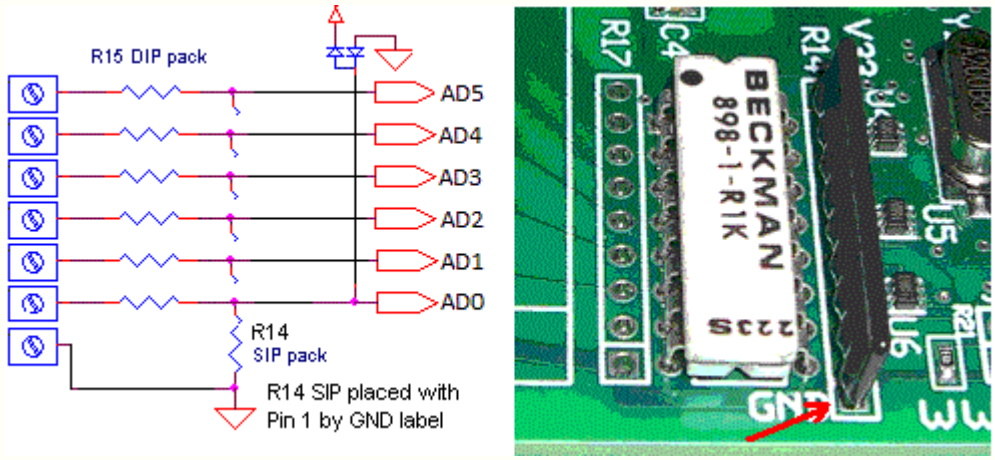
4-20mA sensor --



load 150 ohm SIP into R17

suggested components
  Bourns 4600X Bussed SIP resistor
  Bourns 4100R Isolated DIP resistor

A/D resistor divider --

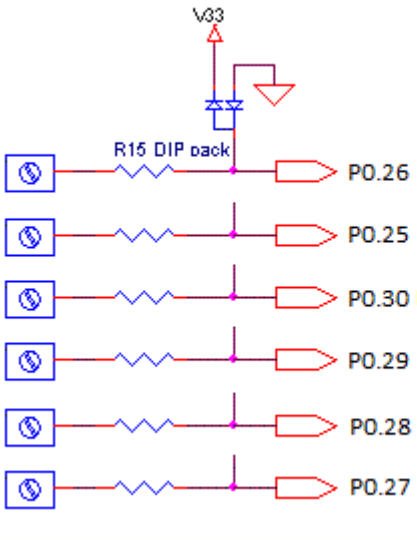


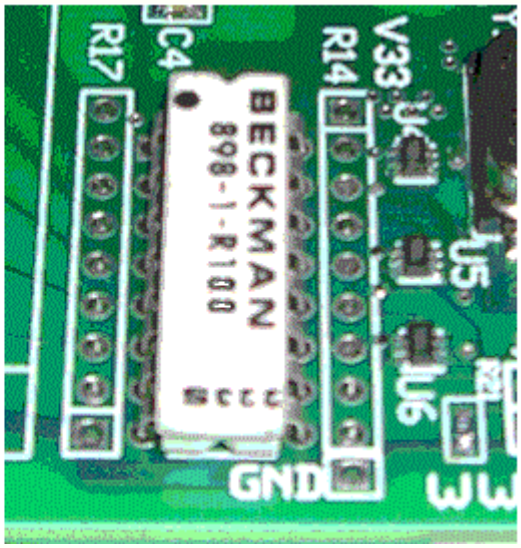
load R15 DIP resistor and R14 SIP with appropriate values

AD = Vin * R14/(R14+R15)

Source impedance to AD should be less than 10K.

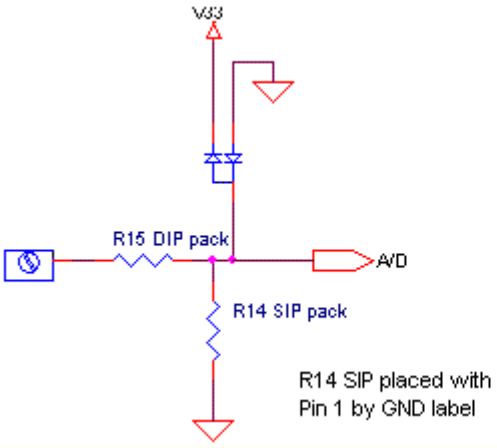
digital IO --

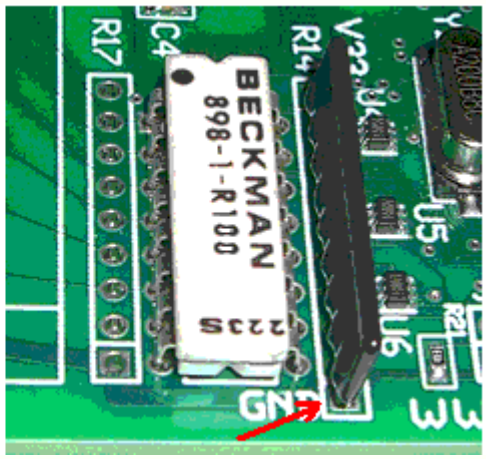




load R15 with 100 or 1K

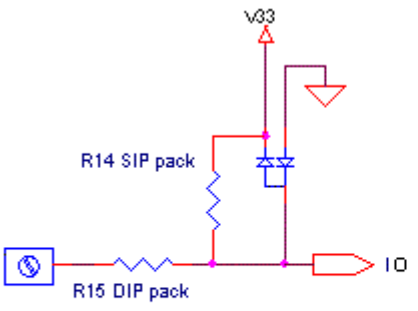
digital IO (pull-down)--





load R15 with 100, R14 with 10K

digital IO (pull-up) --



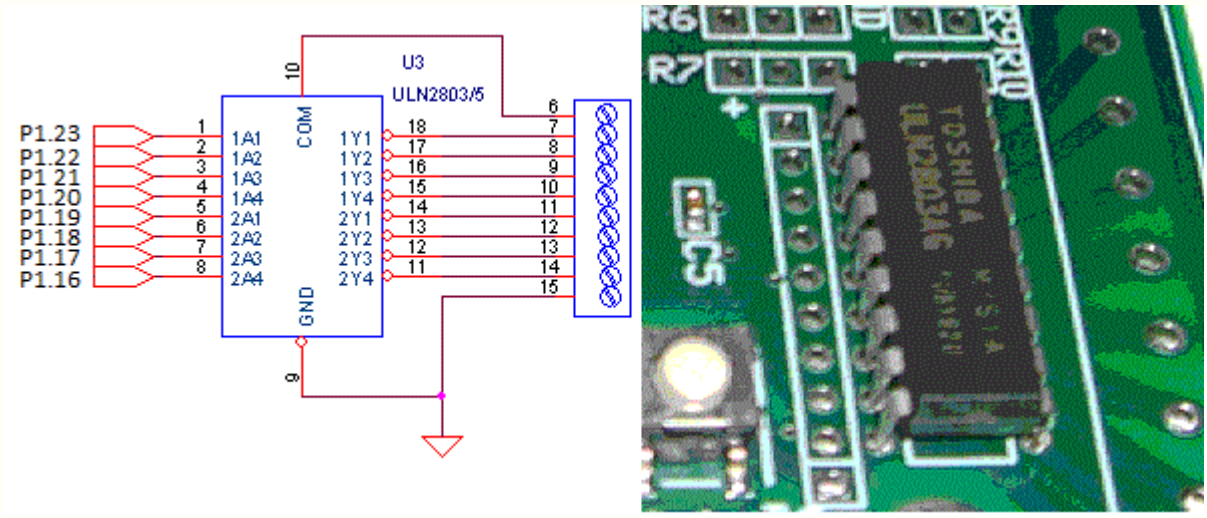
R14 placed with Pin 1 next to
V33 label

load R15 with 100, R14 with 10K

High Current Drivers

These may use a high sink current driver, or configured as digital IOs with optional pull-ups or pull-downs. The ULN2803 is an array of Darlington transistors that either pull a line low (when the corresponding IO is high) or does not conduct at all. A typical application is to use these lines to drive a relay, with the opposite end of the relay tied to a power supply.

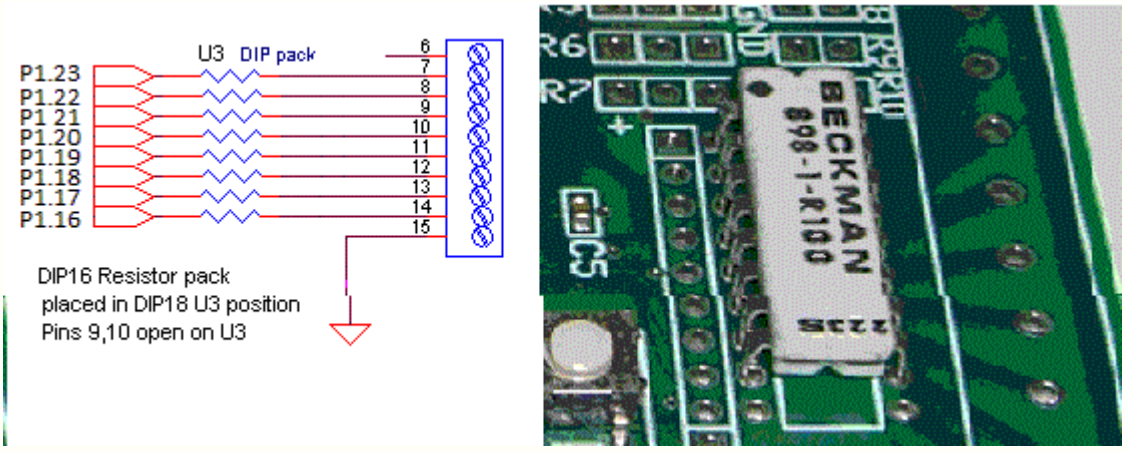
High Current drive --



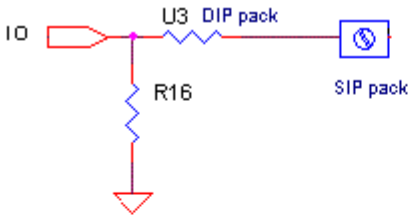
This driver can sink a surge current of 500mA up to 50V, this driver is a **ULN2803** .

suggested components
  TI ULN2803AN
  Toshiba ULN2803APG
  STmicro ULN2803A

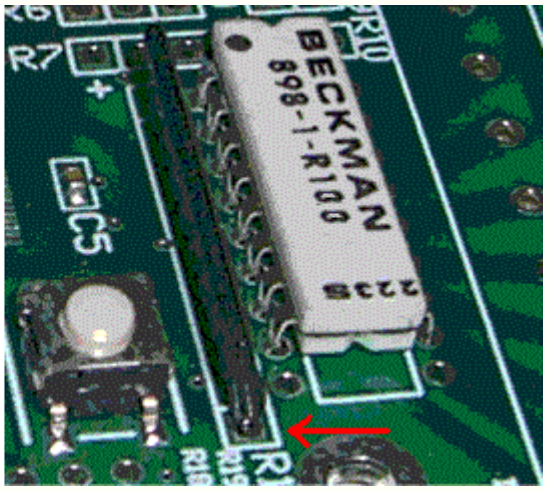
digital IO --



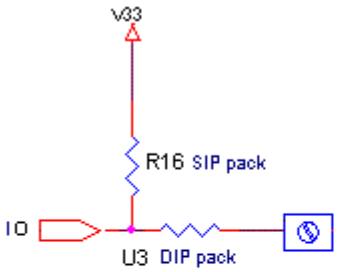
digital IO (pull-down) --



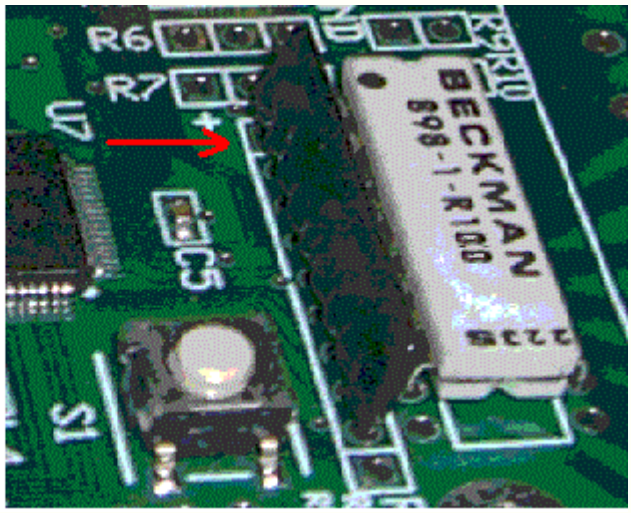
Pin 1 of SIP resistor loaded
near R16 label (arrow)



digital IO (pull-up) --



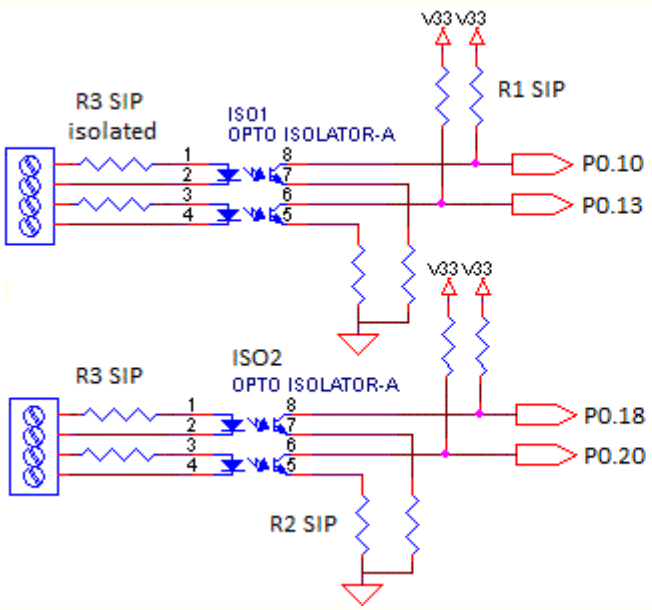
Pin 1 of SIP pullup loaded
near + sign (arrow)

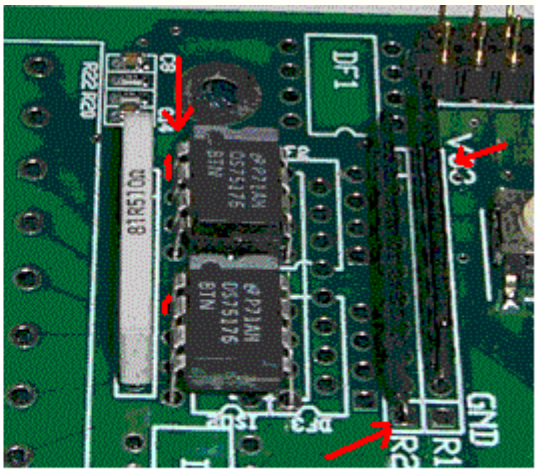


Flexible IOs

These may be configured as 8 digital IOs (with and without pull-up/pull-down), opto-isolated inputs or outputs, or differential inputs or outputs. They are arranged in 2 groups of 4 so that there can be 2 opto-isolated input and 2 opto-isolated outputs.

opto-isolated input --

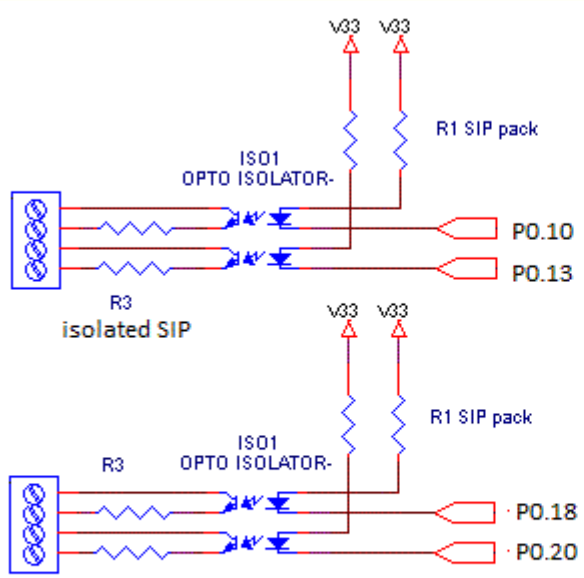


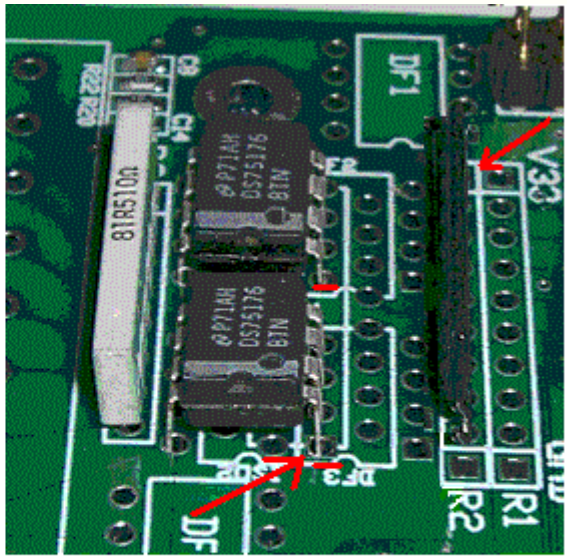


Note Pin 1 orientation of opto-isolators
Pin 1 of R2 near GND, Pin 1 of R1 near V33

suggested components
Liteon LTV-827
Fairchild MCT9001
Toshiba TLP621-2

opto-isolated output --



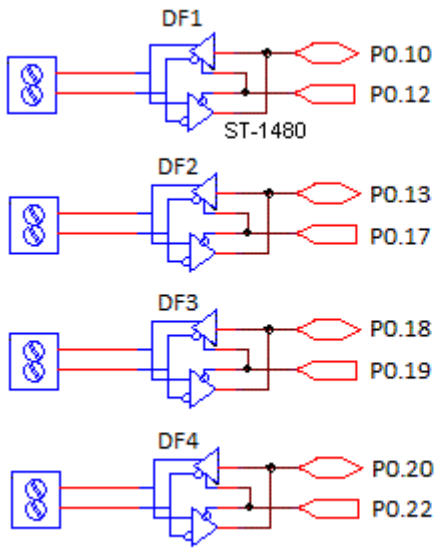


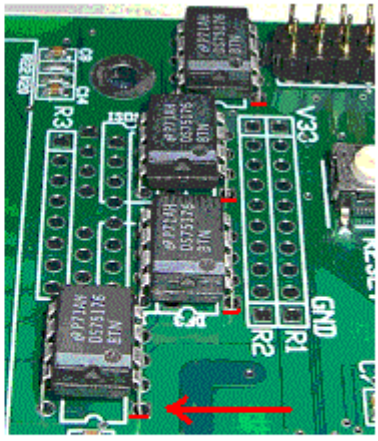
Note Pin 1 orientation of opto-isolators

Pin 1 of R2 near V33 marking

same components as above, rotated 180 degrees

bi-directional RS-422 driver --

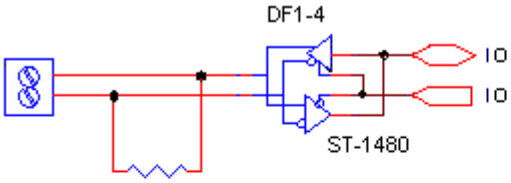




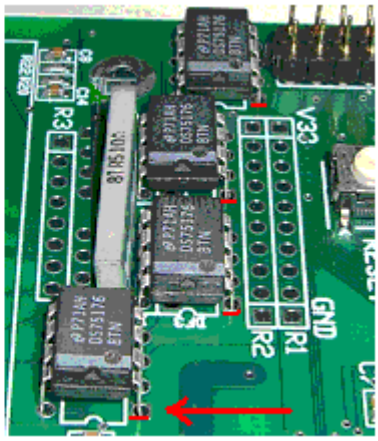
Note Pin 1 orientations

suggested components
National DS75176BN
TI SN75176AP

bi-directional RS-422 driver with termination --

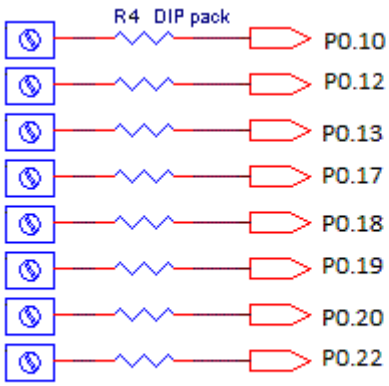


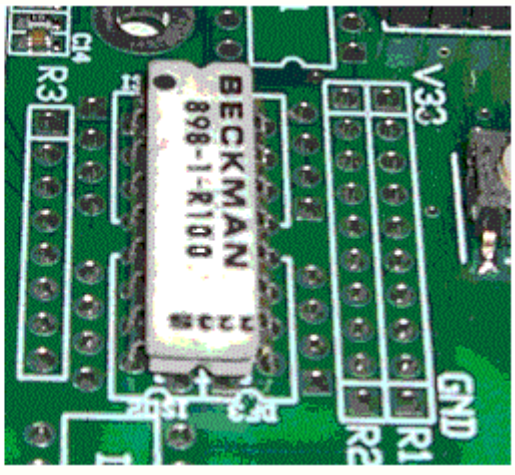
Note Pin 1 orientations



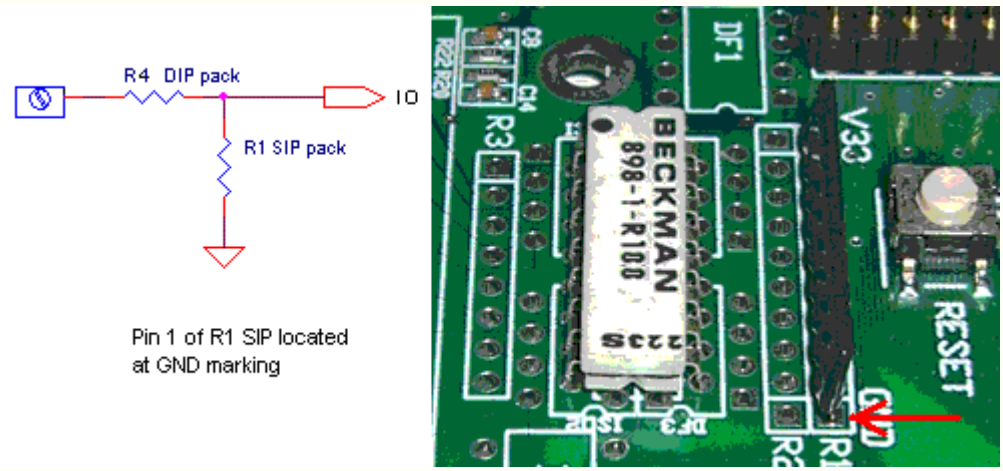
suggested components
Bourns 4600 Isolated SIP resistor

digital IO --

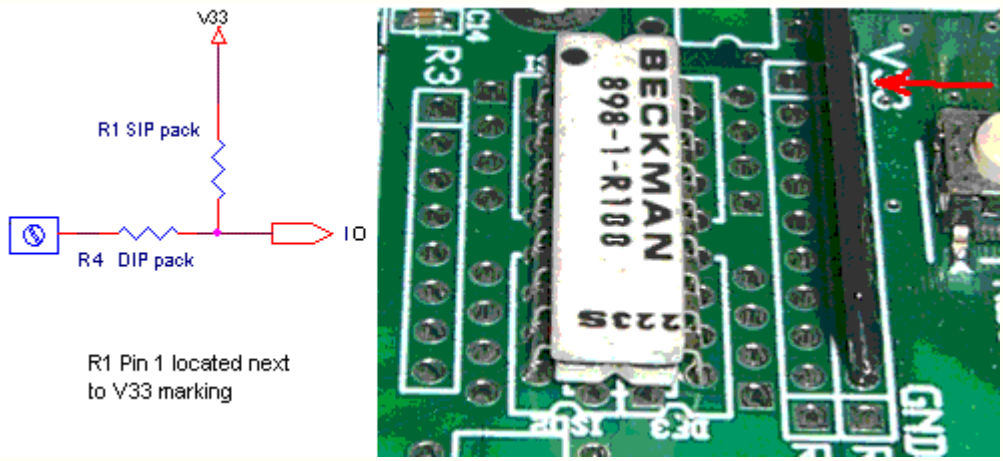




digital IO (pull-down) --



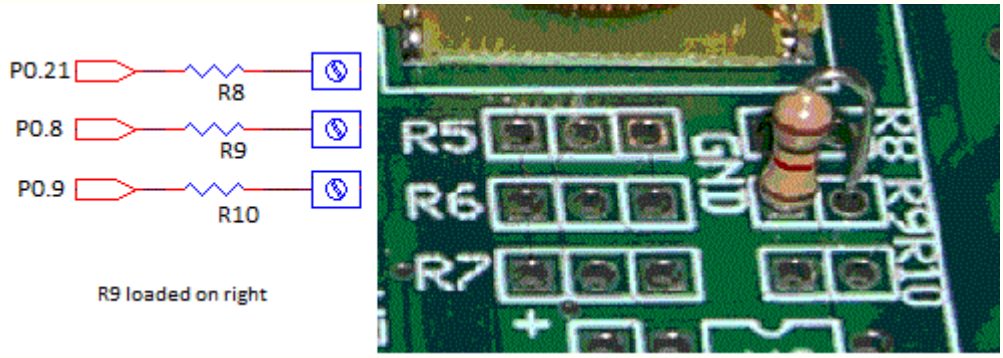
digital IO (pull-up) --



3 digital IOs

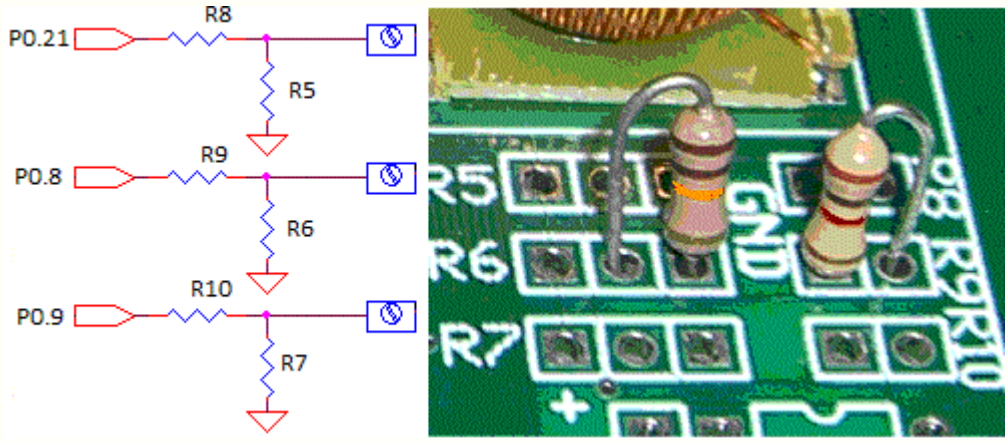
These may be configured as straight thru, or with pull-ups or pull-downs

digital IO --



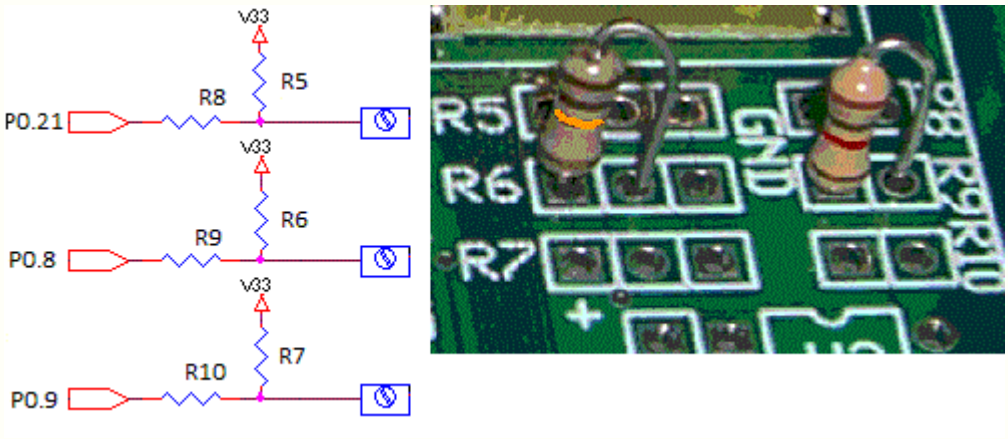
shown with 100 ohm series

digital IO (pull-down) --



shown with 10K pull-down and 100 series

digital IO (pull-up) --

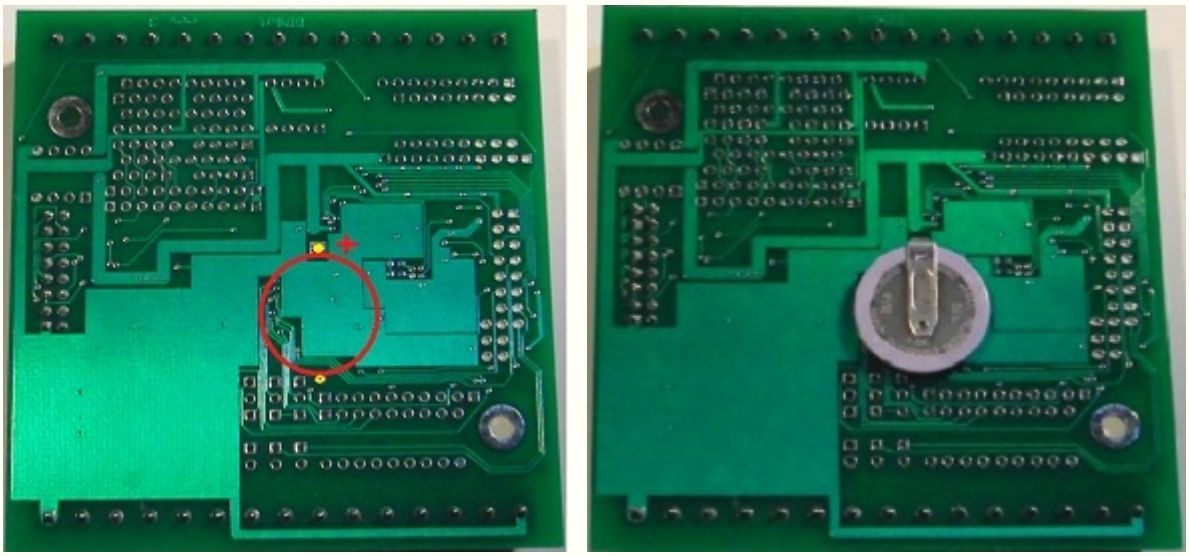


shown with 10K pull-up and 100 series

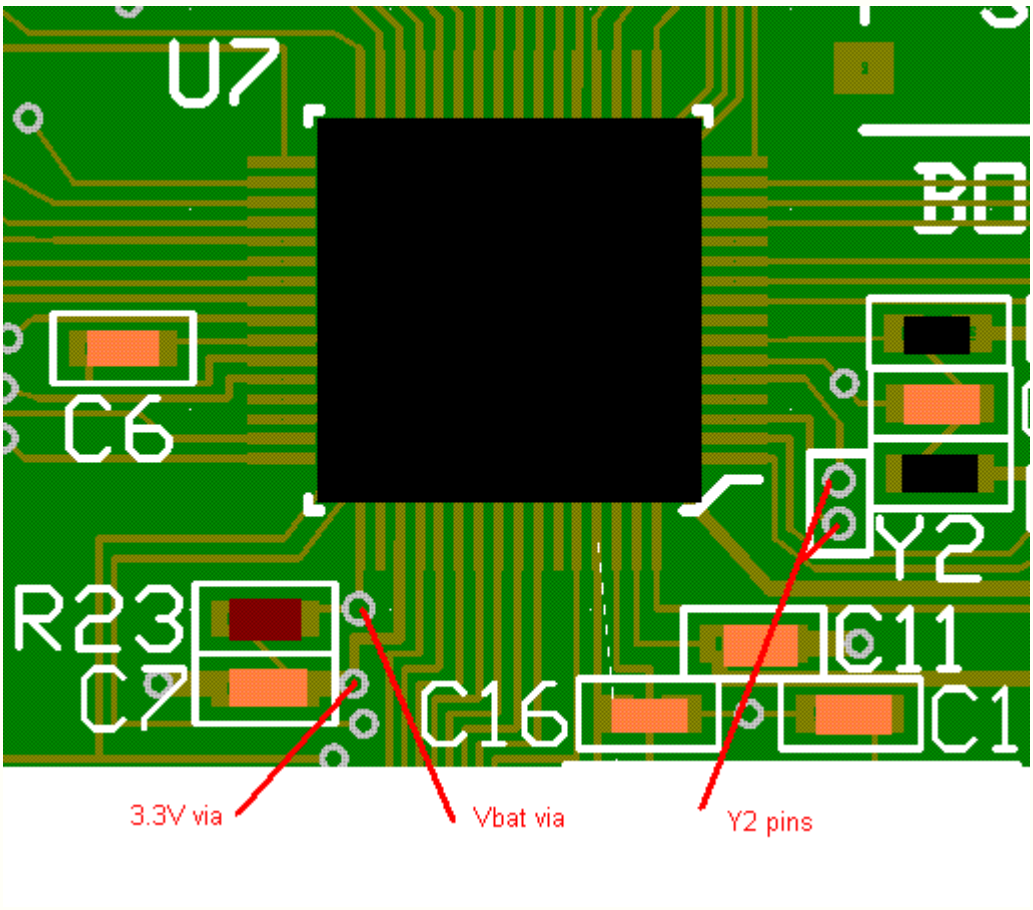
RTC options

Rev 3

This revision adds the diode and resistor needed for charging an ML2020 battery. That battery can be mounted on the backside of the board as illustrated below

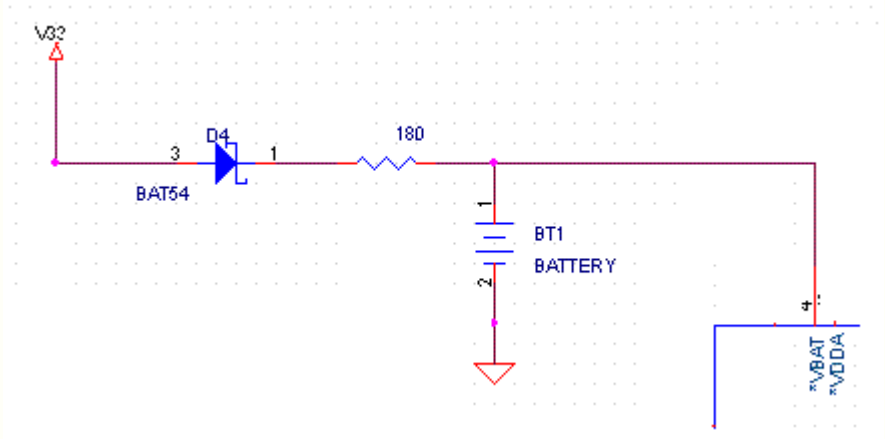


Rev 2



To connect a battery, remove R23, and use the Vbat via to connect, a resistor-Schottky diode-battery connection (suggested schematic below)

GND and 3.3V are available on either side of C7



A 32 KHz crystal (such as the Citizen CMR200TB32.768KDZFTR) can be connected at Y2, with the two 22pF startup caps on the bottom/circuit side of the board.

PRO Pin Description

The PRO is footprint and pin compatible with the Arduino PRO. In addition it has an onboard 5V regulator so it is compatible with 5V shield boards.

BASIC or C programs can be downloaded using the installed test connector using the USB dongle contained in Coridium's evaluation kit or using the **SparkFun USB Basic Breakout board** or FTDI cable from **Digikey** . More details on **these connections here**.

Pins available to the user, 7 of which can be analog inputs

IO(0) IO(1) IO(2) IO(3) IO(4) IO(5) IO(6) IO(7) IO(8) IO(9) IO(10) IO(11) IO(12) IO(13) IO(14)	P0(9) P0(8) P0(27) P0(19) P0(28) P0(21) P0(5) P0(29) P0(30) P0(16) P0(7) P0(13) P0(4) P0(6) P0(20)	RXD(1) TXD(1) EINT0 	PWM1 PWM2 PWM8 PWM4 PWM3 PWM6 PWM7 PWM5	Input/Outputs -- user controlled 0-3.3V level 4mA drive when configured as Outputs 5V tolerant use limiting resistor when connecting to a 5V supply
IO(15)	P0(15)	EINT2		IO15 connected to LED -- no other connection
AD(0) AD(1) AD(2) AD(3) AD(4) AD(5) AD(6) AD(7)*	P0(22) P0(23) P0(24) P0(10) P0(11) P0(12) P0(25) P0(26)	IO(16) IO(17) IO(18) IO(19) IO(20) IO(21) IO(22) IO(23)		10 bit A/D inputs may also be used as digital Input/Outputs IO(16-23) when used as analog lines, voltage levels should not exceed 3.3V AD6 connected to Arduino AREF pin AD7 connected to a via

Dual Use AD pins

On reset or power up the AD pins are configured as AD inputs. To change those to digital IOs, the user must individually specify a control direction using INPUT(x), OUTPUT(x), DIR(x), or IO(x) commands. After that they will remain digital IOs until the next reset or power up.

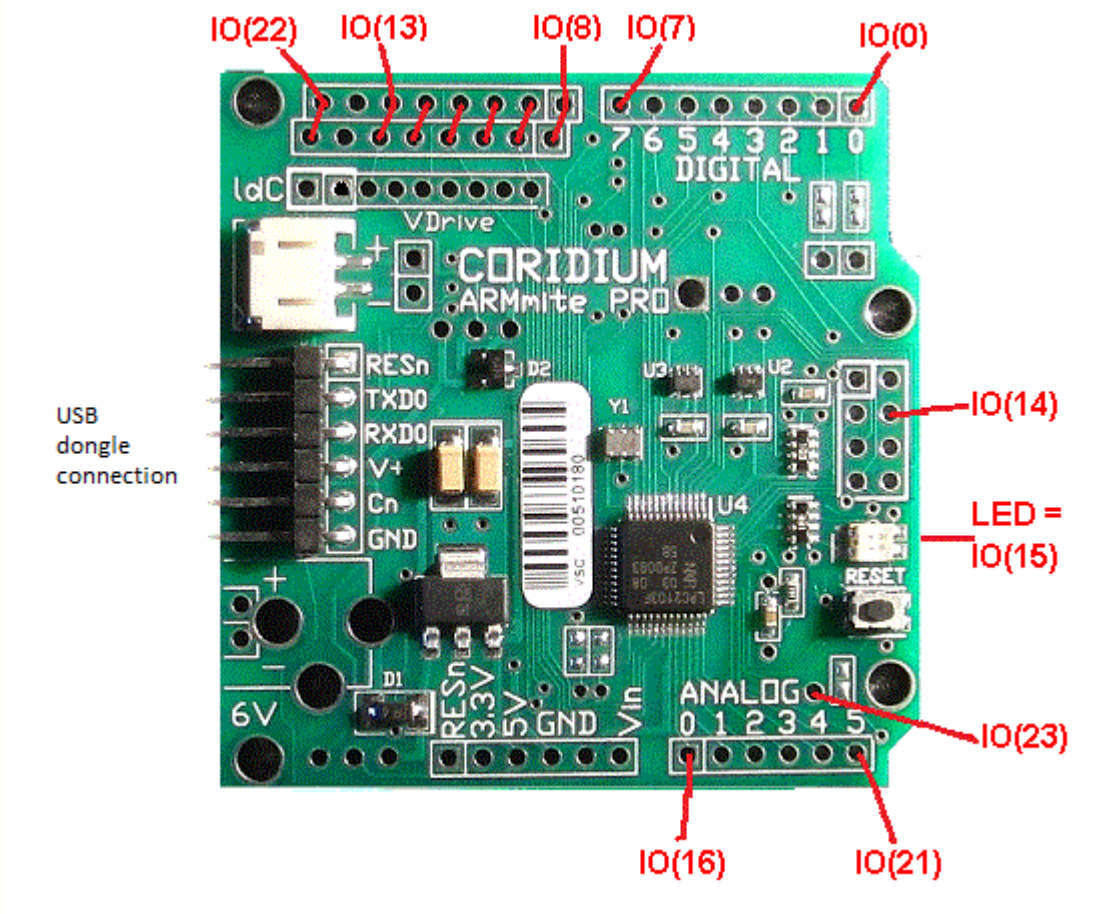
The LPC2103 does not support an external reference for the A/D converters, so the Arduino AREF pin is connected to a seventh converter, AD(6).

PWM pins

All pins can be used for the software PWM function, and 8 pins can be used for the hardware driven HWPWM function.

Digital IO connections

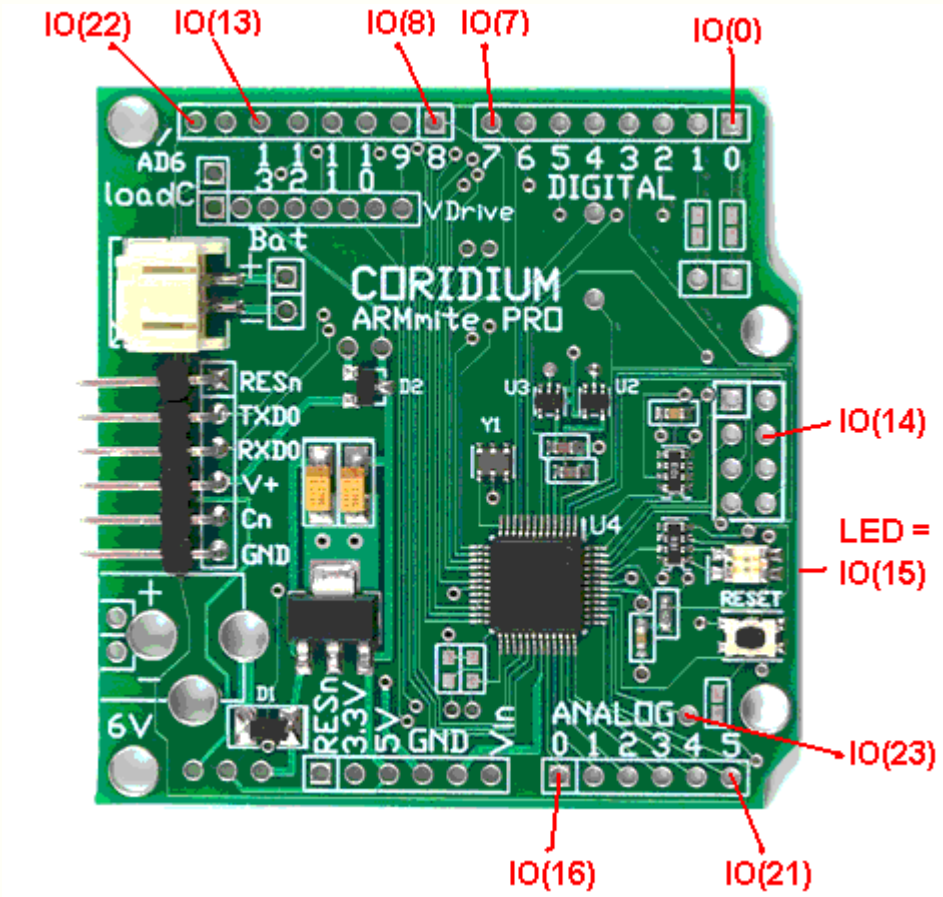
REV4



The major change for rev 4 is to add a parallel connection for the 8 IOs IO(8)-IO(13), GND and IO(22) that is on 0.1" centers in relation to the other connections.

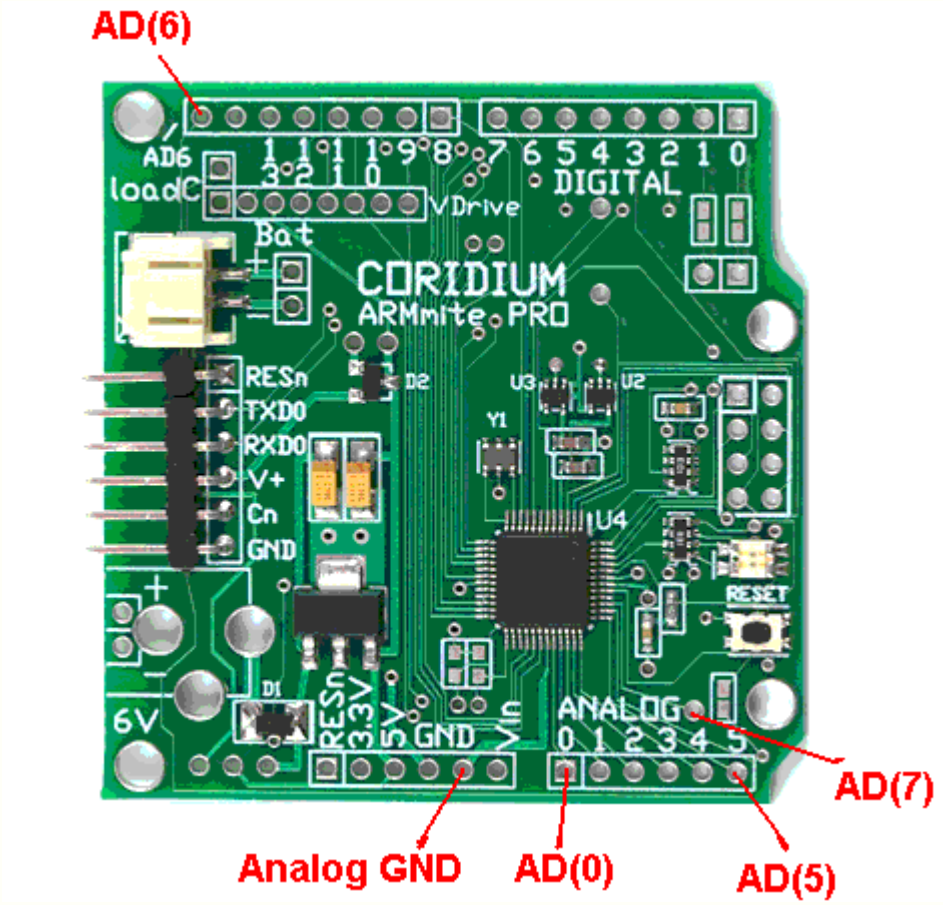
In addition the loadC jumper was rotated 90 degrees to make room for this extra connection. And it is also easier to add a battery to the board, by making 1 cut, and adding a diode, resistor and battery (details below).

REV 3



Picture is for the Rev 3 production board. On the Rev 1, IO(23) is available on the via next to AD(5)/IO(21).

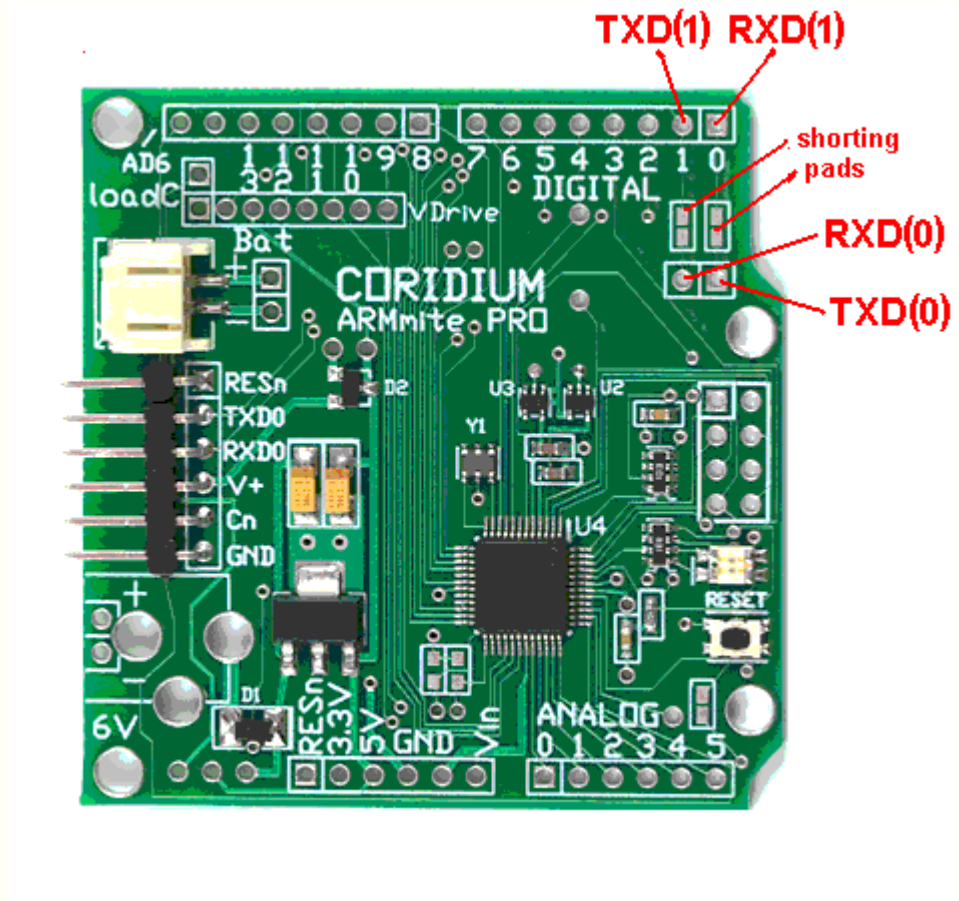
Analog connections



Picture is for the Rev 3 production board. On the Rev 1, AD(7) is available on a via next to AD(5).

Dual Serial Ports

Where the Arduino has only a single serial port, the ARMmite PRO has 2 UARTs. The second UART is connected to IO pins 0 and 1. This allows it to be used simultaneously with the first UART acting as a debug port. In the Arduino, the debug port is connected to these 2 IOs. To allow for this connection as well, the ARMmite PRO has 2 shorting bridges that can be shorted to make this connection.



Power connections

The board is shipped with a **2mm power jack compatible** with a JST PHR/S2B or **SparkFun PRT8671** or various battery packs from SparkFun.

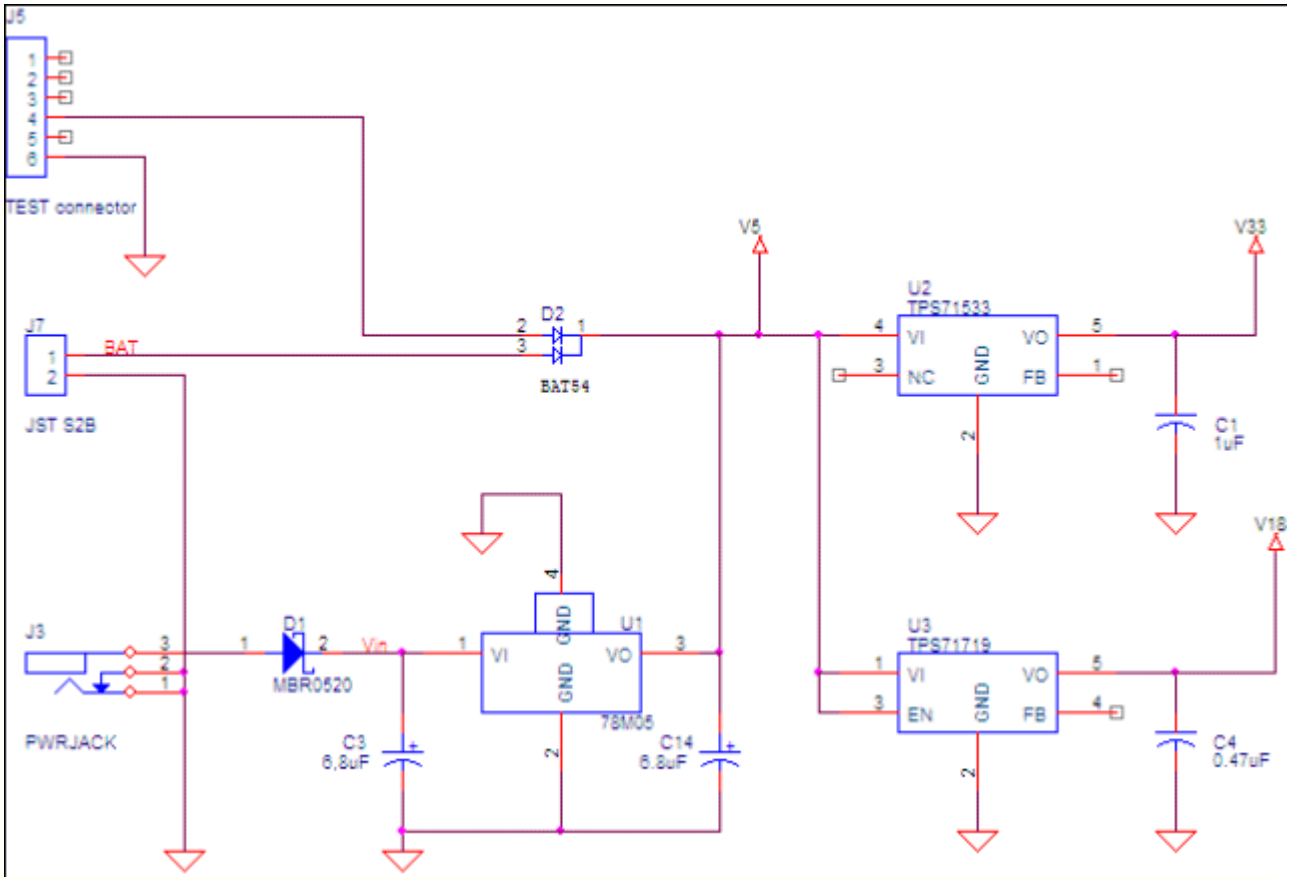
Pads for a Cui PJ-002A or **SparkFun PRT-119** power connector are available in the lower left hand corner.

For both battery and 6V input, 2 pin 0.1" spaced holes are available for wires or headers. When using the battery connector, total current draw for the board must be limited to 200mA. If you want to use more current, you should install a jumper around the D2 diode (holes are available above D2).

Diode steering allows power to be supplied from a barrel connector from a 6V unregulated source, 5V USB test connector, or the battery connector. Because of the Schottky diodes, all 3 power sources can be connected simultaneously. If you are using an unregulated wall transformer, you must check the open circuit voltage and it MUST be less than 12V.

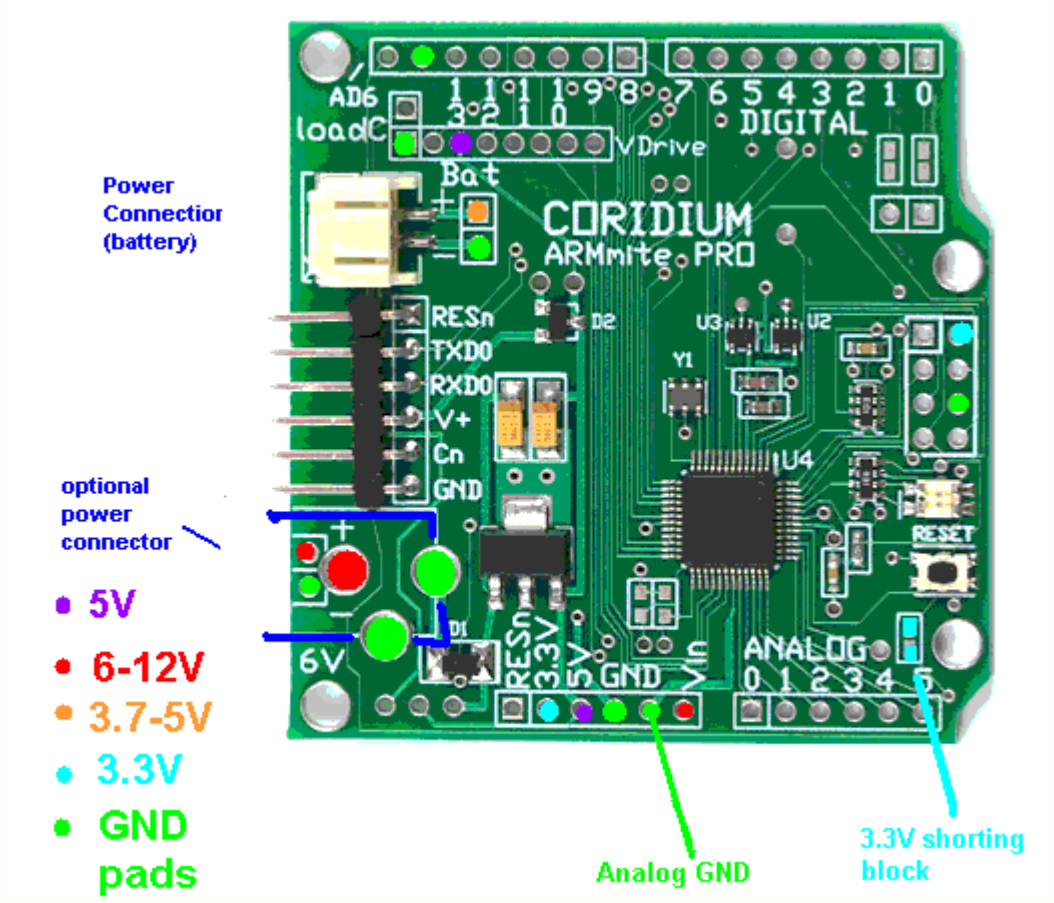
When the 6V source is used, 5V Arduino shields can be powered from the ARMmite PRO.

The schematic describes this circuit



The **full schematic can be seen here**

Power connections details

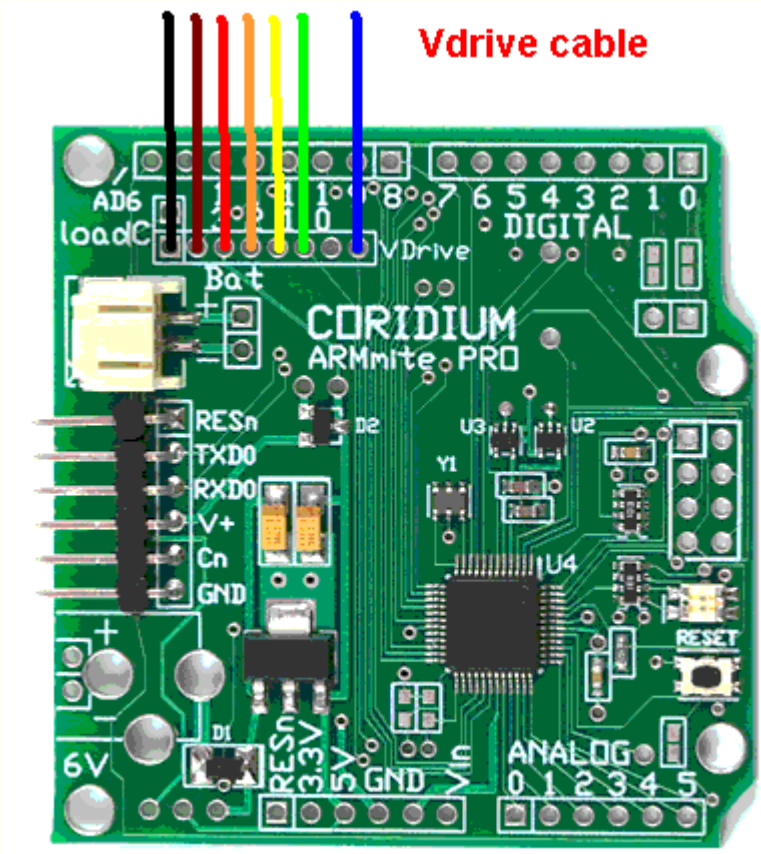


The 3.3V regulator can supply 150 mA, with 50 being used by the LPC2103. The 3.3V connection next to RESn on the lower power connector is only connected if the shorting pads are shorted (NOT the factory default).

The analog GND should be used to connect to the GND of analog inputs. Digital and Analog GNDs are connected together with a small trace, but to minimize noise you should use the analog GND only for analog signals.

Vdrive connection (added in rev 2)

A connection for the Vdrive has been added so it is easy to use an ARMmite PRO to do data logging to a USB Flash. So all that is required is a **Vdrive** and a **2mm header** .

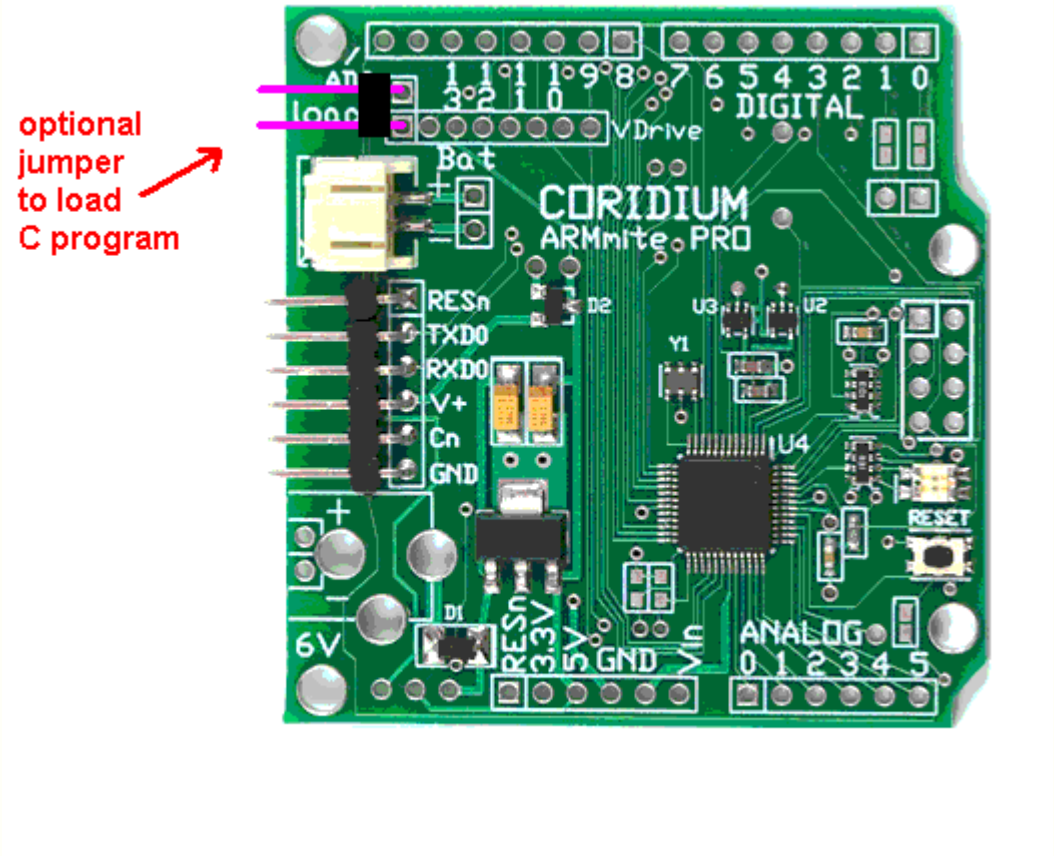


Jumpers and test connector for Program Download

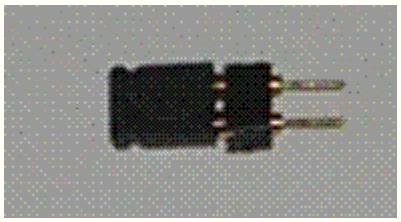
The USB Dongle from Coridium will supply 5V from the USB to power the ARMmite PRO. It also controls the RESET and BOOT signals to automatically load C or BASIC programs using MakeItC or BASICtools.

When using the SparkFun FTDI Basic Breakout Board, a limited amount of power can be supplied from the BBB, but this is limited to 50 mA and after diode drops, its about 2.8V to the LPC2103. In practice this will run, but it is outside the part specifications, so it should be limited in use.

Also with the SparkFun FTDI Basic Breakout Board to load a C program, the LOAD C jumper needs to be installed, then removed to run the program. BASIC programs can be loaded and controlled using the SparkFun board, with no additional steps/jumpers.

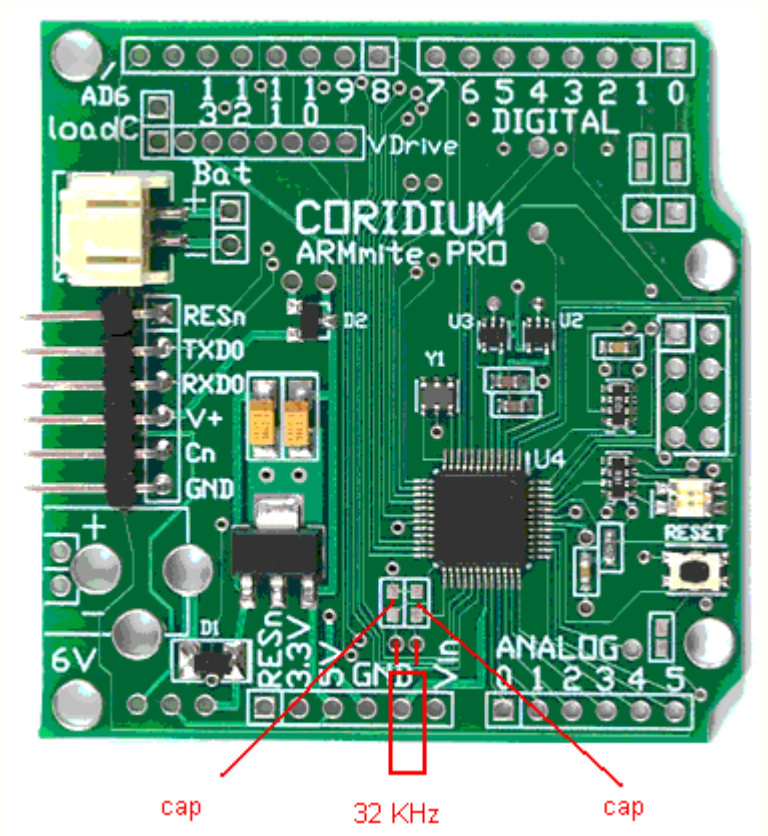


An alternative is to use a 2 pin header with a shorting block (pictured below)



Real Time Clock Oscillator

The ARMmite PRO uses ceramic resonator, which has a 1% accuracy. But there is a provision to load a 32 KHz crystal and 2 cap to use that for the Real Time Clock.

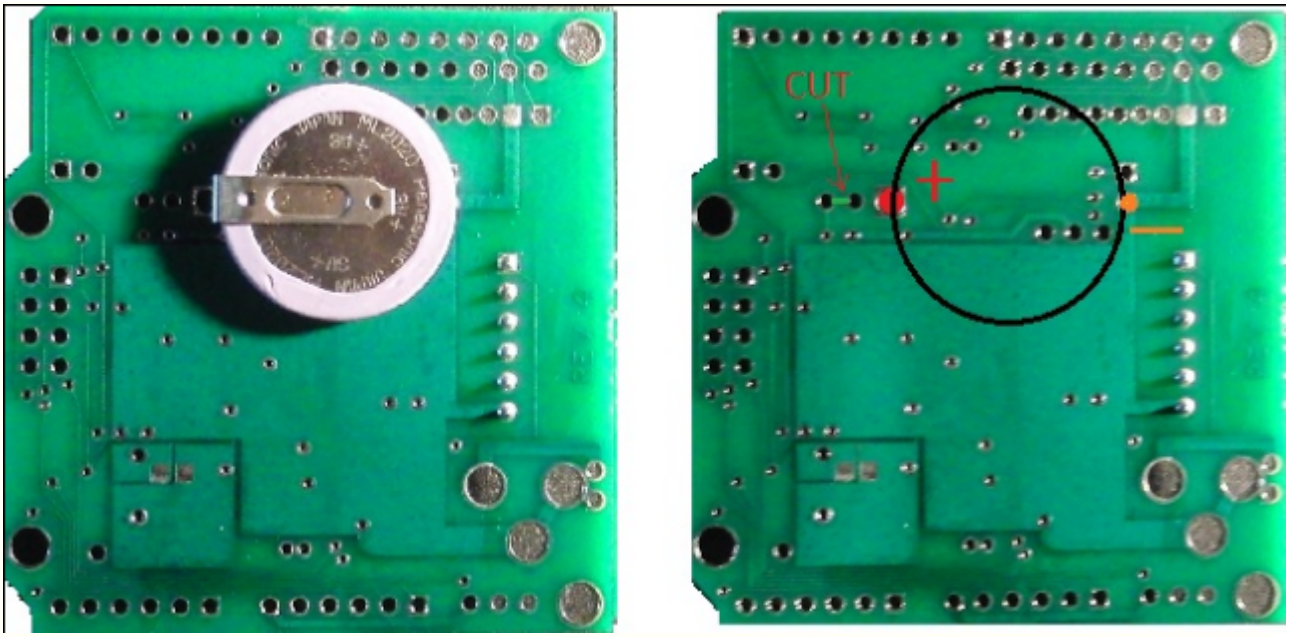


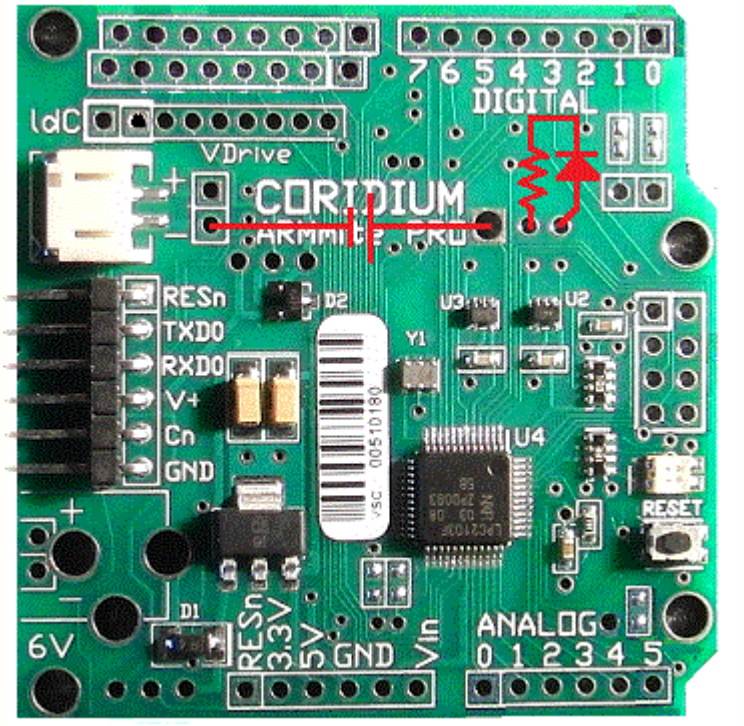
The crystal should be a 32.768 KHz can type, and depending on the rating the capacitors are 0603 size 18-27pF.

If you install this, include the following at the start of your program.

```
#define RTC_CCR   * &HE0024008
RTC_CCR = &H11                          ' clock the RTC with the 32 KHz crystal
```

Rev 4 version of the board makes it easier to add a battery. First cut the trace indicated below, then install a Schottky Diode, 180 ohm resistor and Panasonic ML2020H as shown below. The VL2020/HFN will also work, though it is more expensive and has less power.


CUT
+
–


7 6 5 4 3 2 1 0
DIGITAL
IdC
VDrive
CORIDIUM
RESn
TXD0
RXD0
V+
Cn
GND
D2
U3
U2
Y1
U4
RESET
D1
6V
RESn
3.3V
5V
GND
Vin
ANALOG
0 1 2 3 4 5

Wireless ARMmite Pin Description

obsolete product -- documentation for reference

24 pins available to the user, 8 of which can be analog inputs

Refer to the **Getting started section** for details on selecting your wireless components.

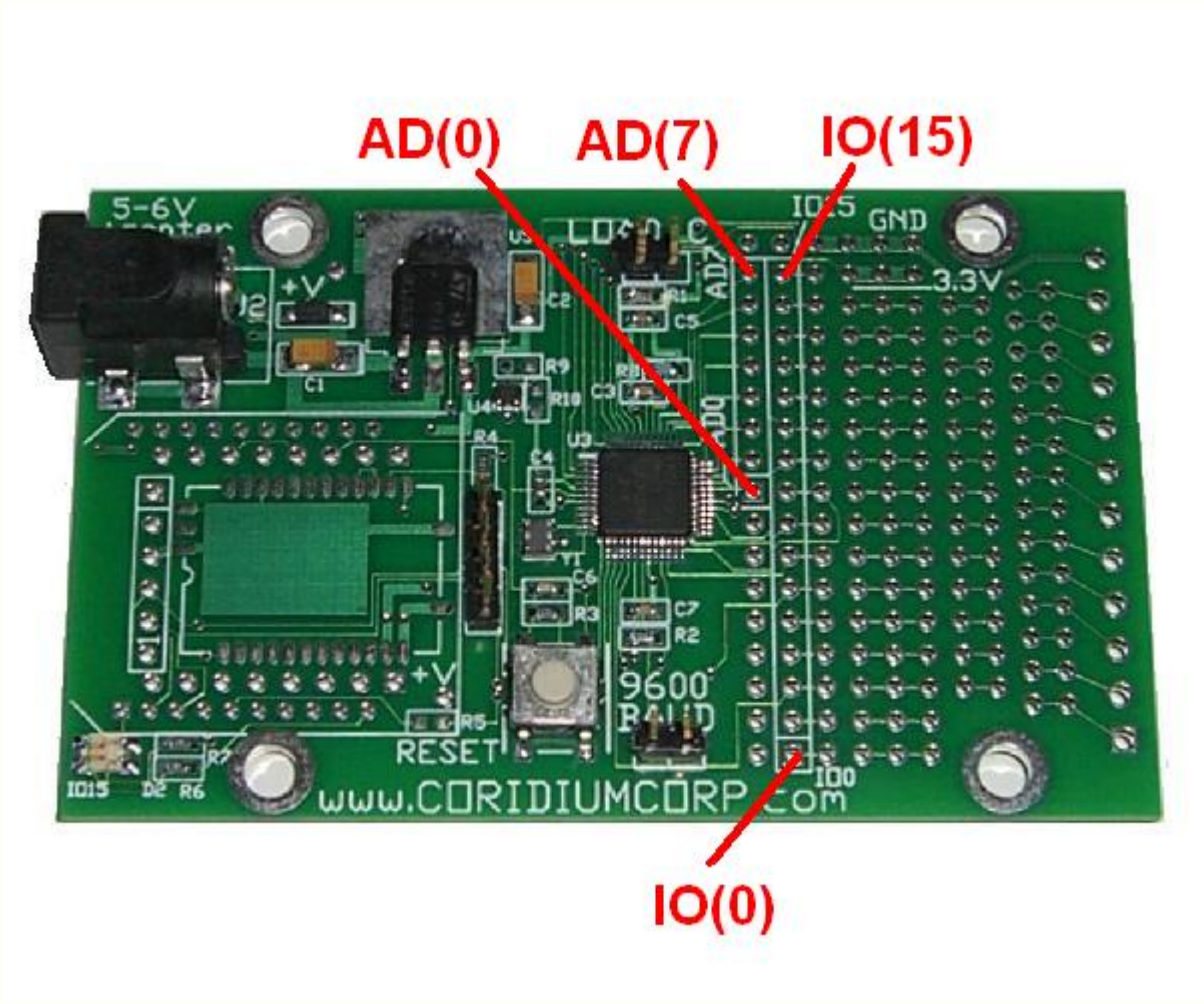
IO0 IO1 IO2 IO3 IO4 IO5 IO6 IO7 IO8 IO9 IO10 IO11 IO14 IO15	RXD1 TXD1 EINT0 EINT2	PWM1 PWM2 PWM3 PWM4 PWM5 PWM6 PWM7 PWM8	Input/Outputs -- user controlled 0-3.3V level 4mA drive when configured as Outputs 5V tolerant - use limiting resistor when connecting to a 5V supply IO15 connected to LED
IO12 IO13			Input/Outputs -- user controlled Open drain 4mA pulldown when configured as Outputs 5V tolerant
AD0 AD1 AD2 AD3 AD4 AD5 AD6 AD7	IO16 IO17 IO18 IO19 IO20 IO21 IO22 IO23		10 bit A/D inputs may also be used as digital Input/Outputs IO(16-23) -- user controlled when used as analog lines, voltage levels should not exceed 3.3V

Dual Use AD pins

On reset or power up the AD pins are configured as AD inputs. To change those to digital IOs, the user must individually specify a control direction using INPUT x, OUTPUT x, DIR(x), or IO(x) commands. After that they will remain digital IOs until the next reset or power up.

PWM pins

All pins can be used for the software PWM function, and 8 pins can be used for the hardware driven HWPWM function.



Jumpers

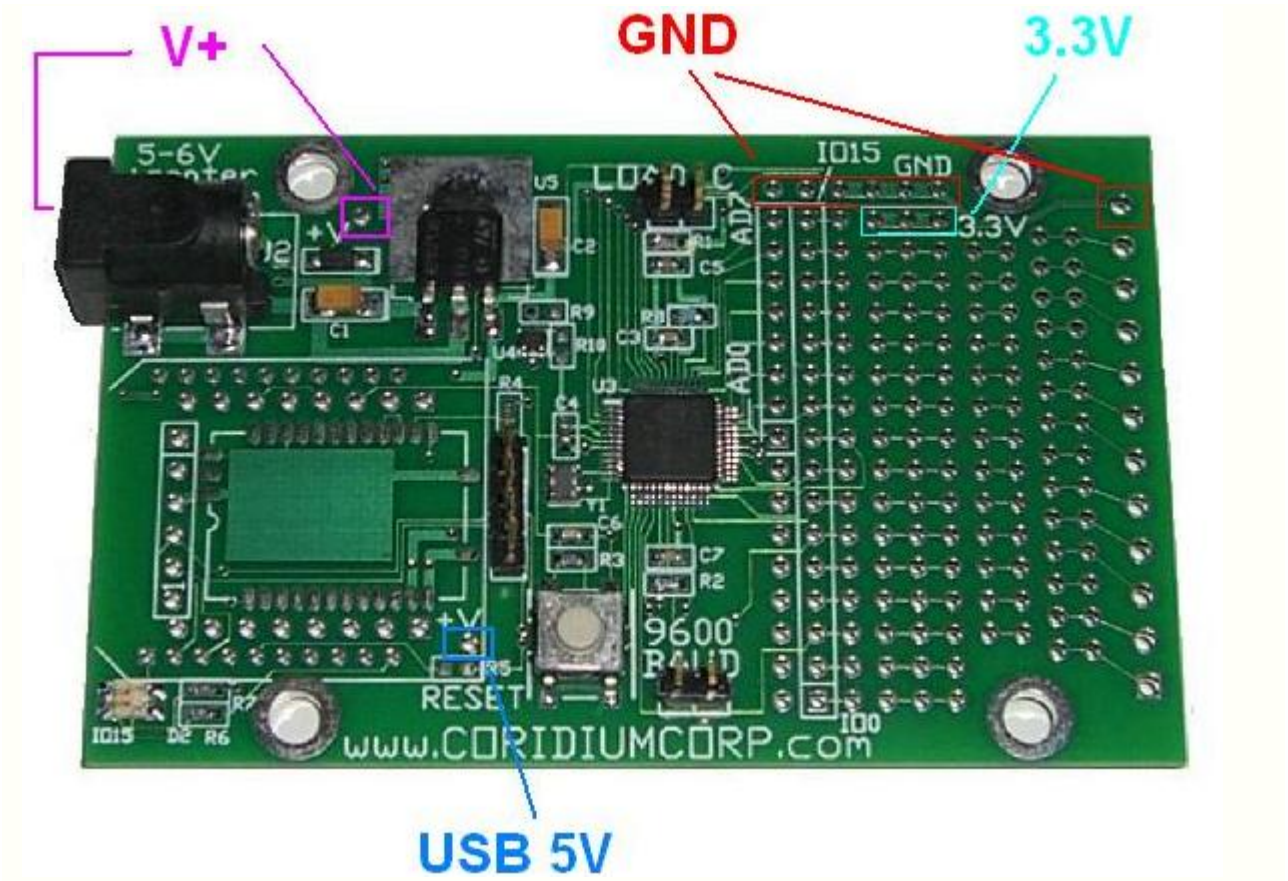
The wireless ARMmite default baud setting is 19.2Kb, and the default setting for the BlueSmiRF and Xbee modules are 9600 baud. While the defaults can be changed for these wireless modules, there is a potential "chicken and egg" problem getting there. So if the 9600 baud jumper is connected on RESET, the ARMmite will come up at that baud rate.

The wireless connections do not have sufficient control lines such that RESET can be controlled from the PC, as well as the RTS line which is used to load C programs. So the BASICtools and MakeItC will prompt you to add a jumper or push the reset button where appropriate.

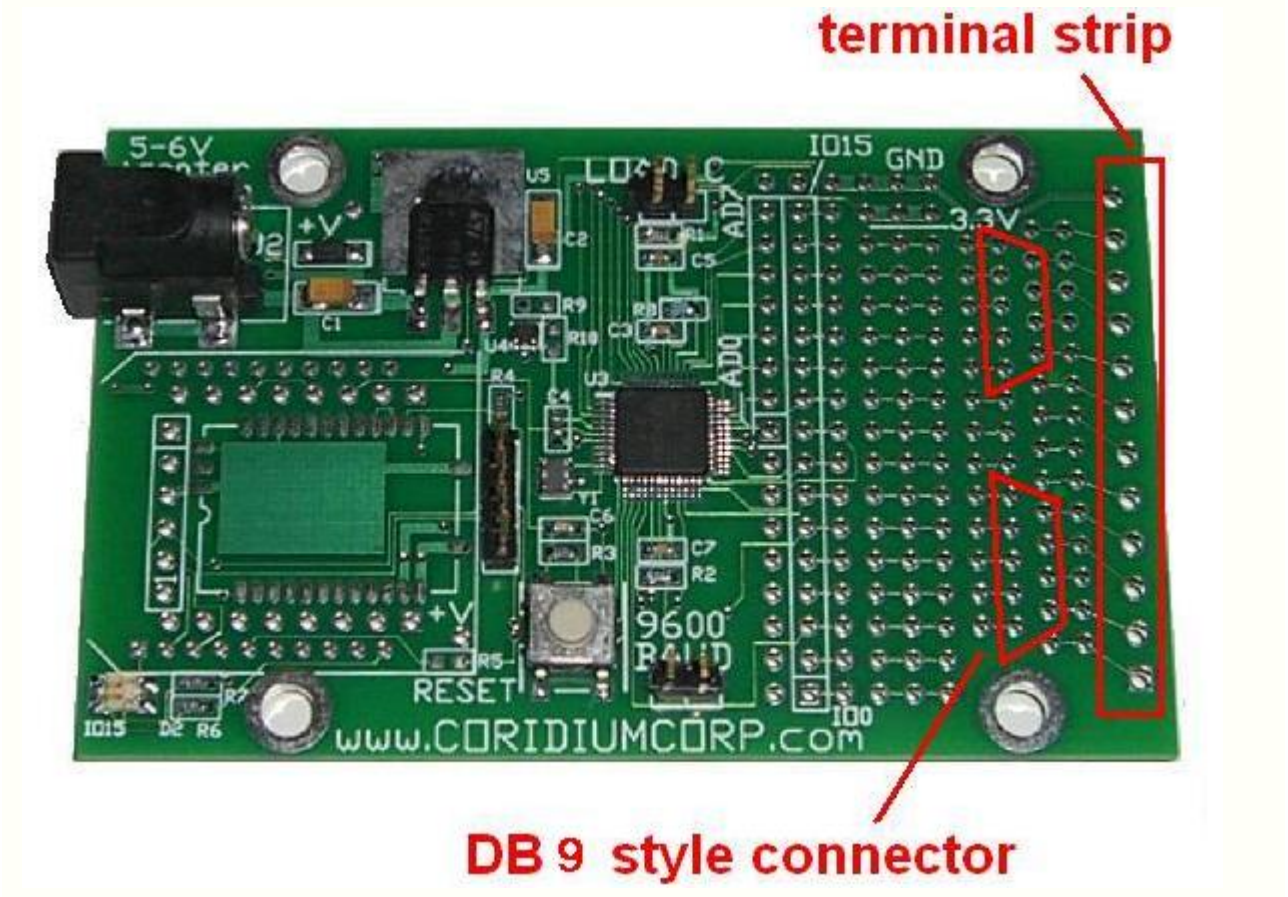
Power

The wireless ARMmite primary power supply is 3.3V. This voltage is available for user circuitry at 3 pins in the prototype area. There is also a pad that is connected to the input power.

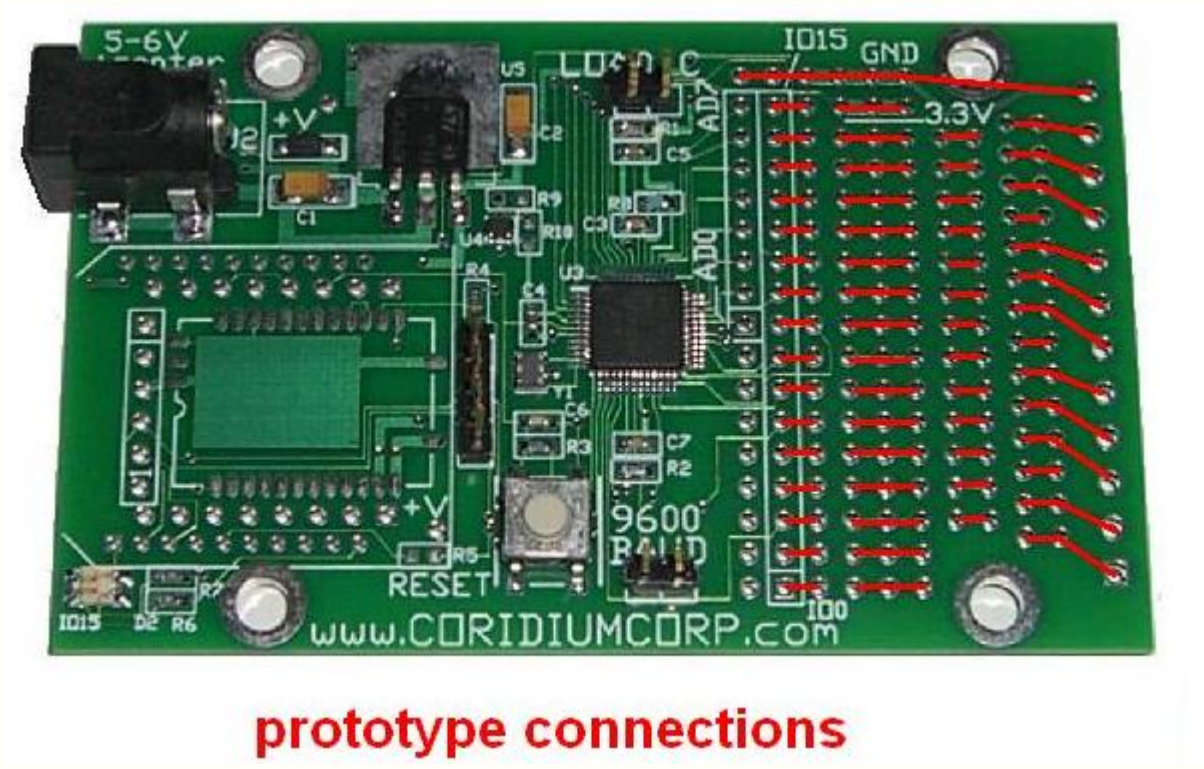
Input power for the wireless ARMmite require 5V or greater. It may be a regulated 5V supply or an un-regulated 6V supply. But it all cases it should not exceed 12V DC. IF YOU ARE USING A BlueSMiRF, this input power is applied directly to the BlueSmiRF and it must not exceed 6V . If you are using an unregulated wall transformer, check the open circuit voltage and make sure it is within these limits.



If the all the connections are made to the USB breakout board then 5V can be supplied from the USB. That is also available at the USB 5V pad. When using power from the USB, it should be the only connection for power (do not connect the 5-6V power).



suggested terminal strip On Shore Tech ED550/12DS or equivalent 3.5mm pitch connector (available at Digikey)



XB Sensor Pin Description

Product Preview

22 pins available to the user, 7 of which can be analog inputs

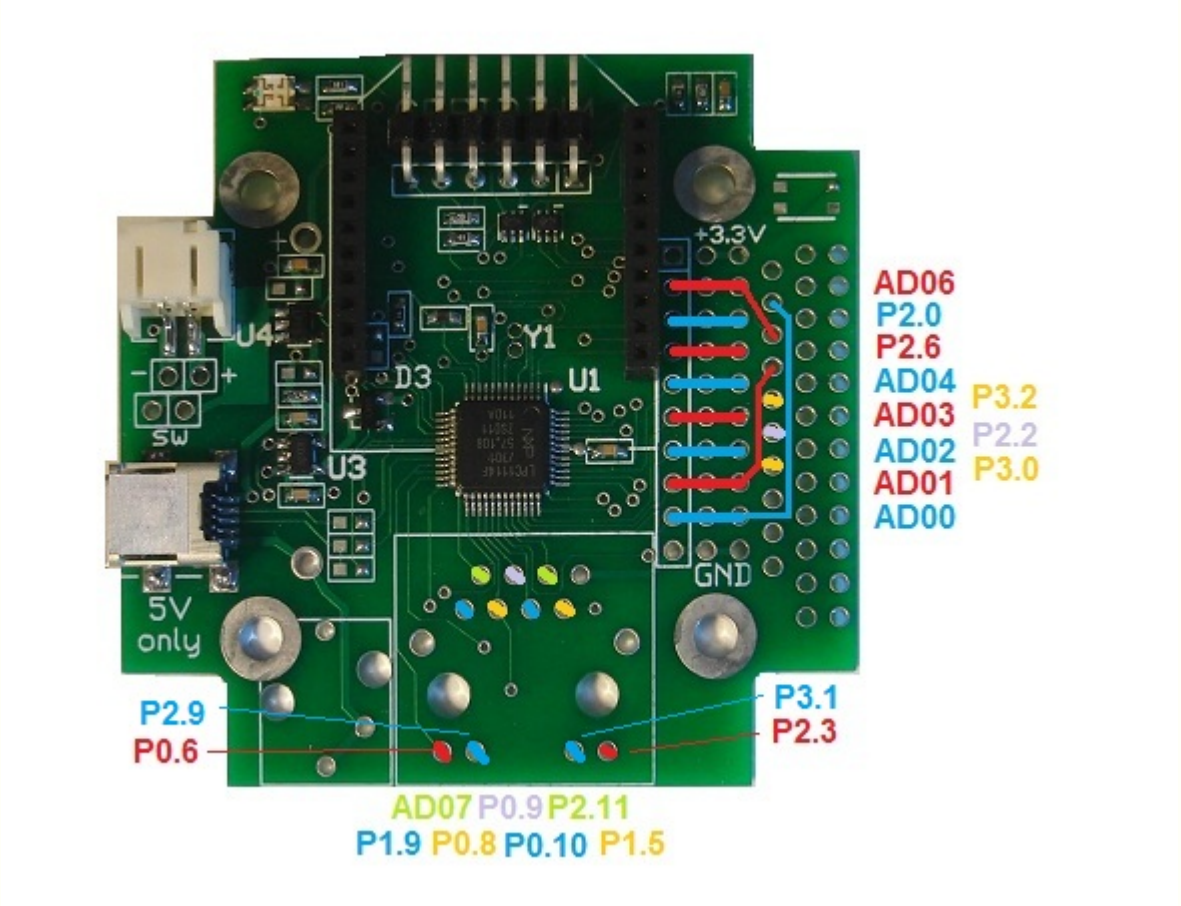
Refer to the **Getting started section** for details on connecting your wireless components.

Dual Use AD pins

On reset or power up the AD pins are configured as digital inputs. To change those to ADs, the user must include the AD11.bas library and call InitAD

PWM pins

All pins can be used for the software PWM function, and 8 pins can be used for the hardware driven HWPWM function.



One Wire interfacing

The optional RJ-45 connection was designed with the idea of driving up to 7 one wire sensors, with P0.7 available as a high current drive that can power up the sensors through a 8 pin SMT (CTS 745C type) resistor on the back side of the board.

That connector is not limited to that, as I2C, SPI or other serial options can be used.

Baud rate settings

The XB sensor default baud setting is 19.2Kb, and the default setting for the Xbee modules are 9600 baud. The Xbee sensor baud rate can be set with the X-CTU utility from Digi. Some details in the **installation section**.

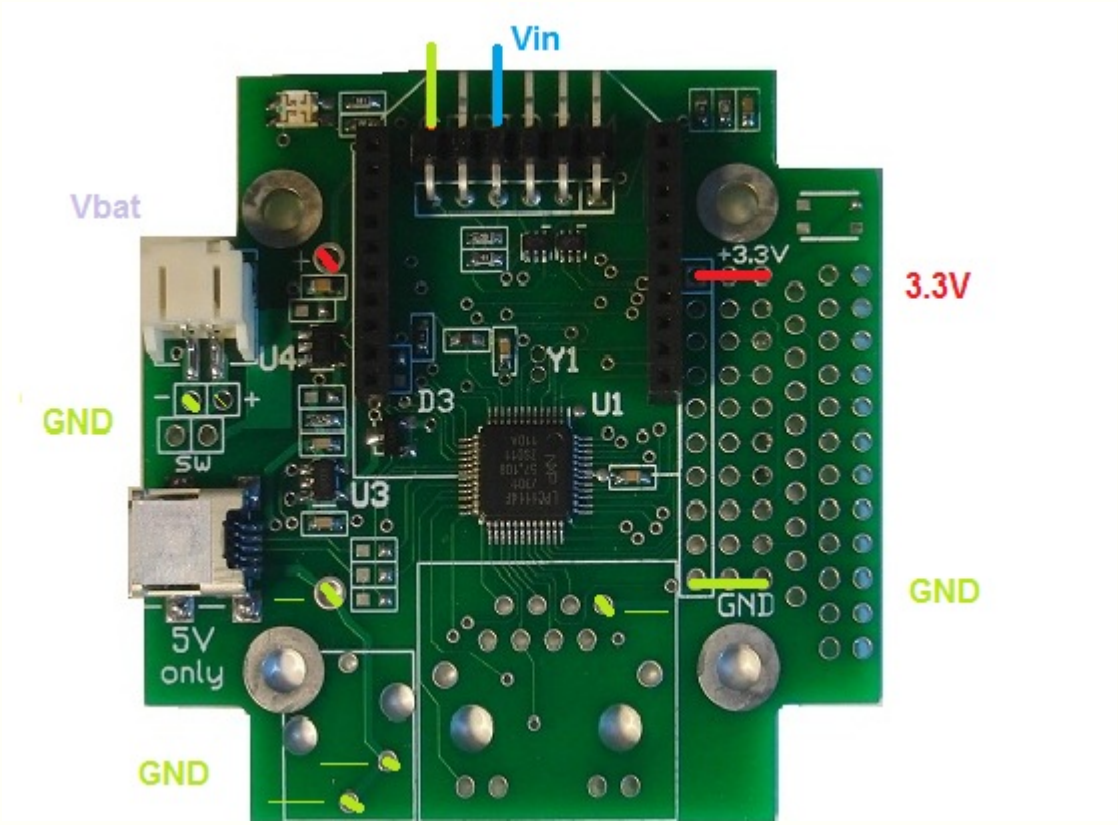
When the USB dongle is connected that takes precedence over the serial connection on UART0 to the Xbee module. This is done by sensing the Vin voltage **(see the schematic)** for details.

Power

The XB sensor primary power supply is 3.3V. This voltage is available for user circuitry at 3 pins in the prototype area. This can be derived from either the battery connection, or the mini-B USB connection. When connected to the USB mini-B connector, a Li-Poly battery can be charged. The charge current is set at 500 mA. These batteries are available in **various sizes from SparkFun**.

During debugging, the USB dongle can be connected and it will supply power to the board through Vin. But the battery should not be charged this way as it may exceed the current capability of the USB debug dongle.

.



Optional components

Coridium does not stock the Xbee modules, as there are too many varieties. We will quote XB sensor boards with Xbee modules for orders of 10 or more boards.

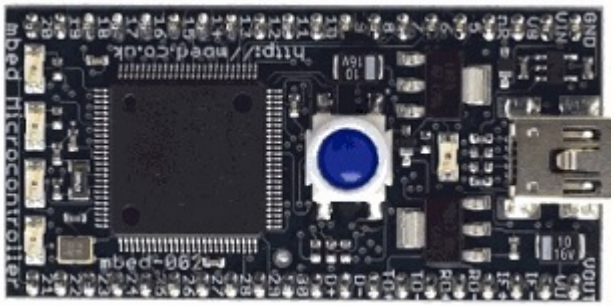
This board is designed to fit into a Polycase LP-11 enclosure

In addition, the board has been designed to accept optional RJ45 vertical connections using Bel Stewart SS-74301-002 or TE connectivity 5556416, available from Digikey.

Also a vertical power connection can be made with Cui PJ-032A 2.1mm barrel connector.

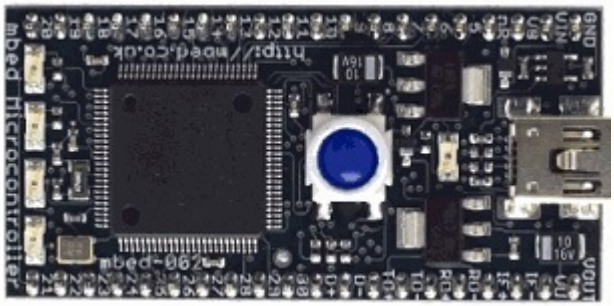
Space is also provided for a reset switch.

ARMweb Ethernet Services



Ethernet Services (mbed 1768) armweb.htm PAGE Controls Page CGI Services CGI Example FTP Services Mail Service Web Services Web BASIC UDP Services Reset Behavior Firmware Update	ARMweb C support

Ethernet Getting Started



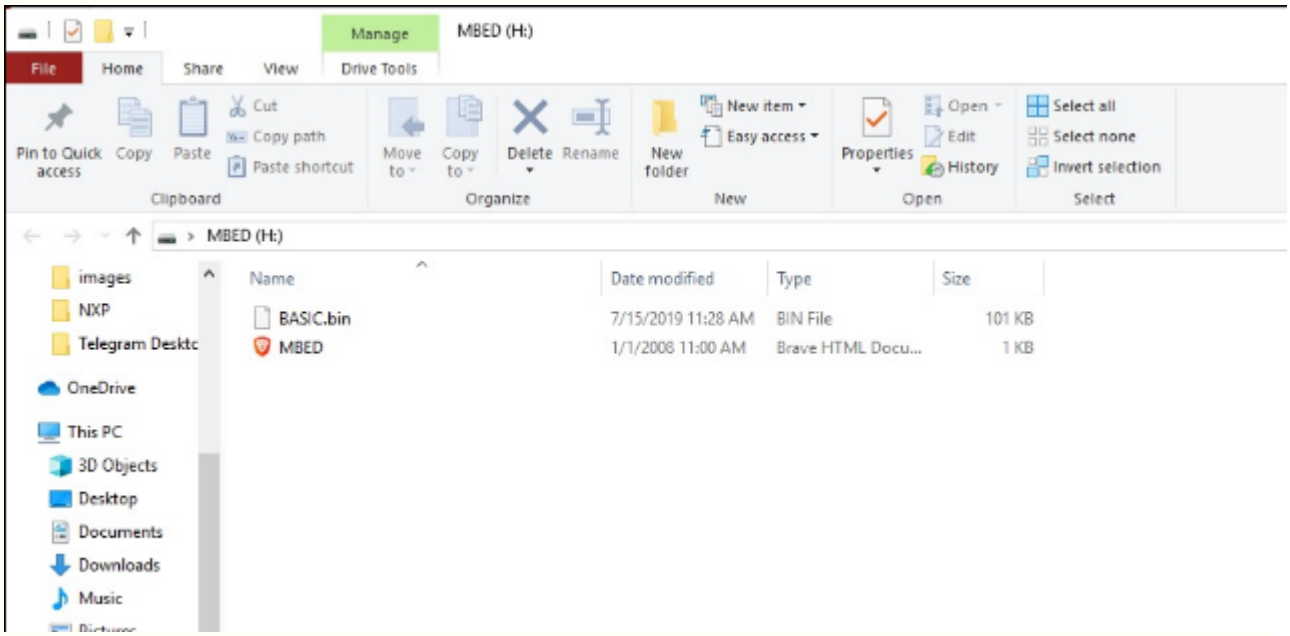
Getting Started

- **Install Software** (BASICtools)
- **Install Firmare** (mbed BASIC binary)
- **Connect Ethernet**
- **USB connection for ARMweb**
- **Writing programs with BASICtools**

USB connection for BASICtools

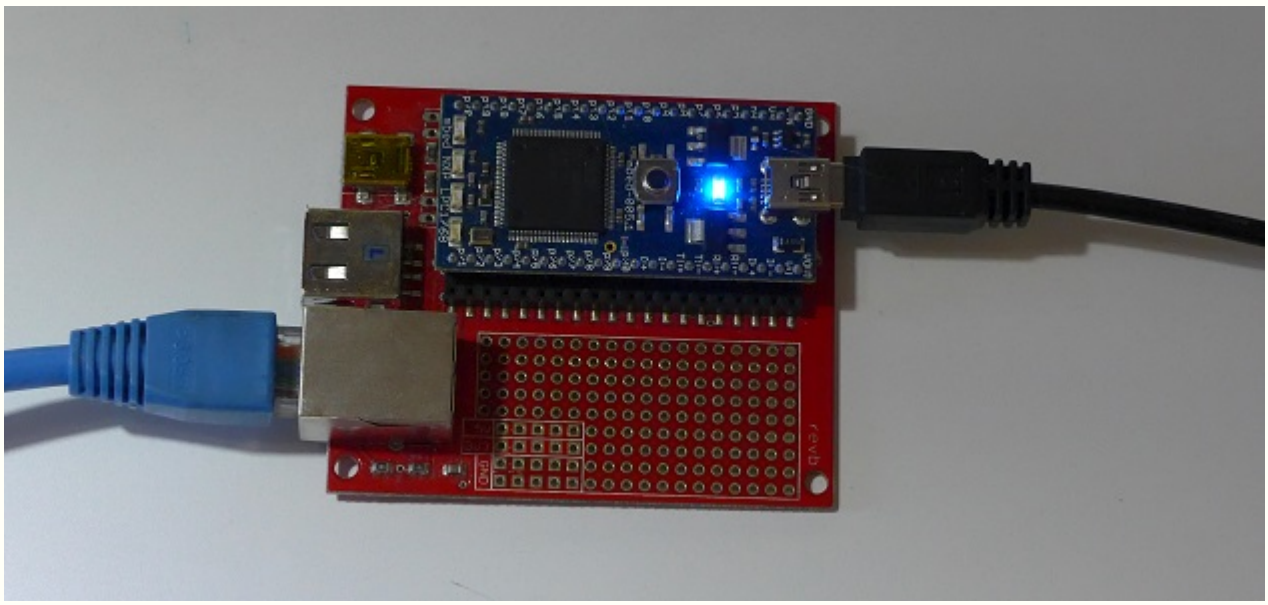
The main user program is loaded through BASICtools via a USB connection. The attachment of the USB and power supply is shown below.

Initially you will have to load BASIC firmware onto the LPC1768. These boards use the mbed protoco, so when you plug the board in for the first time, an MBED disk will appear to the files manager (below)

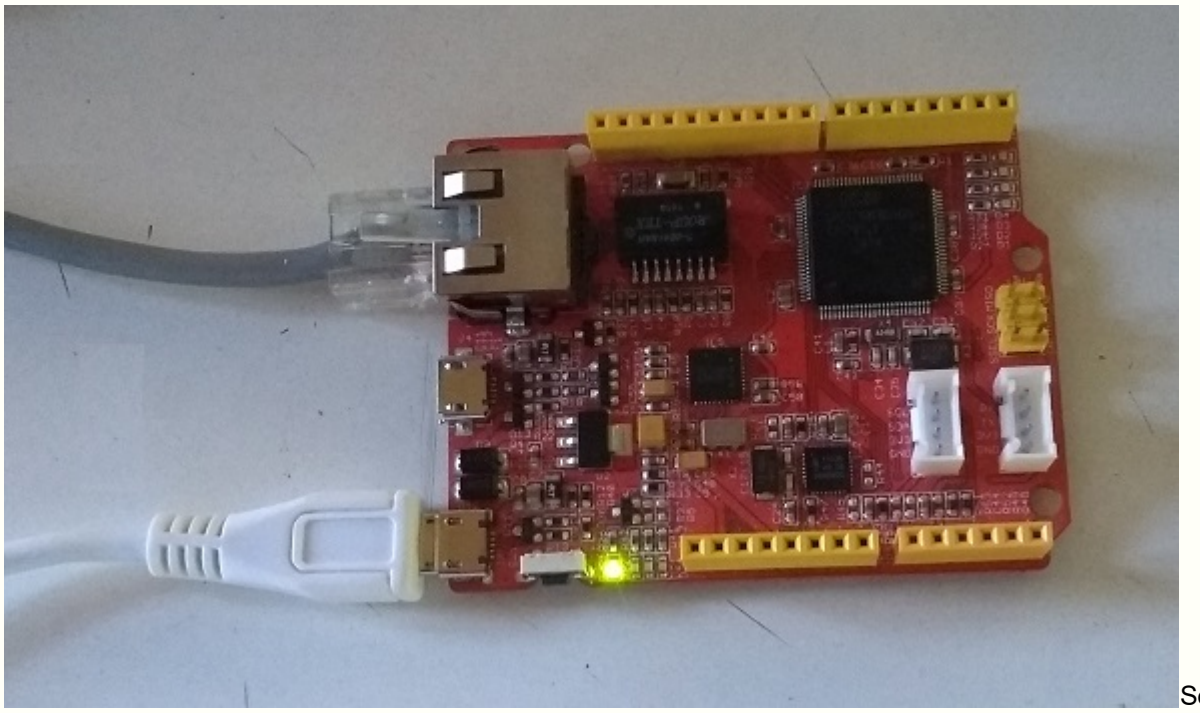


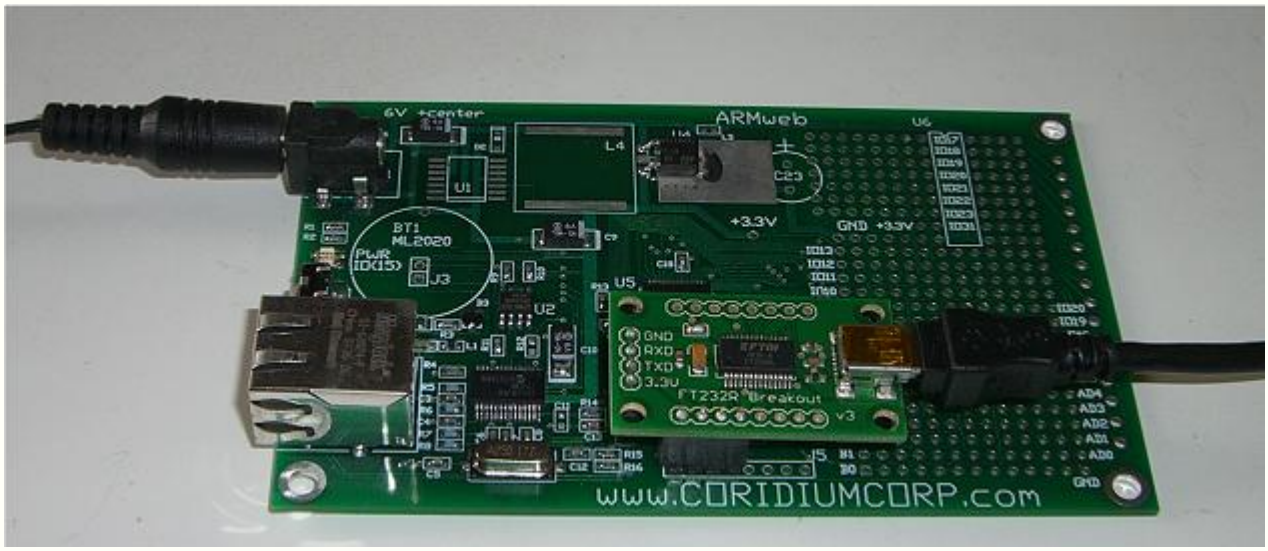
Depending on the mbed firmware version (running on a different CPU LPC11U35 or LPC43xx), you may see a .bin file. Delete any of these and copy the BASIC.bin firmware you downloaded or were emailed from Coridium. And also depending on mbed firmware you may need to reset the board via push button. Not for older ARMweb/ LPC2138 boards use**these directions** .

While an Ethernet connection is not required, if it exists and there is a DHCP server, the ARMweb will boot faster (otherwise each reset the 10 second timeout waiting for DHCP service will occur).

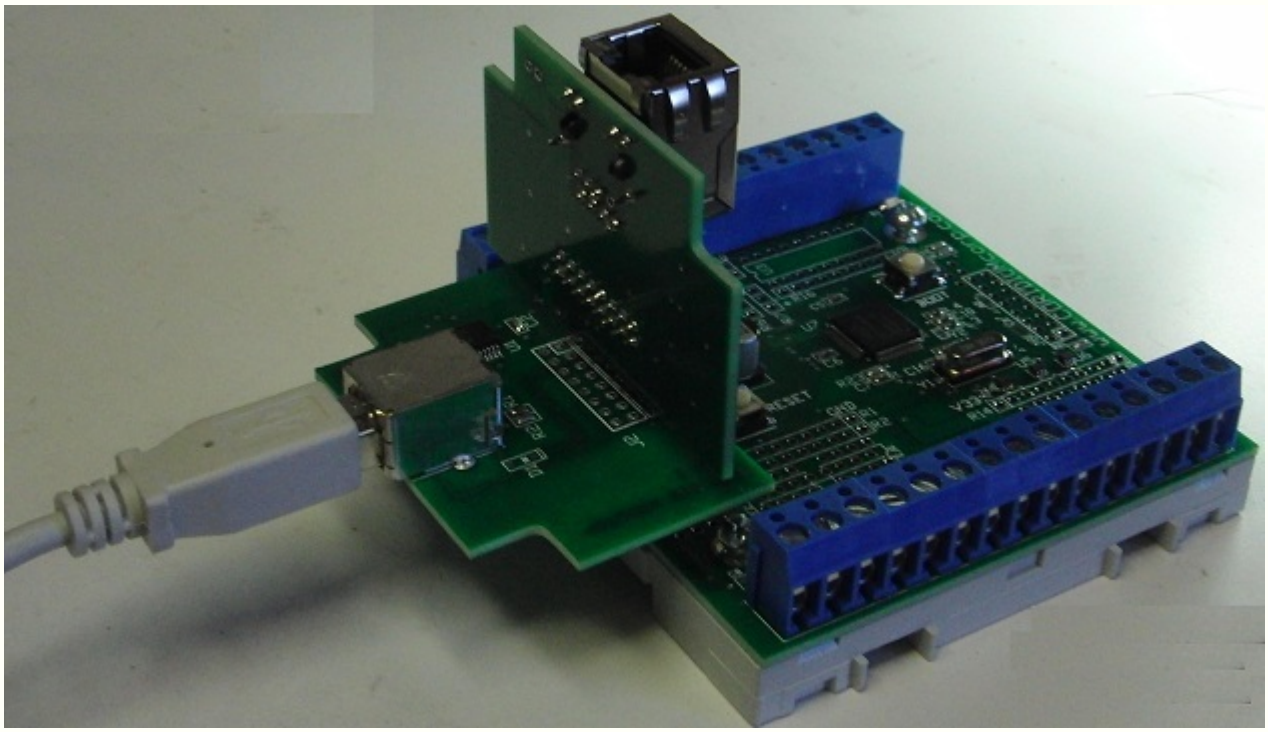


mbed LPC1768


Seeed Studio ArchPro



ARMweb


DINkit - Ethernet.

BASIC and Web page interaction

BASIC can be embedded in the web page served by webBASIC. That BASIC code can access global variables of the user program running on the ARM. BASIC embedded in the web page can **NOT** call a FUNCTION or SUB..

The user (client) can also interact with an webBASIC program via the CGI mechanism.

USB drivers

Most PC's will sound a tone that indicates a new USB device has been connected. Most Windows 7, 8 and 10 systems will either include the FTDI device driver or are able to download it automatically from the network.

If your system is unable to do that. Run the FTDI driver installation setup in the \Program Files\Coridium\Windows_drivers directory. This will install the proper drivers for the FTDI chips we use for interfacing to the USB.

Up to date details are at the **www.ftdichip.com** VCP drivers page.

Continue with the some programming examples.

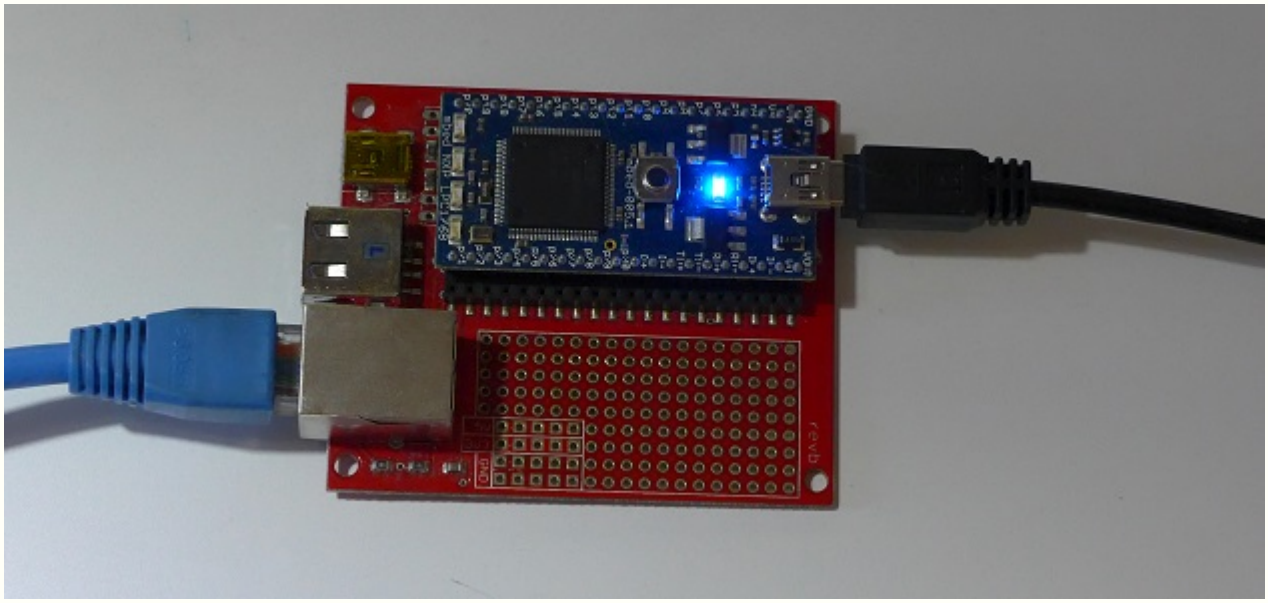
or

More details on webBASIC...

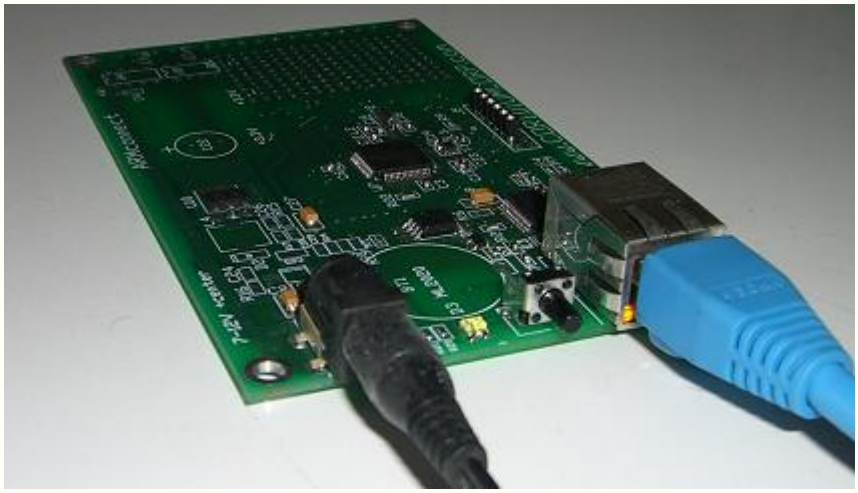
Step 2: Connect Power and Ethernet

Connect Ethernet Cable to ARMweb PCB

You should see a bluue LED on the mBed board and green LED connect light on the lower left side of the Ethernet cable indicate a connection was made. Also your hub normally has a similar type of connection indicator. There should also be some traffic indicated on the right side as the ARMweb looks for a DHCP.



For the mbed LPC1768 power can be supplied from the USB port. The picture above shows the Ethernet connection using a breakout board from Cool Components (unfortunately no longer in production). The Ethernet connection can also be made with a MAGJACK breakout board or mbed Application board from SparkFun. .



The primary power for the ARMweb is 3.3V provided from a linear regulator. The input power for the PCB may be 5V regulated supply or a 6-9V unregulated supply, with a current rating of 250 mA or more. The connector is a standard 2.5mm barrel connector with the + positive side of the supply in the center. A good choice for this power is this **5V regulated supply from SparkFun**

If you don't see the LEDs lit, check your power connections (you should see at least 6V of the + side marked on C1 with an unregulated supply or 5V with a regulated supply, and 3.3V as marked in the prototype area).

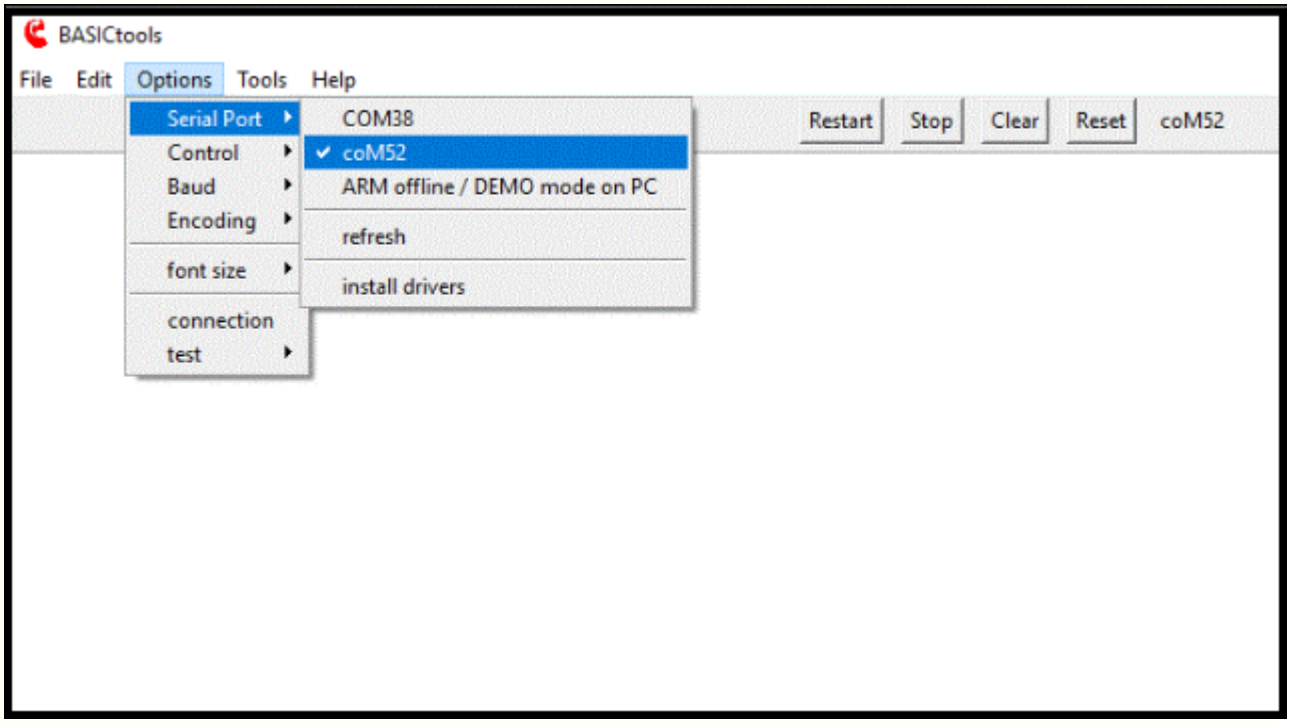
USB connection

You need a serial connection to debug BASIC programs as well as network issues. On the mbed 1768 board

this is through the USB connector on the top of the board.

This can be our **USB dongle** (specific ARMweb version when ordering separately) or some other TTL serial connection. Details on **making the USB connection** here.

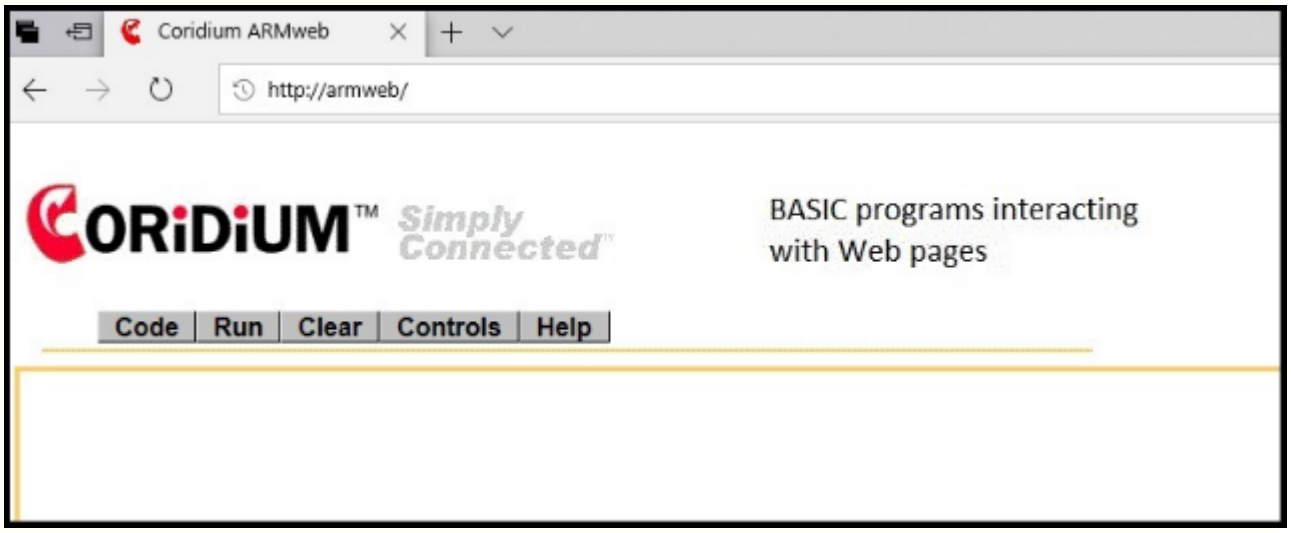
Below is the picture you should see for the mbed 1768 board, with the capital M in coM indicating an mbed serrial device.. If you do not see a coMxx serial device you may need to install the mbed serial device driver (in \Program Files (x86)\Coridium\Windows_drivers.



Finding the card on the network (larger network) -- NetBIOS name service

The ARMweb will configure itself with an IP address assigned by a DHCP server. IP addresses are the way networks organize themselves. If there is no DHCP server found, the ARMweb can provide limited DHCP services in a Diagnostic mode, assuming a single connection on Ethernet with a PC using either a hub or cross-over cable (see the Diagnostic section below).

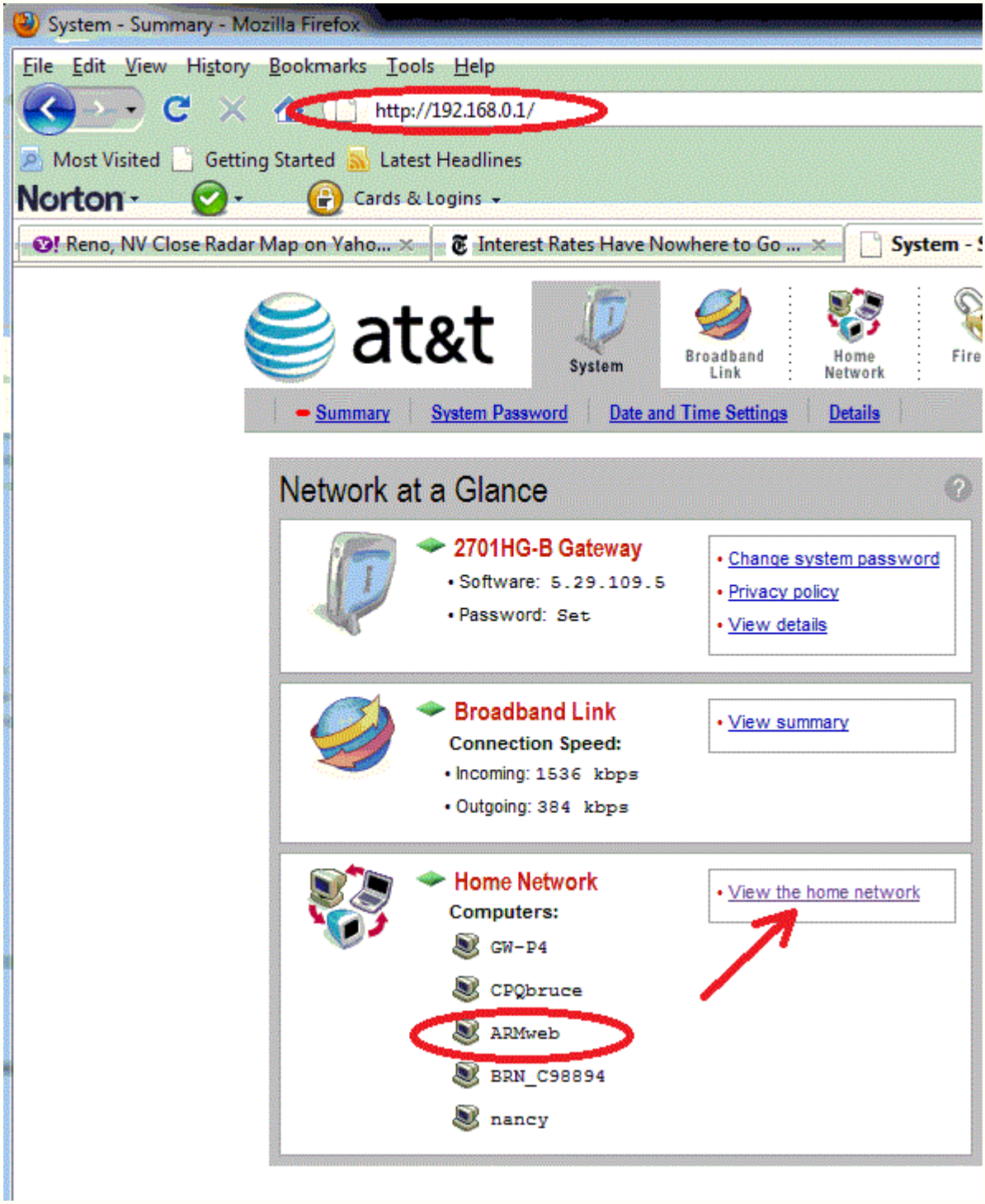
Assuming a DHCP server is available and you are running on a Windows machine, you can use the Windows NetBIOS Name Service. In which case you can find the ARMweb initially with **http://armweb**. **Note that some administrators disable NetBIOS name service** .



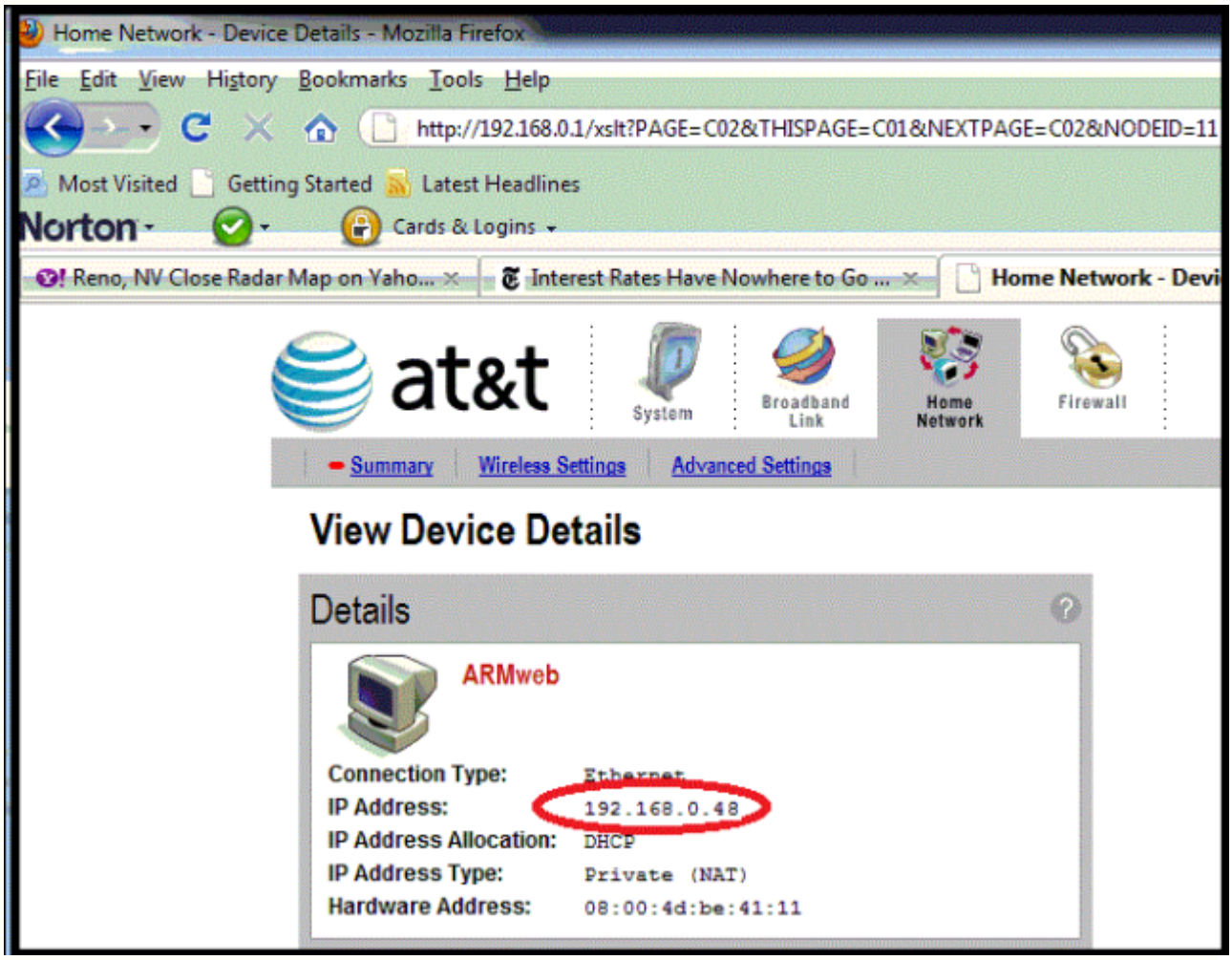
Finding the card using the DHCP server

On most home networks your DHCP will be your internet connection, and its address will share the first 3 bytes with the IP address of your PC. And the final byte being 1. The IP address of your PC is available from the control panel or by typing **IPCONFIG** at a DOS command line. Common values for the DHCP server are 192.168.1.1 or 192.168.0.1 as in the example below.

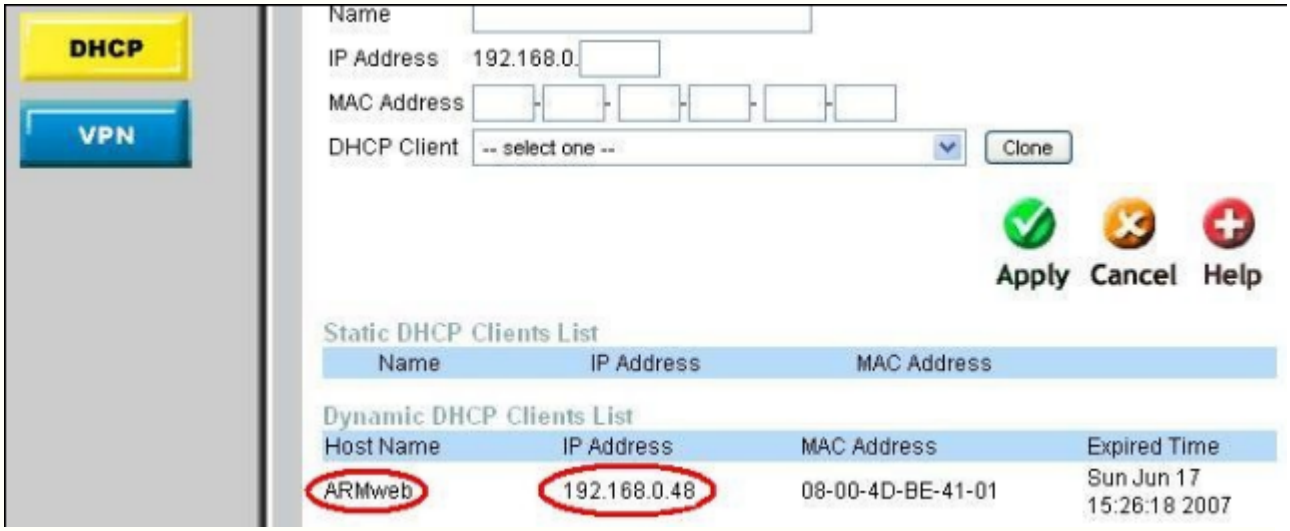
You can navigate to the DHCP server using that IP address from a browser as below.



Most DHCP servers will list client machines which have been assigned an IP address. This 2wire server indicates it on the details view of the home network, and details for the device



Another example is the display from a Dlink Firewall that is also providing DHCP services.



So in this case the ARMweb can be found at **http://192.168.0.48**

Now that you have the IP address of the ARMweb

You can go onto configuration settings, or writing simple programs using BASICtools.

But for this web interface navigate using a browser to **http://w.x.y.z** where w.x.y.z is the IP address of the ARMweb (192.168.0.48 in this example).

DHCP assignment vs. fixed IP addressing

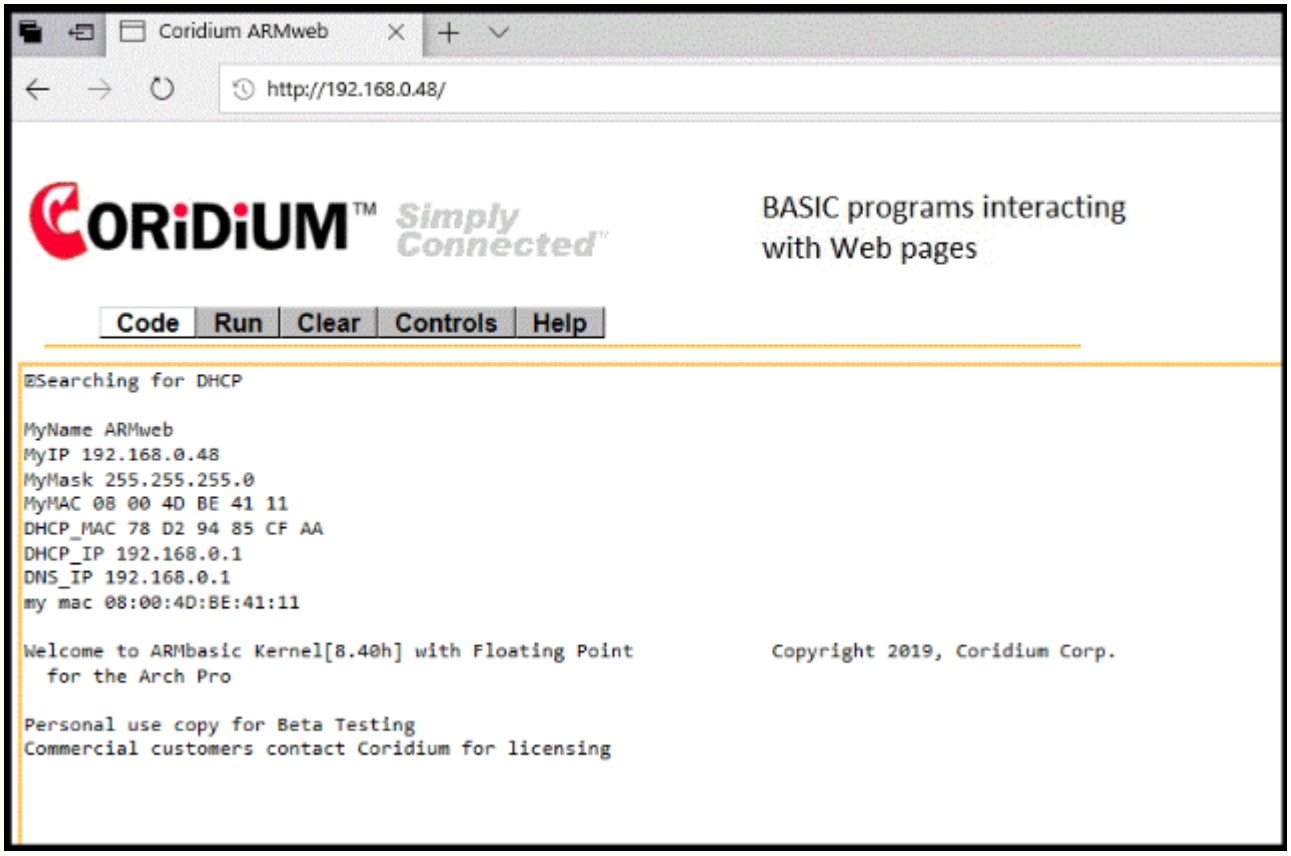
We routinely allow the DHCP server to assign an initial address, but will can a fixed IP address in the final setup. One reason to assign a fixed IP, is to make sure that the IP address assigned never changes, for instance following a power outage. **Details** on setting a fixed IP address.

On to Step 3

armweb.htm PAGE

Description

This page is the main control page for the ARMweb. It can be accessed at the armweb.htm page.



Code :
The default is this page, which mirrors the output to the USB connection of BASICtools..

Run :
This button will run a previously loaded user BASIC program. This function is disabled when a user BASIC program is running. You can only STOP a running program from the BASICtools USB connection.

Clear :
This will erase any user program. This function is disabled when a user BASIC program is running .

Controls :
This accesses the **Controls Page**. It will be disabled when a user BASIC program is running.

Help:
Linked to the Coridium Web Site help files on the internet.

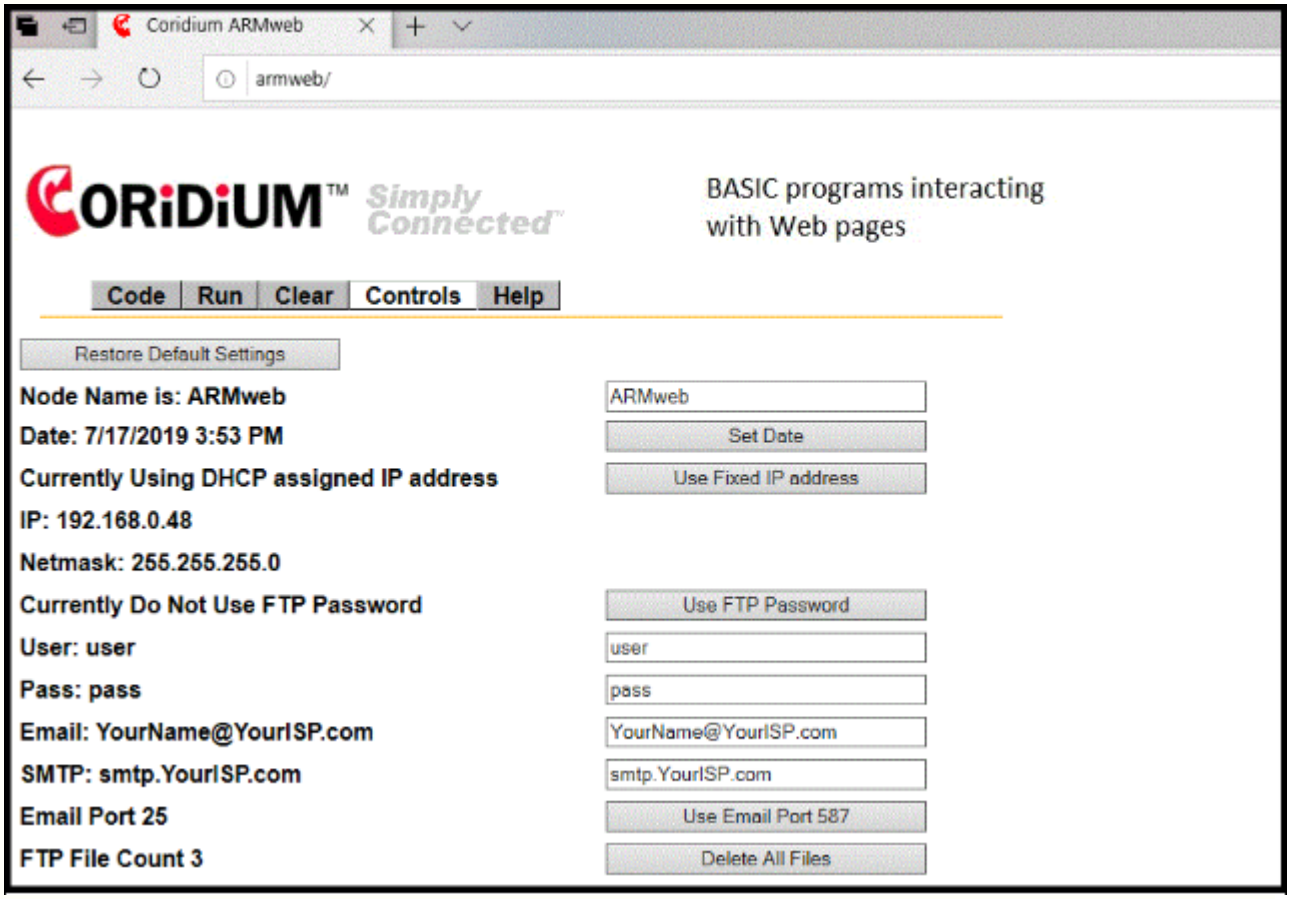
See also

- **UDP Services**
- **FTP Services**

Controls PAGE

Description

This page controls the ARMweb. When the user program is running this page is disabled for security.



Node Name :
The ARMweb's default node name is ARMweb, when you change this main page, the ARMweb will adopt that entry as its node name. The node name will be seen by DHCP servers, as well as the response to **node ping** .

Set Date:
If your browser supports the JavaScript system-time functions this button will access the systems date and time and update the ARMweb registers.

DHCP :
The ARMweb node can either use a DHCP to obtain its IP address, or you can set it to a fixed IP address. The default is to accept a DHCP generated IP address.

We routinely allow the DHCP server to assign an initial address, but will use a fixed IP address in the final setup. One reason to assign a fixed IP, is to make sure that the IP address assigned never changes, for instance following a power outage.

Passwords:
The ftp service can use a password (the default is none or user/pass and password checking turned off). If you do set a different username and password also click the Use Password button.

Email:
The MAIL statement can send an email to the address and server set by these fields. The SMTP server for

name@somewhere.com is normally smtp.somewhere.com .

FTP file count:
This keeps track of the number of files in the FTP area. It also provides a mechanism to clear and reset the FTP space..

See also

- **UDP Services**
- **FTP Services**

CGI Services

Syntax

FUNCTION CGIIN AS STRING

Description

CGIIN functions like a serial channel to the web server. When someone accesses the web page that creates a CGI event (like a button push, or text entry) that data will be sent to a buffer that can be read from the BASIC program.

If no GET request has been made the *string* returned will also be an empty string.

When the ARMweb is accessed from a web page, if the web page contains a ? in the address, data following the ? is passed to the CGIIN routine. There is only one 256 byte buffer available, and that buffer will be available until it is read by a CGIIN, or another CGI request is made. CGI events are only processed when the user's BASIC program is running.

This function requires version 7.36 of the firmware.

Example

```
dim CGIinput(255) as string
 '...

while 1
  CGIinput = CGIIN   ' assumes the form is http : // ... /Input?=#  per the example in CGI example

  if CGIinput(0) then print  CGIinput  ' display on the terminal window -- for debugging

  select  CGIinput(6)
  case "0"
    ' do nothing
  case "1"
    io(16) = 1
  case "2"
    io(16) = 0
  end select
 '...

   CGIinput = ""   ' erase the input line
loop
```

See also

- CGI example
- Web Basic
- FTP Services

Web page Programming

Building a web page on the ARMweb is much like any other web server. An HTML web page is ftp'd to the ARMweb, and it can communicate to a BASIC program running on the ARMweb. The BASIC program can be controlling attached devices. Control or data can be fed back through the web page interface. All sources for this example are at **www.coridiumcorp.com/files/WebBASIC.zip**

Use standard HTML and JavaScript

Build your web page in standard HTML and JavaScript. Include text and graphics in the web page, here the web page includes an image of a logo. In this example user actions on the web page are fed back with a CGI script using JavaScript button action. It also can display a state of the ARMweb, here an LED by running a small BASIC program (included into the HTML).

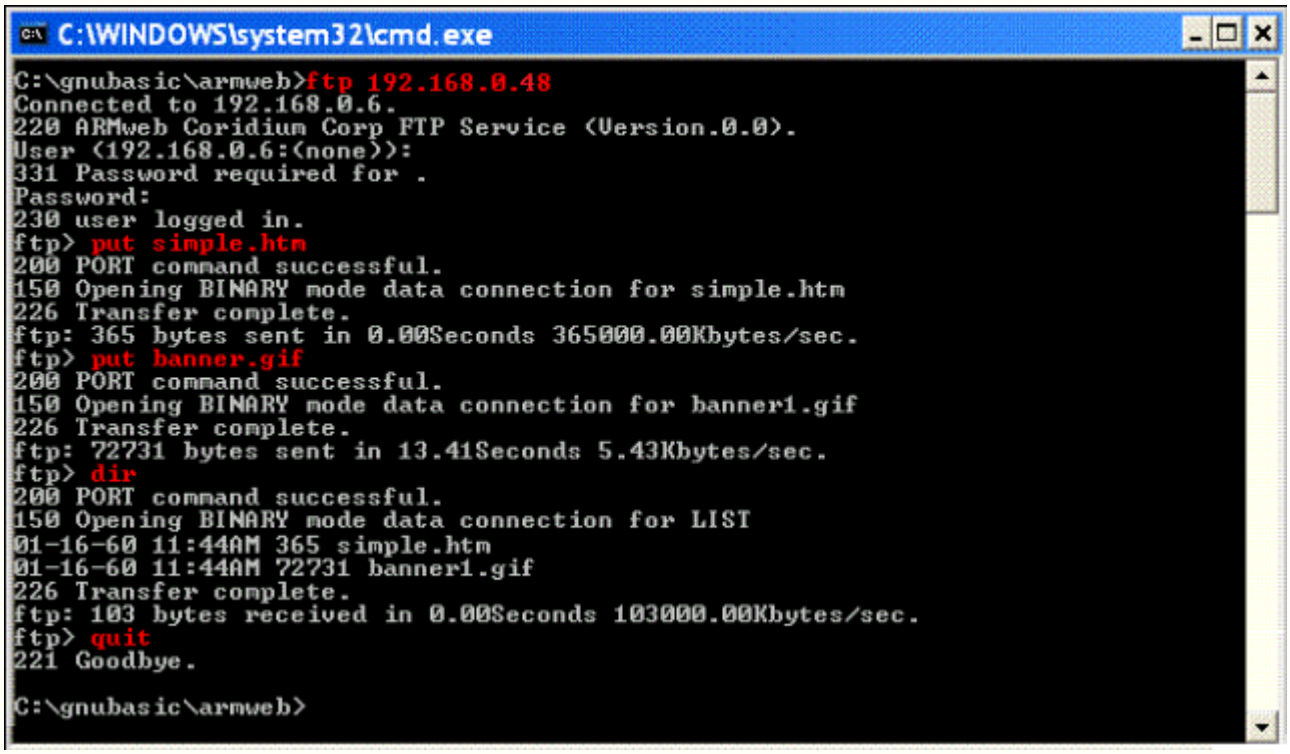
```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/strict.dtd">
<html><head>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
<title>Coridium ARMweb Example</title>
<script type="text/javascript">
function sendValue(s){
  document.location.replace("index.htm?Input="+s);
}
function kd(evt){
  if(evt.keyCode==13)
    sendValue("Text "+document.getElementById("InputText").value);
}
</script>
</head>
<body>
<div style="position:absolute;top:140;left:140">
<img src="banner.gif" style="position:absolute;top:50;left:150" alt="banner">
<br><br>
<br><br>
<br><br>
<br><br>
<br>LED is
<?BASIC
    if IN(16) then print "OFF" else print "ON"
?>
<br><br>
LED controls:    
<br><br>
<div>
<input type="button" value="OFF"    onclick="sendValue('1');">    LED off<br><br>
<input type="button" value="ON"     onclick="sendValue('2');">      LED on<br><br>
<input type="button" value="FLASH"  onclick="sendValue('3');">    2 Hz on LED for 2 seconds<br><br>
<input type="button" value="RAMP"   onclick="sendValue('4');">      Ramp thru PWM on LED<br><br>
<br>Type the message into the input box and press Enter or click a Button below<br>
<input type="text" id="InputText" size="32" maxlength="32" onkeydown="kd(event)"><br><br>
<input type="button" value="UDP"    onclick="sendValue('5');">       Send UDPOUT Get UDPIN<br><
<input type="button" value="EMAIL"  onclick="sendValue('6');">    Send Email<br><br>
</div>
</div>
</body>
</html>
```

run BASIC program from the web page

display data to web page

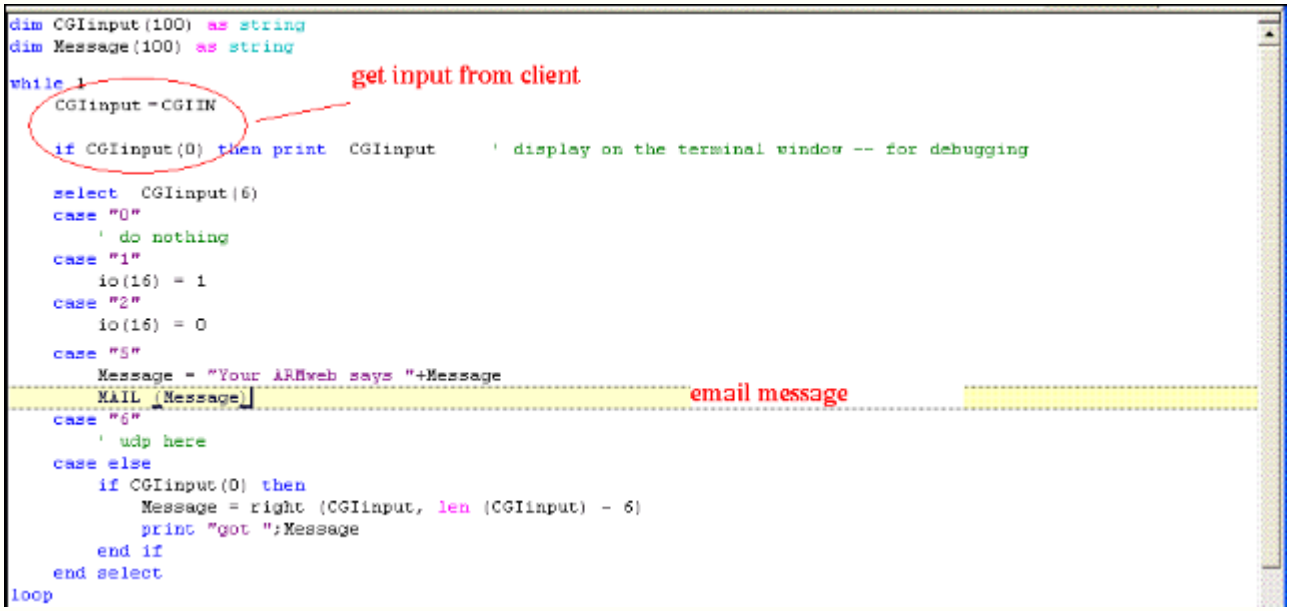
Upload to ARMweb using FTP

No special tools to compile your page, just upload it to the ARMweb. Here the 2 files used for the web page, the main HTML and the banner image.



Interact with a BASIC program running on the ARMweb

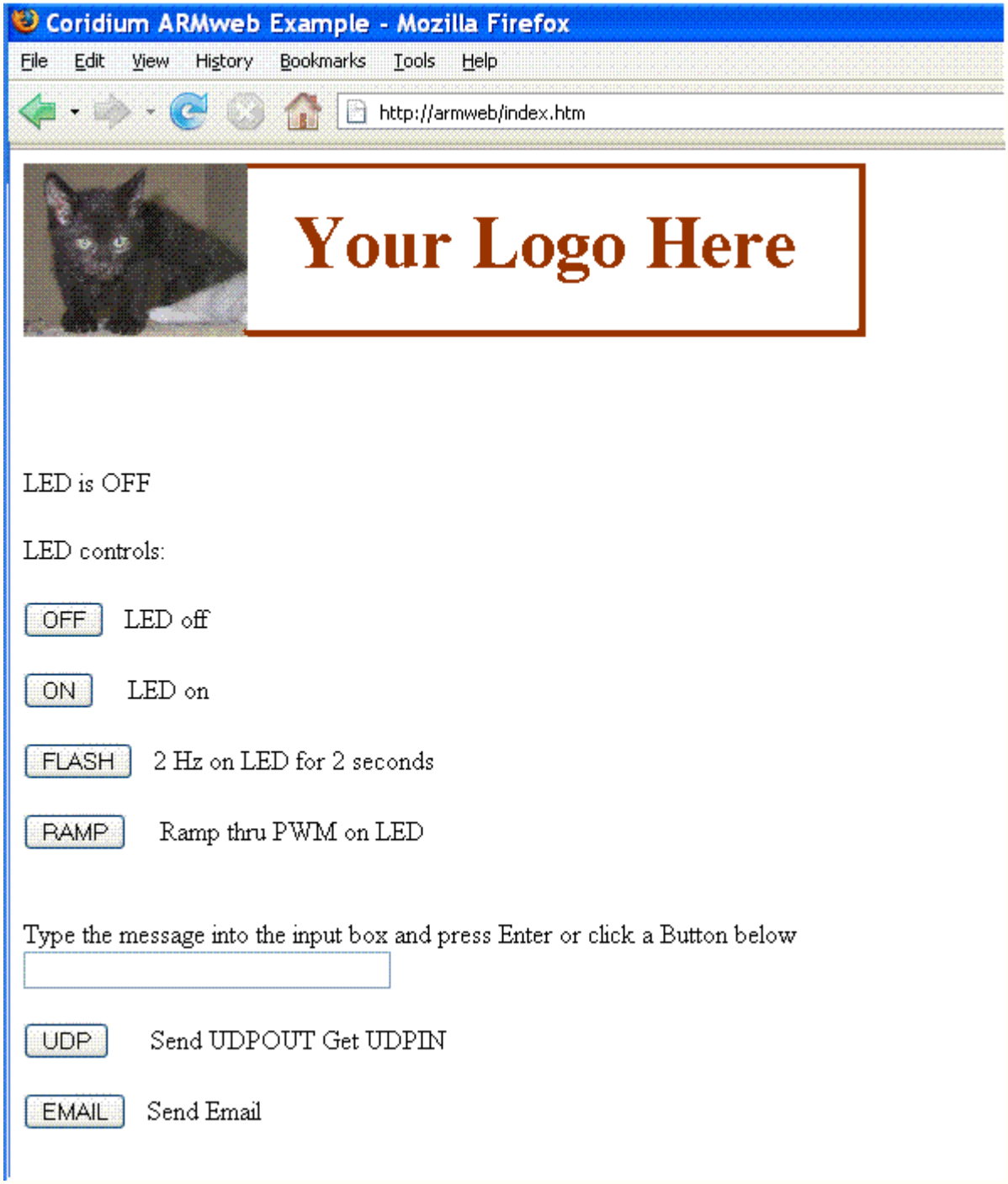
The web page can send data to the ARMweb using CGI that can be read in your BASIC program. It can parse these requests and perform various actions. This allows you to control an ARMweb across the room or anywhere on the internet.




```
dim CGIinput(100) as string
dim Message(100) as string

while 1
    CGIinput=CGIIN

    if CGIinput(0) then print  CGIinput      ' display on the terminal window -- for debugging

    select  CGIinput(6)
    case "0"
        ' do nothing
    case "1"
        io(16) = 1
    case "2"
        io(16) = 0
    case "5"
        Message = "Your ARMweb says "+Message
        MAIL [Message]
    case "6"
        ' udp here
    case else
        if CGIinput(0) then
            Message = right (CGIinput, len (CGIinput) - 6)
            print "got ";Message
        end if
    end select
loop
```

Your Web application running on an ARMweb

This is what will appear on the web, served by the ARMweb.



See also

- CGI services
- Web Basic
- FTP Services

FTP Services

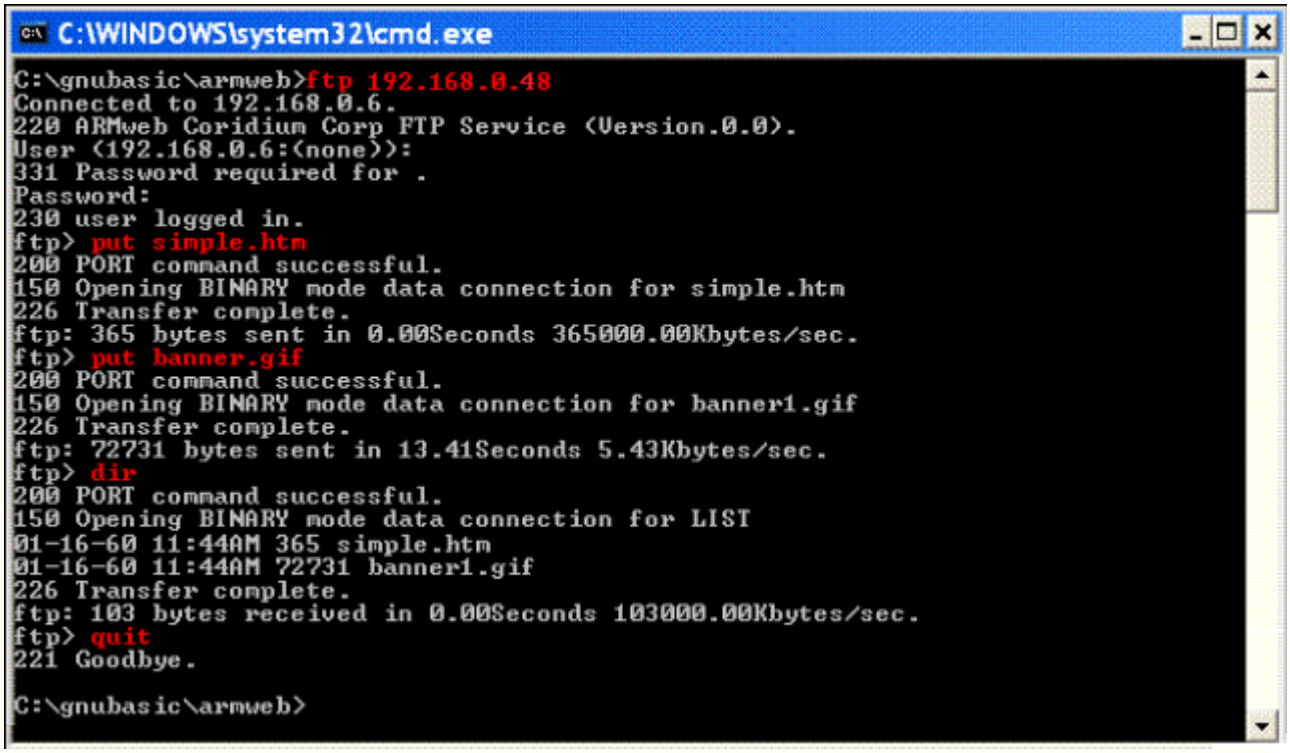
LPC1768 contains a small File System to store additional web pages.

- maximum size of all files combined must be less than 224KB
- there must be less than 76 files
- File names must be 23 characters or less
- File names are case sensitive
- there is only 1 directory and sub-directories are not supported
- the main HTML file must be of the form filename.htm
- ftp put, delete are slow due to the Flash writes, which can take 20 seconds or more
- any BASIC program must be **STOP** 'ed, otherwise ftp will not log you in, or will ignore any requests

By default password protection is not used for FTP.
If logging in from a command prompt simply press enter when asked for Username and Password.
If password protection is desired go to the Controls page of ARMweb and select Use Password.
NOTE : The default user name is "user" and password "pass", when enabled.

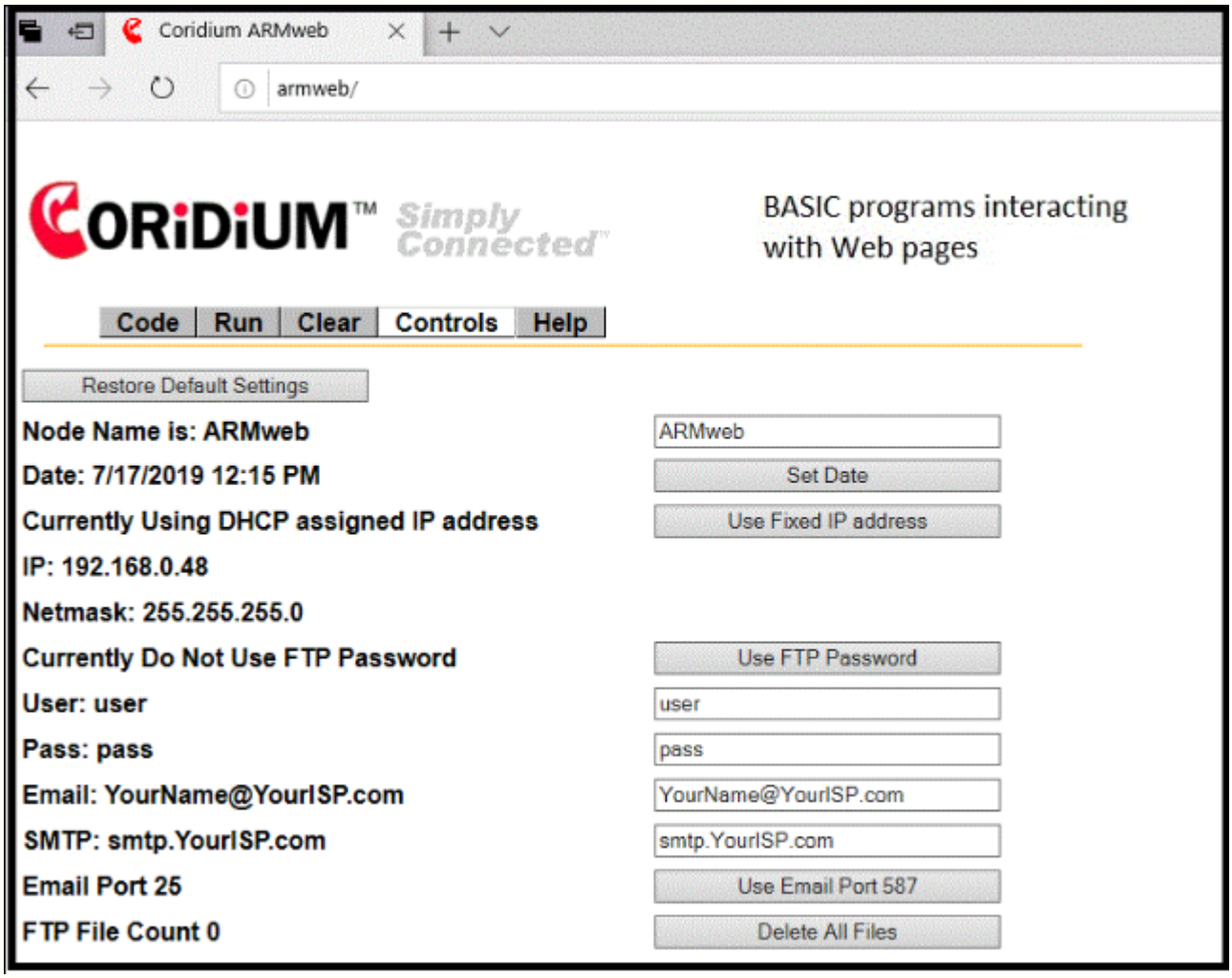
Change these as desired and reset the node to apply changes.
Also on the Controls page, a file from the system may be chosen as the Main Page.
This then becomes the default page when browsing to **http://ARMweb** or to **http://Nodes-IP-address** .

The current implementation is aimed at using the simplest ftp interface, and it may not work with more complex ftp programs or browsers doing ftp. We recommend using the Windows ftp from the DOS command prompt. Here is a sample session used to copy the files for a web page to the ARMweb (192.168.0.48 was assigned by the DHCP, and the default username/pass were used (either enter user and pass, or just hit enter both times).



After the above ftp session a web page has been setup on the ARMweb. It can be viewed at **http://192.168.0.48/simple.htm**.

To make that you are communicating with webBASIC start from the main page..



From then on, when you navigate to **http://192.168.0.48/simple.htm** .

See also

- **Web Basic**
- **Web Services**

HTTP Services

Syntax

SUB HTTP(IPaddress, CGIstring AS STRING)

Description

This allows a BASIC program to do a GET or POST to server somewhere on the web. The BASIC program needs to know the IP address of the server and that is passed to HTTP as a 32 bit value.

The CGI string should include either GET or POST followed by the web address (can be index.html) and may be followed by a CGI request which typically follows a ?. For example this is the same as **http://192.168.0.80/index.html?i=5** which could be a local server on your network. This also works for any IP on the web as well.

The purpose of this is to send a CGI message to a server, but the first 256 bytes that the server returns for the request will be placed into the CGI buffer. Any data after that will be dropped. As this is shared with the CGIIN buffer, you should not be trying to use both as one or the other will be lost.

This function requires version 7.58 of the firmware.

Example

```
DIM A_STRING(256) as string
DIM IP_ADDR as integer
DIM i as integer

' IP Address of my local web server
IP_ADDR = (192<<24)+ (168<<16) +(1<<8) + 4

i = 123 ' some value to send in CGI

' Build a request string, could also be a POST
' In this case include CGI
A_STRING = sprintf("GET /index.htm?i=%d %c%c%c%c",i,13,10,13,10)

' Example read a local page server
HTTP (IP_ADDR, A_STRING)
```

See also

- **CGI example**
- **Web Basic**
- **FTP Services**

MAIL

Syntax

MAIL (*string* **)** ' does not use authentication,

MAIL (*message*, *recipient*, *user_name, pass_word*) ' takes 4 strings and uses authorization

Description

In the first form MAIL will send an email to the address specified in the Controls page. This email is limited to an address on your mail server/ISP, as it is piggybacking on the authentication of your internet connection.

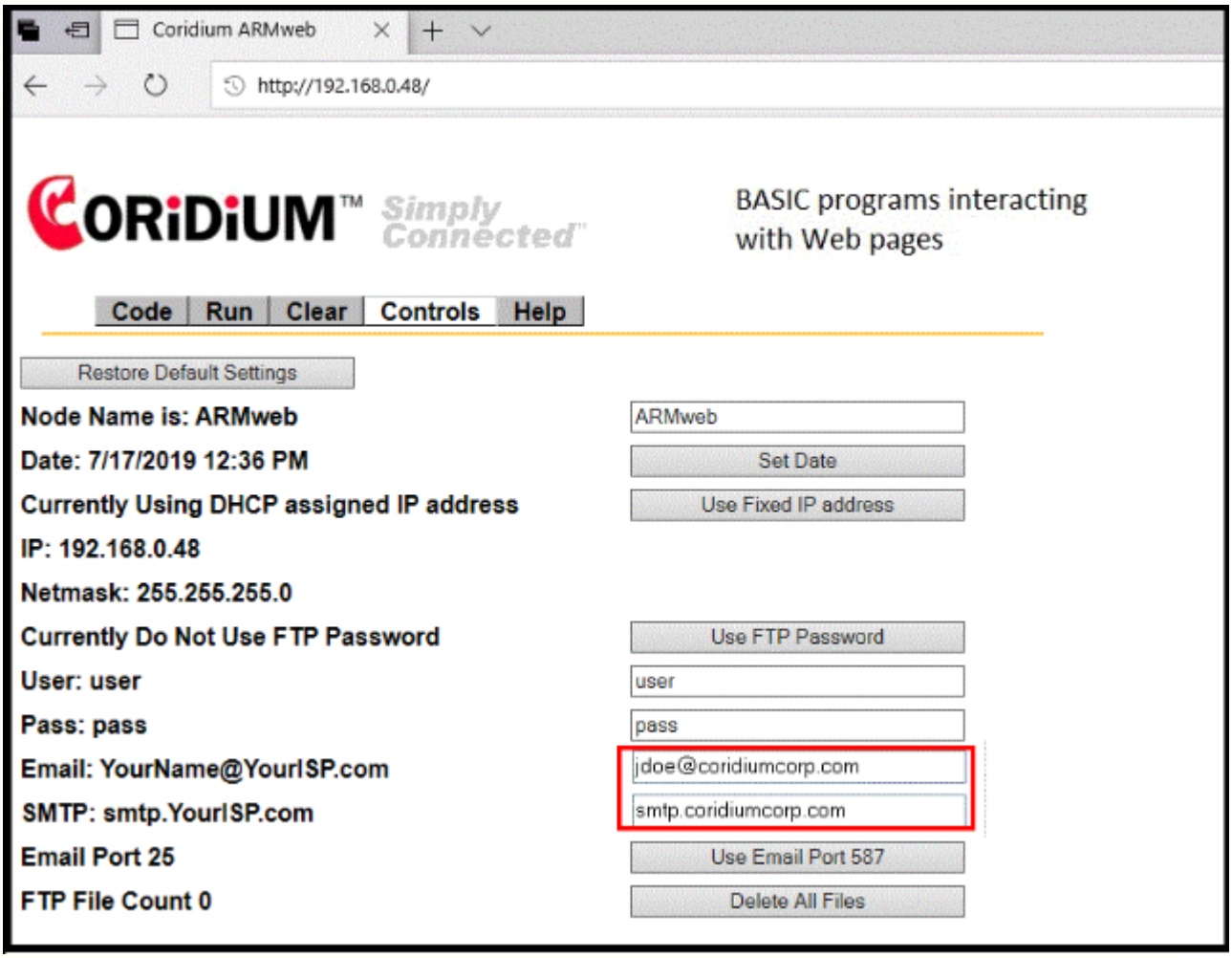
So you can send an email to yourself.

To use email authentication use the second form, in this case it uses the SMTP address of the controls page, and logs in using the *user_name* and *pass_word*. The email *message* will be sent to *recipient* . *recipient* requires the full address like somebody@somewhere.com. *user_name* should NOT include domain.com as that is set in the Controls page smtp server.

In all cases email is limited to 1 email sent every 10 seconds.

Setup

Go to the Controls web page of the ARMweb.
Enter your email address in the Email input box and press enter.
Enter your SMTP server's address in the SMTP input box and press enter.
Example:
jdoe@coridiumcorp.com
smtp.coridiumcorp.com



Reset the node to apply the changes.

The smtp server used must service the email address chosen.

The maximum size of the *MessageList* which will be contained in the email Body is 255 bytes.

Example

```
DIM Astr(10) as STRING
Astr= "the current temperature is "+STR(temperature)
MAIL (Astr)
  '...

MAIL ("operator intervention needed")  ' send a short email to yourself
```

```
MAIL("wake up out there","someone@somewhere.com","my_user_name","my_password")
```

See also

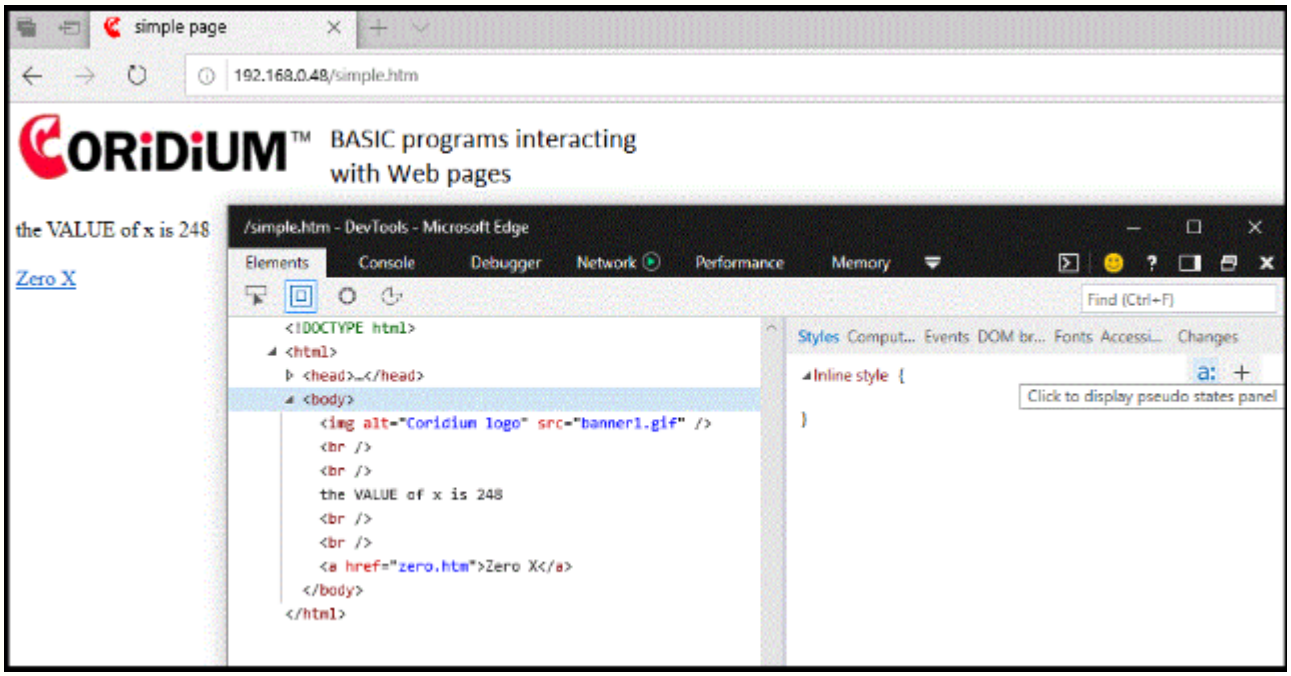
- **UDP Services**
- **FTP Services**

Web Services

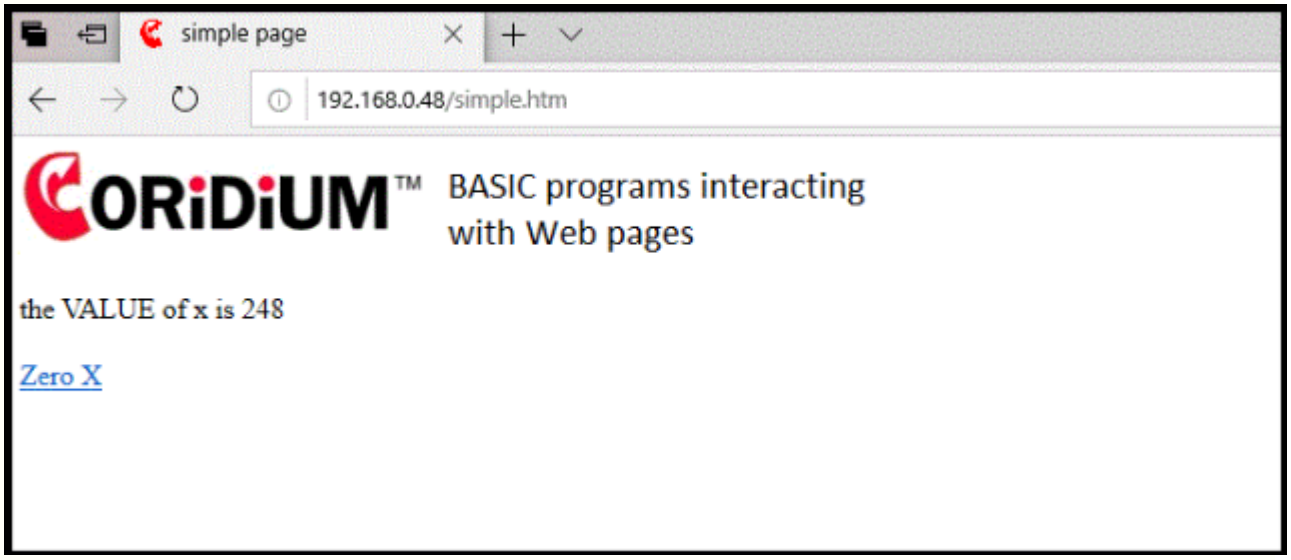
webBASIC may be accessed from any web browser by going to http://ARMweb or if the node's IP is known http://192.168.xx.yy.
From here users may enter code a line at a time, download basic files or access all features of ARMweb.

Building a web page

This is not the venue to teach web page design, but a simple example will be presented here. Various ways can be used to build a web page from FrontPage, DreamWeaver, Mozilla-Composer, to your favorite text editor. This page is built with 2 files, the main page and an image file (banner1.gif). This is the sample source built as displayed in Mozilla Composer.



Once you've built a page, use the **FTP Services** to upload it. Then you will be able to view the page as the main page for the ARMweb-



webBASIC first scans the ftp space for a file named in the browser request (simple.htm in this case). If not

found there it will attempt to find it in the file system on the SD card.

See also

- **Web Basic**
- **FTP Services**

Web BASIC

BASICweb allows for basic code to be embedded in the web pages much like PHP or JavaScript
Global variables may be accessed from the User BASIC program. The intention is not to place your BASIC code in this program, but to interact with your program from a web page. For example if you put an endless loop in the <?BASIC embedded in the web page, the web page will hang. <>BASIC can not call BASIC functions or subroutines in the user code.

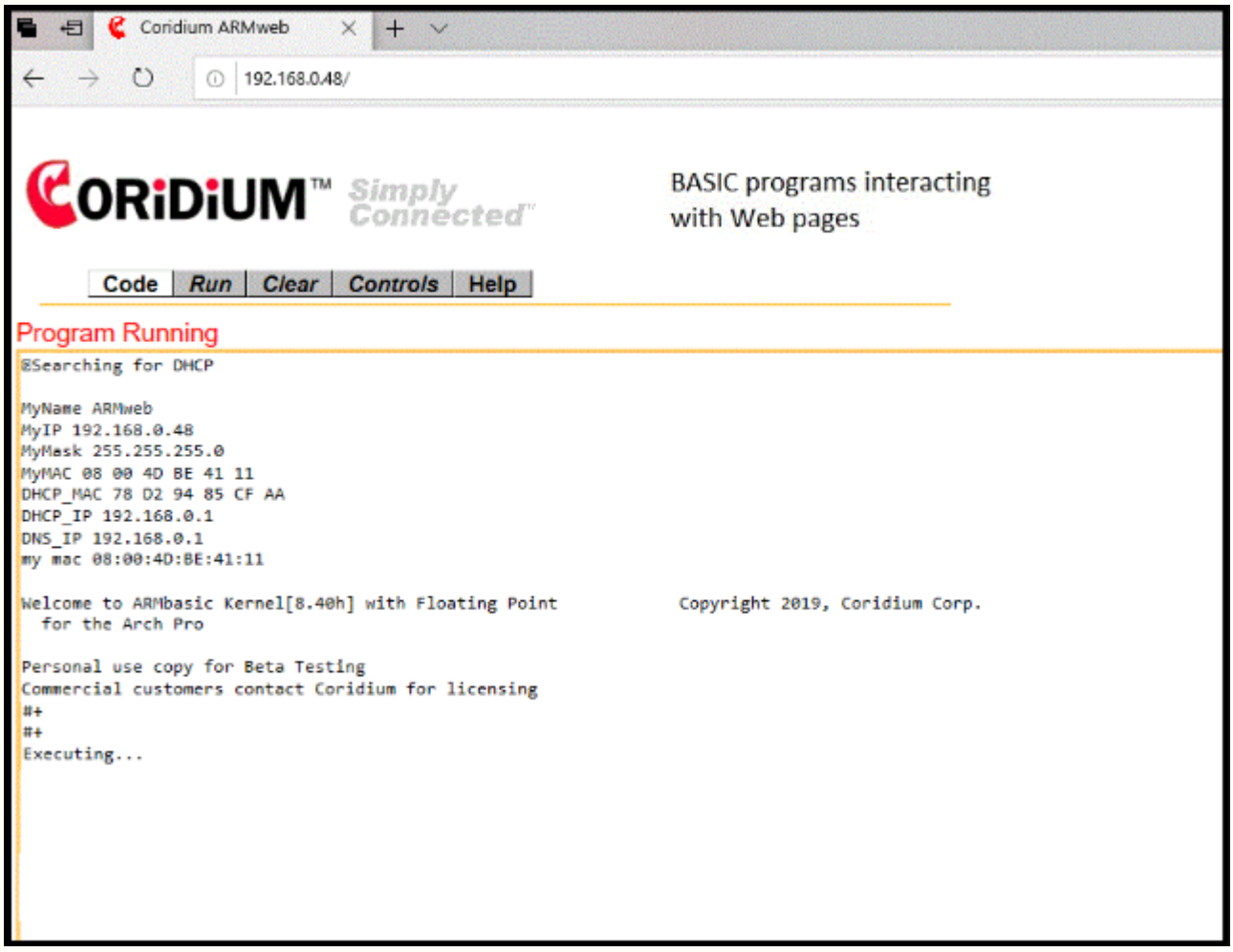
Example: Add reading and setting a User variable through the web page.

Here is a modified version of the web page loaded from **Web Services** and a second webpage that resets the value of X

source for simple.htm

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
  <meta content="text/html; charset=ISO-8859-1" http-equiv="content-type">
  <title>simple page</title>
</head>
<body>
<?BASIC
  print "the VALUE of x is ";x
?>
<br>
<br>
<a href="zero.htm">Zero X</a>
</body>
</html>
```

Now from Code page of ARMweb enter the following program (its can be accessed at armweb.htm)




```
WHILE 1
  IO(55) = X AND 1   ' Flash the LED
  X = X + 1
  WAIT (500)
LOOP
RUN
```

The program is running and the value of X is incremented every half second.
Browse to **http://ARMweb/simple.htm**

Refreshing the browser will show the updated values of X.

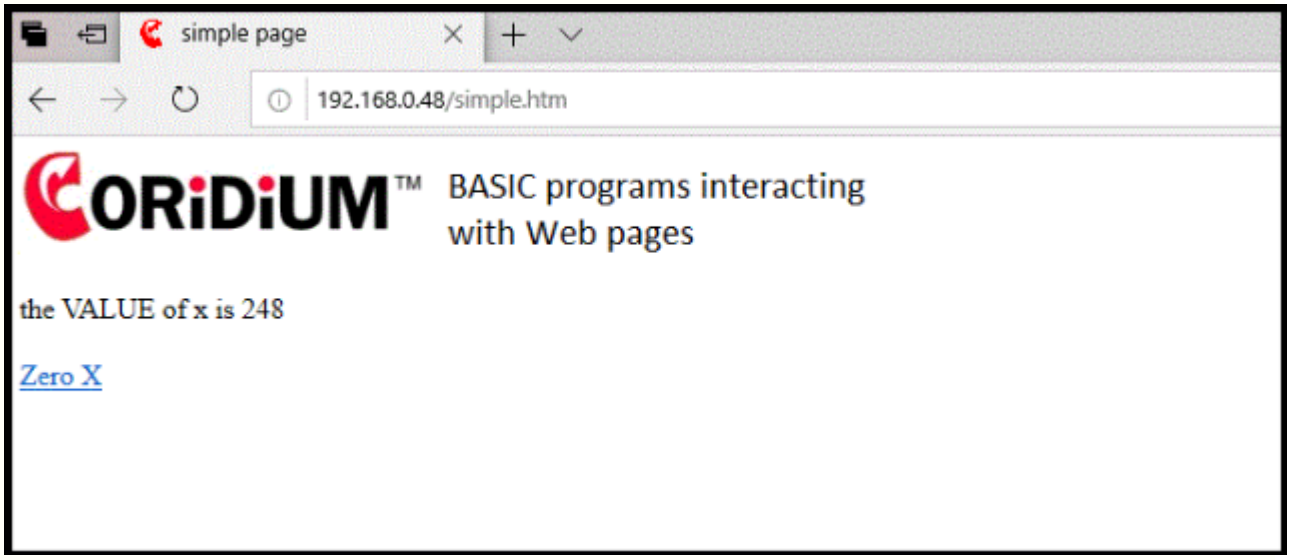
Example: Executing a BASIC command from a web page.

To the above example we will add a method to set the variable x to 0, by accessing another web page that runs a BASIC program. This may also be accomplished with CGI, see the **CGI examples** .

First add an anchor to another web page that will be served by the ARMweb

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
  <meta content="text/html; charset=ISO-8859-1" http-equiv="content-type">
  <title>simple page</title>
</head>
<body>
```

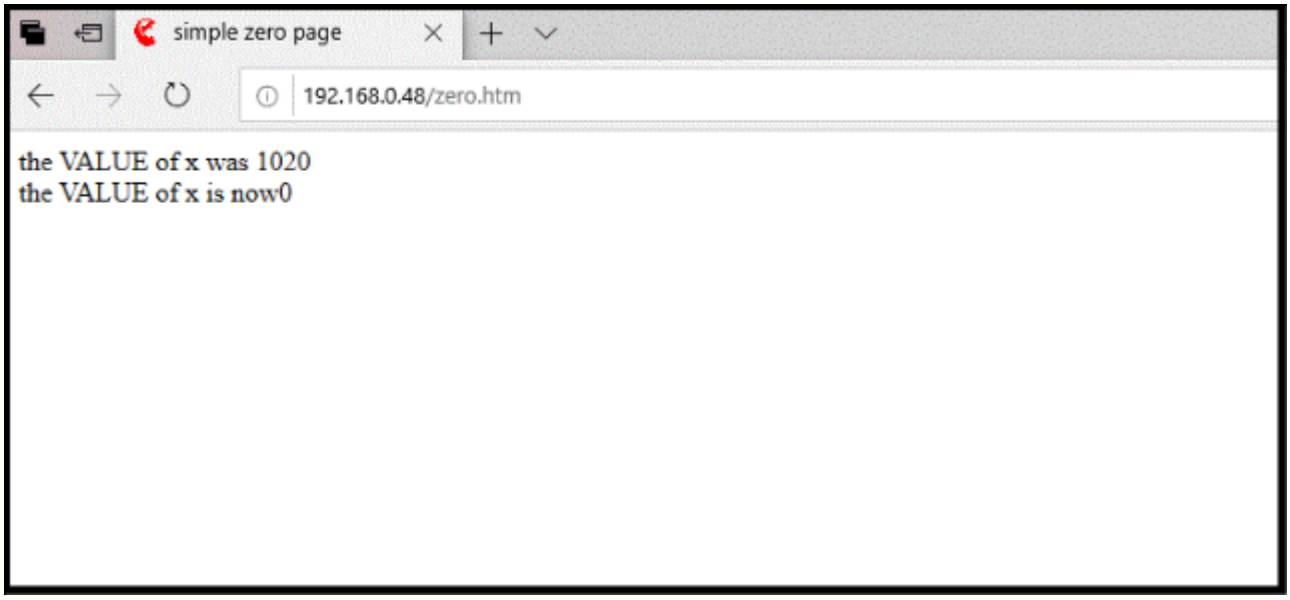
```
<?BASIC
  print "the VALUE of x is ";x
?>
<br>
<br>
<img style="width: 500px; height: 80px;" alt="planes" src="banner1.gif">
<a href="zero.htm">Zero Y</a>
</body>
</html>
```



Next create another page zero.htm that executes a very short BASIC program to zero the variable y. This page also returns to the original page.

source for zero.htm

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
  <meta content="text/html; charset=ISO-8859-1" http-equiv="content-type">
  <title>simple zero page</title>
</head>
<body>
<?BASIC
  print "the VALUE of x was ";x;"<br>"
  x = 0
  print "the VALUE of x is now";x
?>
</body>
</html>
<
/PRE>

<  /FONT>
```



Some notes, currently errors in the BASIC embedded in HTML are not flagged, so be careful, but they will be visible to the console of BASICtools over the USB connection.

The meta tag highlighted will return you to the original page after 1 second, though not all browsers support this.

For a CGI method to accomplish the same see the **CGI examples** .

WEB BASIC limits

The BASIC code between <?BASIC and ?> is limited to 1450 characters.

The output from a web BASIC program must not exceed 1460 characters.

If the web BASIC contains an infinite loop, the server will hang waiting for the loop to complete.

The Pre-processor is not available to WEB BASIC inside the HTML. That includes #include, #ifdef, #define...

The code between <?BASIC ... ?> and the output is also sent to the UART0 (and via USB Dongle or mbed USB serial port to BASICtools).

See also

- **Web Services**
-
- **FTP Services**

UDP Services

Syntax

FUNCTION UDPIN (*PORT*) AS STRING
FUNCTION LASTIP

SUB UDPOUT (*IP, PORT, String* **)**

Description

UDPIN and UDPOUT read or write a packet of data on the network using UDP protocol.
The IP address which the data is sent to or received from is designated by *IPa.IPb.IPc.IPd* eg. 192.168.0.122, which is packed into a 32 bit word.
Broadcast addressing is not supported for UDPIN or UDPOUT.
The port is designated by *PORT*.

NODE PING - A special feature of ARMweb listens on port 49152 (0xC000) for any UDP broadcast. The node will then reply with its Name and IP to identify it on the network.
According to iana.org, The Dynamic and/or Private Ports are those from 49152 through 65535.
User applications should use ports above 49152 to avoid other conflicts.

UDPOUT automatically sets the node to listen on the given port.
This allows any responses to be buffered and subsequently read with UDPIN.
If an application wishes to just read UDPIN it is advised to call UDPIN once to clear any buffered data first.
Each call to UDPIN will wait up to one half second to receive data or return immediately upon receipt.
If no data was read the port is left open for reading, any incoming data will be buffered and available for subsequent calls.
The maximum size of the returned by UDPIN is 255 bytes.

This function requires version 7.36 of the firmware

Example

```
'send a string to UDP port 50000 of 192.168.0.122
UDPOUT ((192<<24)+(168<<16)+(0<<8)+122, 50000, "9876543210")

DIM Astr(100) as STRING

'sit and listen for any incoming UDP on port 50000

Astr = ""

WHILE Astr(0) = 0

  Astr = UDPIN  (50000)
```

x = LASTIP
LOOP
PRINT Astr; " from "; x>> 24; "."; x>>16 and 255; "."; x>>8 and 255; ".";x and 255
Executing...
ABCDEFGHIJ from 192.168.15.122

See also

- **Web Basic**
- **FTP Services**

ARMweb setup

Getting Started

- **Power on behaviour**
- **Firmare Update** (mbed BASIC binary)

Power On Behavior

Initial Power on conditions

On power up all pins are tri-stated on the ARMweb.

If P0.14 is low during reset, the NXP ISP (in system programming) routine starts. This is how we load firmware.

If P0.14 is high the Coridium firmware starts up. It looks for a cable plugged into the PHY. If there is none the board will not start the user program, but drops into the BASIC firmware monitor.

If there is an Ethernet cable connected, the ARMweb tries 5 times to get an IP address from a DHCP. There is a pause of 5 seconds between each try.

If no DHCP responds, and the ARMweb has never seen a DHCP response it goes into a mini-DHCP server mode. In this mode a PC with a cross-over cable may be directly connected to the ARMweb (or a hub and standard cables). The ARMweb will act as DHCP server to the PC. This mode is for diagnostic purposes and is NOT intended for normal use.

If the DHCP responds the ARMweb accepts the IP address and the boot process continues.

The ARMweb waits 0.5 seconds for an ESC character, which if received on UART0 stops the user program from running. If no ESC is received the process continues.

If the ARMweb security setting on the controls page has been set, the user program will start. If it is not set it will drop into the BASIC firmware monitor.

Restoring Factory Defaults

Press and hold the button on pin P0.7 during RESET (on J8.9 in DINkit). The firmware will erase all user programs, settings and files in the ftp area.

Regaining control with BASICtools

Hit the STOP, which disables web access and enters the monitor. Type in a small program that terminates, which will erase the looping program. Hit RESET which will drop back into a non AUTORUN state.

BASIC Boot Loader serial commands

When the user program is not running or not at a STOP, the BASIC firmware monitor is functioning.

The ARMweb has a full compiler ready to compile BASIC programs line by line. This can be used with the TclTerm terminal emulator or the web interface of the ARMweb. When running BASICtools programs are compiled on the PC and downloaded to the ARMweb. The ARMweb also supports the commands used by all the others, and these are used to load and control BASIC programs-

- **:**20.... Coridium hex format line, copy this data into the code buffer
- **:**00000001FF write the code buffer into the appropriate Flash space
- ARM responds by sending XOFF, writing the Flash, then sends XON followed by **+**
- **?** get vectors for ARMbasic compiler running on the PC
- **^** launch any user program contained in the Flash space
- @HHHH dump memory starting at HHHH which is a hex value without a preceding $
- **@** dump memory starting from last address + 32
- **"**message echo message back
- **!** reserved
- ctl-C or ESC on reset run the BASIC bootloader rather than the User program

Firmware Update

ARMweb allows for firmware updates in the field. The following steps should be used.

After version 7.36, firmware versions will require update via USB. Note what com port the USB is configured as, you will need that information below.

Download load21xx.exe from the **Yahoo ARMexpress Forum Files** section.

Also download the latest ARMweb firmware. The name will be of the form webXXXX.hex. As of March 2011, web0746.hex is the latest release.

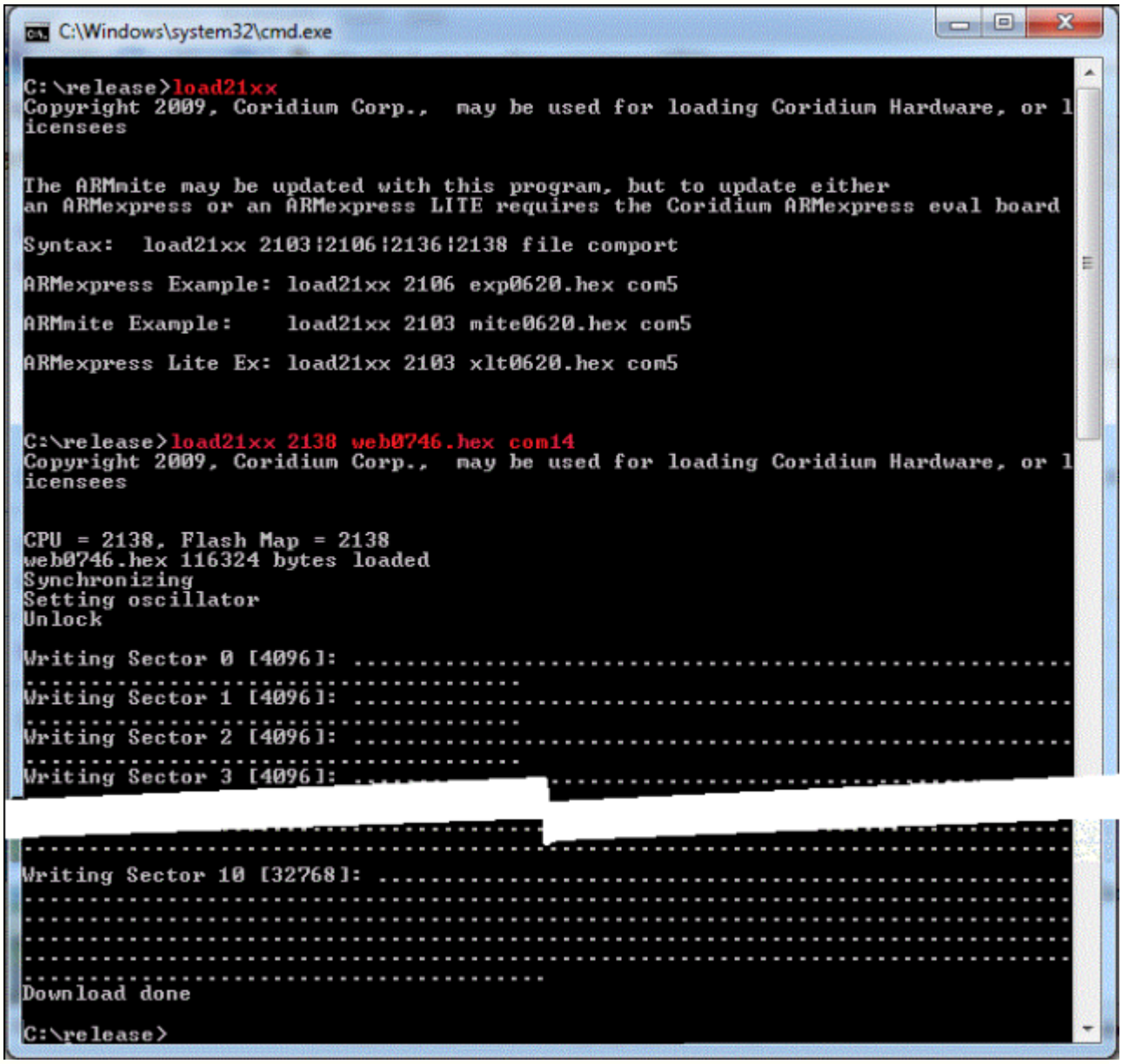
From a command line run load21xx.exe.

It will prompt you for the proper format of the command to update, the CPU is a 2138 -- see below for an example session.

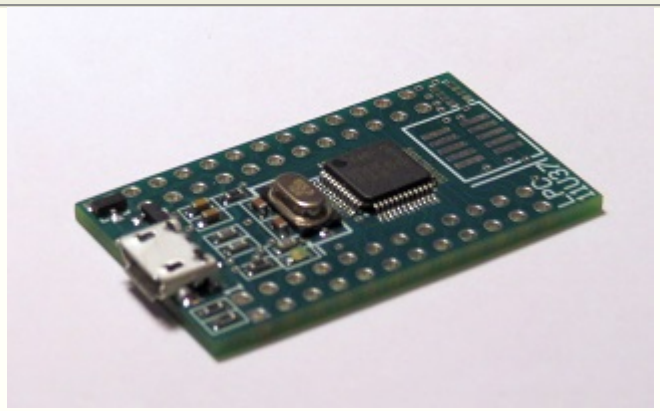
Restoring Factory Defaults

Press and hold the button on pin P0.7 during RESET (on J8.9 in DINkit). The firmware will erase all user programs, settings and files in the ftp area.

firmware update session --


C:\Windows\system32\cmd.exe
C:\release>load21xx
Copyright 2009, Coridium Corp., may be used for loading Coridium Hardware, or l
icensees
The ARMmite may be updated with this program, but to update either
an ARMexpress or an ARMexpress LITE requires the Coridium ARMexpress eval board
Syntax: load21xx 2103¦2106¦2136¦2138 file comport
ARMexpress Example: load21xx 2106 exp0620.hex com5
ARMmite Example: load21xx 2103 mite0620.hex com5
ARMexpress Lite Ex: load21xx 2103 xlt0620.hex com5
C:\release>load21xx 2138 web0746.hex com14
Copyright 2009, Coridium Corp., may be used for loading Coridium Hardware, or l
icensees
CPU = 2138, Flash Map = 2138
web0746.hex 116324 bytes loaded
Synchronizing
Setting oscillator
Unlock
Writing Sector 0 [4096]:
Writing Sector 1 [4096]:
Writing Sector 2 [4096]:
Writing Sector 3 [4096]:
Writing Sector 10 [32768]:
Download done
C:\release>

Tables



Tables

- ASCII Character Codes
- Bitwise Operators
- Operator Precedence
- Variable Types

ASCII Character Codes

ARMbasic uses the standard "ASCII extended" character set. The compiler uses the character set values 32 to 126 which corresponds to SPACE through TILDA.

Characters outside this range may have a special meaning and are interpreted by the terminal emulation program that is controlling the ARM processor. Those would include BACKSPACE, TAB, CR and LF. These characters cause changes in the stream of characters going to or from the ARMexpress module. These characters may be interpreted differently on a PC vs. a Mac.

However when using RXD(0) or TXD(0), there is no special interpretation of characters, so all codes 0 to 255 may be sent without any change.

The ARM processor requires BASICtools to know whether the user **ARMbasic** code is running. So now when a program starts a SOH (001) character is sent and when the program finishes an EOT (004) character is sent. User code should avoid using these character codes if BASICtools is being used for communication with the module or board.

Dec	Hex	Meaning	Dec	Hex	Meaning
000	000	NUL (Null char.)	064	040	@ (AT symbol)
001	001	SOH (Start of Header)	065	041	A
002	002	STX (Start of Text)	066	042	B
003	003	ETX (End of Text)	067	043	C
004	004	EOT (End of Transmission)	068	044	D
005	005	ENQ (Enquiry)	069	045	E
006	006	ACK (Acknowledgment)	070	046	F
007	007	BEL (Bell)	071	047	G
008	008	BS (Backspace)	072	048	H
009	009	HT (Horizontal Tab)	073	049	I
010	00A	LF (Line Feed)	074	04A	J
011	00B	VT (Vertical Tab)	075	04B	K
012	00C	FF (Form Feed)	076	04C	L
013	00D	CR (Carriage Return)	077	04D	M
014	00E	SO (Shift Out)	078	04E	N
015	00F	SI (Shift In)	079	04F	O
016	010	DLE (Data Link Escape)	080	050	P
017	011	DC1 (XON)	081	051	Q
018	012	DC2 (Device Control 2)	082	052	R
019	013	DC3 (XOFF)	083	053	S
020	014	DC4 (Device Control 4)	084	054	T
021	015	NAK (Negative Ack)	085	055	U
022	016	SYN (Synchronous Idle)	086	056	V
023	017	ETB (End of Trans. Block)	087	057	W
024	018	CAN (Cancel)	088	058	X
025	019	EM (End of Medium)	089	059	Y
026	01A	SUB (Substitute)	090	05A	Z
027	01B	ESC (Escape)	091	05B	[(left bracket)
028	01C	FS (File Separator)	092	05C	\ (back slash)
029	01D	GS (Group Separator)	093	05D	] (rightbracket)
030	01E	RS (Request to Send)	094	05E	^ (caret)
031	01F	US (Unit Separator)	095	05F	_ (underscore)
032	020	SP (Space)	096	060	`
033	021	! (exclamation mark)	097	061	a
034	022	" (double quote)	098	062	b
035	023	# (number sign)	099	063	c
036	024	$ (dollar sign)	100	064	d

037	025	%	(percent)	101	065	e	
038	026	&	(ampersand)	102	066	f	
039	027	'	(single quote)	103	067	g	
040	028	(	(left parenthesis)	104	068	h	
041	029	)	(right parenthesis)	105	069	i	
042	02A	*	(asterisk)	106	06A	j	
043	02B	+	(plus)	107	06B	k	
044	02C	,	(comma)	108	06C	l	
045	02D	-	(minus or dash)	109	06D	m	
046	02E	.	(dot)	110	06E	n	
047	02F	/	(forward slash)	111	06F	o	
048	030	0		112	070	p	
049	031	1		113	071	q	
050	032	2		114	072	r	
051	033	3		115	073	s	
052	034	4		116	074	t	
053	035	5		117	075	u	
054	036	6		118	076	v	
055	037	7		119	077	w	
056	038	8		120	078	x	
057	039	9		121	079	y	
058	03A	:	(colon)	122	07A	z	
059	03B	;	(semi-colon)	123	07B	{	(left brace)
060	03C	<	(less than)	124	07C	|	(vertical bar)
061	03D	=	(equal sign)	125	07D	}	(right brace)
062	03E	>	(greater than)	126	07E	~	(tilde)
063	03F	?	(question mark)	127	07F	DEL	(delete)

Bitwise Operators

Y = A AND B

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

Y= A OR B

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

Y = A XOR B

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

Y = NOT A

A	Y
0	1
1	0

Operator Precedence

Description

When several operations occur in a single expression, each operation is evaluated and resolved in a predetermined order. This called the order of operation or operator precedence. There are three main categories of operators; arithmetic, comparison, and logical. If an expression contains operators from more than one category, arithmetic operators are evaluated first, comparison operators next, and finally logical operators are evaluated last. If operators have equal precedence, they then are evaluated in the order in which they appear in the expression from left to right. Comparison operators all have equal precedence.

The following table gives the operator precedence for each operator in each category. Operators lower on the list have a lower operator precedence. Operators on the right have lower precedence than ALL operators in the column to the left. Arithmetic operators are evaluated before comparison operations, and logical operators are last.

Parentheses can be used to override operator precedence. Operations within parentheses are performed before other operation. However, within the parentheses operator precedence is used.

Arithmetic	Comparison	Bitwise/Logical
- (Negation)	= <> < > <= >=	AND
*, / (Multiplication and division)		OR XOR
MOD (Modulus Operator)		NOT
+, - (Addition and subtraction)		
<<, >> (Shift Bit Left and Shift Bit Right)		

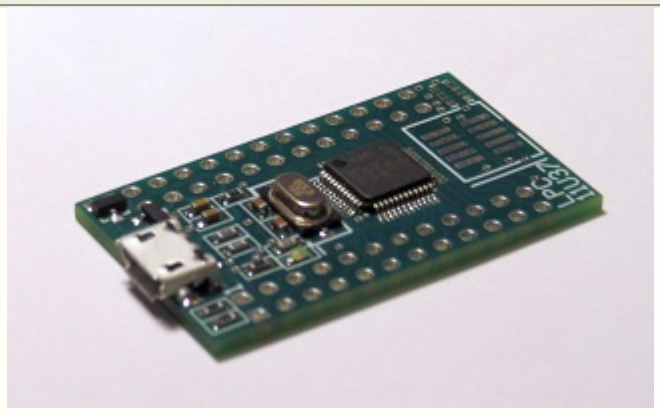
See also

- Operator List

Variable Types

NAME	BITS	FORMAT	MIN VAL	MAX VAL
INTEGER	32	signed integer	-2147483648	+2147483647
SINGLE	32	signed float	-1.0000E+38	+1.0000E+38
ARRAY	fixed length	signed integer	-2147483648	+2147483647
SINGLE ARRAY	fixed length	signed float	-1.0000E+38	+1.0000E+38
STRING	variable/max length 256 bytes	zero terminated	0	+255
BYTE	used as byte array no max length		0	+255

Support



Support

- How to contact the developers
- How to report a bug
- Contributors
- Notices

Trouble Shooting

Before you start make sure you have the latest version of the tools.

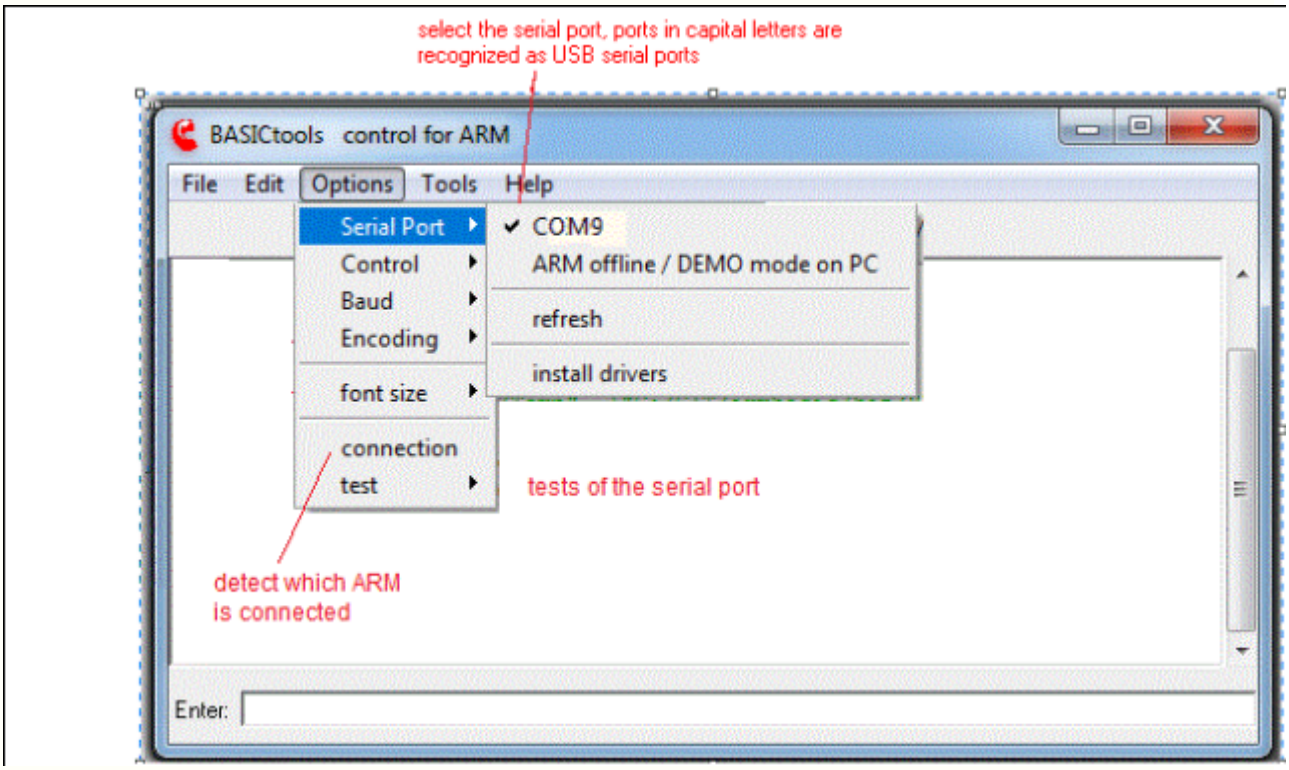
The latest tools are always available at the support page on the Coridium Website-

Check your cables, check the LED

See **Connect USB** All Coridium boards have an Green LED that is driven when power is applied. If that LED is not on, check you connection, or using the **Schematics** trace the power connections.

Determining which COM port should be used

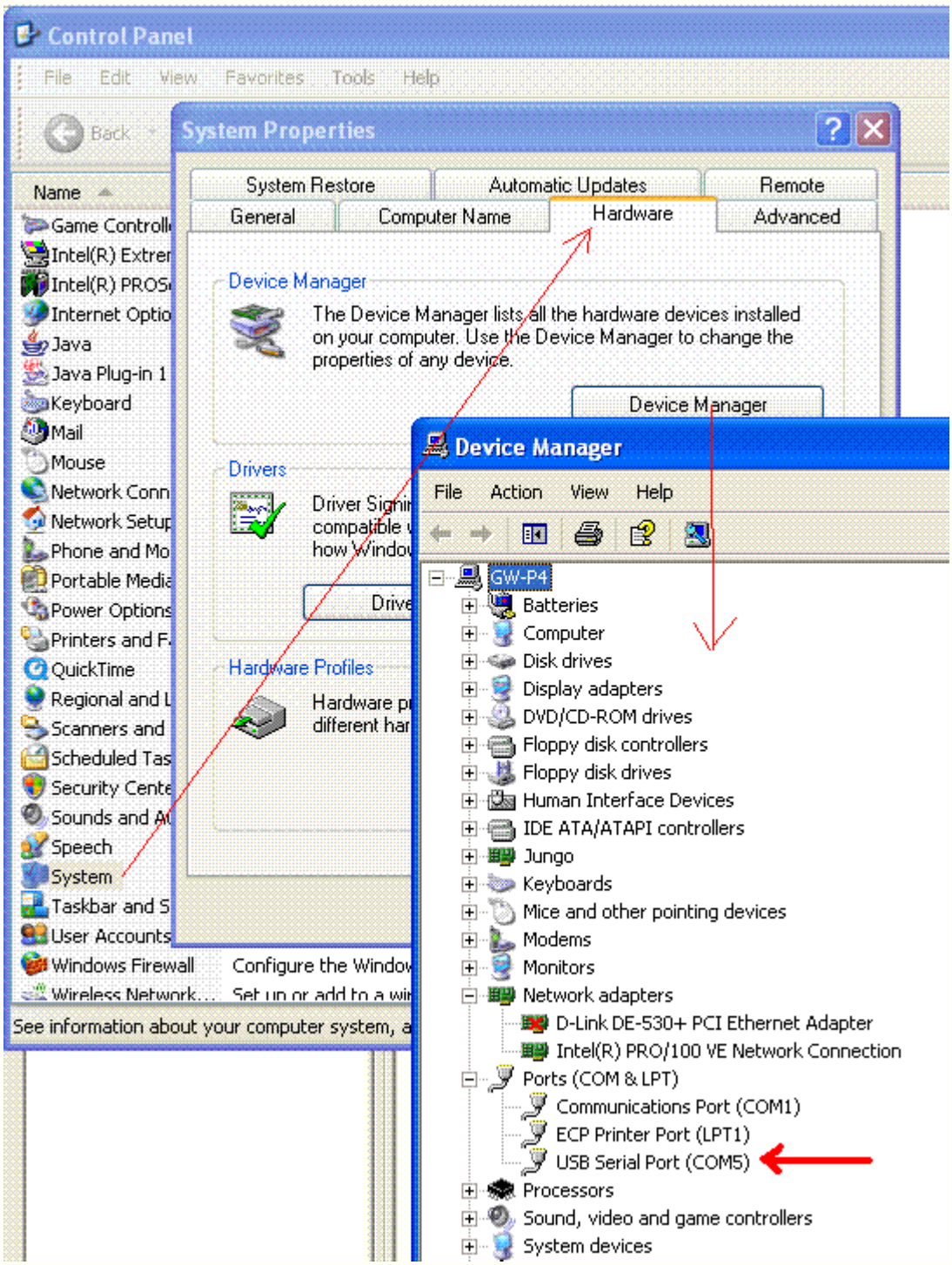
The tools will query the Window registry and will list the available COM ports. If you are using a Coridium USB dongle, a Coridium card with built in USB connection, a SparkFun USB dongle or a FTDI USB cable; then those COM ports will be listed in capital letters. Those in lower case are NOT using a Coridium dongle or built in USB port, do not select those. If you are using some other serial connection, refer to the section on **ISP checks** .



In the example above, COM9 is a Coridium board, com1 and com2 are serial ports built into the PC.

You can also identify an FTDI port using the Device Manager.

Open the Control Panel>System>Device Manager



USB(COMx not appearing)

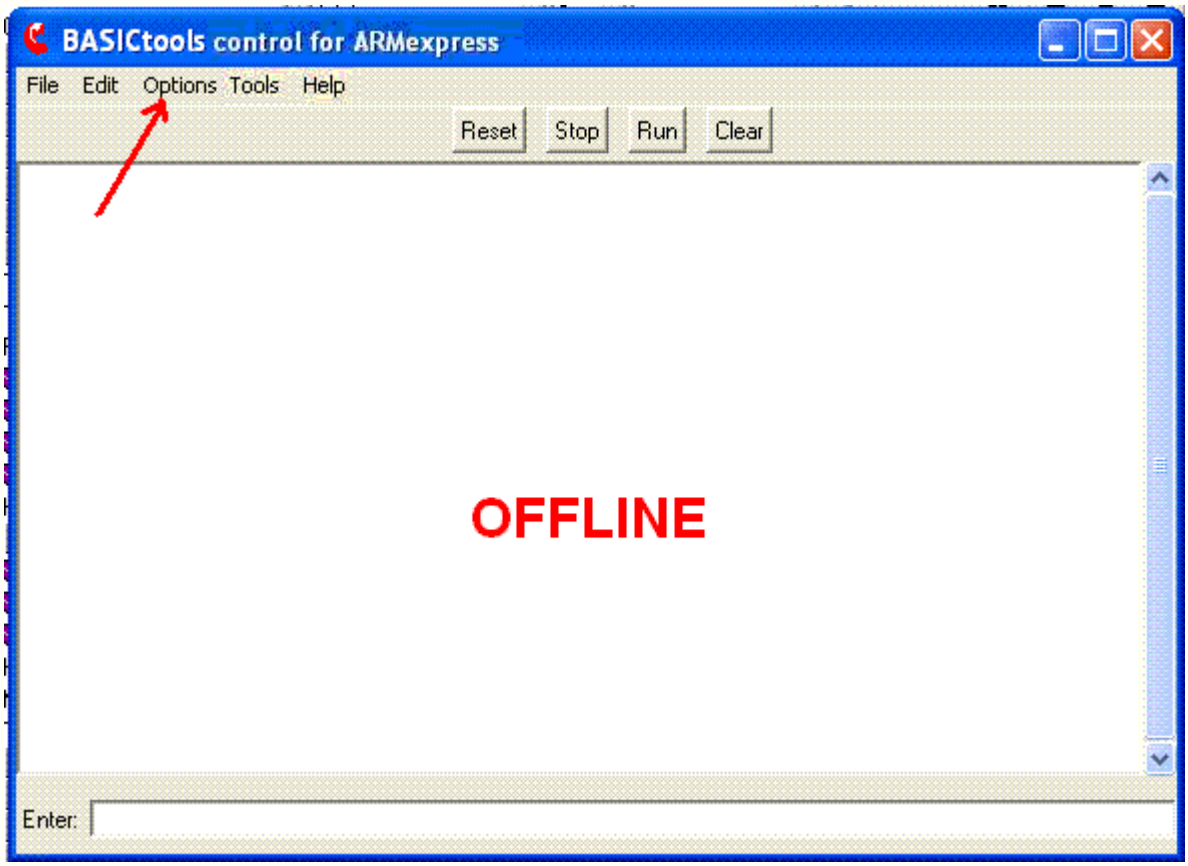
If the USB Serial Port does not appear, check the connection and if its still not there install the driver.

Open the /Program Files(x86)/Coridium/Windows Drivers folder. Run the .exe file in that folder. This will install the FTDI driver. The FTDI FT232RL is the USB device we use on the Coridium USB dongle.

We use the FTDI VCP **driver**, more information on its installation can be found at their **website** .

Offline indicator

This will be shown if the port you were using last time the program was run is no longer available. You must reselect a Port using the Option Menu to reestablish communication with the ARM. Make sure any other copies of MakeItC or BASICtools are closed, as you can not open a port simultaneously with more than one program.



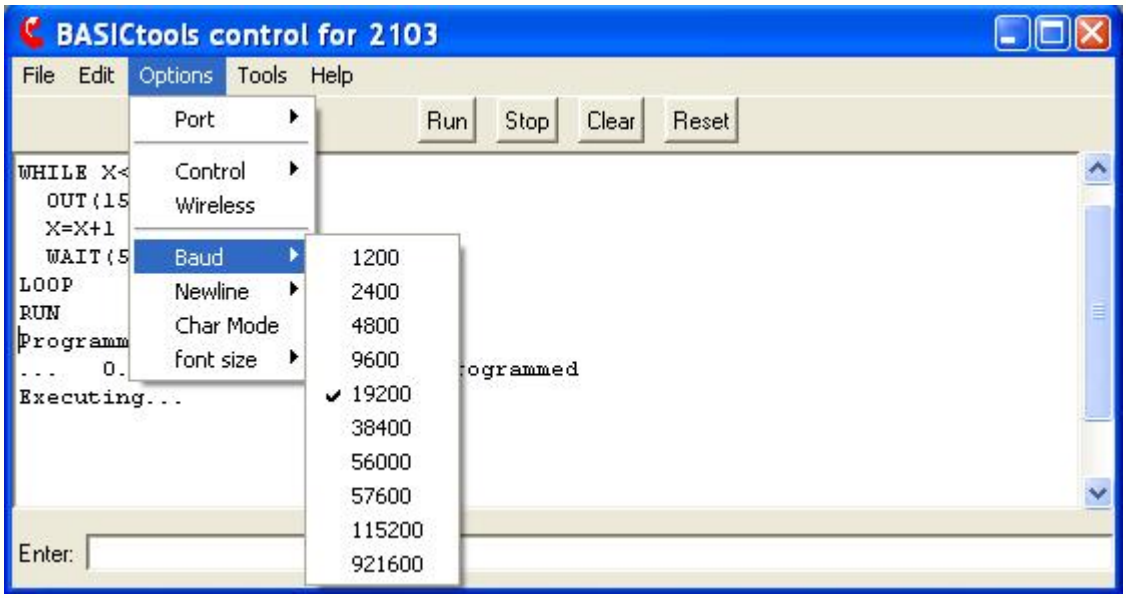
Reset ARM shows no message

If your BASIC program does not appear to start up, hit the STOP button. This will stop any known existing program, even those that alter the UART baudrates. If your program sends a large amount of data to the UART, it may take a while for the STOP to respond. You may need to close BASICtools and re-open them. When BASICtools starts up it halts any existing user program. At that point you can load a new program -- something short like PRINT 1234 ' will erase a run away program.

After STOP or restarting BASICtools you should get a welcome message.

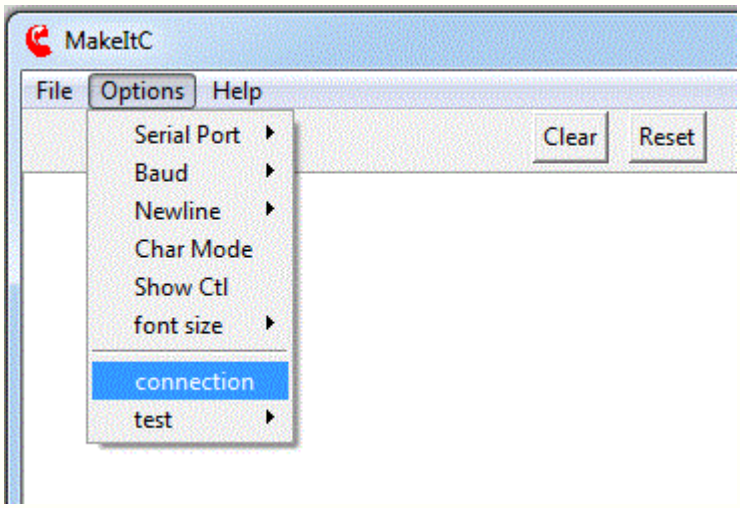
Check Baud Rate

Or you might not have the correct baud rate selected. Make sure it is set to 19200 or 115200 baud. Baud settings in the Device Manager do NOT affect the MakeItc or BASICtools. Firmware versions after 8.34 use 115.2 Kb, this includes Teensy, mBed and many boards we built after 2014.

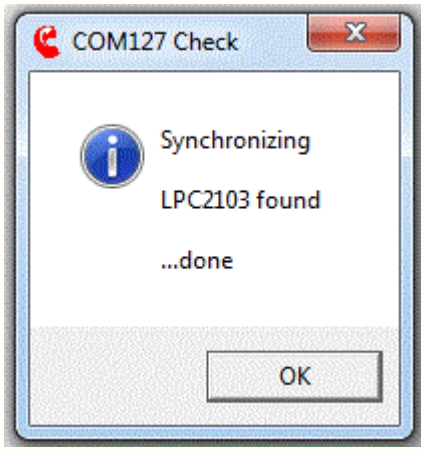


Check ARM connection

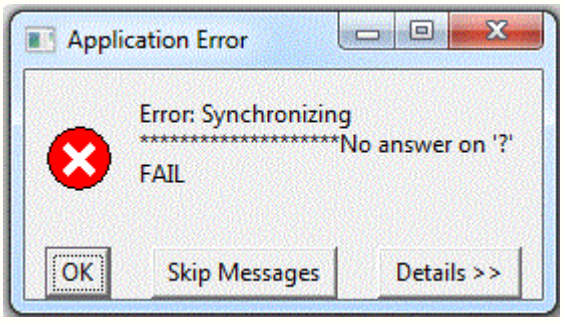
TclTerm has some tools to verify a connection to the ARM chip.



The connection test will use the Coridium USB dongle or built in USB connection to place the ARM into ISP mode (holds P0.14, P2.10 or P0.1 low during RESET, depending on the part -- details in the corresponding NXP User manual). Then a ? will be sent followed by "Synchronized". If an appropriate answer is received a command to ID the part will be sent and that will be reported --

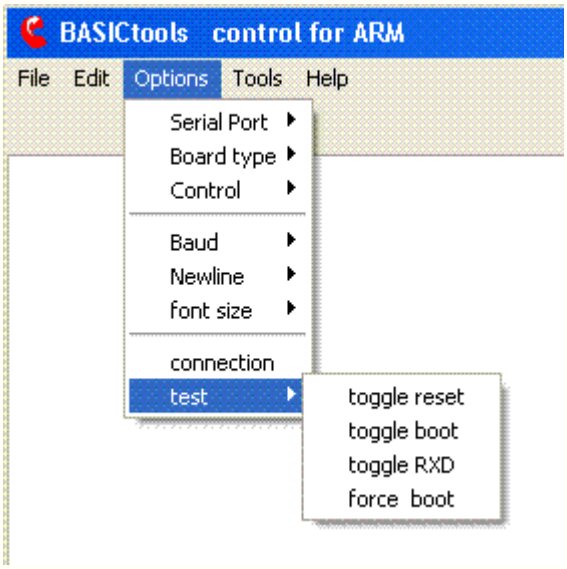


Or if no part is found



If no part is found go back check the connection, power connections, or which port has been selected.

If your USB/serial connection is something homebrew, and if you followed the Coridium reference schematics, you can use the test menu under options to wiggle the required RESET, BOOT and RXD pins to verify they are connected to the CPU.



How to contact the developers

You should contact the **ARMbasic** developers through Coridium Corp.

- **www.coridiumcorp.com**

Tech Support monitors the following groups.

- **groups.yahoo.com/group/ARMexpress**
- **groups.yahoo.com/group/gnuarm**

Coridium has done custom ports of ARMbasic to other platforms.

- **techsupport@coridiumcorp.com**

See also

- **Reporting a bug**

How to report a bug

Before reporting a bug, try to make sure it's a bug in **ARMbasic** and not a bug in your own code. Try to write a small test that reproduces the problem you are encountering. Read any relevant documentation. If you show people that you have tried to solve your own problem, rather than immediately running for help, you will be more likely to find people willing to help you.

Be as specific as you can - "The HWOWM library function fails when it is called with a value of 1234" is much better than "It crashes".

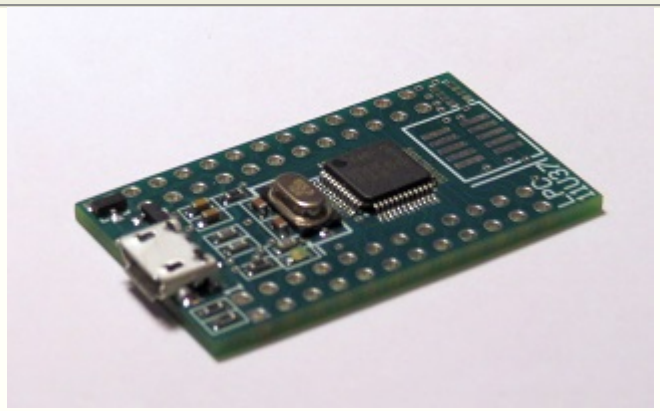
Also describe your setup, which board, which PC OS and other configuration details.

The first place to go in the case you believe you've encountered a bug is **groups.yahoo.com/group/ARMexpress**

If you have isolated a compiler bug completely, and you have steps to reproduce it and a small piece of sample code, you can also file a bug report with tech support at **support @coridiumcorp.com**.

DO NOT file general "it doesn't work!" bug reports in the **groups.yahoo.com/group/ARMexpress** system. Only isolated, reproducible bugs should be posted there.

Updating ARMbasic Firmware



The ARMbasic compiler can be freely downloaded. There is no charge to run BASIC or C on Coridium Products.

We do offer for sale a BASIC firmware that can be installed on OTHER vendors hardware. There is a demo version that allows you to try it before you buy it. That demo version limits the code and data space.

This utility is protected. You will need to obtain this program from Coridium which is part of the order process. For now this will be emailed to you manually from Coridium, until this process is fully automated

Upgrading Firmware on Coridium boards Install Software	**Installing Firmware on other vendors boards** Install Software Install Demo Firmware Installing purchased full feature Firmware

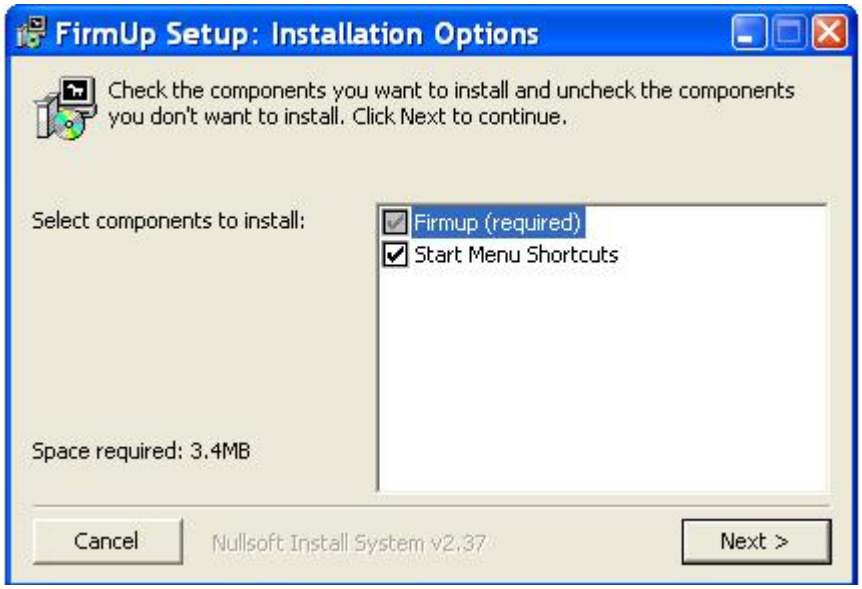
Step 1: Install Software

The **ARMbasic** compiler runs on the PC, in combination with a BASIC support library that is installed on the ARM. This support library (firmware) will be updated from time to time to support new features. To upgrade that firmware you will need to purchase the upgrade.

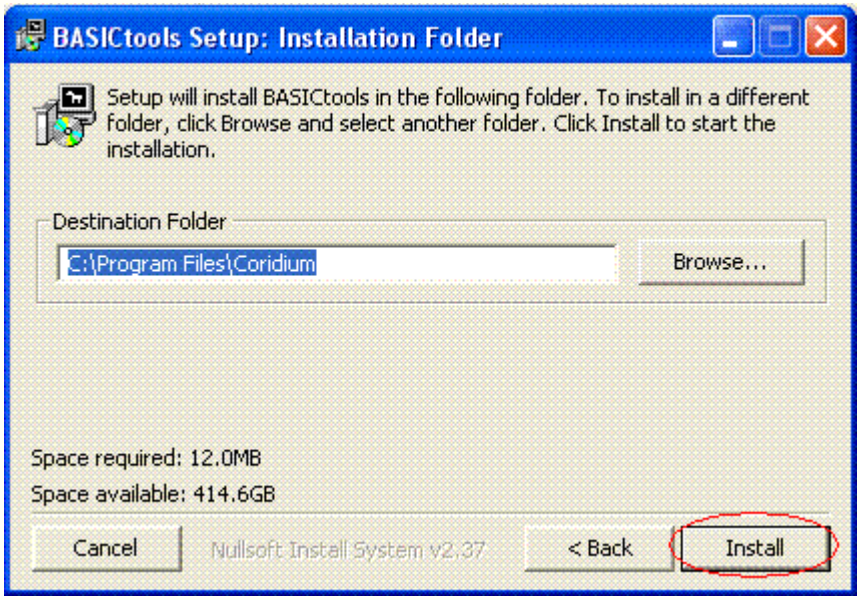
Purchase at Coridium Web store

This installer is meant for Windows either NT, XP or XPx64, Vista and Win 7 (both 32 and 64 bits).

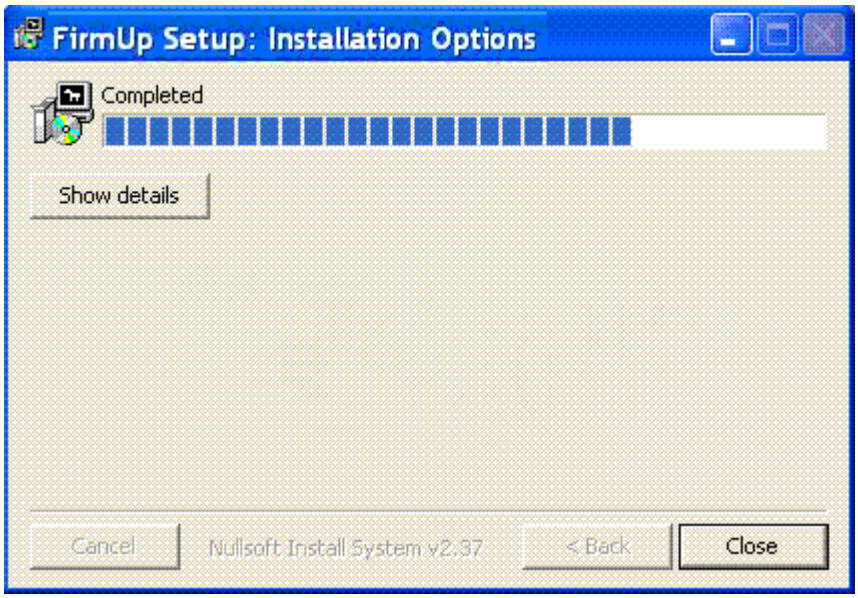
The software is downloaded from the web, and run as an installer SETUP program.



Click **Next** to get started.



Accept the defaults and **Install**. You may chose a different target directory.



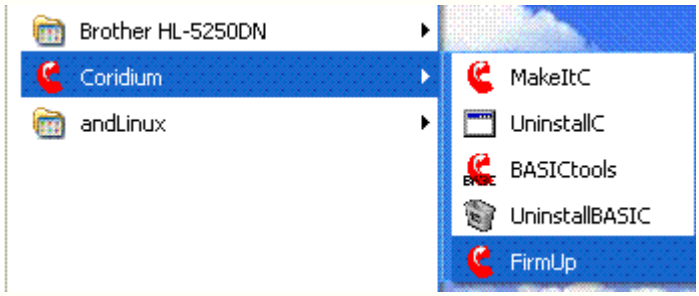
The installation will now run, and when it finishes hit **Close** .

And its as easy as that.

On to Step 2

Step 2: Writing the Firmware.

The software installed in the previous step would either be FirmUp for firmware upgrades.

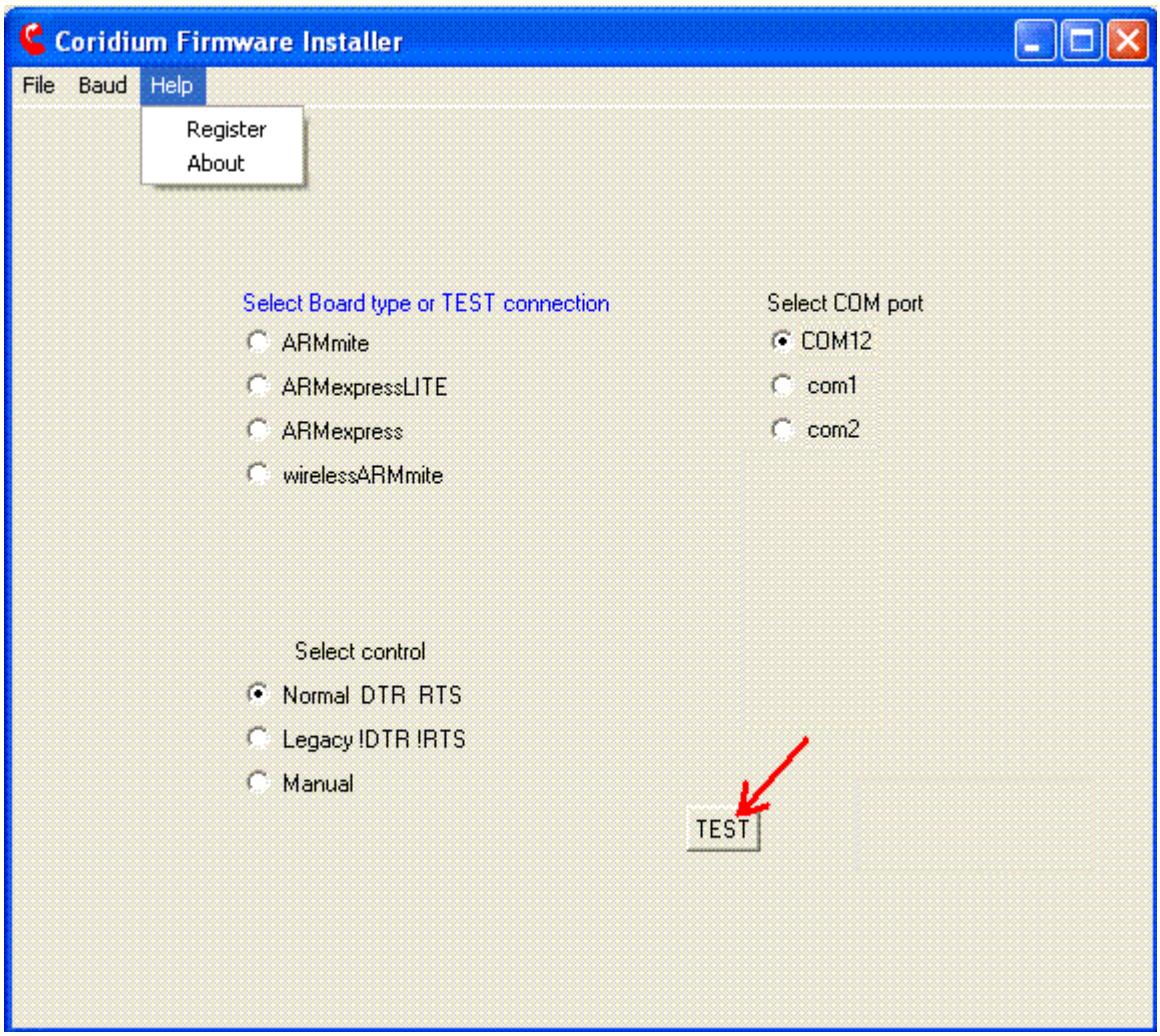


To run FirmUp you must have network access, as information is downloaded from the Coridium website.

Step 2: Establish communication

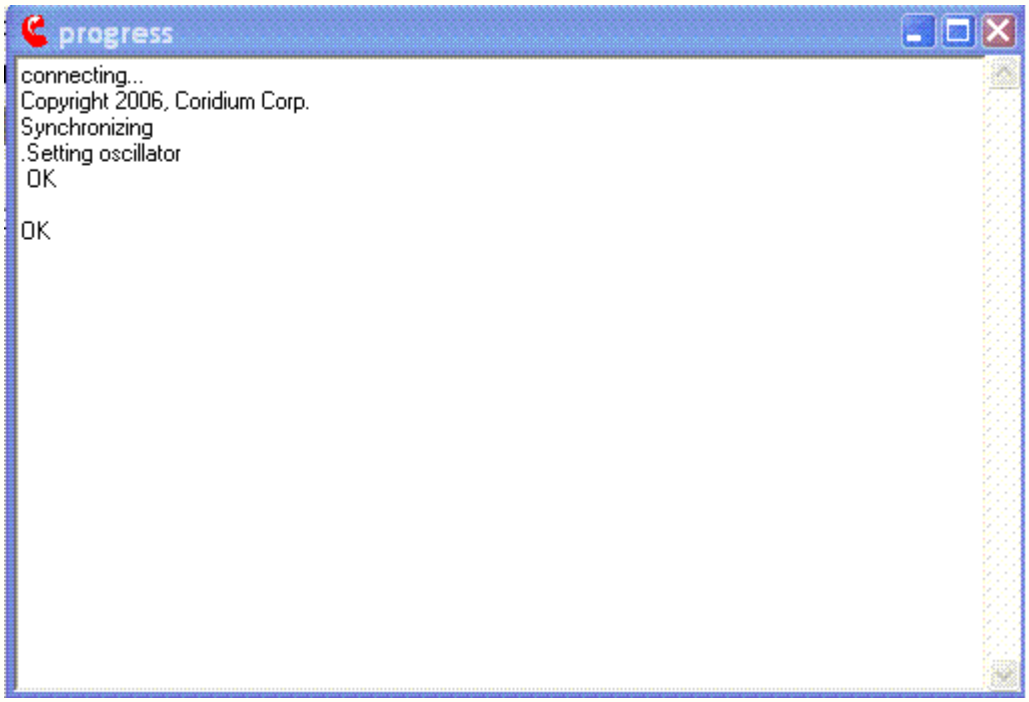
Before you can run **FirmUp** you must be able to communicate with the board that contains the NXP LPCxxxx ARM, and then load **ARMbasic** firmware onto that board. These 2 steps are accomplished with the FirmUp program. The installation of Step 1 has installed a Start Menu shortcut.

FirmUp allows you to choose the serial port on the PC from a list of known ports. Ports in that list that are capitalized were determined to be using FTDI USB serial devices. You must also set the control type, which for most will be Normal mode. Legacy mode is for those users who have inverted the control signals, for instance to run Hyperterm or Linux, details **here** . For wireless boards, Manual mode should be chosen.



So select your comport and choose the control method. To test that push the soft button TEST on the FirmUp program. It will prompt you for any action required (like pushing buttons on the target board), and then test the communication with the PC.

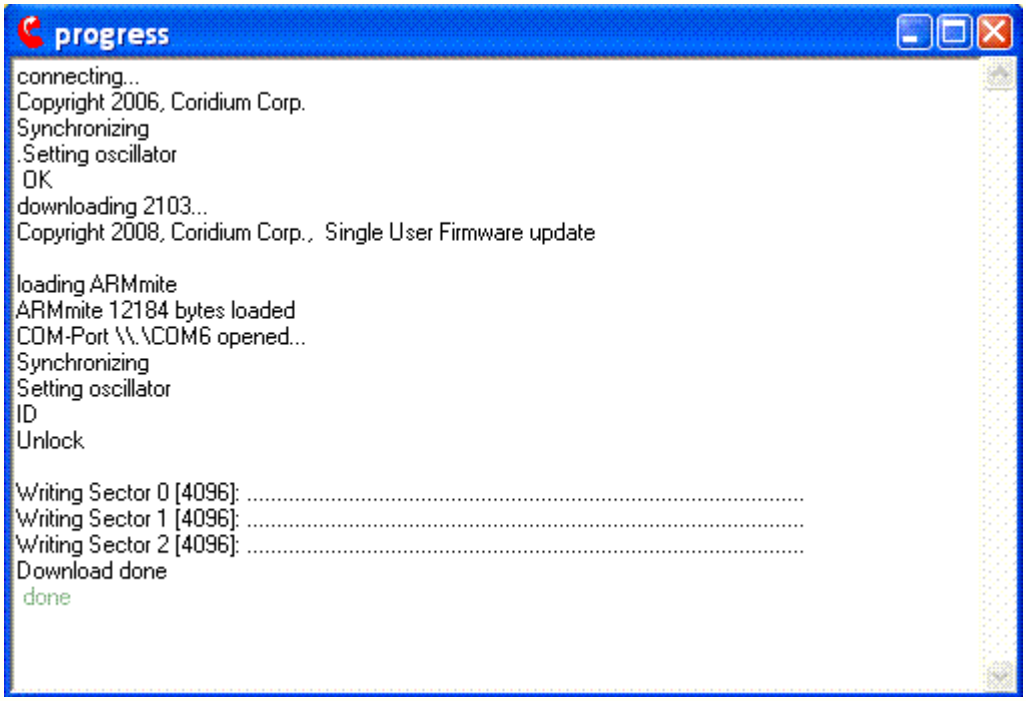
If this does not pass, then you cannot go on to the next step.



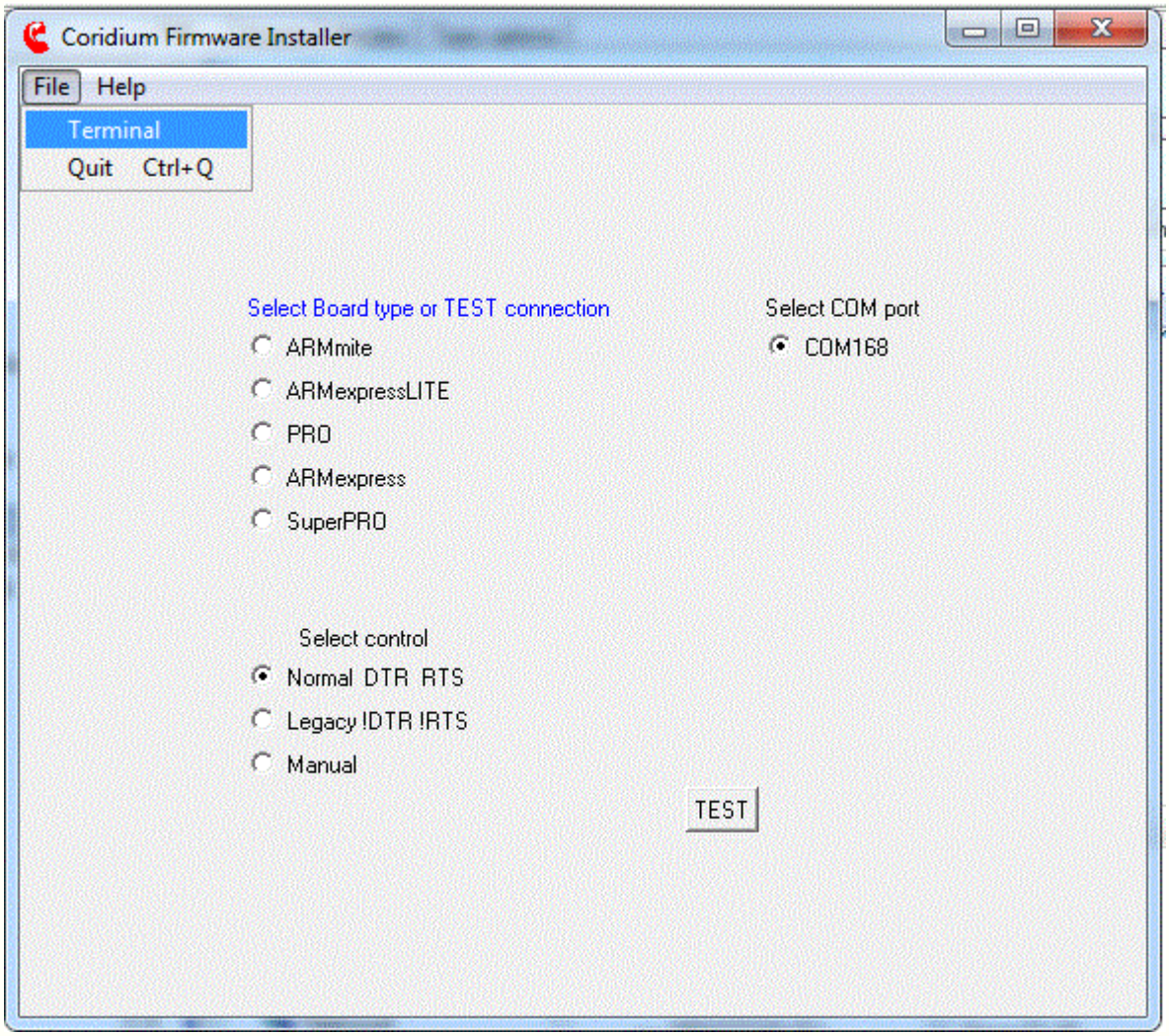
Step 3: Install Firmware on ARM

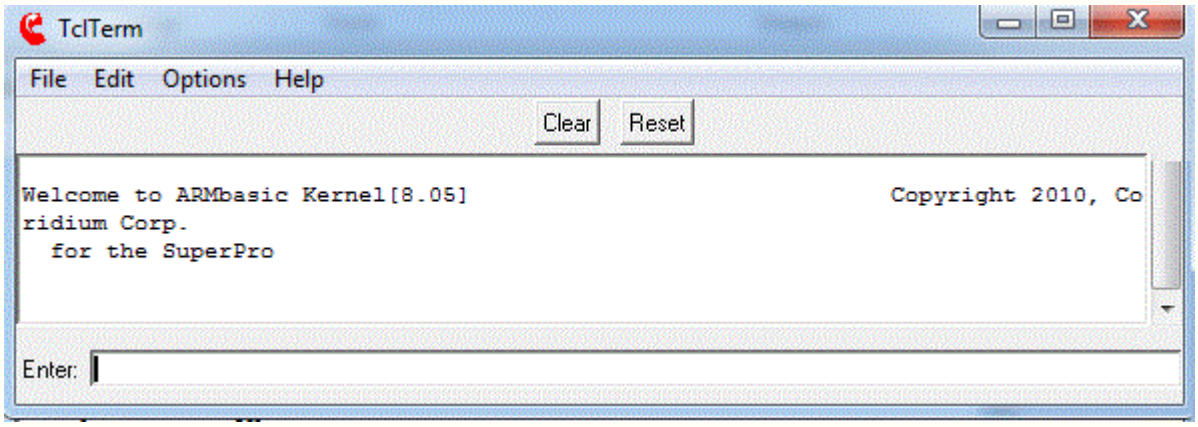
This part of the install needs to be run once to place a base set of libraries on the ARM processor. This firmware includes the initialization code, communication routines, and a set of subroutines called from the user ARMbasic program.

Once the TEST above passes and the device type has been selected, the TEST button will change to UPDATE. Select this button ONCE. You should see a dialog similar to the one below.



Firmware has been successfully loaded, you can open a terminal window here to verify that.


Coridium Firmware Installer
File
Help
Terminal
Quit Ctrl+Q
Select Board type or TEST connection
ARMmite
ARMexpressLITE
PRO
ARMexpress
SuperPRO
Select COM port
COM168
Select control
Normal DTR RTS
Legacy !DTR !RTS
Manual
TEST


TclTerm
File Edit Options Help
Clear
Reset
Welcome to ARMbasic Kernel[8.05] Copyright 2010, Co
ridium Corp.
for the SuperPro
Enter:

Contributors

The **ARMbasic** compiler itself is property of the Coridium Corp. and all rights are reserved.

Mike and Bruce began this project in 2003. The original target was a Cygnal 8051 using the Keil Compiler. As part of the development, the BASIC was compiled on a PC in both Visual C and GCC. This allowed quicker development of the language parser. Then a need arose for a hardware debugger on an ARM based cell phone that used the CodeWarrior compiler. To check out hardware such as new displays and camera subsystems a new approach was required. At the time it took 3-5 hours to make a change in the main software on the platform. The BASIC made it possible to verify interfaces in minutes. Then Zilog introduced the websurfer and the BASIC was ported to that platform with a web interface replacing the serial port. Later it moved to the Rabbit 3100 modules and was productized on the 3710. This product is the BASIC-8. For performance the interpreter was replaced with code compiler that performed a two pass compile-link step. The speed of code increased by at least an order of magnitude. Now Coridium has moved this compiler back to the ARM using GCC. This time it includes a single pass BASIC compiler that incrementally builds programs in Flash. Code tables are maintained even after the program is "run" which allows the user the look and feel of an interpreter. Its easy to check the value of variables when the program has stopped, or to even change them. Also during this time the BASIC-8 product's web interface was translated to Japanese and is available as the NAPI-BASIC server.

As you can see the compiler has been around the block, and now the world too. Its quite portable as having lived on 6 different C-platforms. As it has been used extensively, its also quite stable. Coridium will continue to add features as needed and offer customizations for OEM customers.

A number of utilities have been used to produce the ARMexpress system.

Freewrap is used to generate BASICtools from a Tcl/Tk script.

The MinGW cpp is used for pre-processing the BASIC.

The Tcl'ers Wiki Oscilloscope was the source for the basis of the LogicScope code.

ARMbasic was compiled with Winarm GCC.

The **ARMbasic** documentation has been based on the documents of the GPL WikiPedia and FreeBASIC project. This document is also covered under the **GFDL** license.

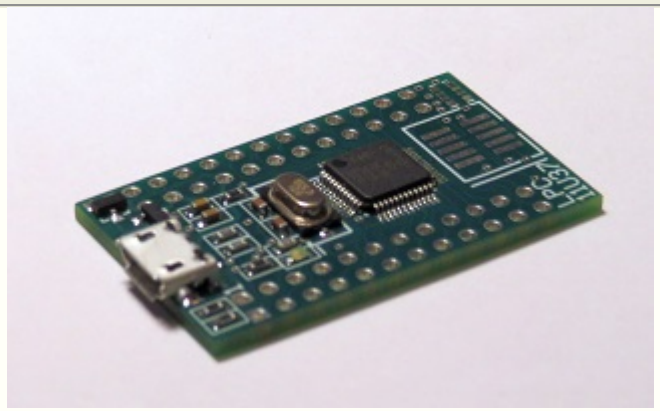
Notices

NO WARRANTY

1. THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. CORIDIUM PROVIDES THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

2. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL CORIDIUM BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Obsolete



Obsolete

These functions have been superceeded by more modern implementations.

- **Obsolete PBASIC functions**
- **DATA, READ and RESTORE replaced by CONST**
- **ON replaced by interrupts**
- **RXD0, TXD0 replaced by Serial Array**

PBASIC type functions -- OBSOLETE, included for reference

Library

#include <PULSE.bas> ' PBASIC library, these routines were written for ARMexpress, and included for general reference

C

- COUNT

P

- PULSIN
- PULSOUT
- PWM

R

- RCTIME

Interface

```
' duration in microseconds
' timeperiod in milliseconds
' duty 0-255

FUNCTION COUNT (pin, timeperiod)
FUNCTION PULSIN (pin, level)
SUB PULSOUT (pin, duration)
SUB PWM (pin, duty, timeperiod)
FUNCTION RCTIME (pin, state)
```

Description

COUNT the number of pulses low-high-low or high-low-high on *pin* over a *timeperiod* of *milliseconds*, returning the FUNCTION value.

PULSIN measures an input pulse on *pin* at *level*, returning the value in microseconds. The IO direction of *pin* will be set to input. If *pin* is already at *level* when PULSIN is called it will wait to a transition to the opposite *level*. PULSIN will wait 1 second for *pin* to go to *level*. The minimum pulse that can be measured is 1 microseconds. If *pin* does not go to level or remains at *level* longer than 1 second 0 is returned..

PULSOUT will generate an output pulse on *pin* for *duration* microseconds. The IO direction of *pin* will be set to output. The level of the output will be switched, driven for *duration* microseconds, then switched back to its initial level. The minimum pulse period is 1 microseconds.

PWM will generate a pulse corresponding to an analog signal on *pin* for *timeperiod* in milliseconds with a *duty* cycle of 0 to 255. A *duty* cycle of 255 corresponds to an output value of 100%. The IO direction of the pin will be set to output, the PWM pulse train is output, and then the pin is set to tri-state (input). If the pin is connected to an RC filter, then the voltage will stay on the capacitor for a period of time determined by the load.

RCTIME will measure the time which *pin* remains at *level*, returning the value in microseconds(us). The minimum time measured is 1 microseconds. If *pin* is not at *level* when RCTIME is called -1 is returned. If *pin* remains at *level* longer than 1 second 0 is returned.

COUNT -- Obsolete PBASIC routine

Syntax

#include <PULSE.bas> ' PBASIC library, these routines were written for ARMexpress, and included for general reference

FUNCTION COUNT (*pin, milliseconds*)

Description

Count the number of pulses low-high-low or high-low-high on *pin* over a duration of *milliseconds*, returning the value to *variable*.

Example

```
#include <PULSE.bas>
'Report the number of transition cycles on pin 7 during a 10 second interval

ct = COUNT ( 7, 10000 )
PRINT "Pin 7 transitioned "; ct; " times"

Pin 7 transitioned 3 times
```

Differences from other BASICs

- no equivalent in Visual BASIC
- different syntax from PBASIC, and times in milliseconds rather than "ticks"

See also

- **RCTIME**
- **Hardware Pulse Routines**

PULSIN -- Obsolete PBASIC routine

Syntax

' PBASIC library, these routines were written for ARMexpress, and included for general reference

#include <PULSE.bas> ' source in /Program Files/Coridium/BASIClib

FUNCTION PULSIN (*pin, level*)

Description

Measure an input pulse on *pin* at *level*, returning the value to *variable*.

The IO direction of *pin* will be set to input.

If *pin* is already at *level* when the function is called it will wait to a transition to the opposite *level*.

The function will wait 1 second for *pin* to go to *level*. The length of time is measured in microseconds(us). The minimum pulse that can be measured is 1 microseconds. If *pin* does not go to level or remains at *level* longer than 1 second *variable* is set to 0.

Example

```
#include <PULSE.bas>

'Wait for pin 7 to go high then low.
'Print the number of microseconds pin 7 was high.

tim = PULSIN (7, 1)
PRINT "Pin 7 pulse high for "; tim; " us"
```

Differences from other BASICs

- no equivalent in Visual BASIC
- Times are measured in microseconds rather than CPU dependent ticks in PBASIC

See also

- **RCTIME**
- **COUNT**
- **Hardware Pulse Routines**

PULSOUT -- Obsolete PBASIC routine

Syntax

```
' PBASIC library, these routines were written for ARMexpress, and included for general reference

#include <PULSE.bas>                ' source in /Program Files/Coridium/BASIClib

SUB PULSOUT (  pin, microseconds )
```

Description

Generate an output pulse on *pin* for *microseconds*.

The IO direction of *pin* will be set to output. The level of the output will be switched, driven for *microseconds*, then switched back to its initial level. The minimum pulse period is 1 microseconds.

Example

```
#include <PULSE.bas>

' Generate a 1 second high pulse on pin 4.
IO(4)=0
PULSOUT (4, 1000000)
```

Differences from other BASICs

- no equivalent in Visual BASIC
- measures time in microseconds rather than CPU dependent ticks in PBASIC

See also

- **PULSIN**
- **Hardware Pulse Routines**

PWM -- Obsolete PBASIC routine

Syntax

```
' PBASIC library, these routines were written for ARMexpress, and included for general reference

#include <PULSE.bas>                    ' source in /Program Files/Coridium/BASIClib

SUB PWM ( pin, duty, milliseconds)
```

Description

Generate an analog signal on *pin* for *milliseconds* with a *duty* cycle of 0 to 255.
A *duty* cycle of 255 corresponds to an output value of 100%.

The IO direction of the pin will be set to output, the PWM pulse train is output, and then the pin is set to tri-state (input). If the pin is connected to an RC filter, then the voltage will stay on the capacitor for a period of time determined by the load.

Example

```
#include <PULSE.bas>

 '  Generate a 1.65 volt (half of 3.3V) on pin 4 for 6 seconds.

PWM (4, 127, 6000)
```

Differences from other BASICs

- no equivalent in Visual BASIC
- duration in PBASIC is CPU dependent and measured in ticks

See also

- **HWPWM**
- **PULSOUT**
- **Hardware Pulse Routines**

RCTIME -- Obsolete PBASIC routine

Syntax

```
' PBASIC library, these routines were written for ARMexpress, and included for general reference

#include <PULSE.bas>                    ' source in /Program Files/Coridium/BASIClib

FUNCTION RCTIME ( pin, level )
```

Description

Measure the time which *pin* remains at *level*, returning the value to *variable*.
The length of time is measured in microseconds(us). The minimum time measured is 1 microseconds.
If *pin* is not at *level* when the function is called *variable* is set to 1.
If *pin* remains at *level* longer than 1 second *variable* is set to 0.

Example

```
#include <PULSE.bas>

DIR(7)=0

'... some external device has set input pin 7 to low or 0 volts

tim = RCTIME (7, 0)
PRINT "Pin 7 low for "; tim; " us"

'... function waits for input pin 7 to go to high state

Pin 7 low for 50 us
```

Differences from other BASICs

- no equivalent in Visual BASIC
- results in microseconds rather than CPU dependent ticks in PBASIC

See also

- **PULSIN**
- **Hardware Pulse Routines**

REV (moved to library)

Syntax

#include <PBASIC.bas>

FUNCTION REV (value, *number of bits*)

Description

Function returning a reversed (mirrored) copy of a specified number of bits of a value, starting with the rightmost bit (LSB).

For instance, 0xFEED REV 4 would return 0xB, a mirror image of the last four bits of the value.(The binary representation of 0xD being 1101 and 0xB 1011)

Differences from PBASIC

- no equivalent in Visual BASIC
- same as PBASIC

See also

- **AND**
- **XOR**
- **NOT**

DATA (obsolete)

replaced by CONST array

See **CONST** array

ON (only on ARM7 parts with integer BASIC firmware)

ON is NOT used with BASICchip, PROstart, PROplus and SuperPRO see INTERRUPT SUB

ON is no longer supported on the Floating point versions of firmware, Use the INTERRUPT SUB or revert to the integer version of firmware can be found in the **Coridium Forum**.

Syntax

ON TIMER *msec label*

or

ON EINT0|EINT1|EINT2 RISE|FALL|HIGH|LOW *label*

Description

These statements will initialize interrupt service routines so that when the interrupt occurs the code at label will be executed. *Label* must have been pre-defined and can either be a SUB (without parameters) or code beginning with a *label*: and ending in a RETURN. The interrupt response time is approximately 3 usec. Other interrupts may make this time longer.

TIMER interrupts will occur every *msec* milliseconds. *msec* may be a variable or constant, expressions are not allowed. The value for *msec* must be greater than 1. If TIMER interrupts are used, then only 4 hardware PWM channels are available.

EINT0 and EINT2 are 2 pins that will interrupt when the defined event occurs. RISE and FALL are the preferred method and will generate interrupts on rising or falling edges on those 2 pins. HIGH and LOW are supported, but if the pin remains in that state interrupts will be continuously generated.

EINT1 is connected to the RTS line of the PC, and is normally high, so it can be used by a program on the PC to interrupt the ARMmite, rather than having to reset the board. EINT1 is also available on the ARMexpress modules (pin 21), and should also be kept normally high if used.

Each time the ON statement is executed the interrupt will be initialized, so it is possible to change routines within the program. Multiple interrupts can be used, but they are serviced in the order received, and each interrupt service routine will complete before the next one is handled (interrupts that occur while one is being serviced will be handled after the current interrupt is processed).

Interrupt routines should normally be short and simple. The state of the other user BASIC code will be restored after the interrupt, with the exception of **string** functions, which should **NOT** be done inside an interrupt. PRINT statements use strings, so other than a temporary debug to see if the interrupt occurs, they should not be inside an interrupt routine.

To disable the interrupt use the following #define

```
#define VICIntEnClear  *&HFFFFF014

#define TIMERoff    VICIntEnClear = &H20
#define EINT0off    VICIntEnClear = &H4000
#define EINT1off    VICIntEnClear = &H8000
#define EINT2off    VICIntEnClear = &H10000
```

ON added in version 7.09

The LPC2106 based ARMexpress supports ONLY ON LOW, due to hardware limitations.

ON is a statement that is executed, so if multiple ON statements are in a program the last statement executed will be active command.

Cortex M3 and M0 do not support ON, but use INTERRUPT SUB

Example

```
IO15up = 0              ' serves to declare IO15up

 '...
SUB IO15count
  IO15up = IO15up + 1
ENDSUB

 '...
main:

ON EINT2 RISE IO15count

IO15up = 0
while 1
  if IO15up <> lastIO15count then
    print IO15up
    lastIO15count = IO15up
  endif

 '...

loop
```

```
every20msec:
  checkIO0 = checkIO0 + (IO(0) and 1)
  IO0samples = IO0samples +1
RETURN

 '...
main:

ON TIMER 20 every20msec

 '...

PRINT "Percentage of time IO0 is HIGH =", 100*checkIO0 / IO0samples

 '...
```

Differences from other BASICs

- no equivalent in VB
- no equivalent in PBASIC

See also

- **GOTO**
- **RETURN**

READ (obsolete)

replaced by CONST Array

See **CONST** array

RESTORE (obsolete)

replaced by CONST Array

See **CONST** array

RXD0, TXD0 BAUD0, RXD1, TXD1, BAUD1 (obsolete)

replaced by RXD(channel) TXD(channel) BAUD(channel)

Earlier hardware versions had only 1 or 2 UARTs. Now there can be up to 8 in some parts, so these dedicated routines were replaced by **RXD(channel)** , **TXD(channel)** and **BAUD(channel)**

If you need to support earlier code use the pre-processor --

```
#define RXD0  RXD(0)
#define TXD0  TXD(0)
#define BAUD0  BAUD(0)

#define RXD1  RXD(1)
#define TXD1  TXD(1)
#define BAUD1  BAUD(1)
```

Logic Scope -- no longer supported

Logic Scope
- Timed Samples
- User Defined Sampling
- Stand Alone Analyzer

Timed sampling with Logic Scope

Timing setup

The ARM can sample the up to 32 data lines at 1 MHz rates in BASIC. The software library LogicScope.bas is used to coordinate this sampling. Other sample rates that are multiples of 40 uSec are also supported.

While sampling data the CPU is consumed gathering the 400 samples and then sending them to the PC, at which point processing of the user program can continue.

Example

Example

```
#include <LogicScope.bas>         ' call in support for LogicScope functions
#include <HWPWM.bas>
 '...

' user code to generate the stimulus -- the ScopeDemo engages the HWPWM


HWPWM (1,200,10)
HWPWM (2,200,20)
HWPWM (3,200,40)
HWPWM (4,200,80)
HWPWM (5,200,16)
HWPWM (6,200,32)
HWPWM (7,200,40)
HWPWM (8,200,45)

  ...

while 1
  call doLogicScope (50,0,0)     ' 50 uSec, and trigger on any state (mask =0, trigger =0)
  stop                           ' stop needed only to handshake with the PC for continuous tracing
loop
```

keyw ords: Logic Scope

User sampling with Logic Scope

Random sampling setup

LogicScope is setup to display 400 samples of 16 IO lines. The user can generate these samples by sprinkling the sample call into their program.

The sample data call is completed in less than 3 uSec, except on the 400th sample where the data is sent to the PC. If you don't have 400 samples, but want to see the data in the sample buffer call the FlushScopeSamples routine.

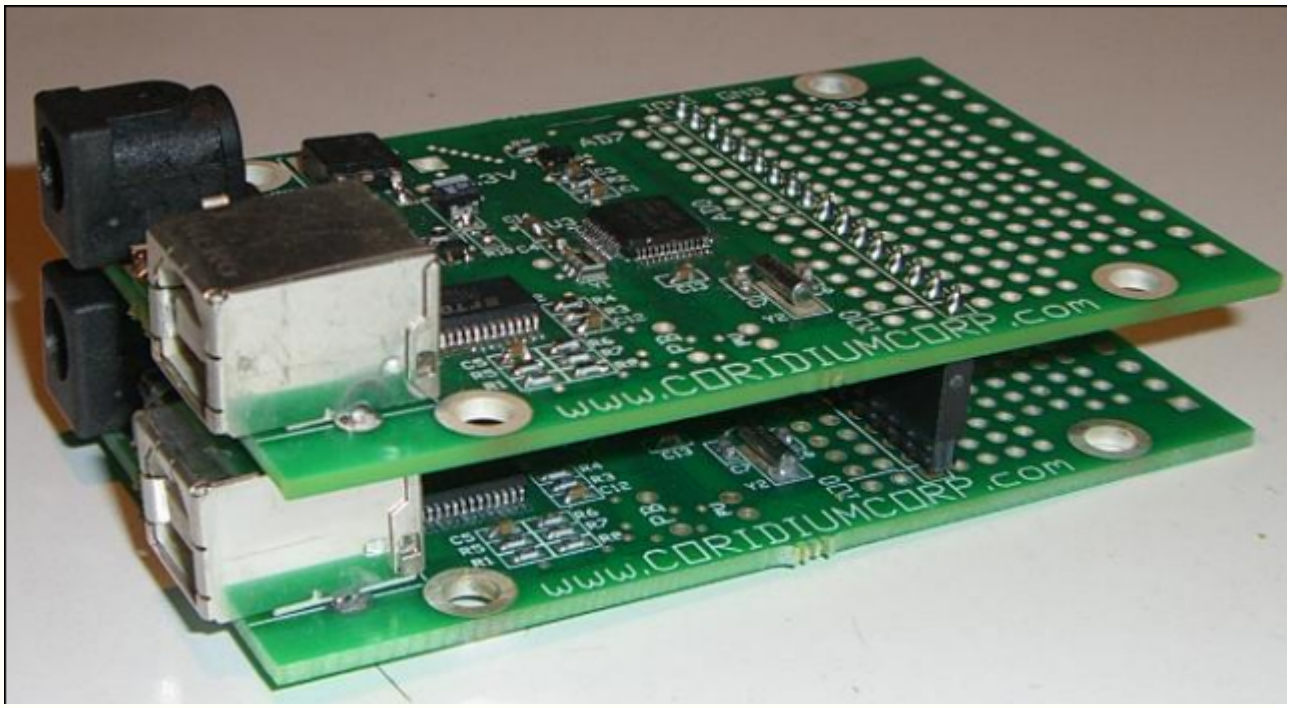
Example

Example

```
#include <LogicScope.bas>        ' call in support for LogicScope functions

#define DoSample    CALL doScopeSample     ' use this version to watch code
'#define DoSample                           ' use this version to remove LogficScope watch

'...

CALL setupLogicScope             ' initialize the sampling routine

'...

' user code for a custom serial interface

for i=0 to 8
  x = (x << 1) or (IN(3) and 1)
  DoSample
next i

 '...

CALL FlushScopeSamples           ' view any data in the buffer
```

keyw ords: Logic Scope

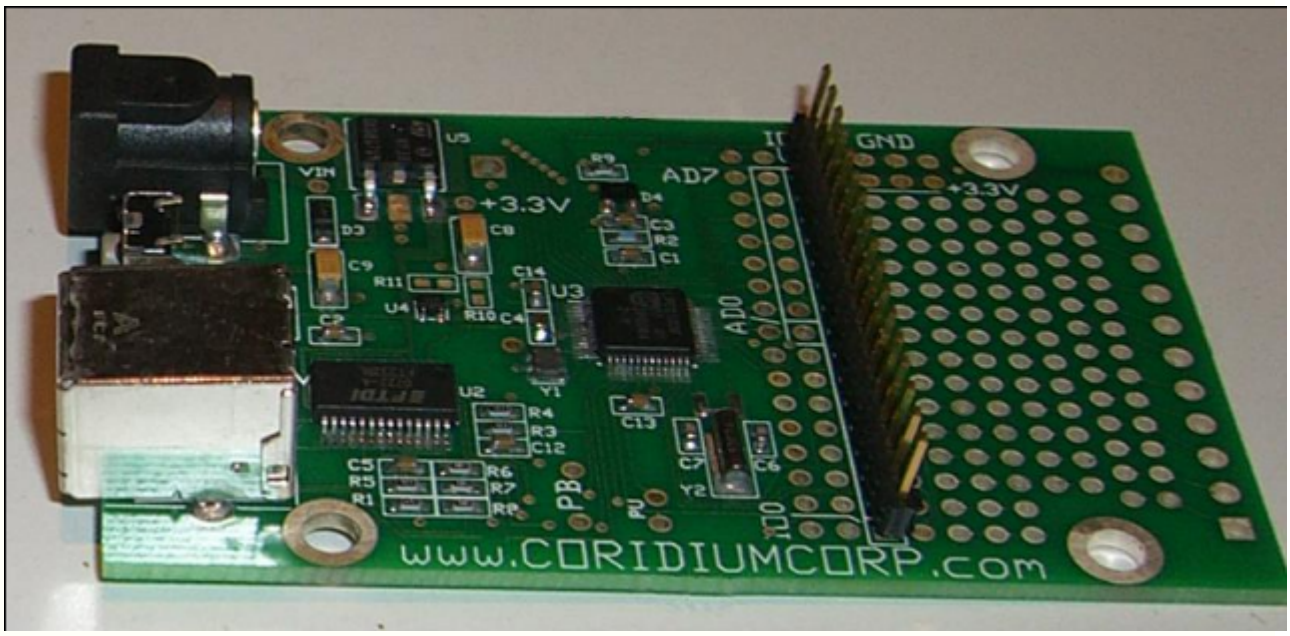
Stand Alone Logic Scope



The ARMmite is a flexible solution to capture the logic state of your project. The ability to program the control of sampling in BASIC can be a powerful tool. Using a second ARMmite means that the timing of your code will not be affected when using LogicScope.

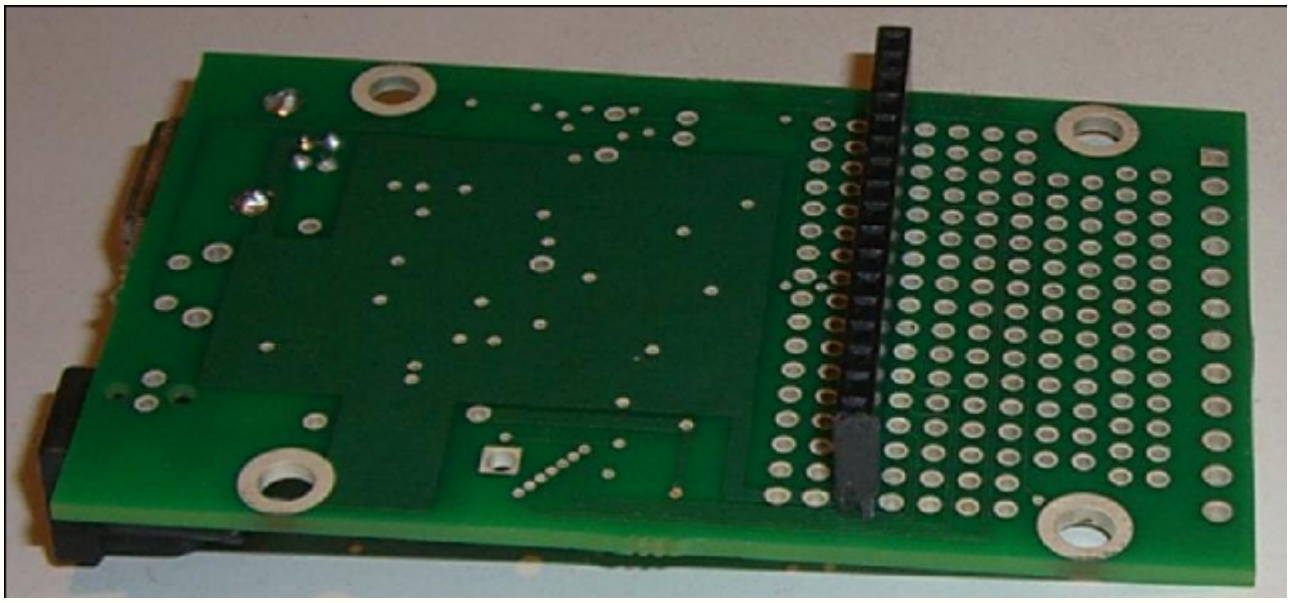
Board under test setup

..

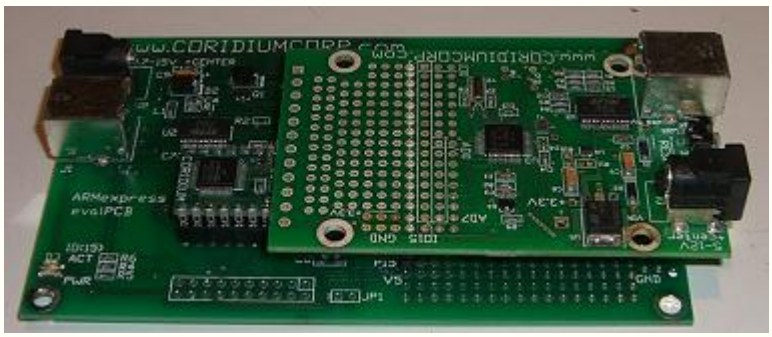


Analyzer board setup

..



ARMmite sampling data from ARMexpress/ ARMexpress LITE evaluation board



keyw ords: Logic Scope

-

Index

▪ TXFREE **U** ▪ UDPIN ▪ UDPOUT ▪ unreachable ▪ UNTIL ▪ USB Connections	▪ YEAR **Misc.** ▪ &H $ constants