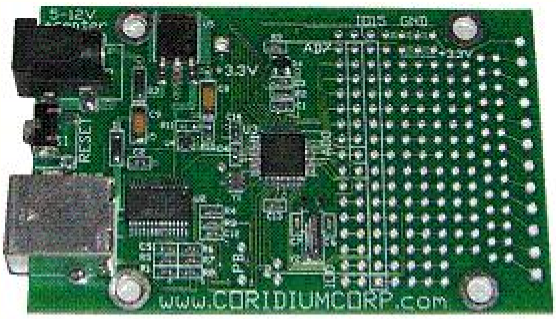
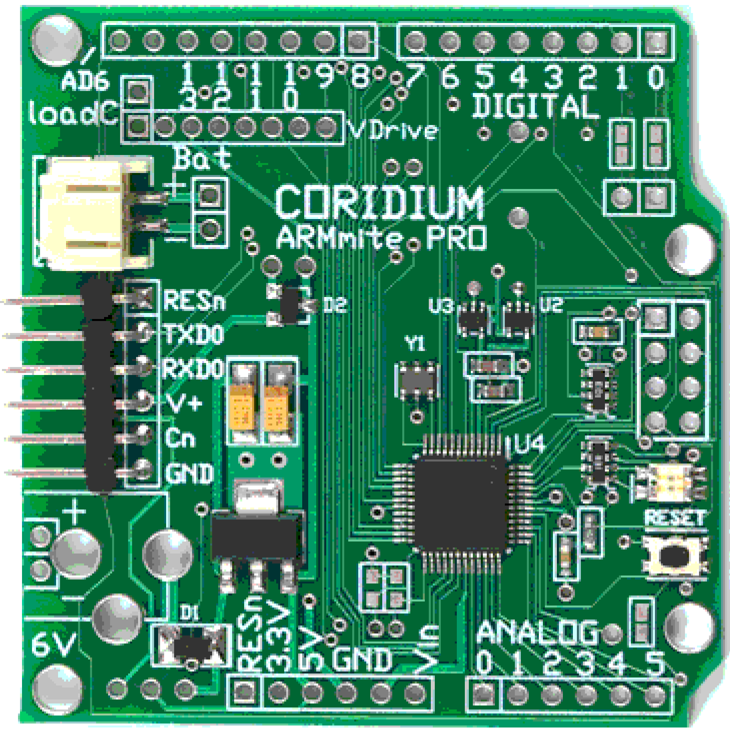
Table of Contents



Getting Started

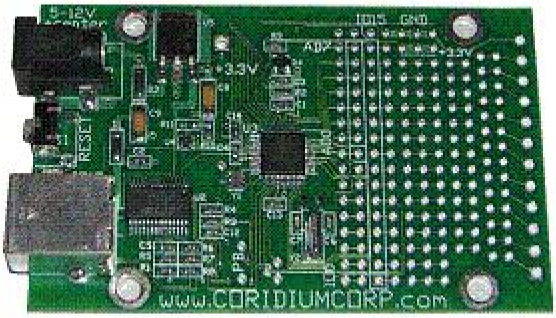


Getting Started

- PRO, PROplus and SuperPRO
- ARMmite and ARMexpress
- ARMweb and DINkit(ethernet)
- wireless ARMmite
- ARMbasic for non-Coridium Hardware

ARMmite, ARMmite PRO and ARMexpress Getting Started

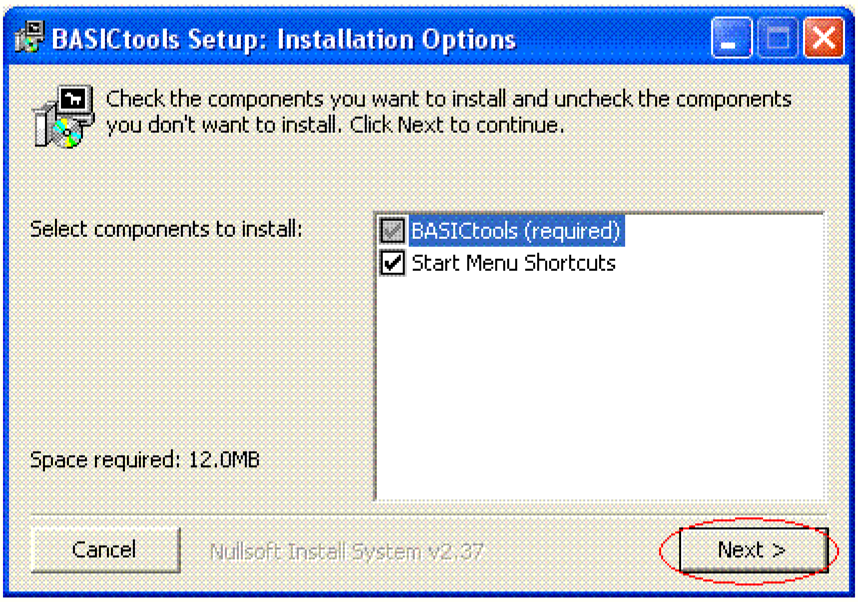
Getting Started	Programming the IO
Install Software	More complex programs
Connect ARMmite	Trouble Shooting
Connect PRO family	BASICtools Features
Writing your first program	



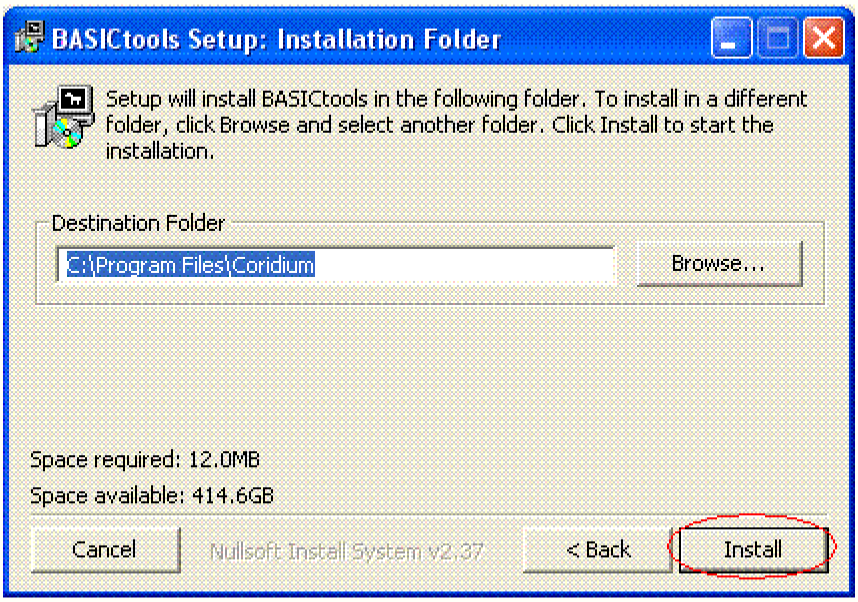
Step 1: Install Software

The **ARMexpress** family 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 ARMexpress and ARMmite. Also, a number of help files and documents about the ARMexpress will be installed on the machine at this time. This installer is meant for WIndows either 98, NT, XP or XPx64 and Vista.

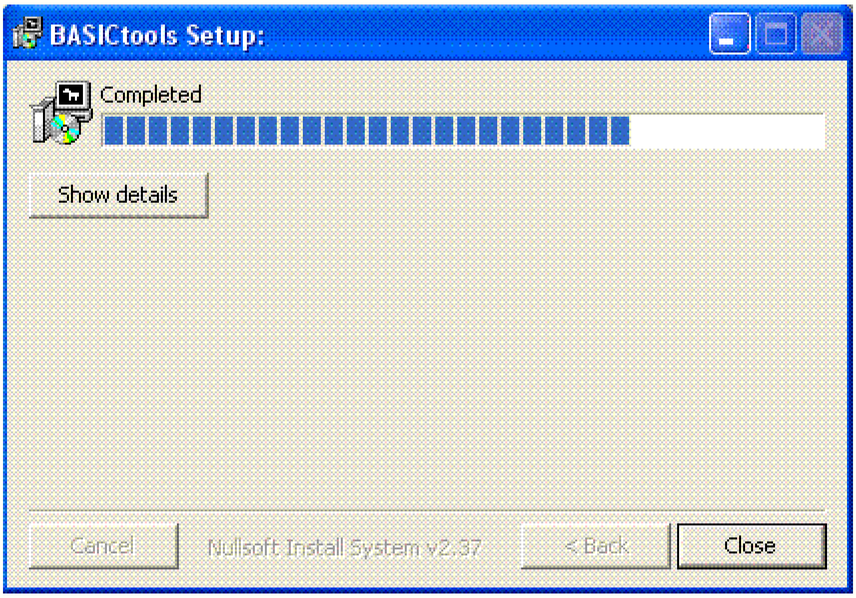
If you are installing from the CD, then it will automatically run the install program when the CD is inserted. If downloading from the web, run the SETUP program to start the installation.



Click **Next** to get started.



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



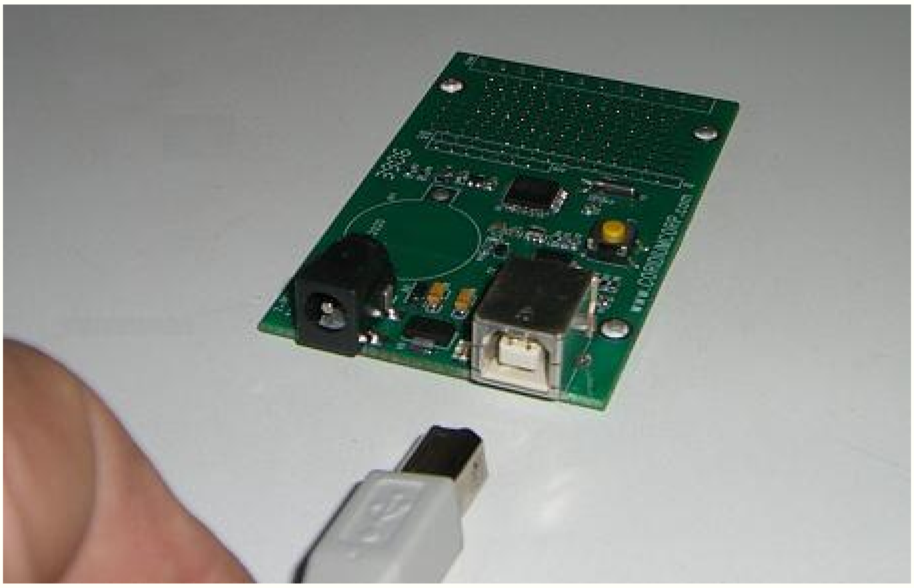
The installation will now run, and when it finishes hit **Close** .

And its as easy as that.

On to Step 2

Step 2: Connect USB

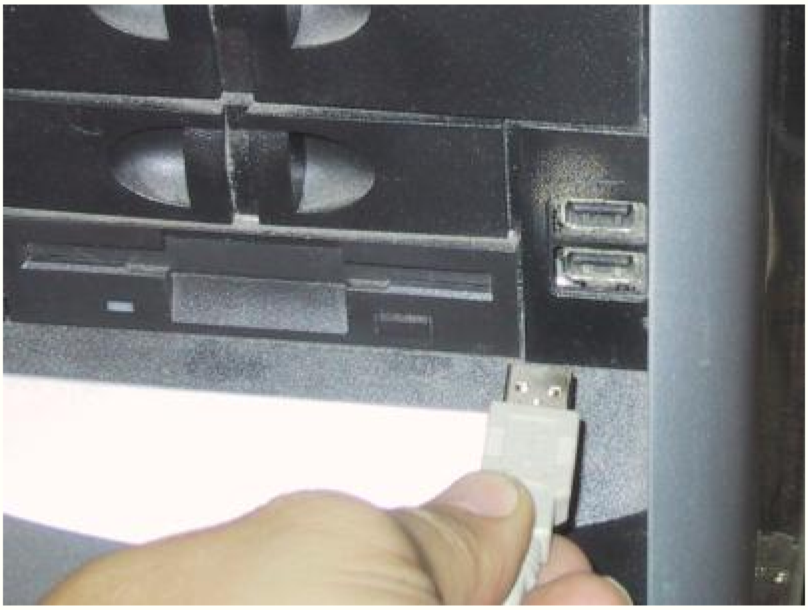
Connect USB Cable to ARMmite/ARMexpress Eval PCB/ARMmite PRO



For details on connecting the ARMmite PRO visit this page.

The **ARMmite** / **ARMexpress** Eval Kit comes with a USB cable. This cable allows you to connect the **ARMmite**/**ARMexpress** directly to a computer equipped with USB. Locate the USB jack on the side of the Eval PCB and plug one end of the USB cable into it. 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 USB Cable to Computer



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

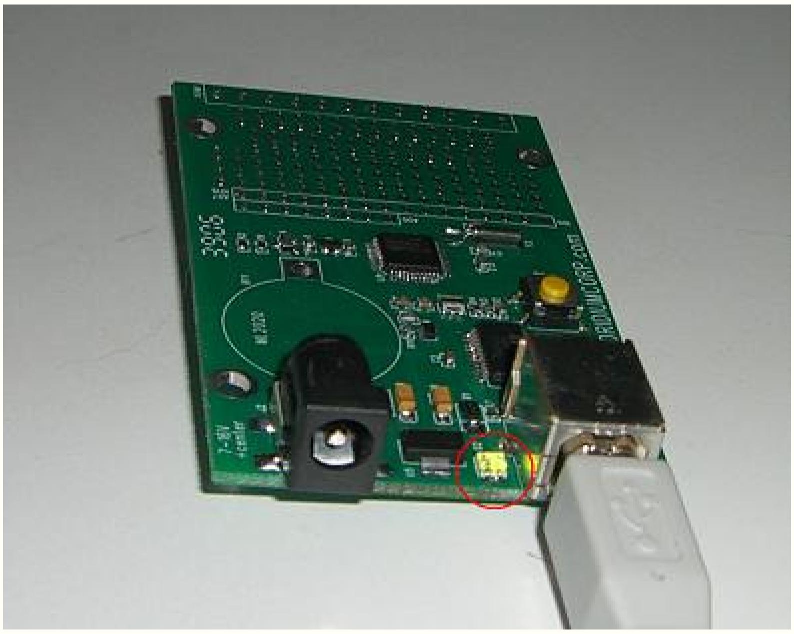
Please Consult Installation Guides

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. 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.

Driver Installation Complete, Confirm USB Connection



The Eval PCB or the ARMmite will be powered from the USB bus. It may also be connected to a 5-12V DC power source simultaneously.

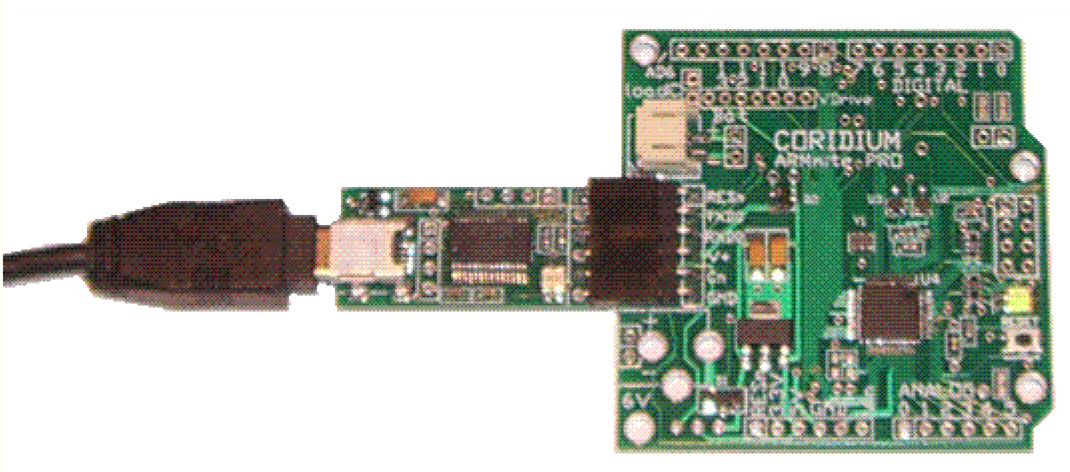
To verify connection with the USB and PC the LED on the Eval PCB should light up.

On to Step 3

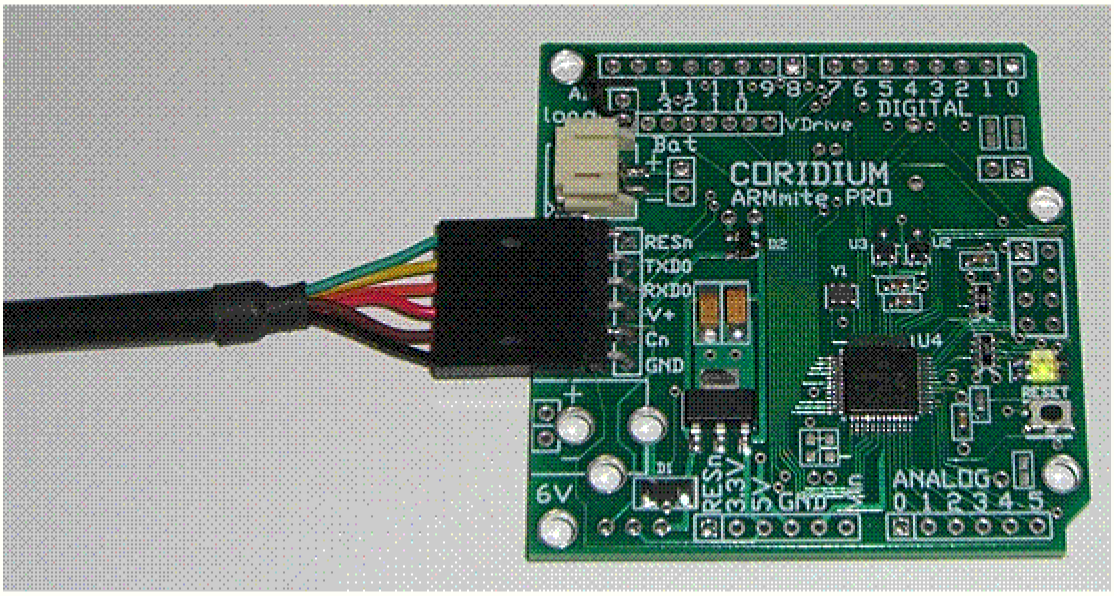
Step 2: Connect USB on ARMmite PRO family

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.

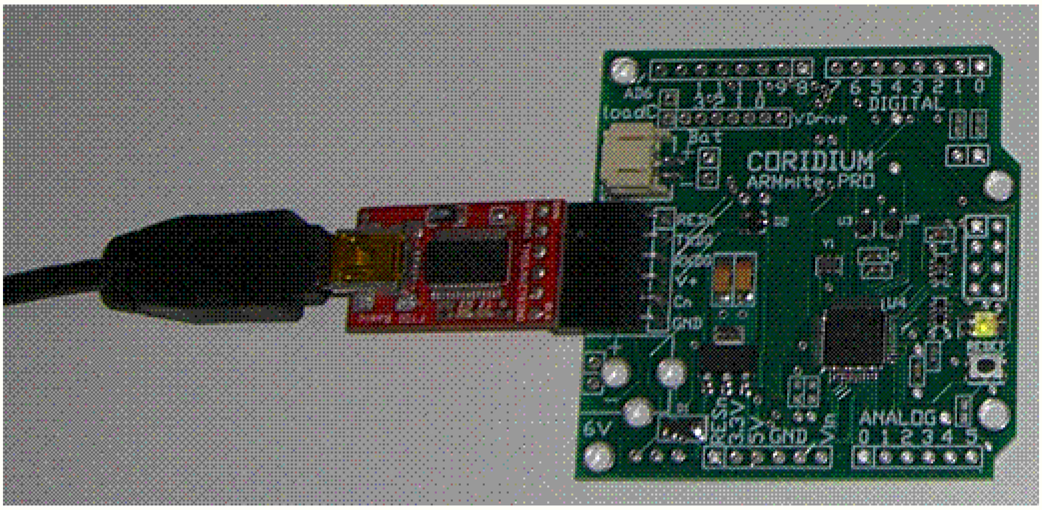


Connect FTDI cable to ARMmite PRO



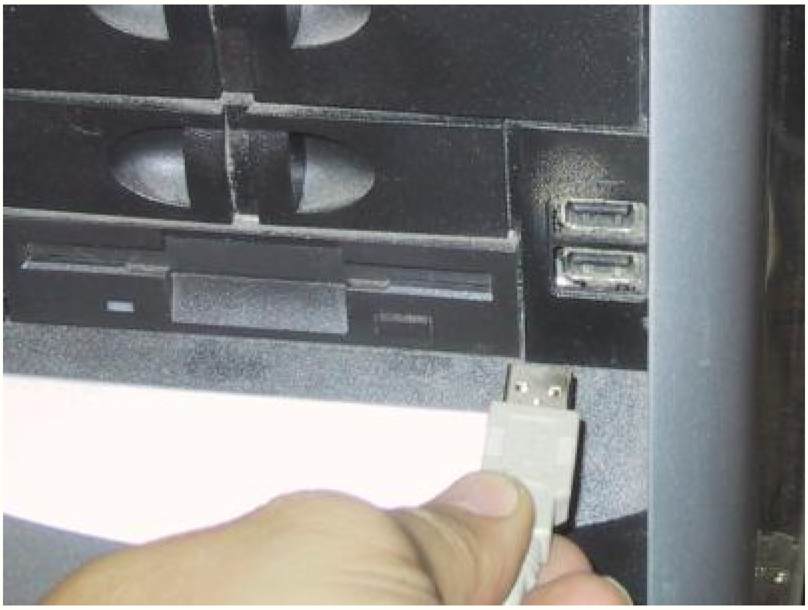
Connect black wire to GND. This cable is available at **Digikey** or the **Makershed**. This cable connects RTS to RESETn, BASICtools support this.

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.

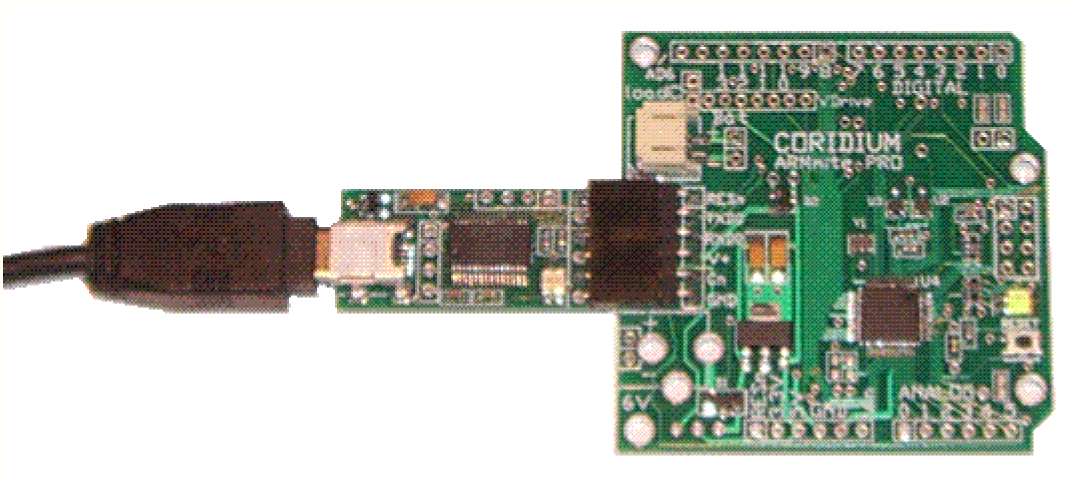
Please Consult Installation Guides

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. 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.

Driver Installation Complete, Confirm USB Connection



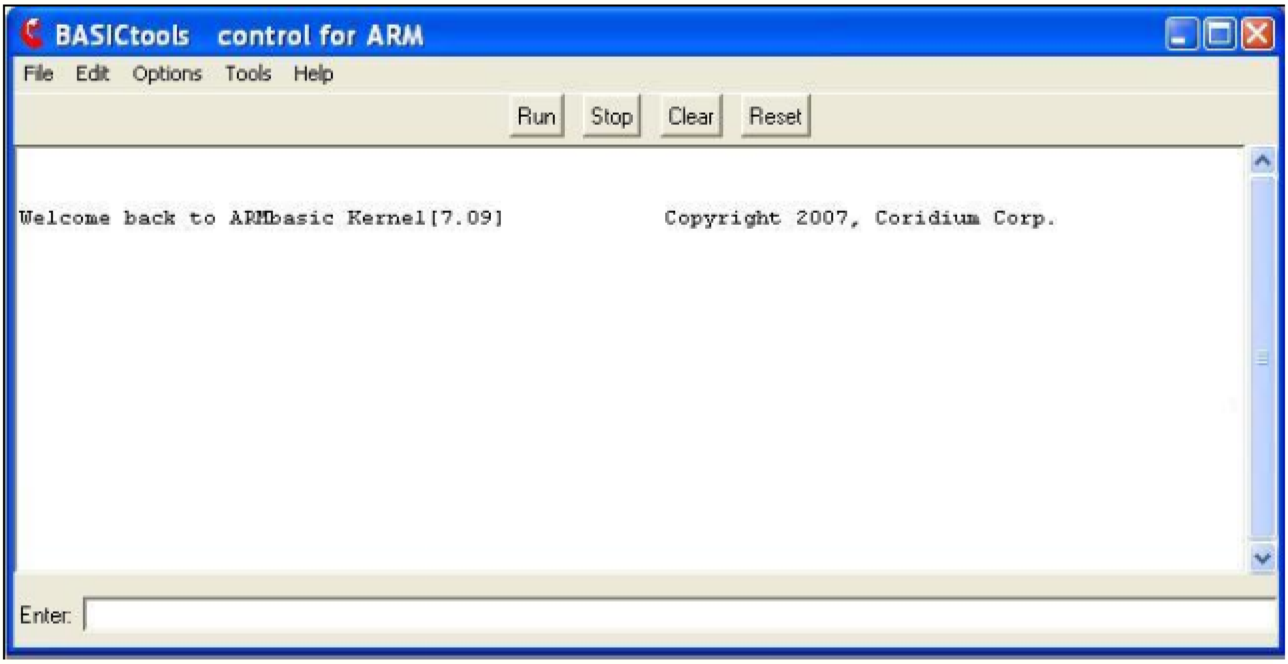
The ARMmite PRO will be powered from the USB bus, when using either the Coridium Dongle or the FTDI 5V cable. It may also be connected to a 6-7V DC power source simultaneously.

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

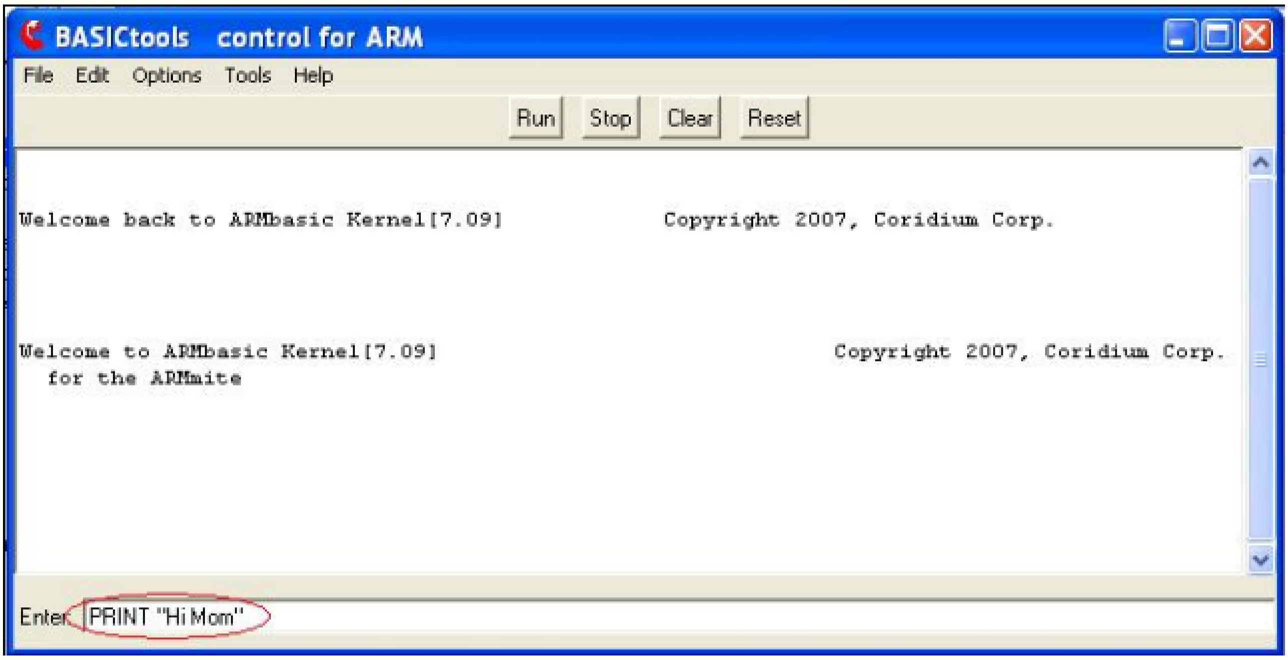
Start the BASICtools from the StartMenu or from the Desktop Icon.   You should see a welcome message which has been sent from the ARMmite or ARMexpress-



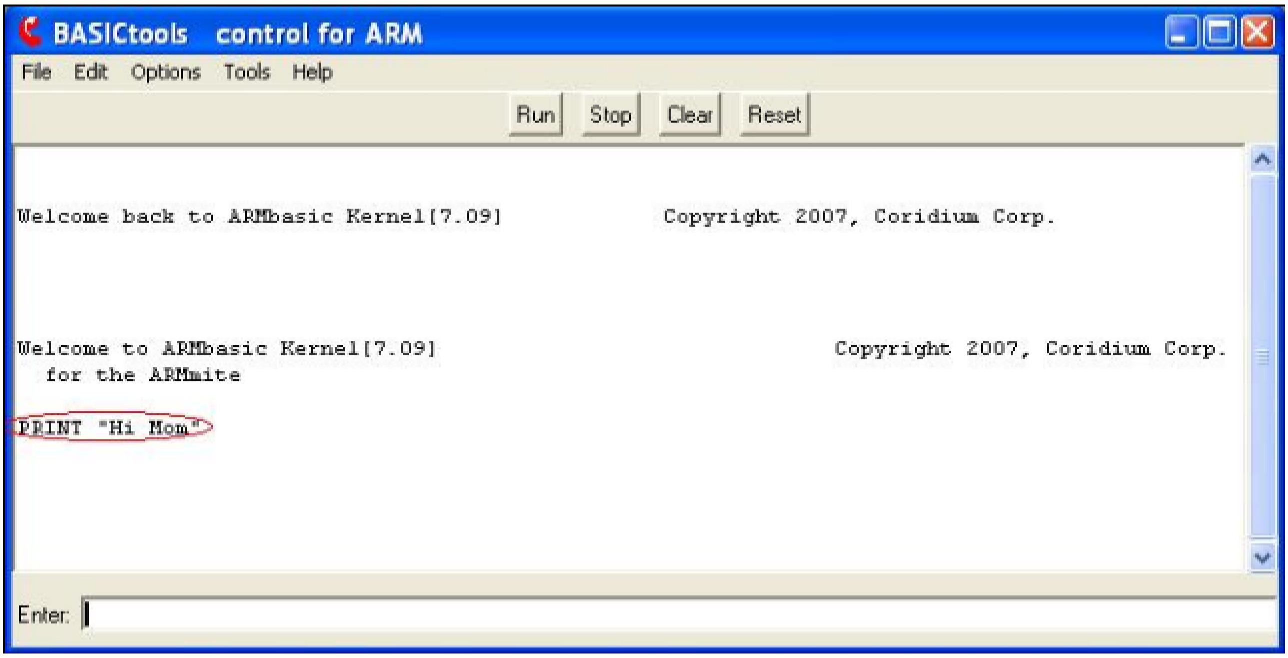
If you do not see this welcome, even after pushing the RESET button, then communication has not been established.

- 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**

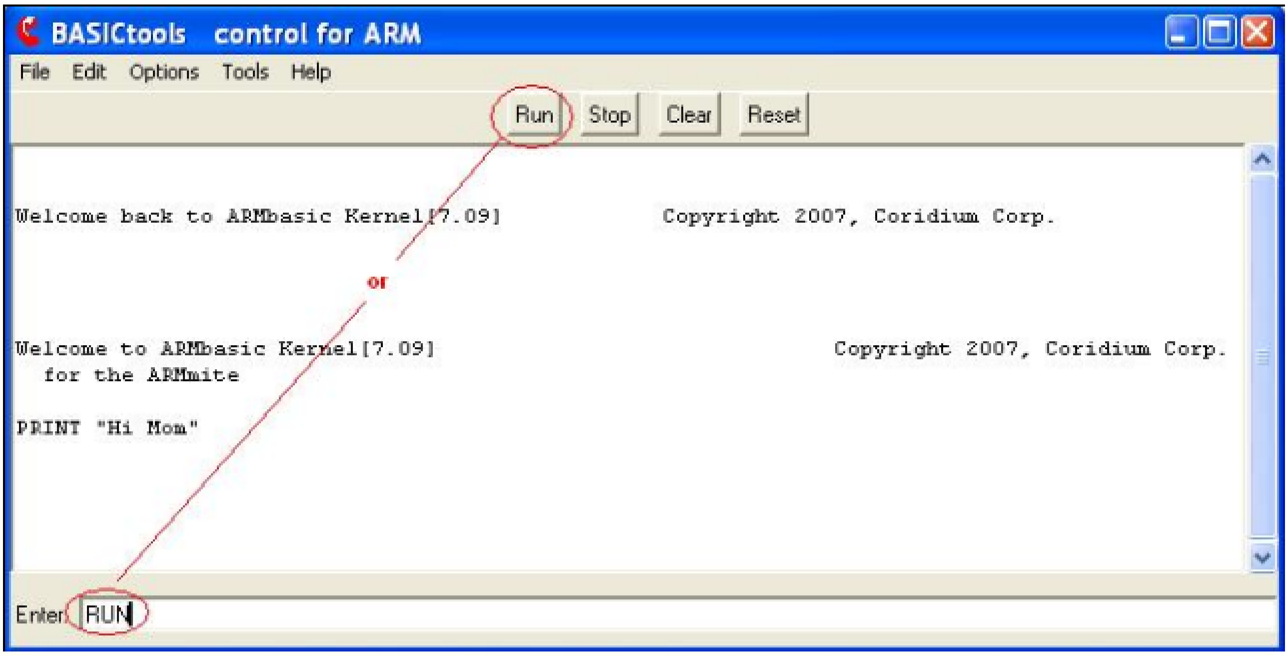
The traditional "Hi Mom" program



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

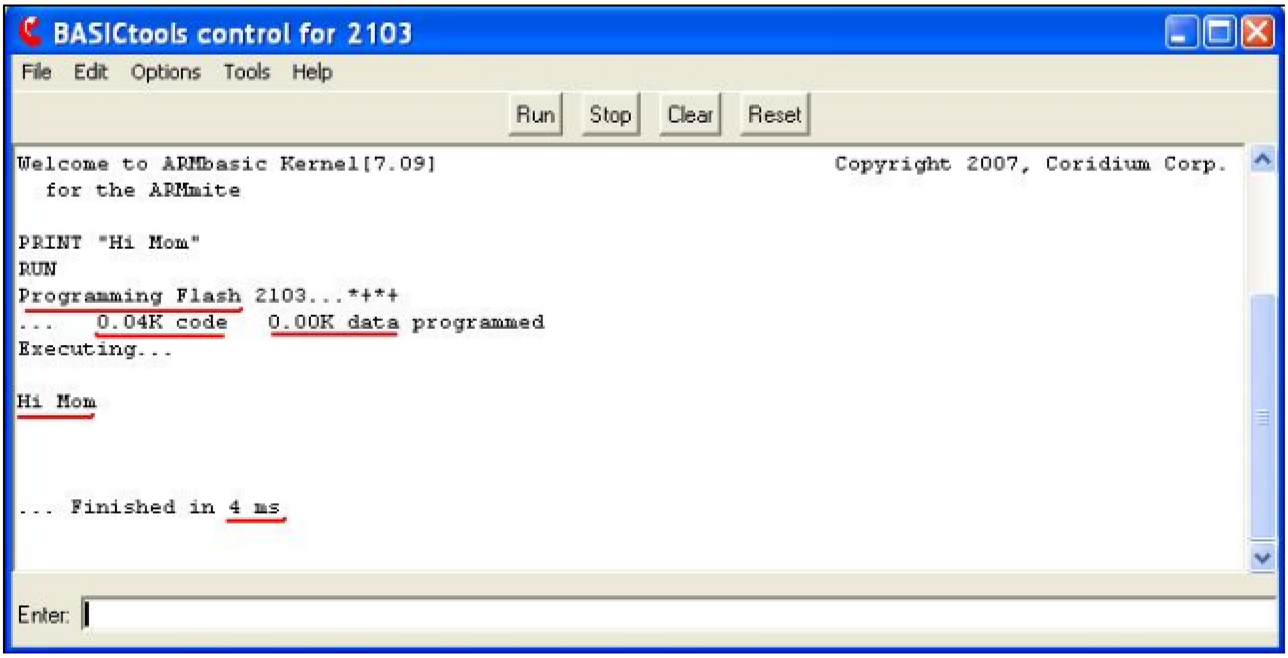


Now RUN the program



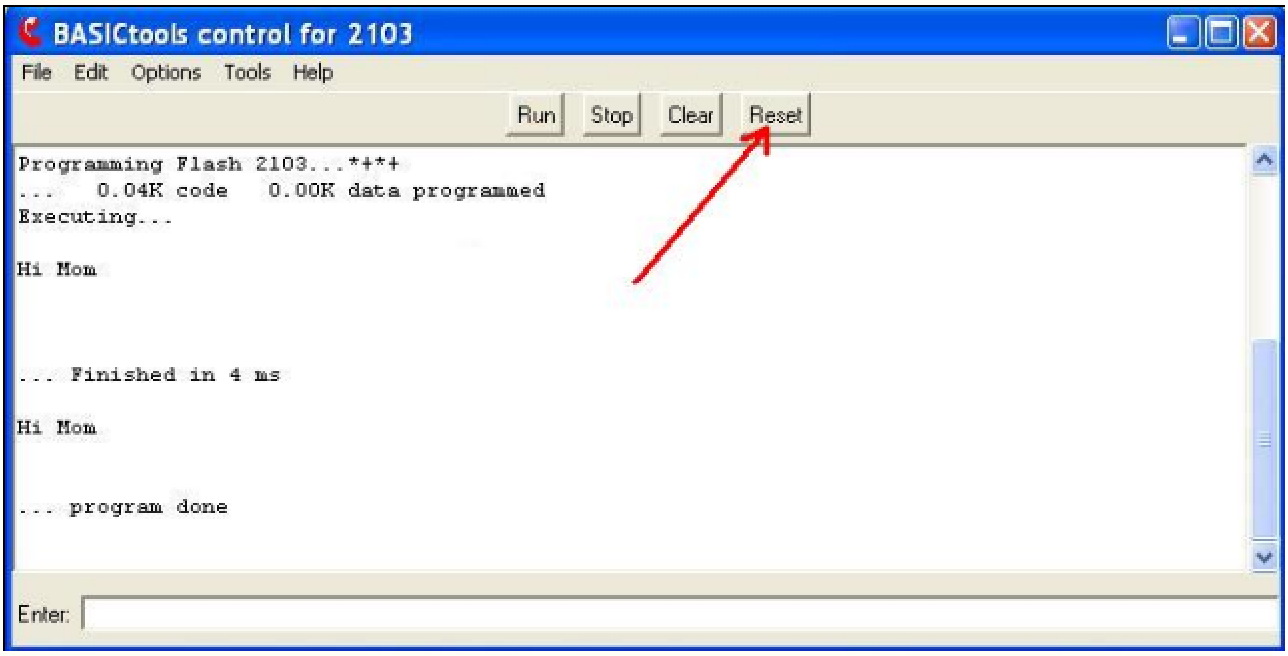
Which you can do by either typing RUN or hitting the RUN button at the top of the screen.

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 Mom" to the console. ARMexpress also reports back how long the program executed, in this case 4 msec, which is mostly startup time.

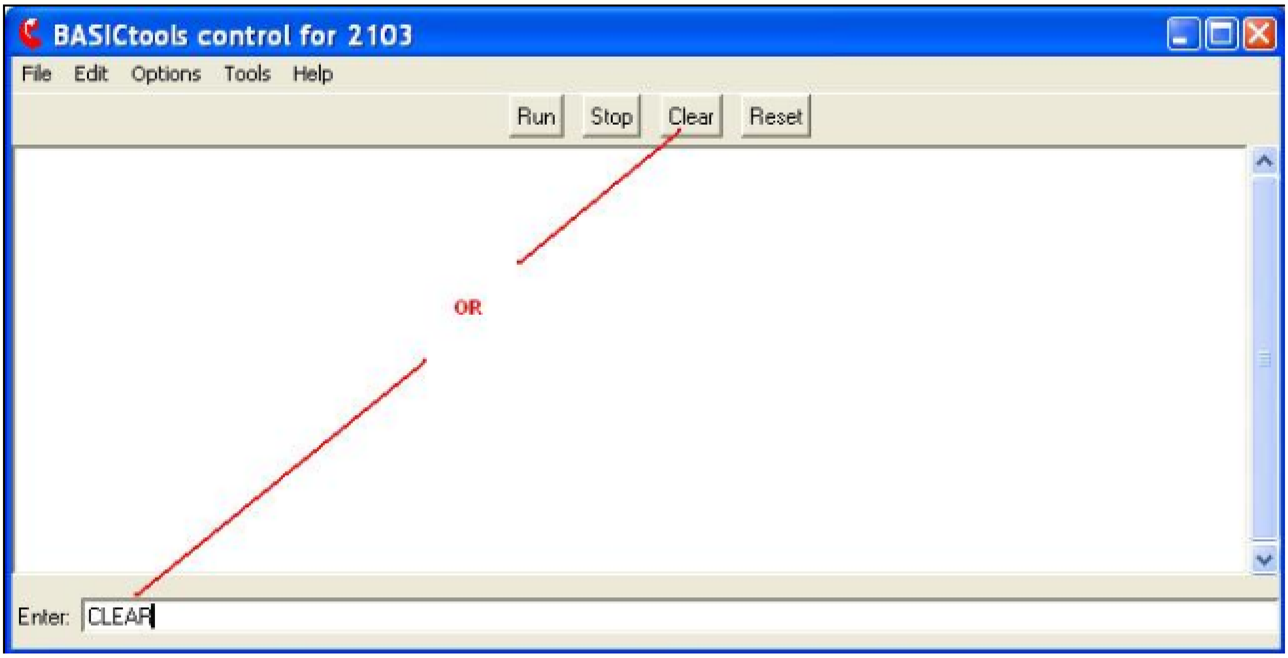
Also your program is now saved in the ARMmite/express Flash memory. And it will be executed the next time the board is RESET. So try that...


BASICtools control for 2103
File Edit Options Tools Help
Run Stop Clear Reset
Programming Flash 2103...*+*+
... 0.04K code 0.00K data programmed
Executing...
Hi Mom
... Finished in 4 ms
Hi Mom
... program done
Enter:

On to Step 4

Step 4: Programming the IO

Clear previous ARMmite/ARMexpress 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

Type the following program in the console window. (below)

```
DIR(15)= 1                ' enable pin 15 as an output
WHILE X<30
  OUT(15) = X AND 1       ' drive pin 15 high when x is odd, low when x is even
  X=X+1
  WAIT(500)
LOOP
```

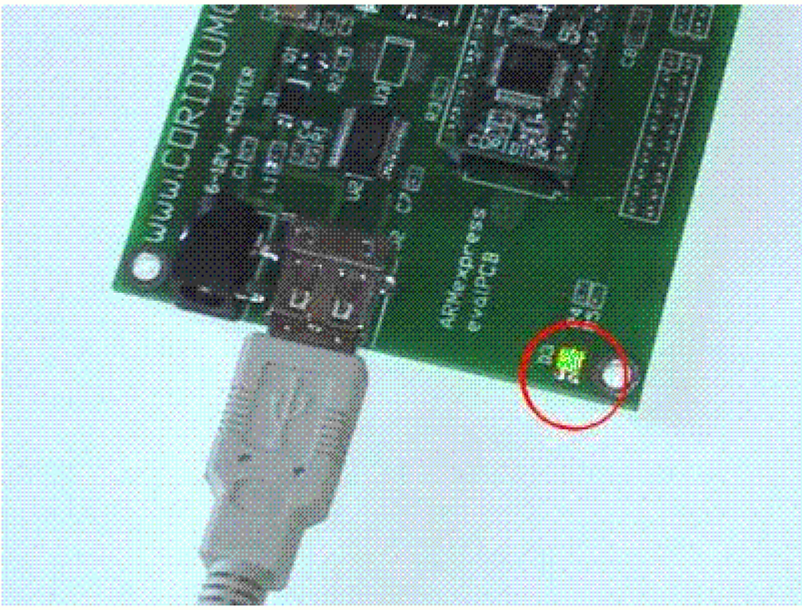
For the SuperPRO and PROplus, the LED is connected to P2(10). Use the following

```
  FIO2DIR = &H2009C040   ' this is the DIR register for port 2, its also defined in #include <LPC17xx.bas>
  *FIO2DIR = 1<<10
  WHILE X<30
     P2(10) = X and 1
      X=X+1
     WAIT(500)
  LOOP
```

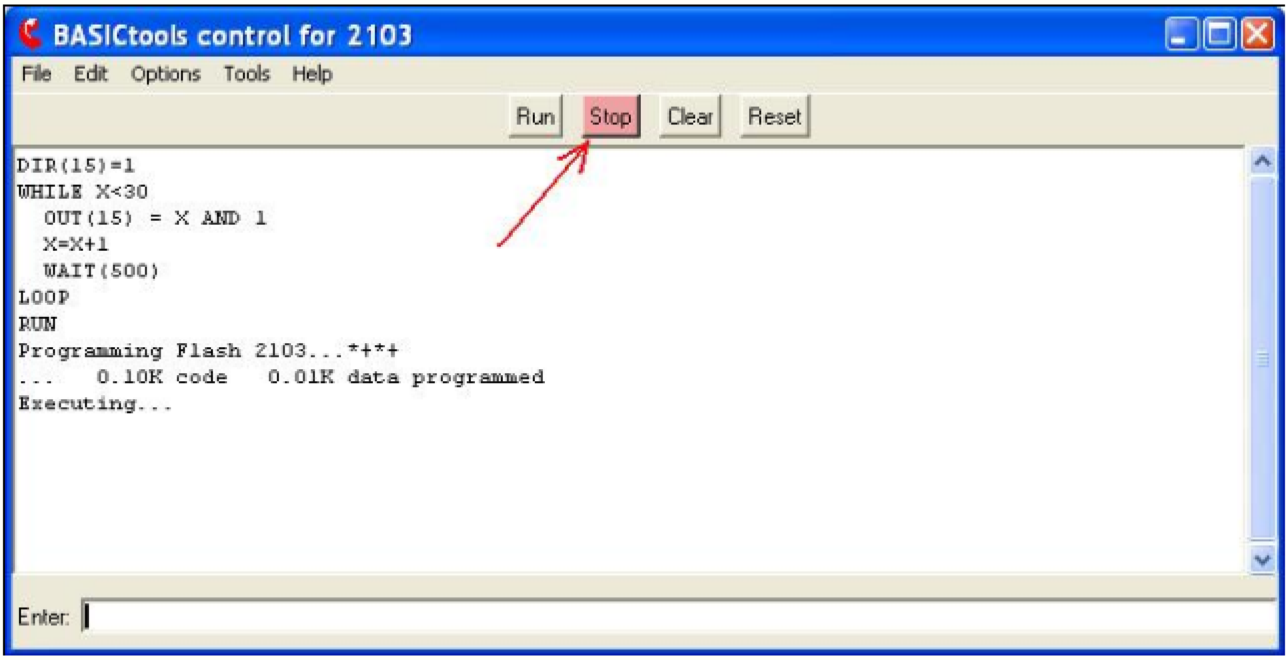
Now RUN the program

The LED on the PCB should pulse 15 times.

<u>And see the results</u>



<u>Stop the program</u>



To stop a running program simply press the Stop button.

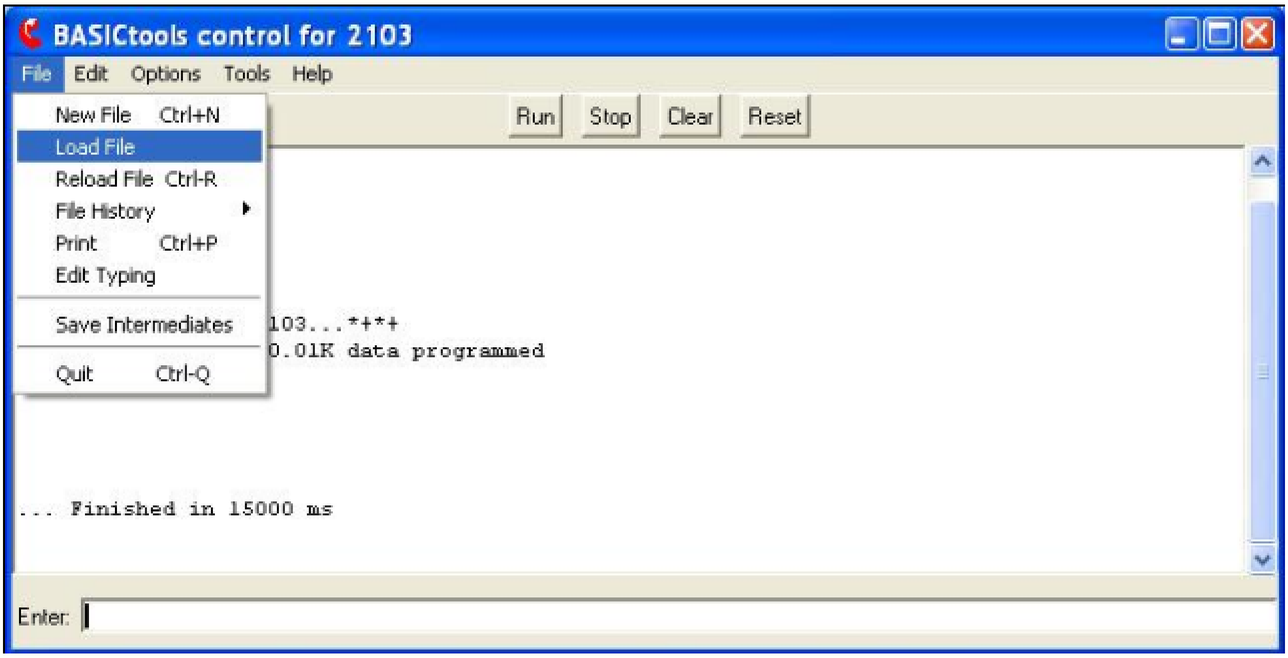
On to Step 5

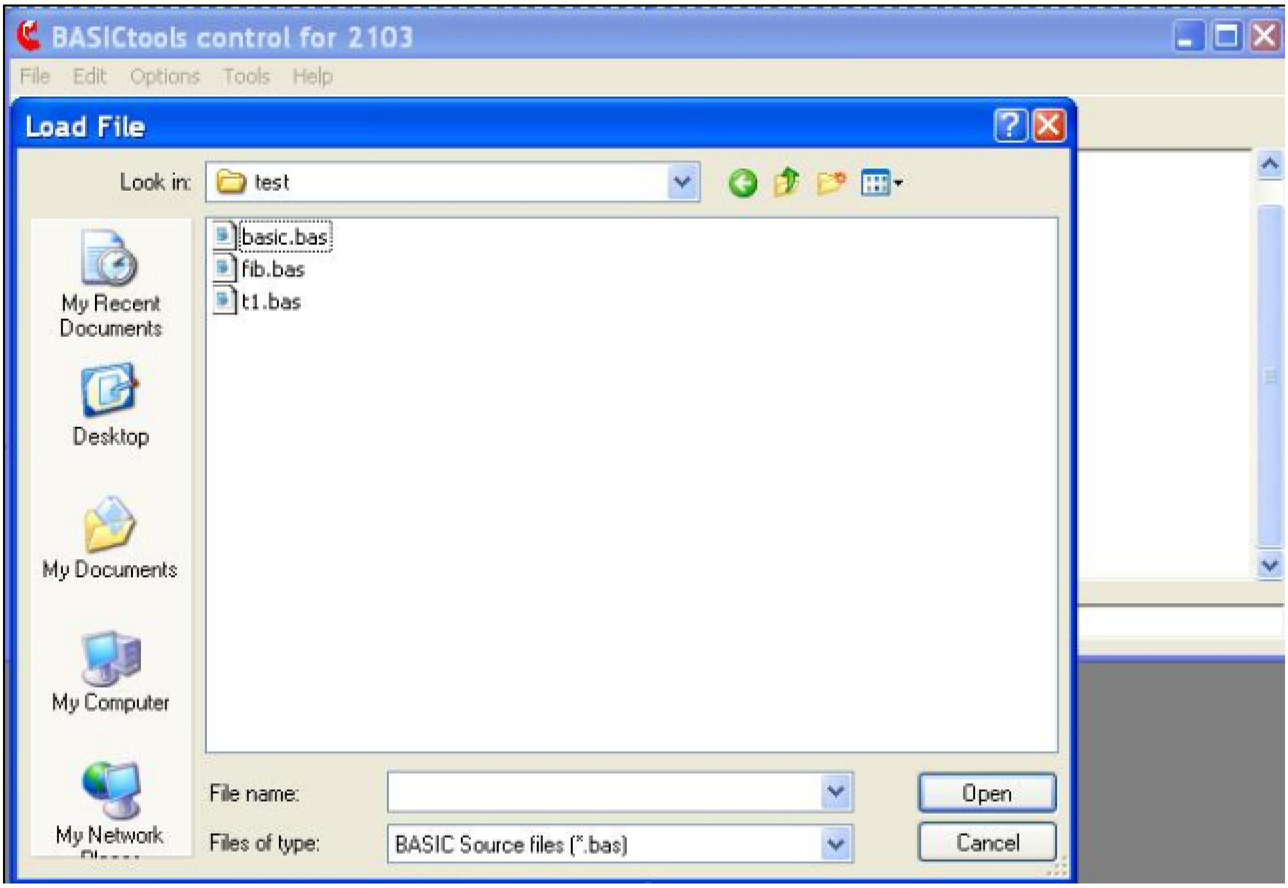
Step 5: More Complex Programming

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 **Yahoo Forum** for files and help from other users of ARMbasic products. There are also examples on the **Coridium Website Programming pages.**

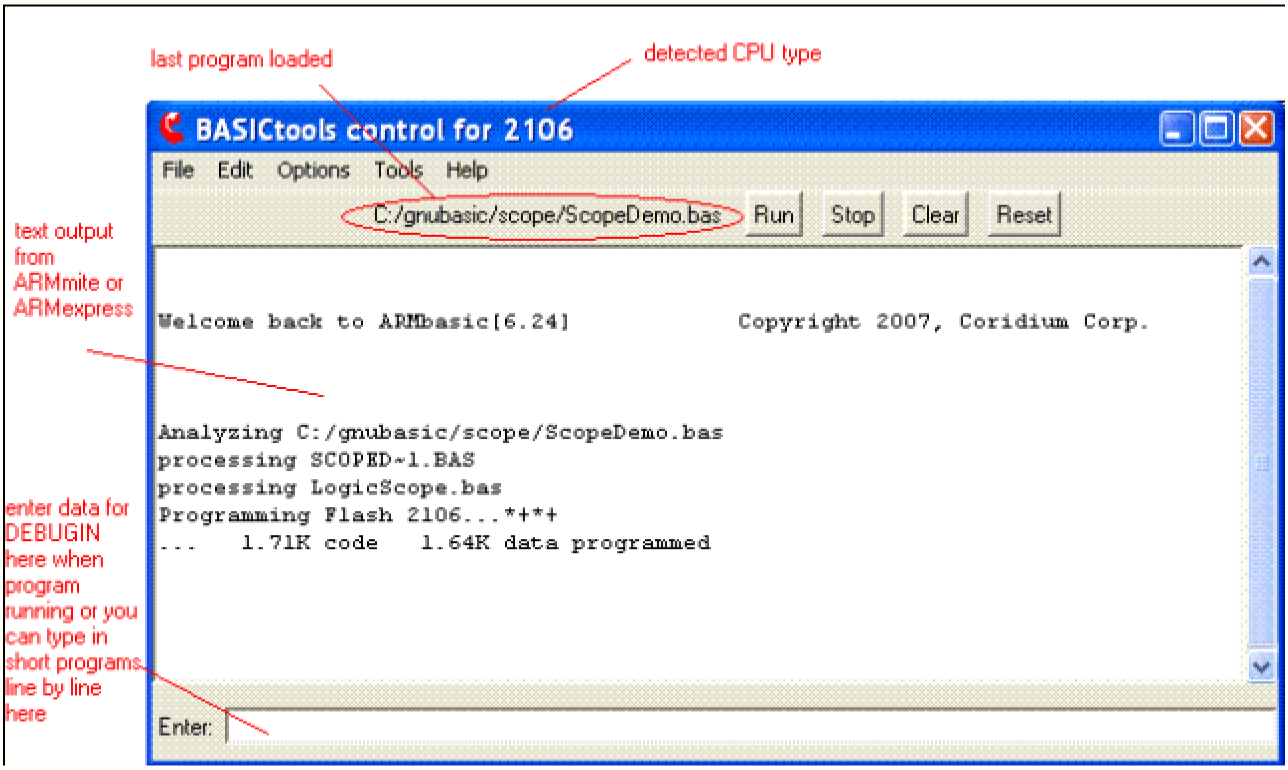
For more details on the BASICtools IDE check the **next page**.

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, close BASICtools and then restart it (you may need to use the Task Manager to exit). It will STOP your spewing program.

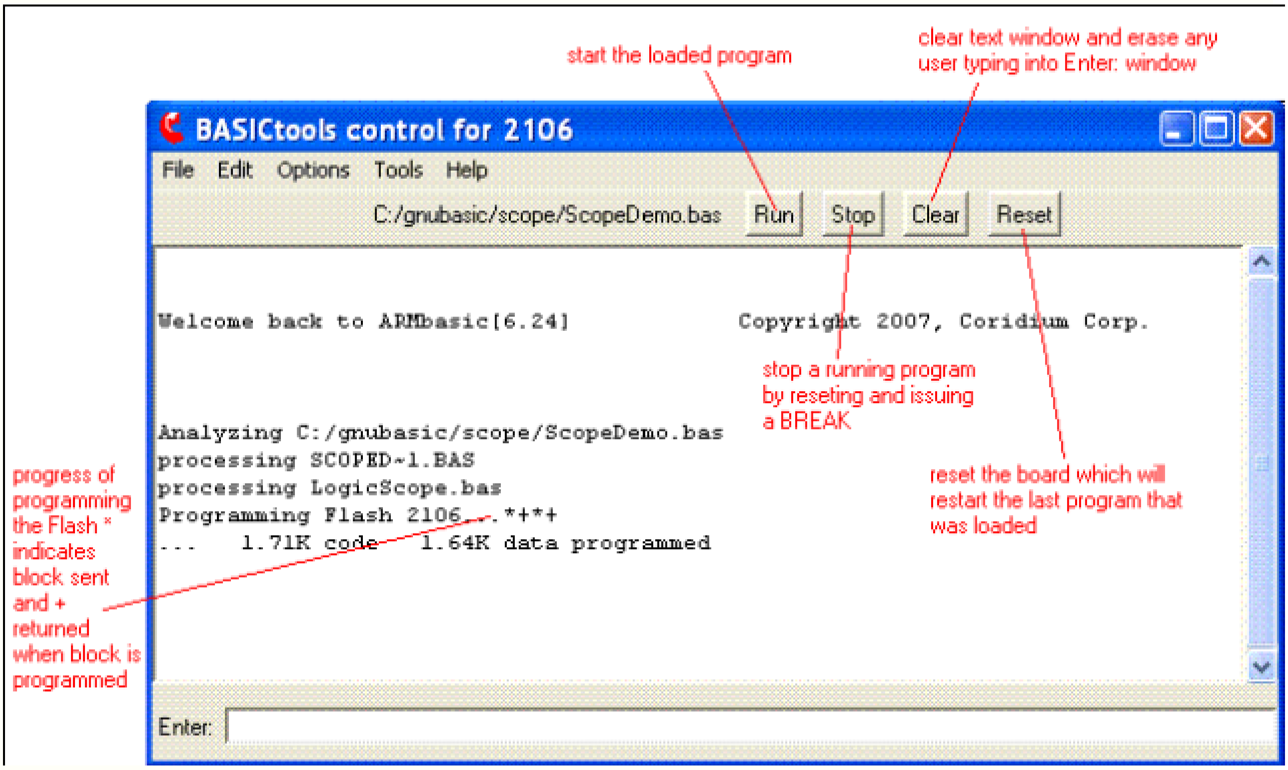
BASICtools Layout



keyw ords: enter line debugin type BASIC commands

Buttons

..



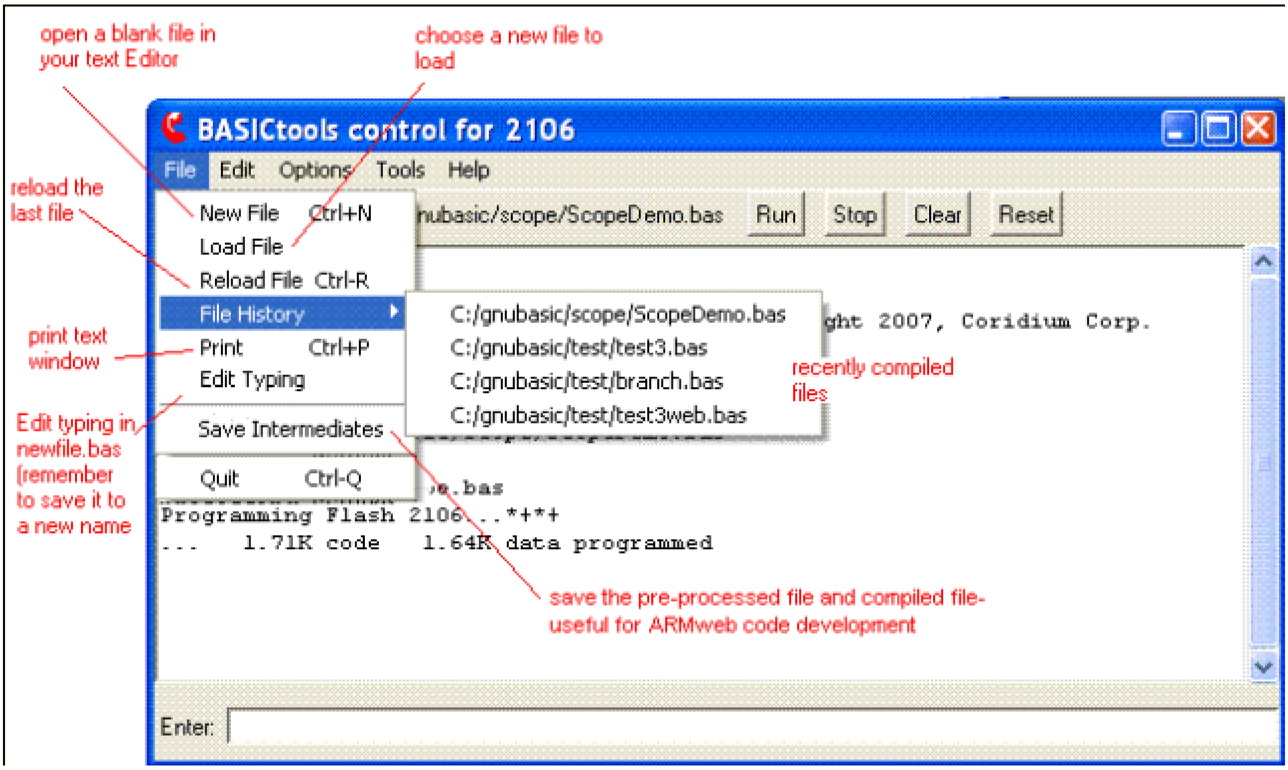
The CLEAR button only erases the display screen and the buffer on the PC of statements you have typed into the Enter window.

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

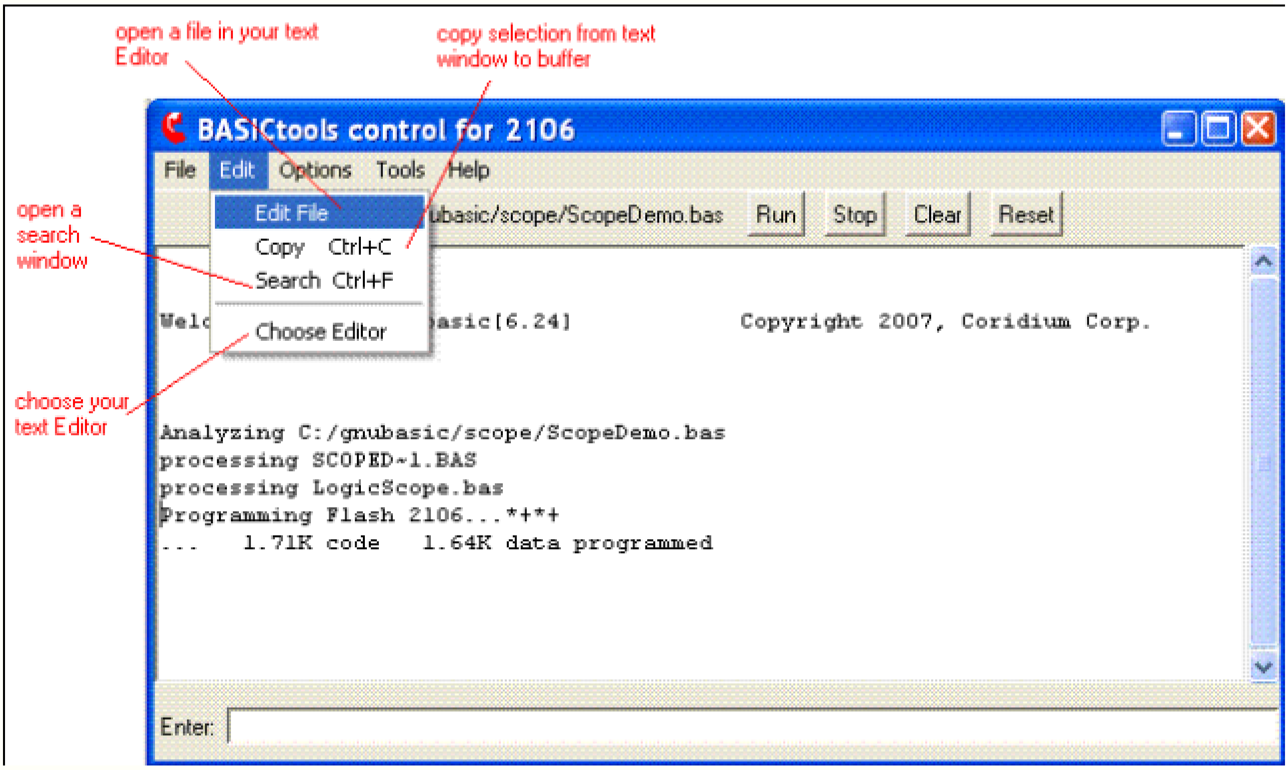
..



file load file reload file print save file quit

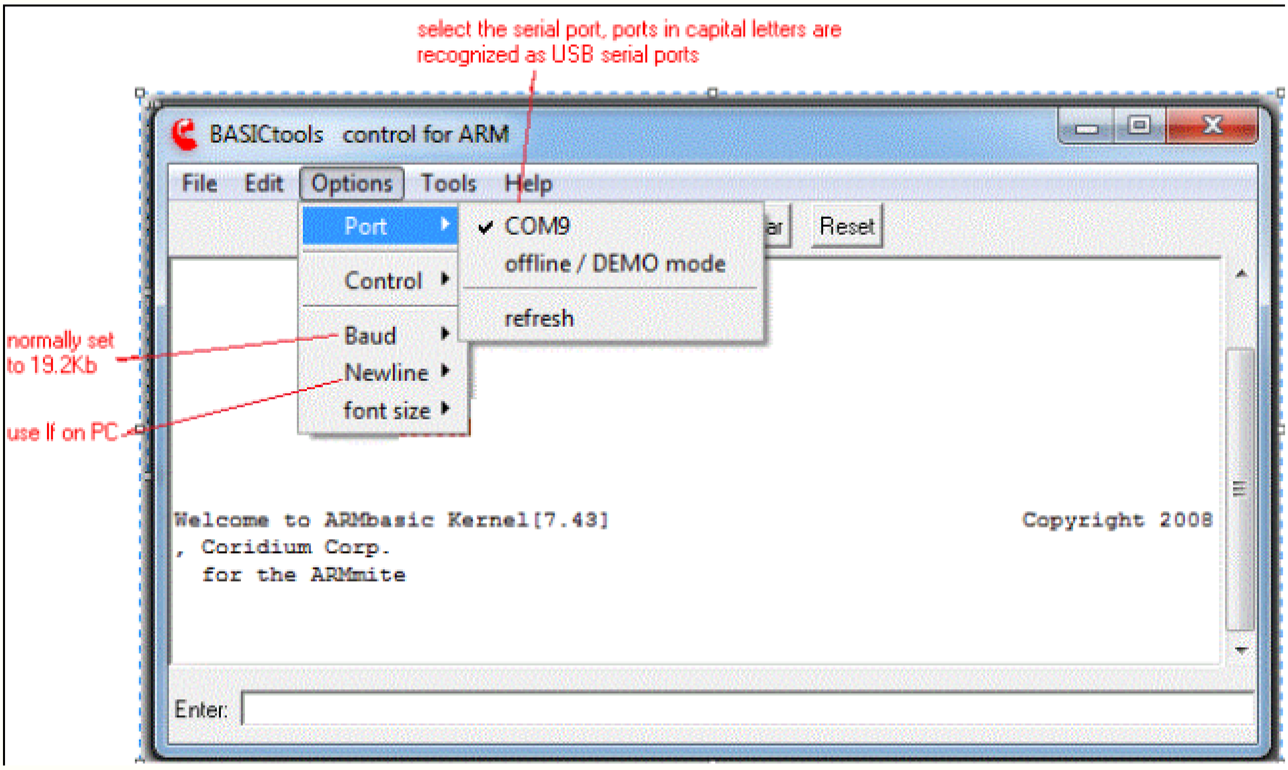
Edit Menu

..



keyw ords: edit choose editor

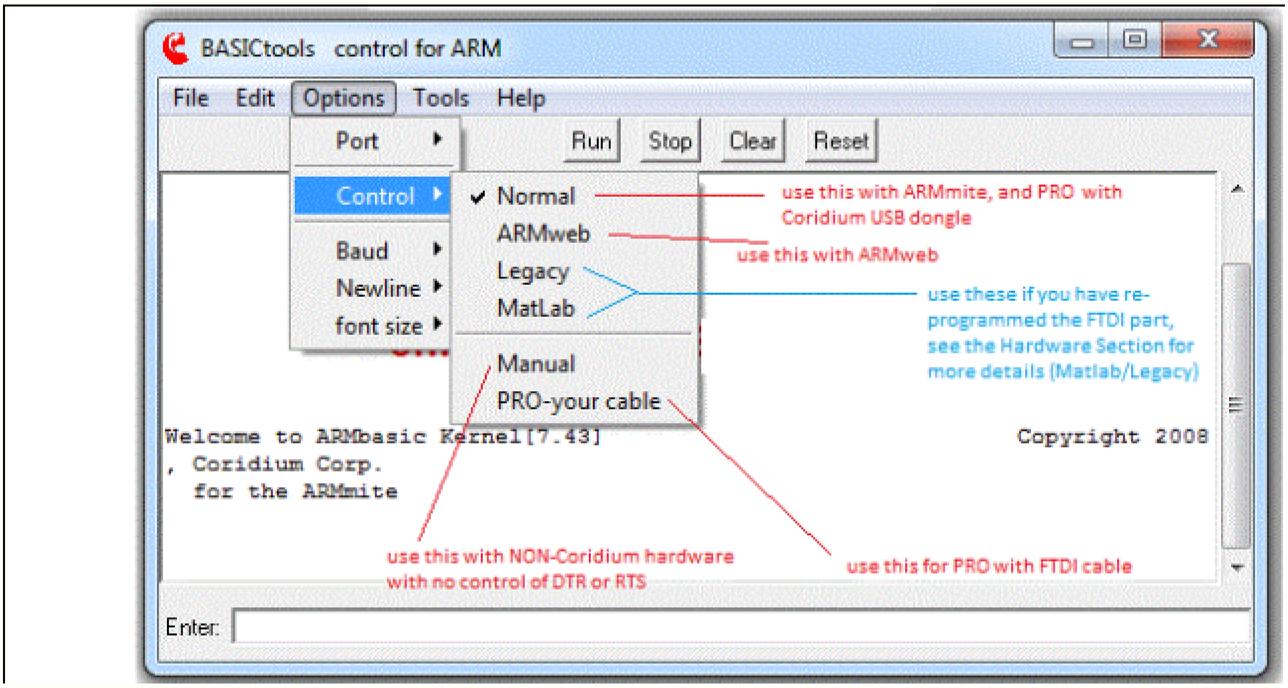
Options Menu



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

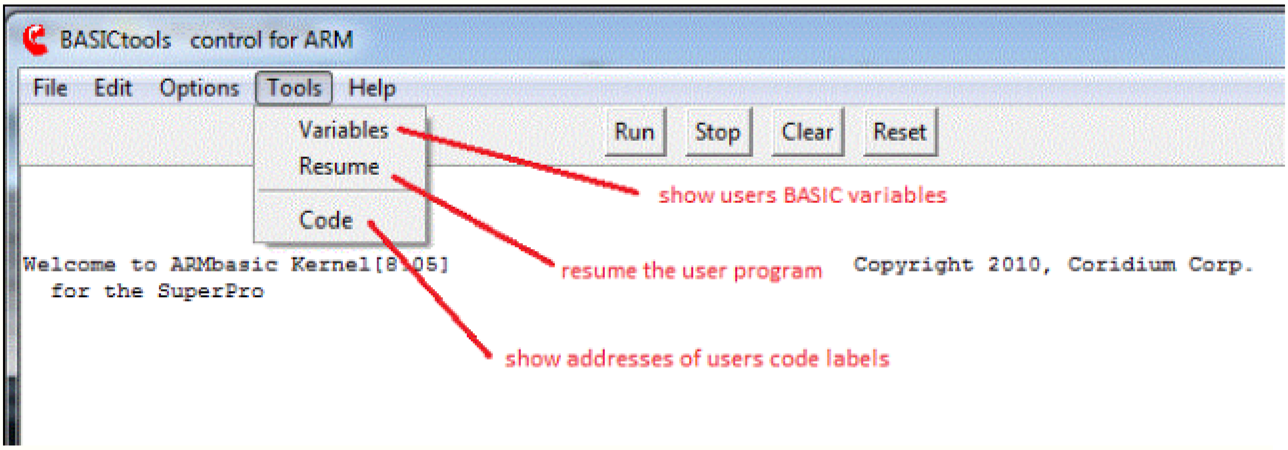
Control Menu



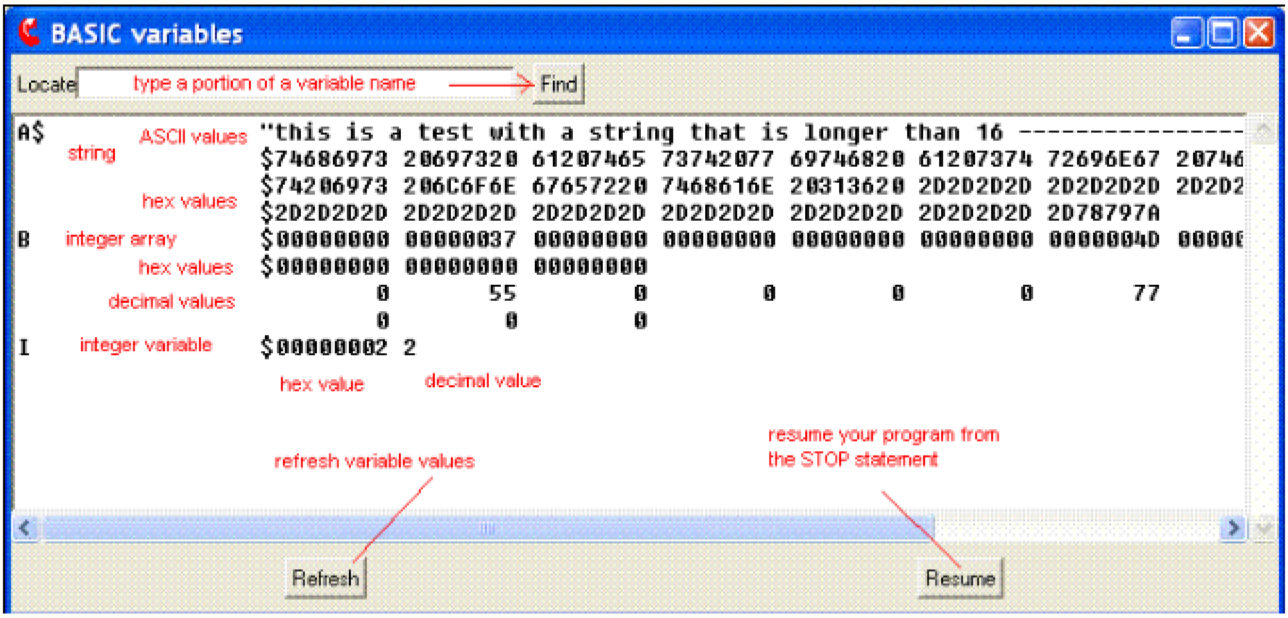
keyw ords: options port baud new line char mode PC compile control throttle

BASIC variable viewer

Open this window from the Tools Menu (variables)

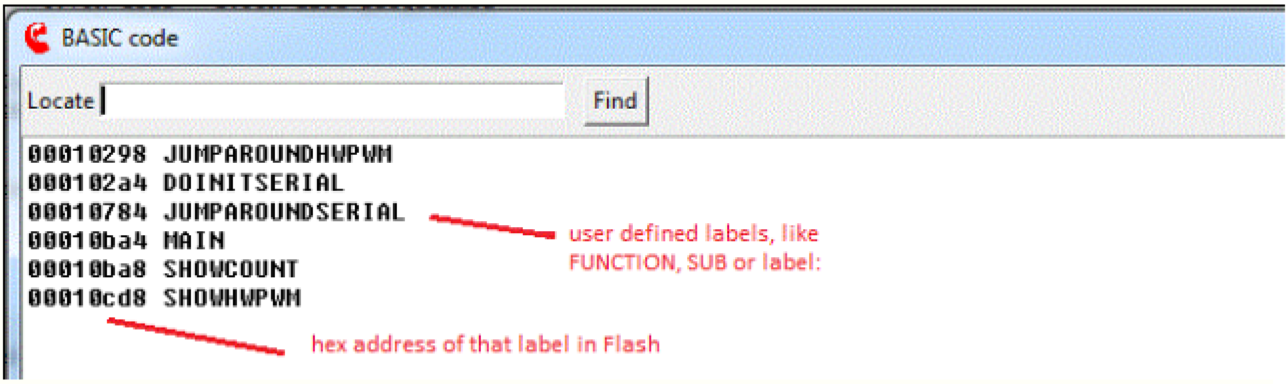


variables window



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

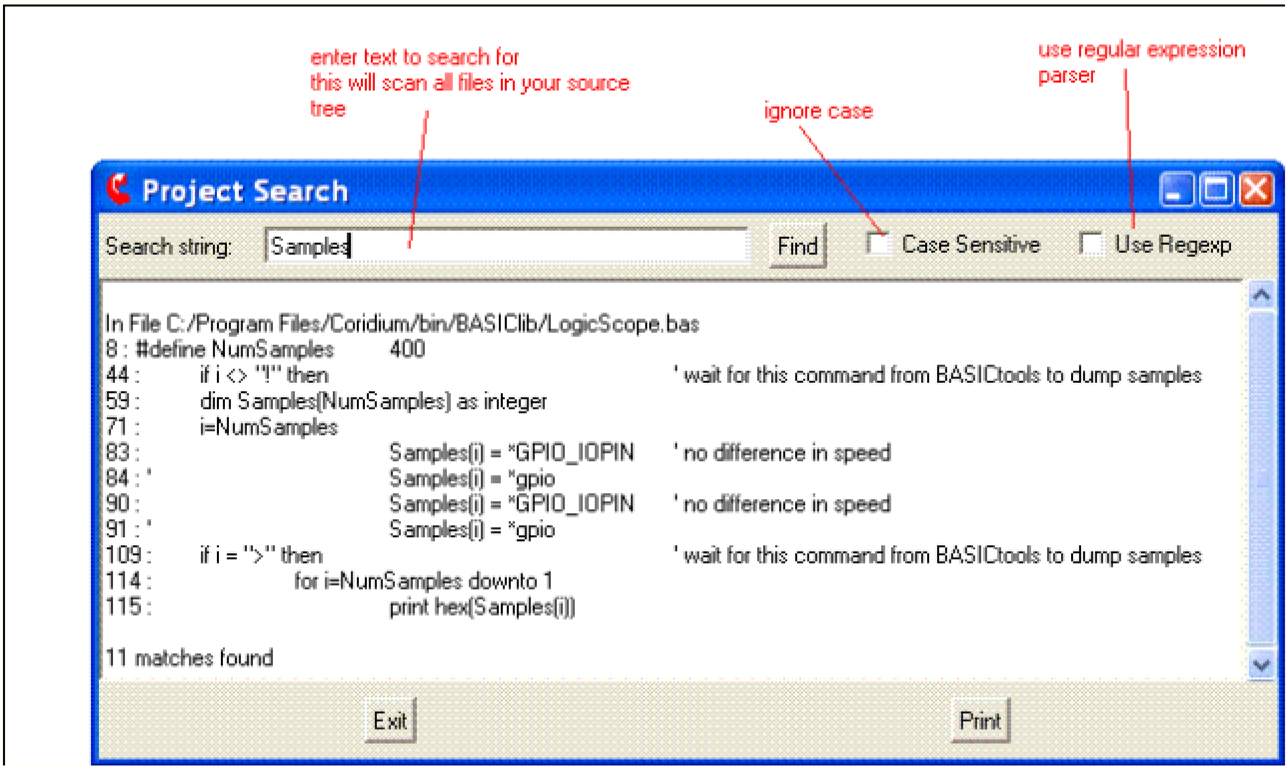
code window



keyw ords: variable dump breakpoint STOP view memory

Search Window

Open this window from the Edit Menu



keyw ords: search

Logic Scope Window

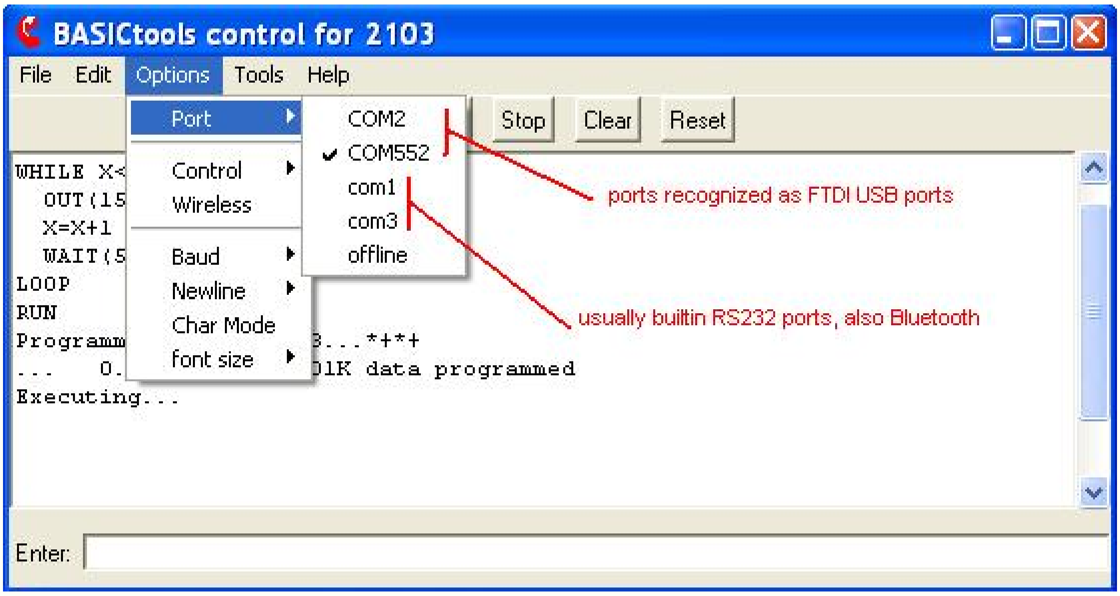
This module must be included in your BASIC program. It will monitor the pins for a period of time when called from your program.

See the example program ScopeDemo.bas and details in the **Logic Scope Section** .

keyw ords: oscilloscope logic analyzer logic scope

Trouble Shooting

Reset ARMexpress shows no message

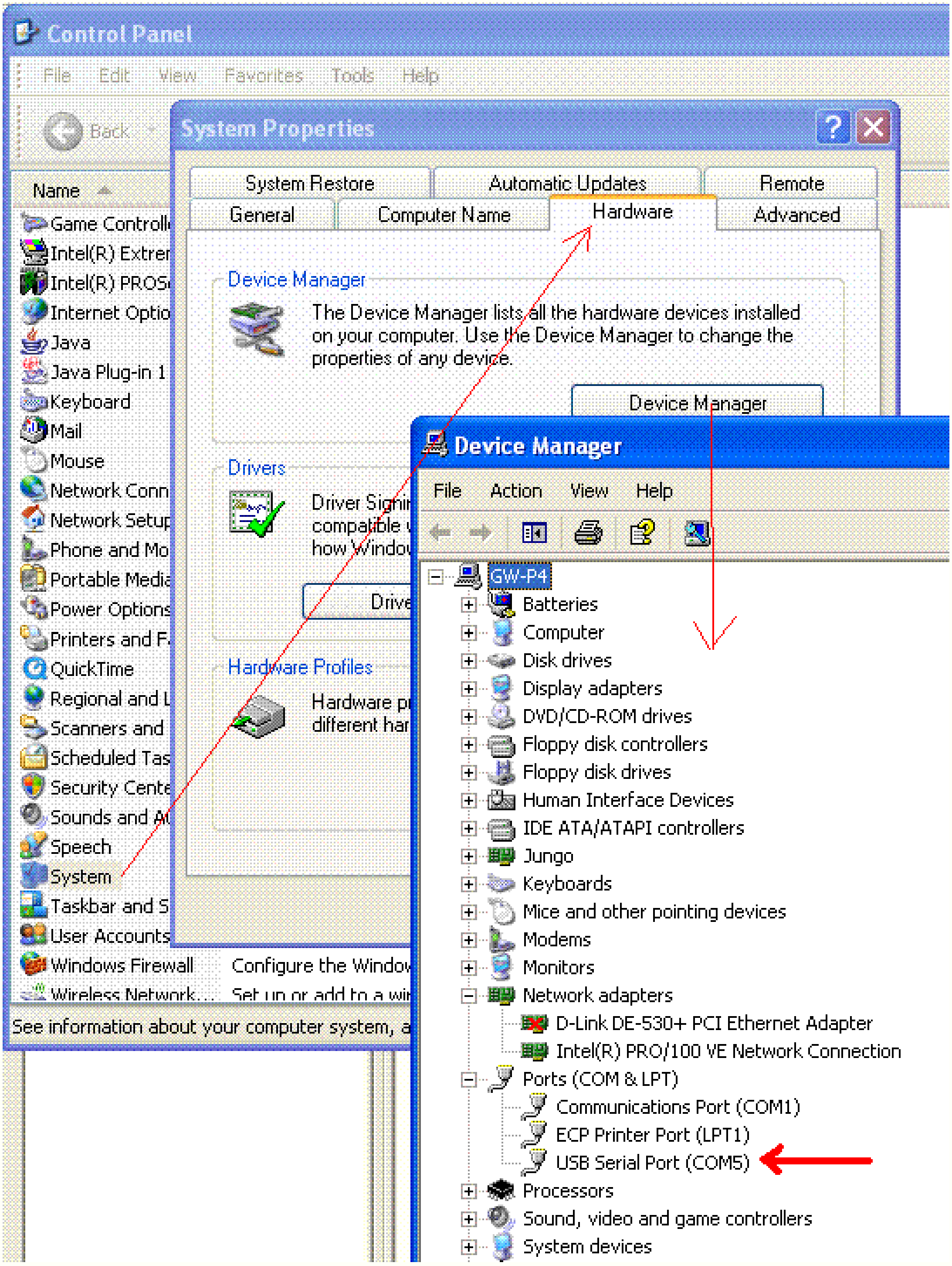


Most PCs have a number of COM ports, you might not have the correct port selected, you can change that in the Options>Port Menu   This window lists all the available ports, those in capital letters are recognized as FTDI USB serial ports and are usually the location of the ARMexpress Eval PCB or the ARMmite.

One other reason that communication could be lost, is that the driver can lose sync with the card if it is disconnected and reconnected with the USB, especially when BASICtools or TclTerm (under MakeItC) is running and connected to the card. When this happens it is often necessary to restart the PC. Because the serial port is being emulated, and the Windows enumerator gets involved, when the USB is disconnected, the various pieces of software can get confused if the port is open. If you are using the original hardware serial port, normally with COM1 this is not an issue.

Determining which COM port should be used

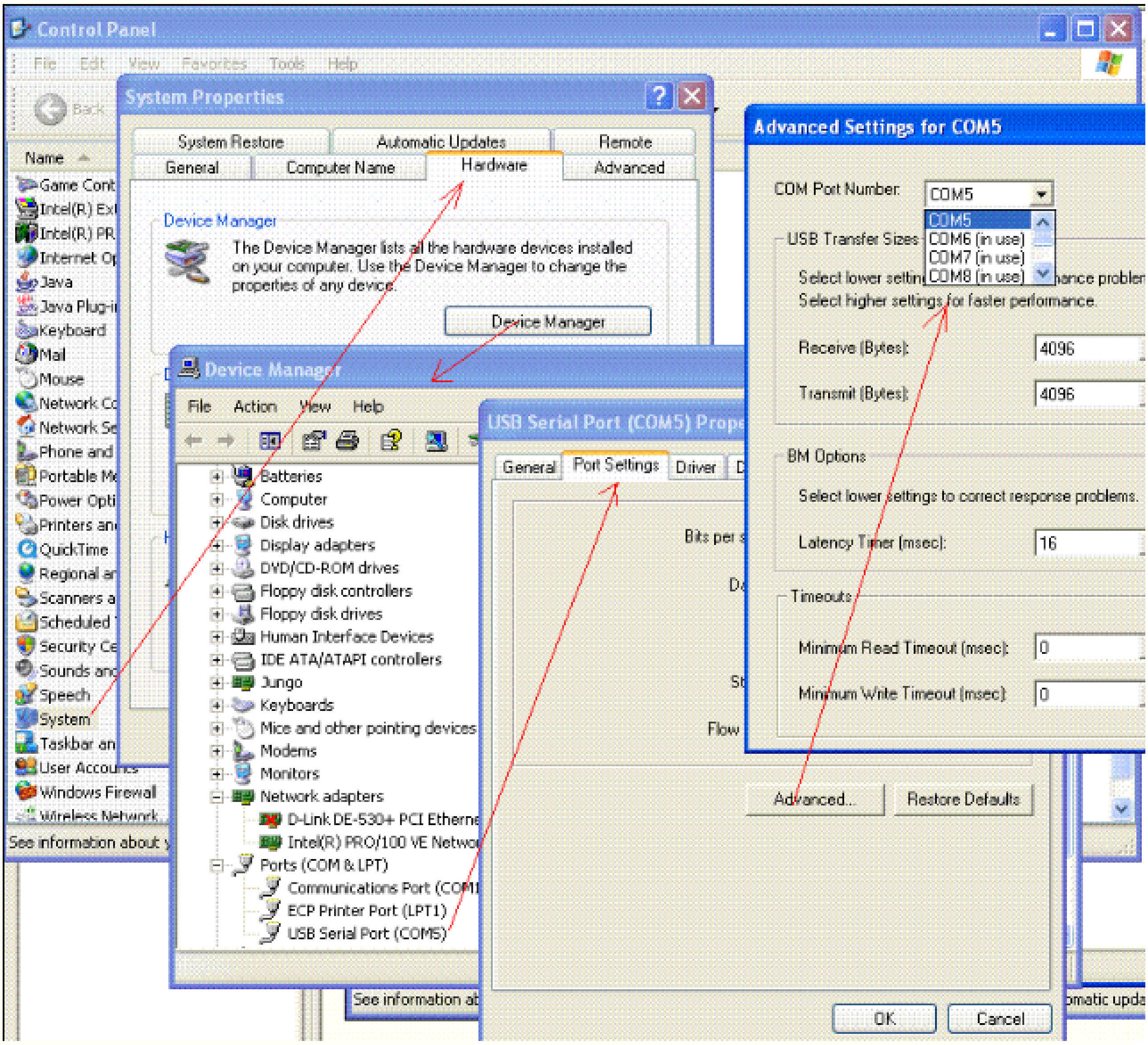
This can be found in the Control Panel>System>Device Manager



COM port conflicts

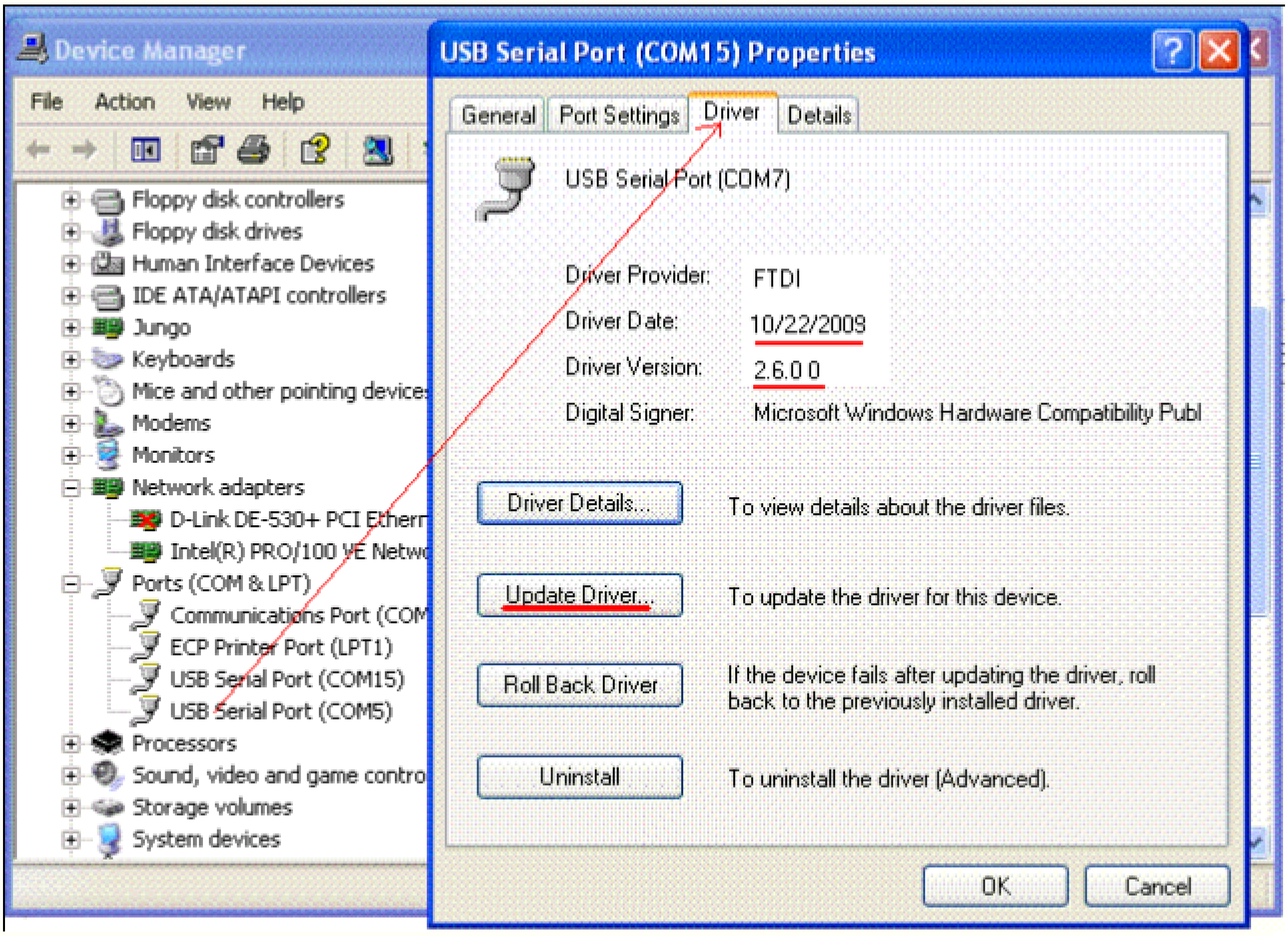
While rare there are systems out there with non-plug and play serial ports, or its possible for 2 com ports to have the same address. The address can be changed from the Control Panel.

Control Panel> System> Hardware> Device Manager> Ports> Port Settings> Advanced



Check the USB Driver version

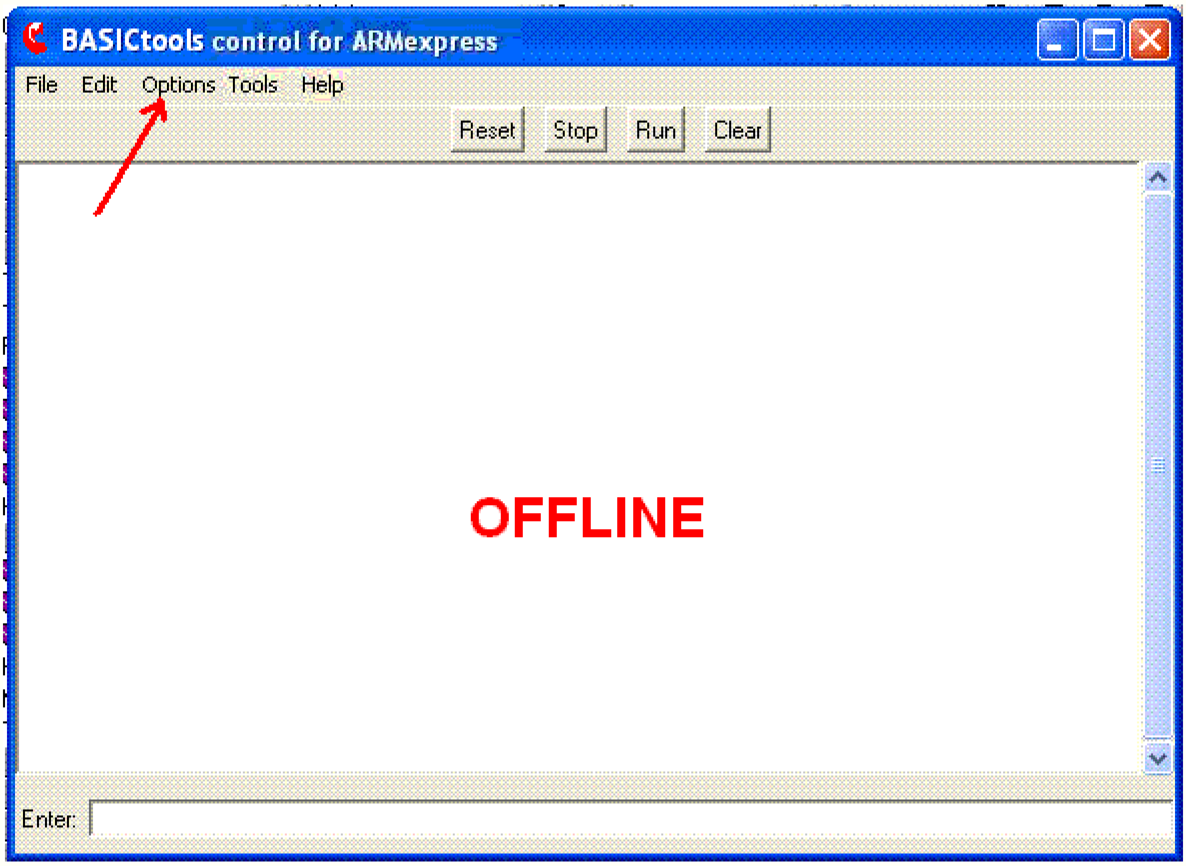
Our software does not reinstall the USB drivers if they already existed. We expect to be running version 2.6.0.0 dated 10/22/2009. Find this in the Control panel>Driver properties



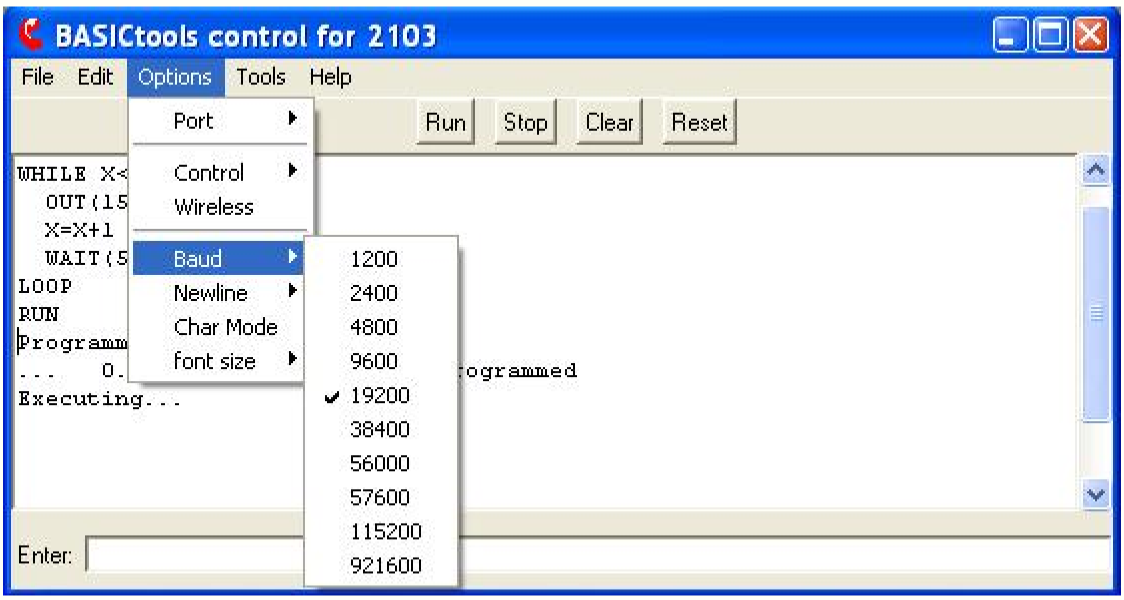
If this does not match, then you have an older driver and it should be updated...

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 ARMmite or ARMexpress.



Check Baud Rate



Or you might not have the correct baud rate selected.

Check your cables, check the LED

The green LED should be on if the USB connection is made for the ARMmite, or when power is connected for wireless ARMmite or ARMweb.

See **Connect USB**

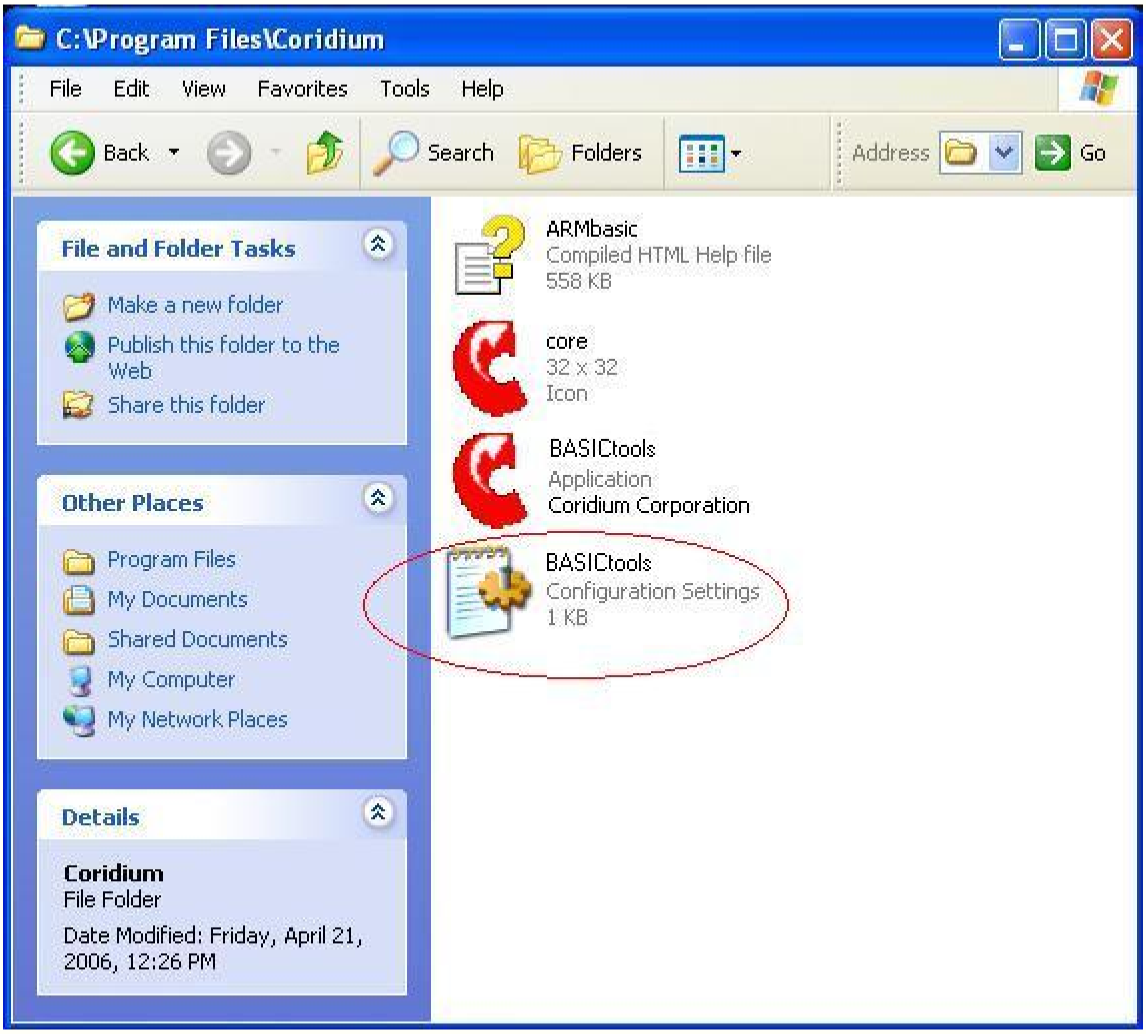
Wireless Serial link

When debugging the serial connection for the wireless ARMmite, make sure both modems are set to the same baud rate, otherwise one way communication is possible. Check your solder connections. Use a USB breakout board to monitor the communication of either side, connect the RXD pin of the USB breakout to either TXD or RXD on the ARMmite to monitor the serial communication.

If you can see the Welcome message following pressing the reset button on the ARMmite wireless, then communication is running one direction. Type a ? at the enter line, you should see a number of 4 digit hex numbers come back. At this point communication is running in both directions.

You can also use BASICtools to send repetitive data through the serial port. To do this check the Char mode under options, this will send out any key you hold down from the enter box, rather than the normal line buffering. Then you should be able to see the data on a scope. Remember to uncheck Char mode when done.

Odd behavior following Windows Update



In rare cases, when the Windows Update has automatically rebooted while BASICtools was running, the serial port settings of BASICtools have been corrupted. To correct this, reboot the system, and if the problem persists delete the BASICtools configuration settings (BASICtools.ini, it will be regenerated when you run BASICtools). This file is located in the %AppData%/Coridium directory or in older versions of BASICtoos in Program Files\Coridium directory. If you don't know where the %AppData% directory is, open a DOS command line and type **echo %AppData%** .

Have Fun!!

ARMweb Getting Started

Getting Started

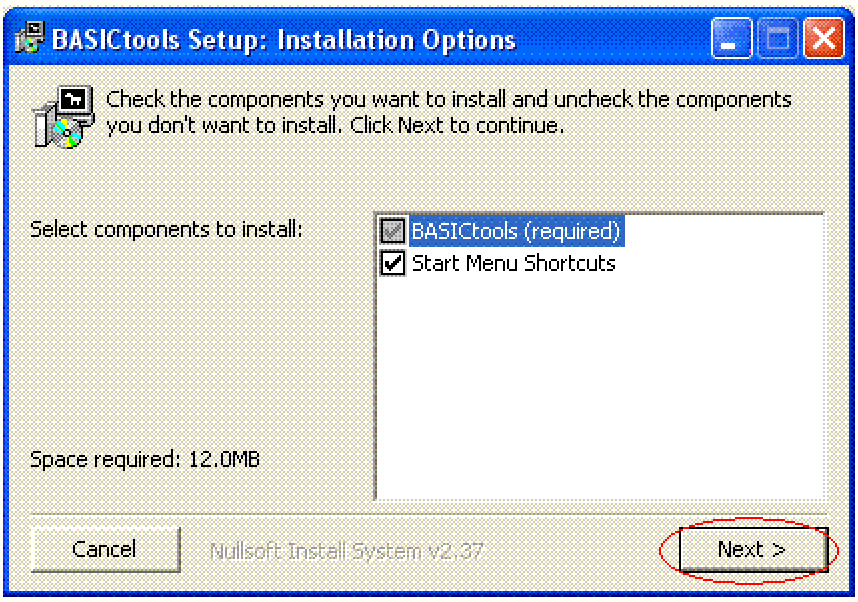
- Install Software
- Connect Ethernet
- USB connection for ARMweb
- Writing simple programs via the web
- Writing programs with BASICtools

Step 1: Install Software

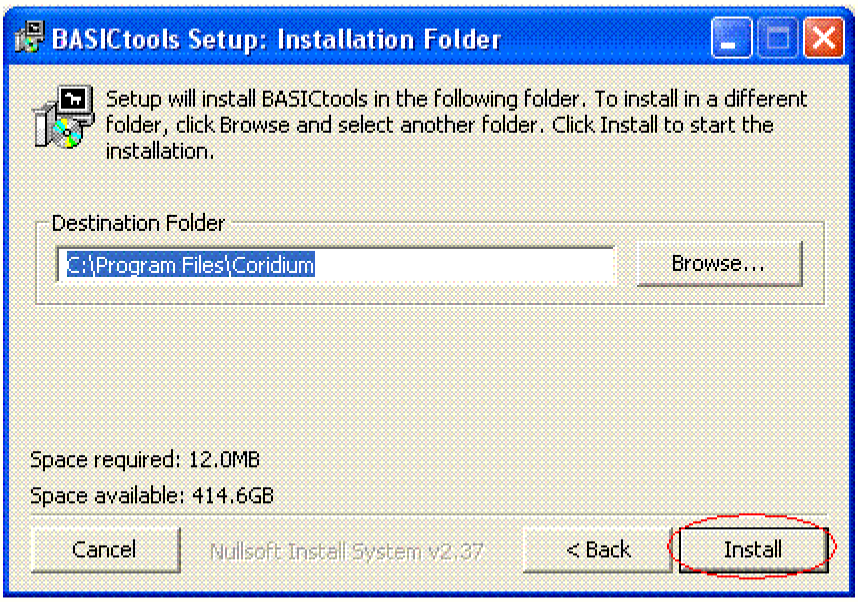
Actually much of the software you need for the **ARMweb** is already on your computer. The interface to the **ARMweb** is through any web-browser. That's why we call this **Simply Connected™** technology.

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 webpages served by the ARMweb. Your main BASIC program should be debugged and loaded via BASICtools over a USB connection.

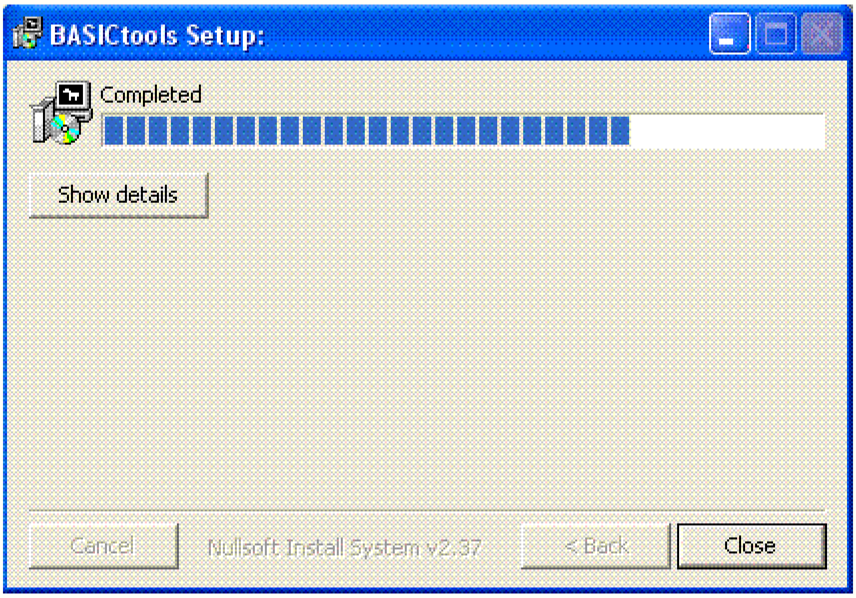
You will want to run the setupBASIC installation, to get access to documentation about ARMbasic and the PC based main BASIC compiler.



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** .

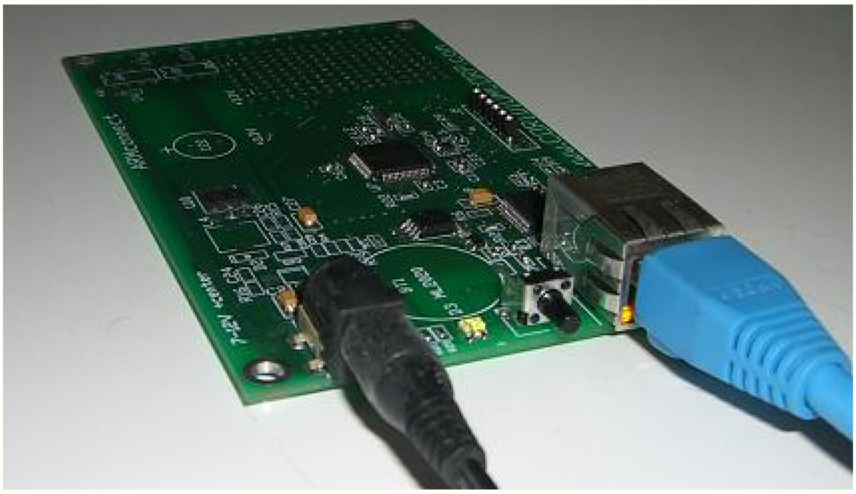
And its as easy as that.

On to Step 2

Step 2: Connect Power and Ethernet

Connect Ethernet Cable to ARMweb PCB

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** .



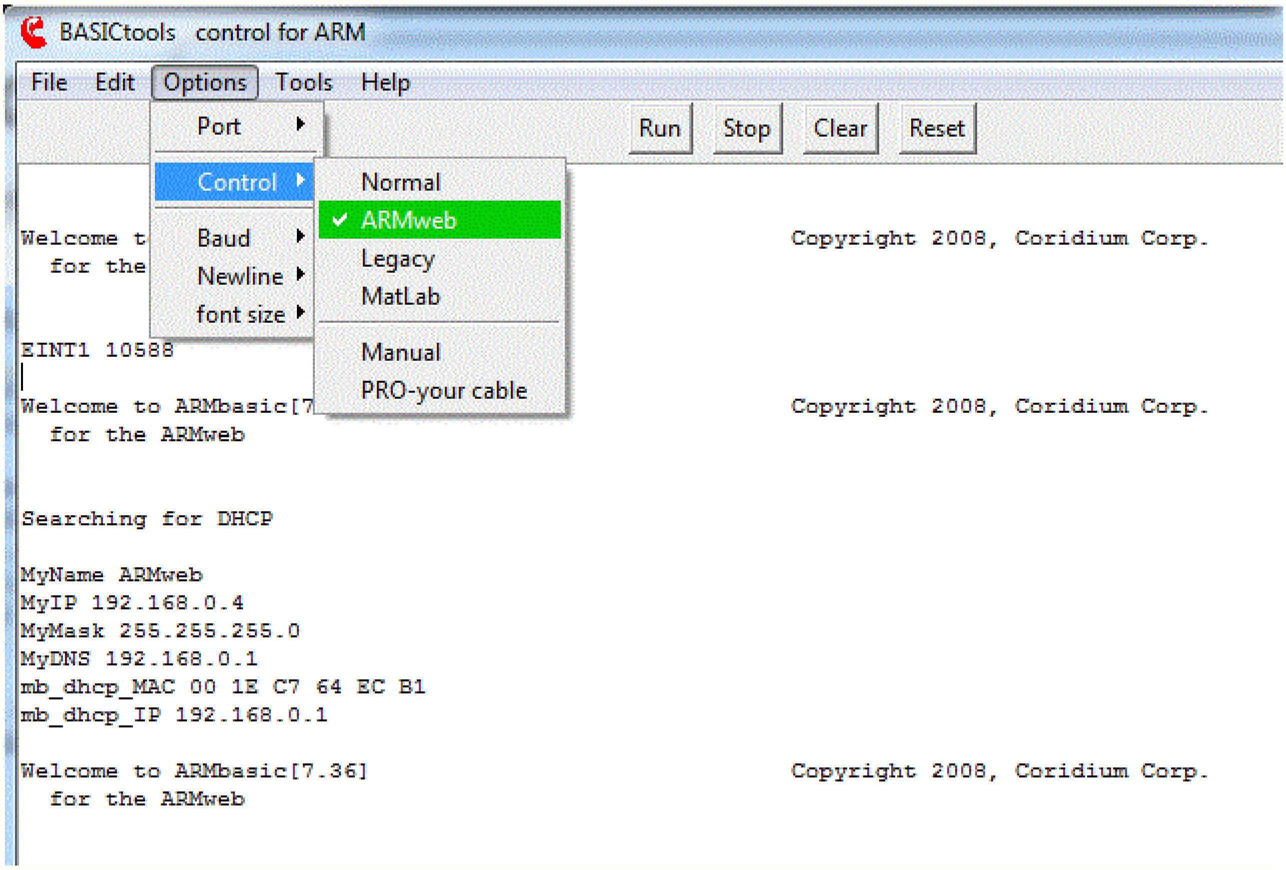
You should see a green LED connect light on the lower left side of the ethernet cable indicate a connection was made. Also your hub normally has a similar type of connection indicator. There should also be some traffic indicated on the right side as the ARMweb looks for a DHCP.

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

We recommend that you have at least one USB connection to debug BASIC programs as well as network issues. This can be our **USB dongle** or some other TTL serial connection.

Below is the picture you should see. Depending on which version of firmware and which USB dongle you may see an EINT1 interrupt message. EINT1 was being used for network debug in earlier firmware versions. You should disable that by choosing ARMweb control under the Options. After that you should see the ARMweb "Searching for DHCP" and if there is one it will report the DHCP IP address and the IP address assigned by the DHCP (MyIP)

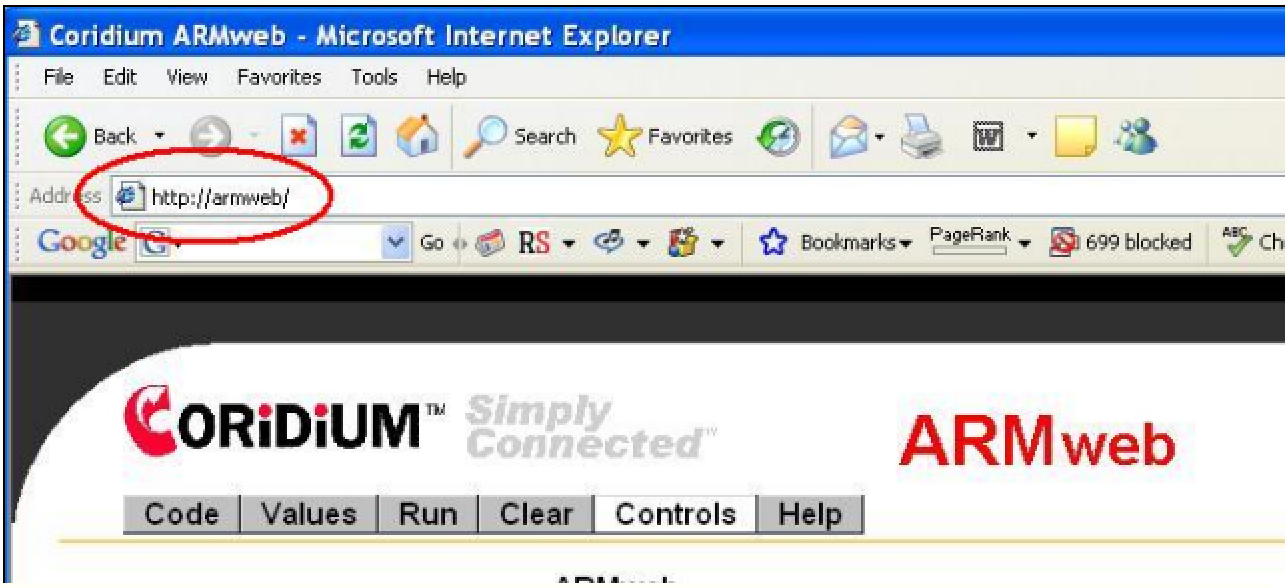


Again, if you don't see the LEDs or this display, check your power connections (you should see at least 6V of the + side marked on C23, and 3.3V as marked in the prototype area), check your com connections **(details in Troubleshooting section**).

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).

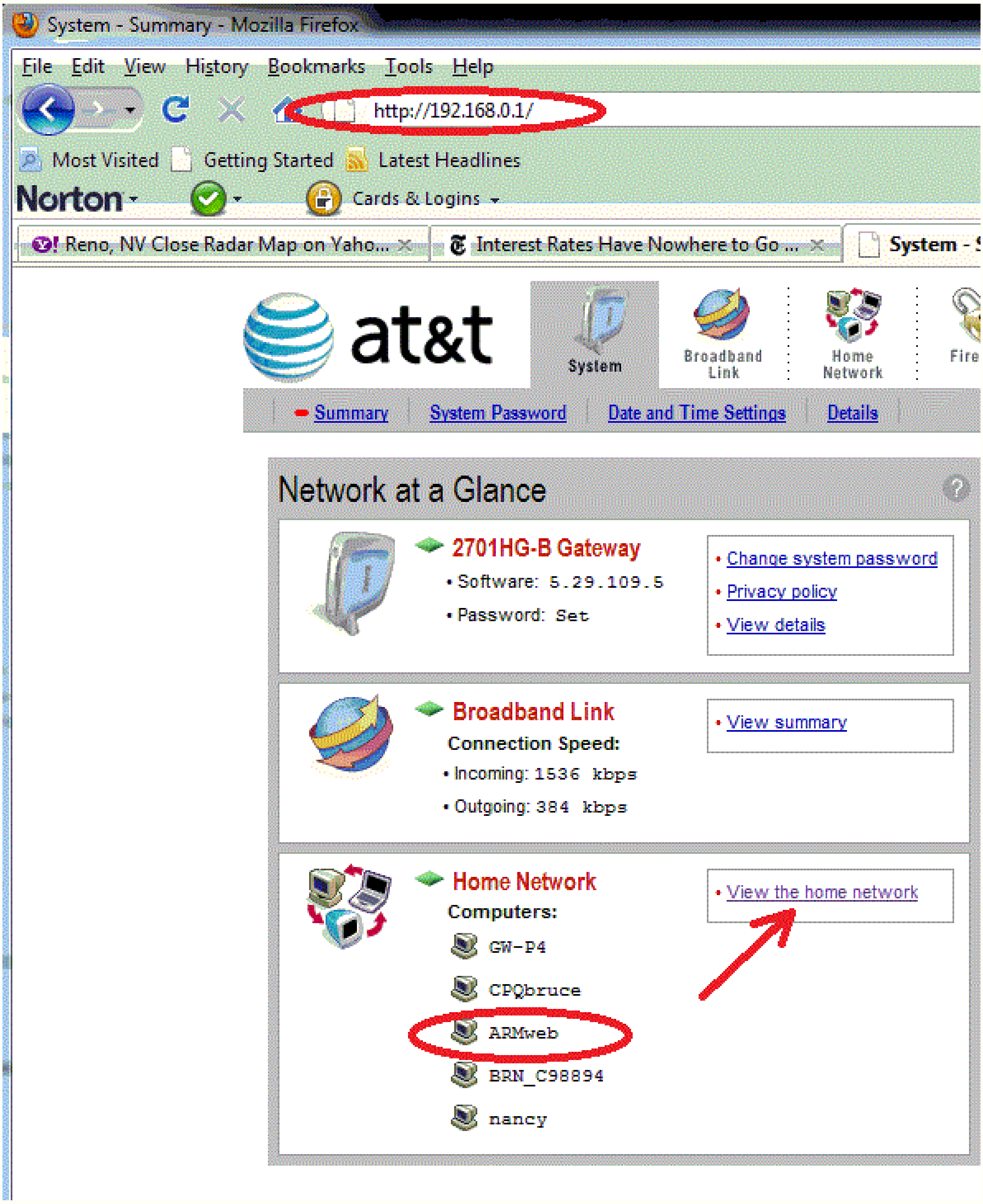
Assuming a DHCP server is available and you are running on a Windows machine, you can use the Windows NetBIOS Name Service. In which case you can find the ARMweb initially with **http://armweb**. **Note that some administrators disable NetBIOS name service** .



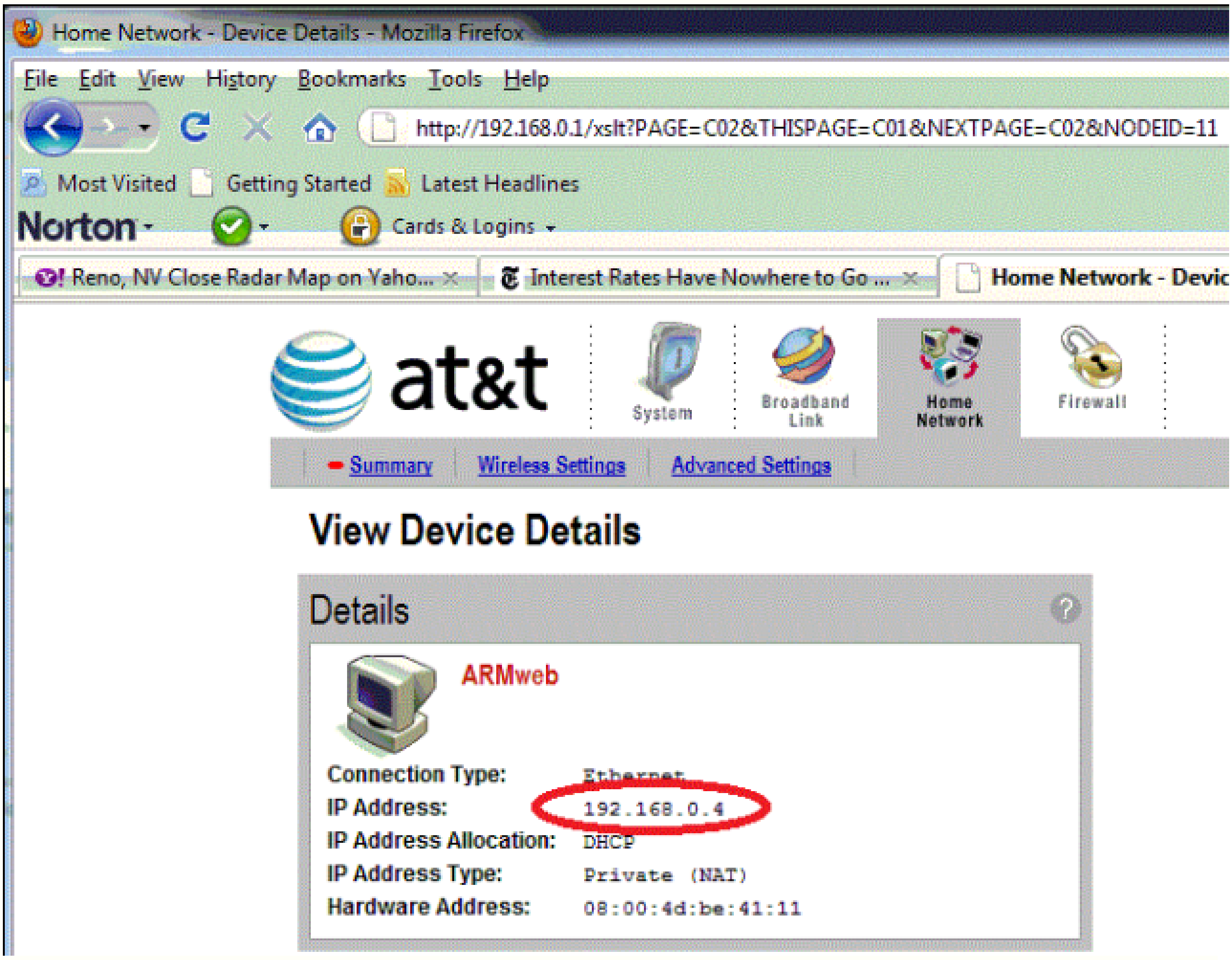
Finding the card using the DHCP server

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.

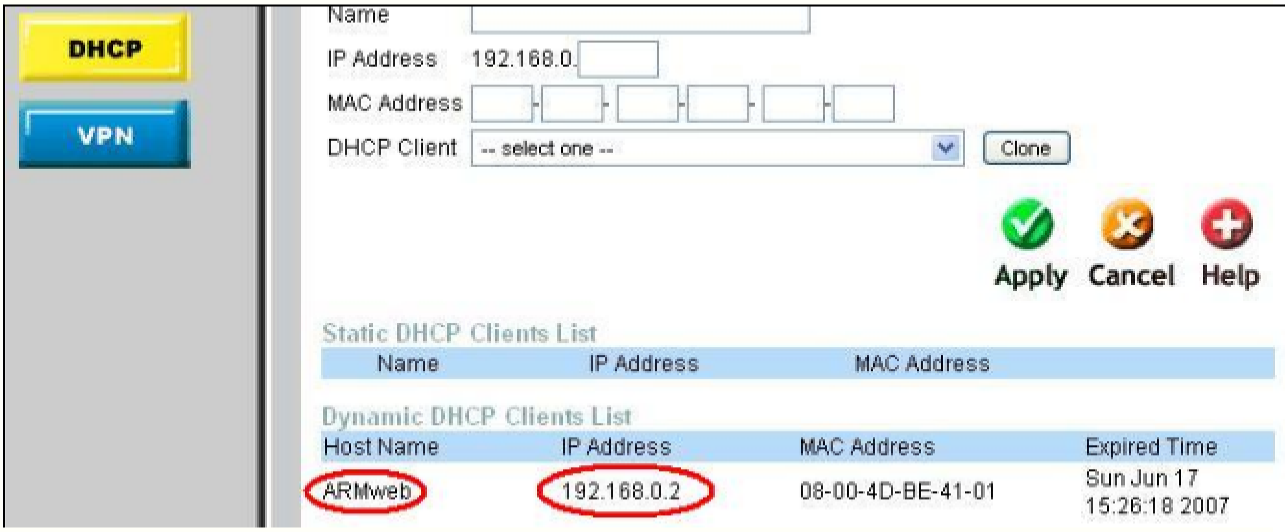
You can navigate to the DHCP server using that IP address from a browser as below.



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://192.168.0.2**

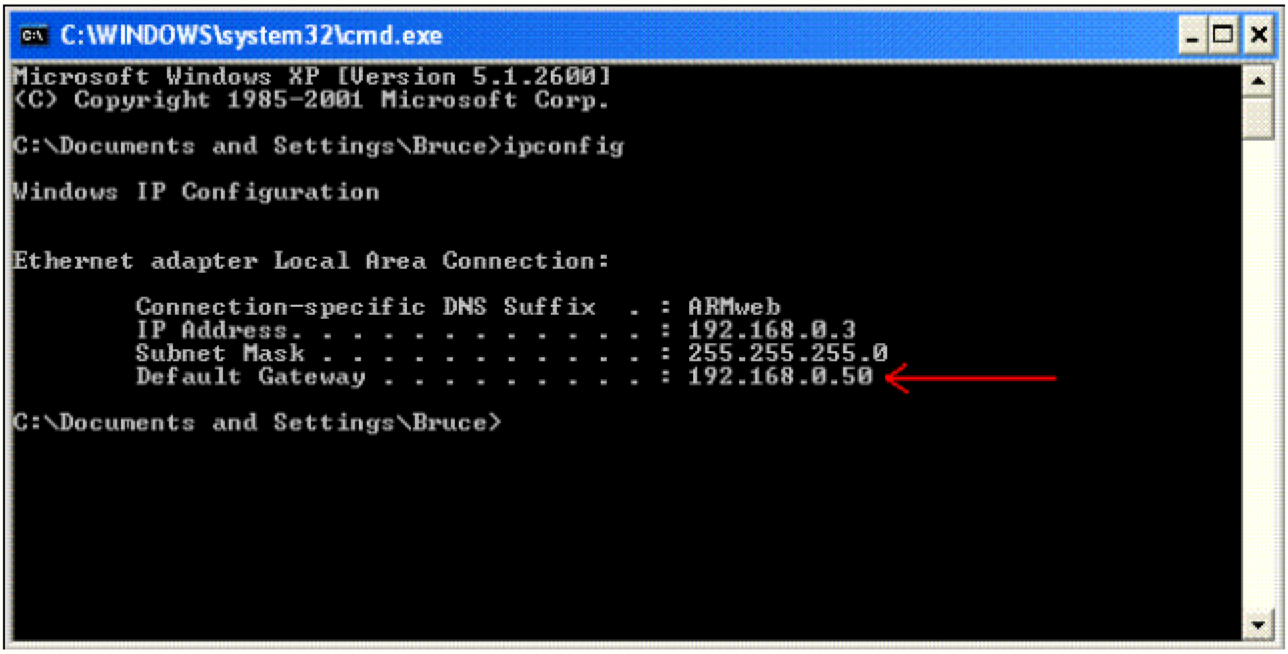
Diagnostic Mode -- only to be used in special situations

A minimal configuration is an ARMweb connected to a PC with a cross-over cable. This can be useful for configuring an ARMweb **prior to connecting with a larger network**, In this case no DHCP server will be found, and after 10 seconds the ARMweb will provide limited DHCP services, assigning an IP address to the PC. **However, this miniDHCP service will be terminated if the ARMweb is ever connected to a**

network with a DHCP server . To restore this miniDHCP service and the factory defaults, hold the push-button for 5 seconds while cycling the power.

The ARMweb will normally be located at **http://192.168.0.50** unless it has been reconfigured before, in which case it will use the last assigned IP address.

If you can not find the ARMweb at **http://192.168.0.50** or **http://ARMweb** as above, then you can locate its IP address with the DOS command line program IPCONFIG. The ARMweb will appear as the default gateway in this case. Also if your ARMweb has been connected to a network serviced by a DHCP it will not function as a limited DHCP server (this would cause confusion in a large network).



If you're not seeing this make sure your PC Network configuration is set to Obtain an IP address automatically. (Control Panel -> Network Connections -> Local Area Network -> Properties -> TCPIP -> Properties)

Now that you have the IP address of the ARMweb

You can go onto configuration settings, or writing simple programs using the web interface (the web interface is only meant for simple programs, to do more extensive programs will require a USB connection and 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

DHCP assignment vs fixed IP addressing

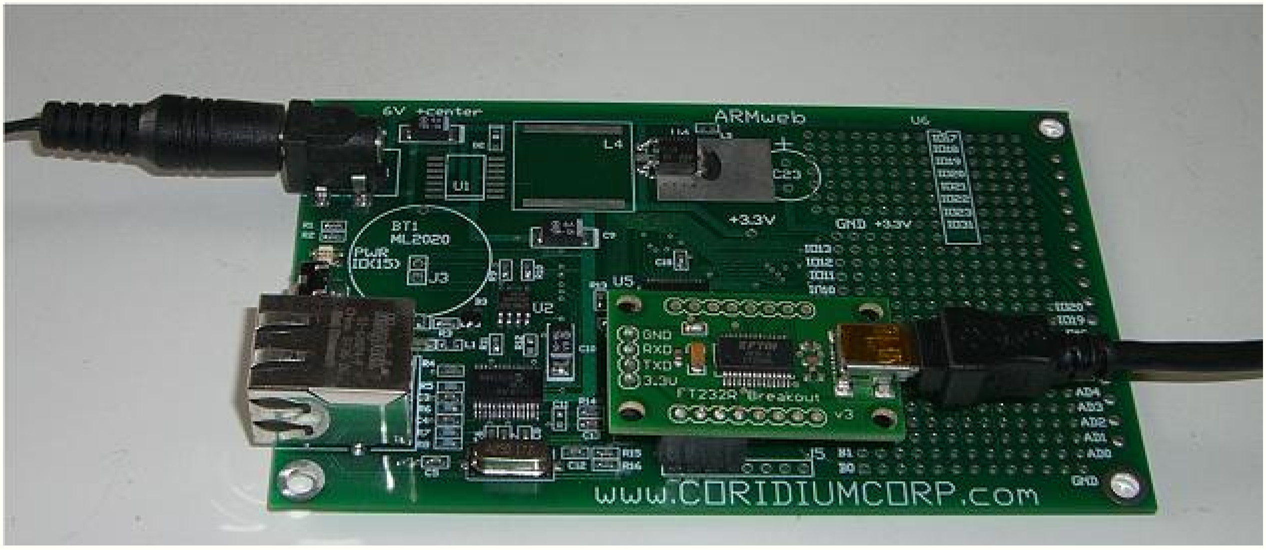
We routinely allow the DHCP server to assign an initial address, but will use a fixed IP address in the final setup. One reason to assign a fixed IP, is to make sure that the IP address assigned never changes, for instance following a power outage. **Details** on setting a fixed IP address.

On to Step 3

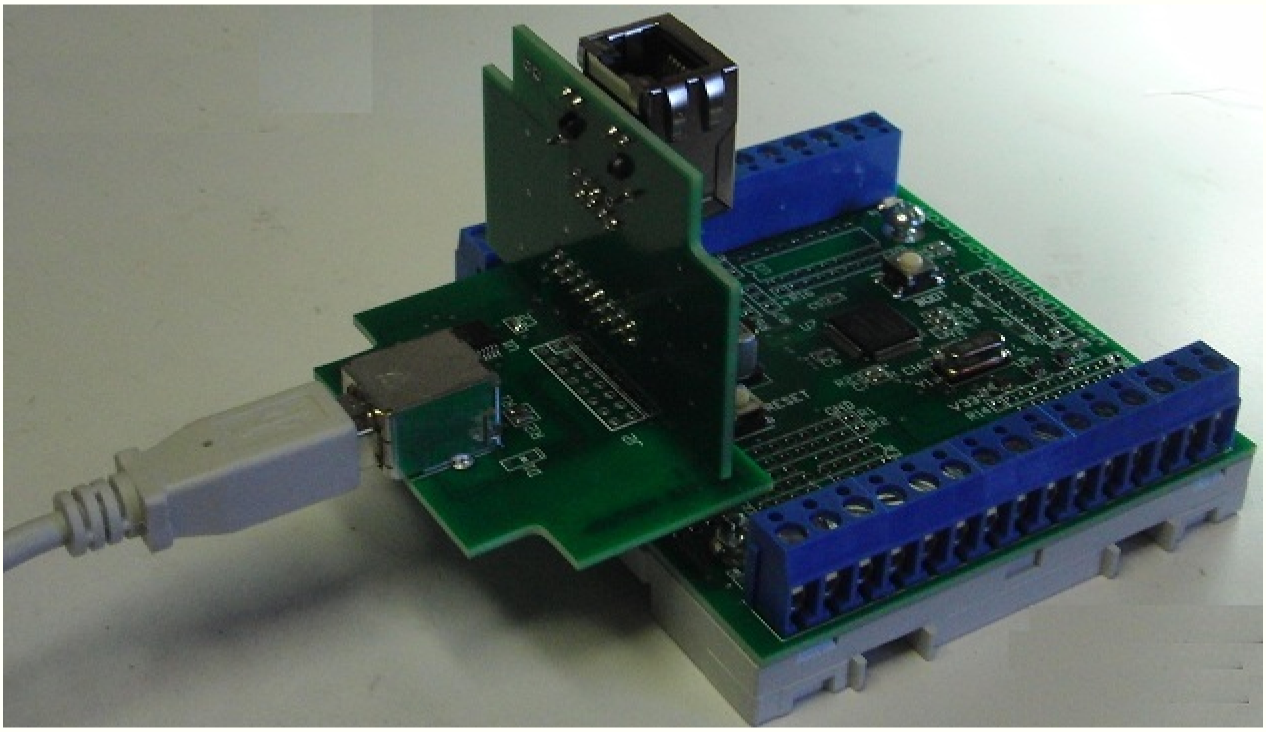
Optional: USB connection for BASICtools

While the ARMweb can be programmed through the webpage, during the development cycle BASICtools can be used via a USB connection. BASICtools has a much faster response than a browser.

The attachment of the USB and power supply is shown below. 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).



ARMweb



DINkit (ethernet)

Why use BASICtools?

Browsers are very slow when refreshing a webpage, so the interaction with the programmer is better with BASICtools.

#include can not be used from a webpage, as the ARMweb does not have direct access to the #include'd file

The BASIC compiler on the PC has more memory for the symbol table and can handle larger programs than when compiling on the builtin ARMweb compiler.

The variable dump tool is available in BASICtools. Debug messages are sent to the USB port, as well as <?BASIC ... ?> source and output when processing web requests. When your program is debugged and AutoRun is turned on the USB port is turned off. You can improve the performance of the web server BASIC compiler by increasing the speed of UART0, by changing baud settings in BASICtools and executing BAUD0(937500) in your main program.

For an introduction to BASICtools refer to the **ARMmite sections** .

BASIC and Webpage interaction

BASIC can be embedded in the webpage served by the ARMweb. That BASIC code can access global variables of the user program running on the ARMweb. At present, BASIC embedded in the webpage can not call a FUNCTION or SUB (this will be a future enhancement).

The user (client) can also interact with an ARMweb BASIC 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 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. 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.

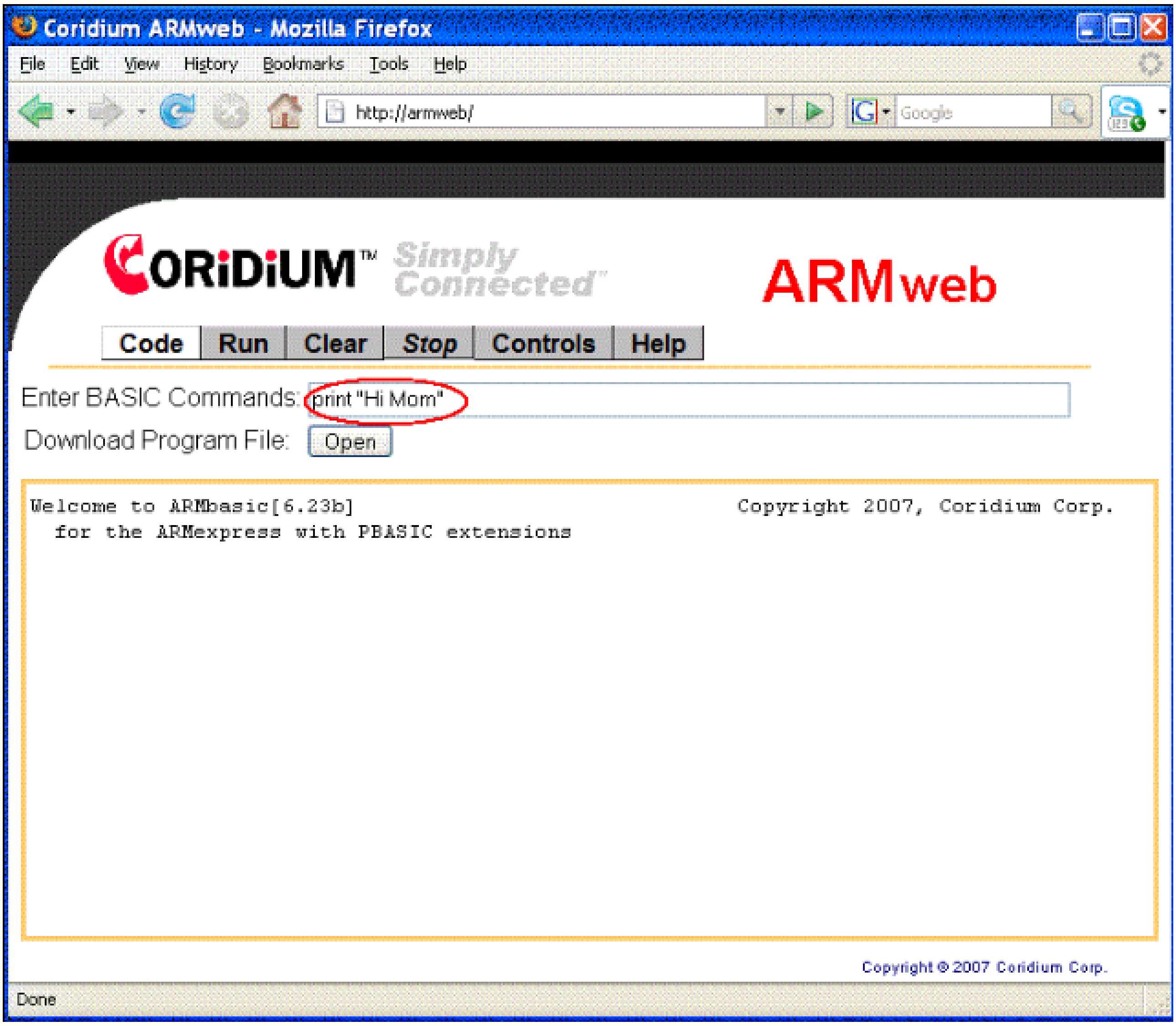
or

More details on ARMweb and BASIC...

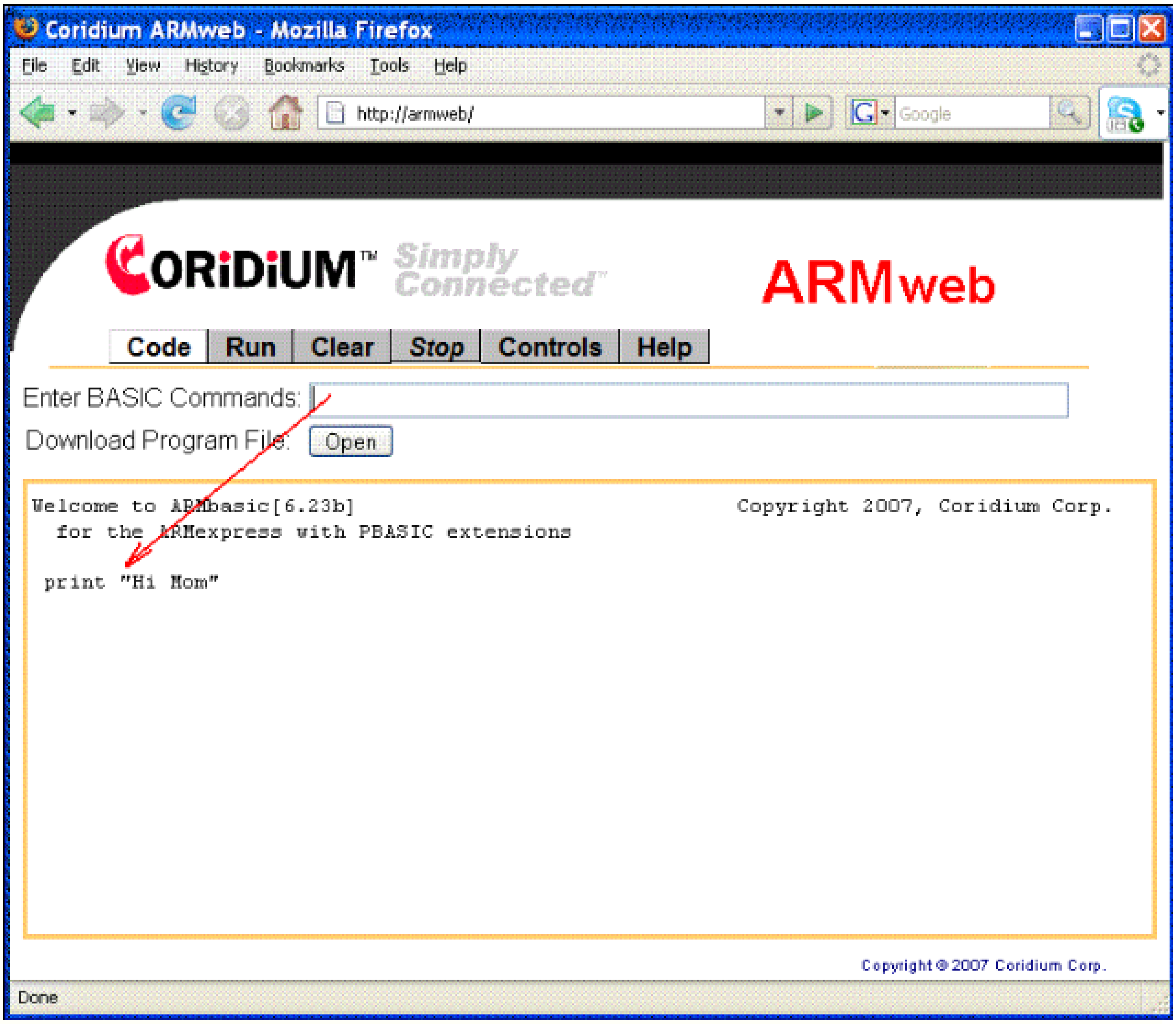
Step 3: Writing a simple Program with the web interface

The traditional "Hi Mom" program

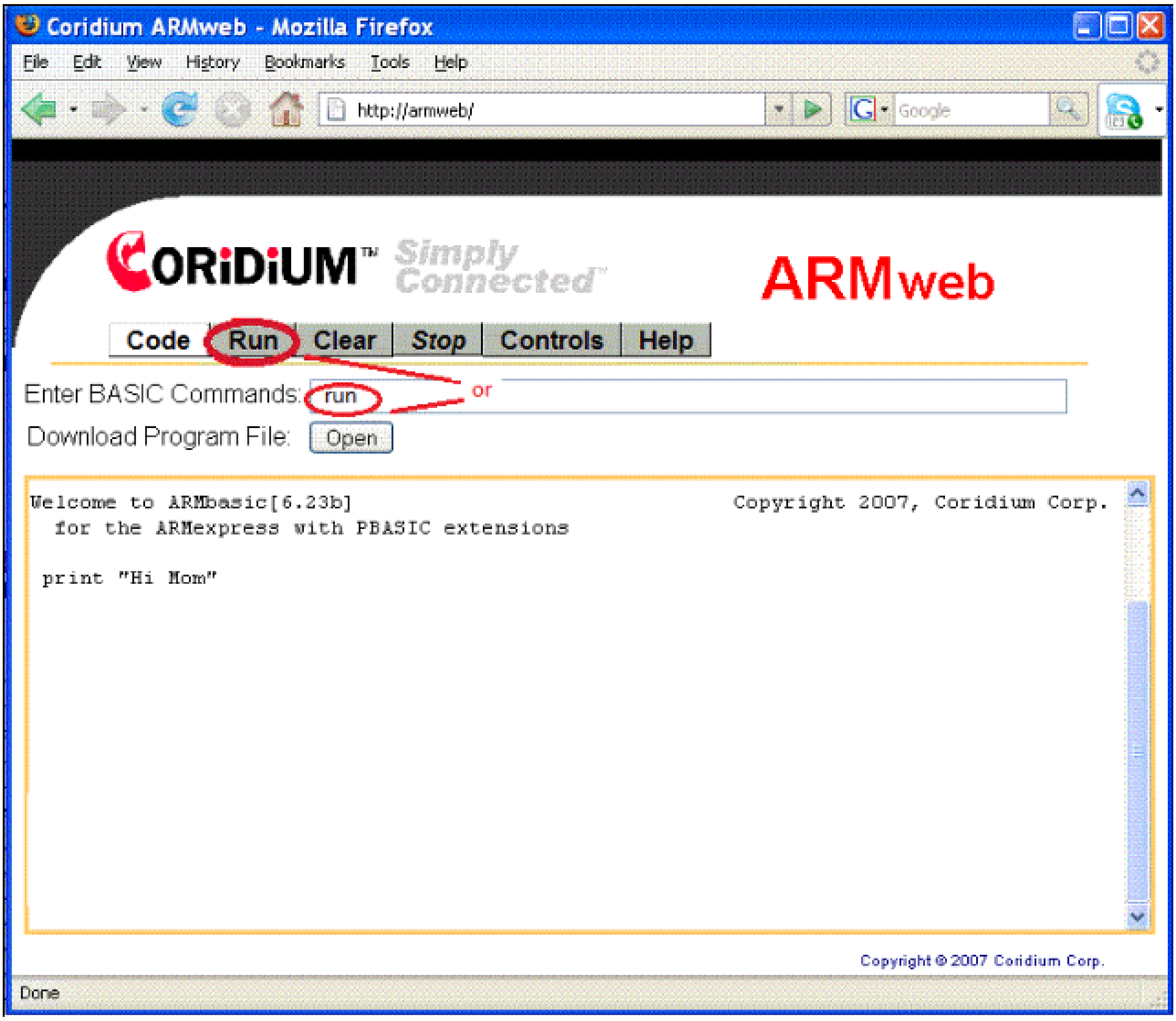
This section describes writing programs with the web interface, which is fine for small programs. But you will really want to use the USB interface to write larger programs, covered in the **next section** .



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

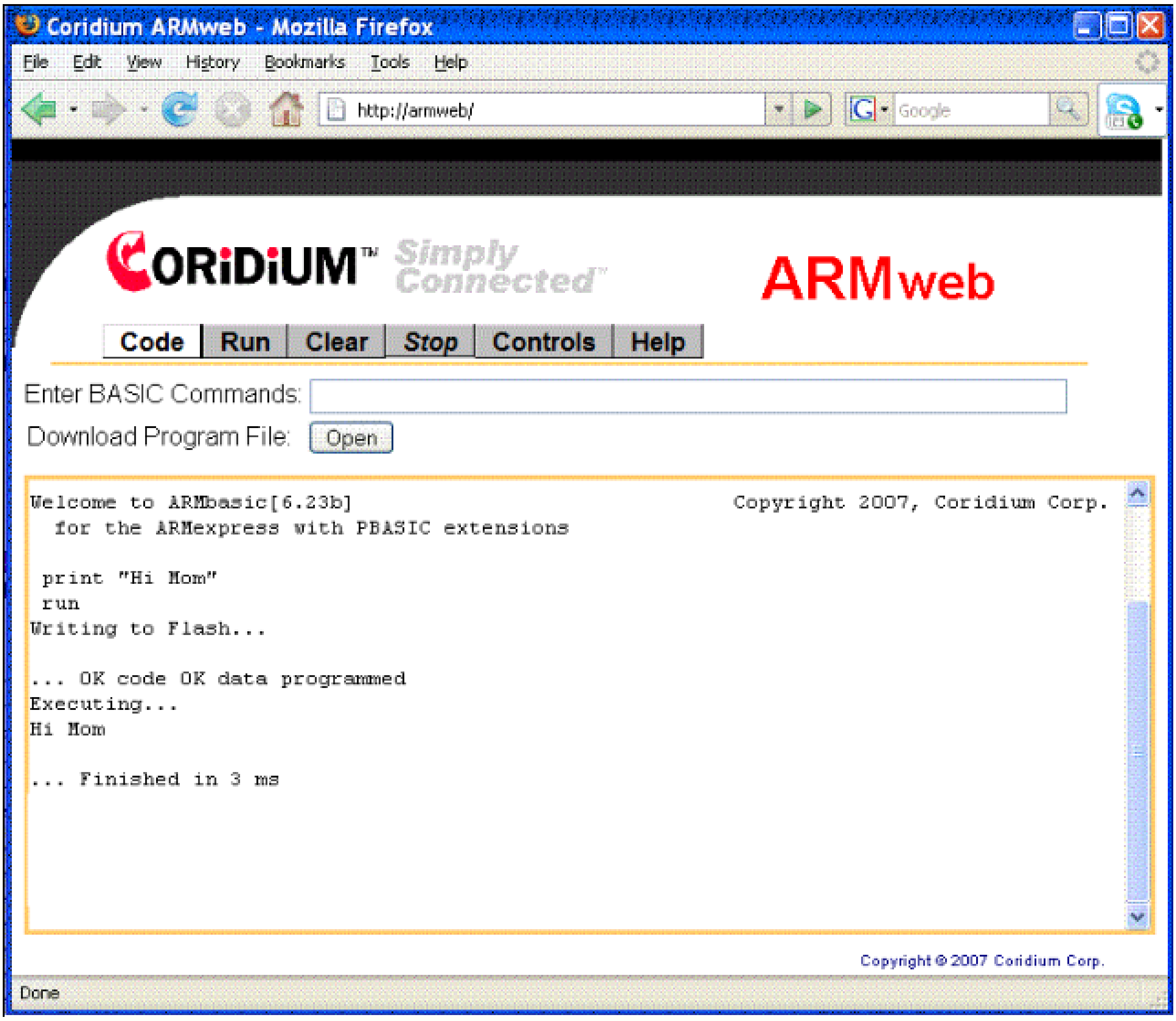


Now RUN the program



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

And see the results

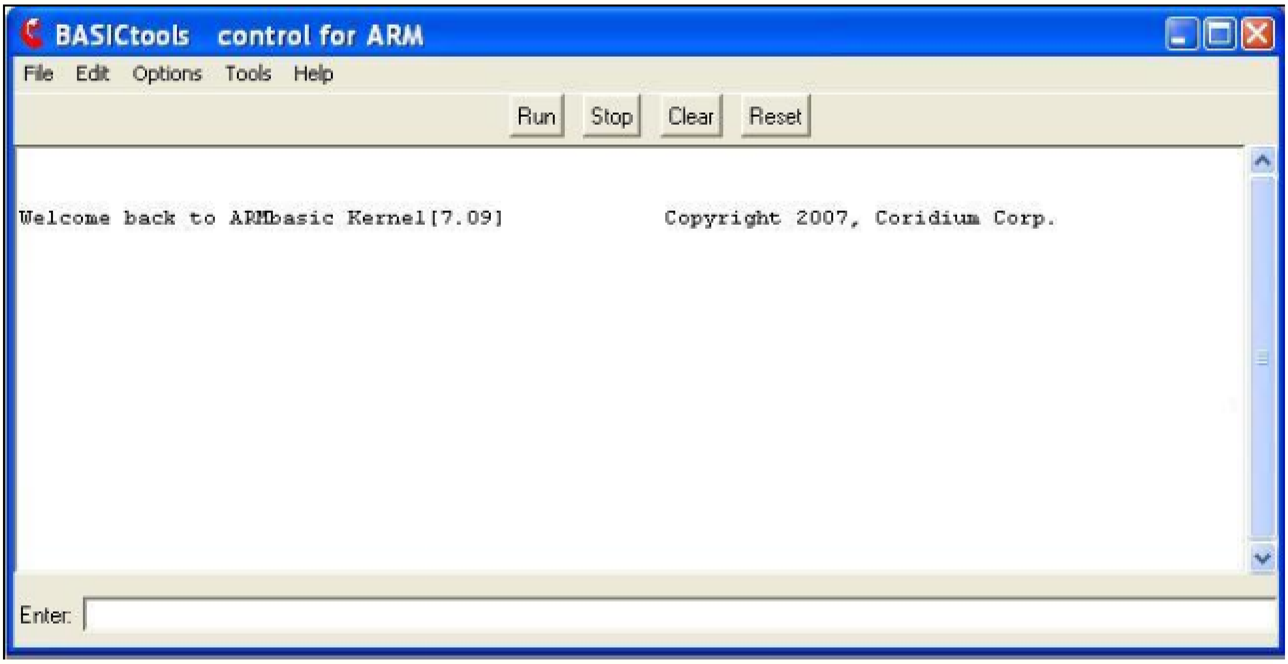


You can notice a number of things. First the program is compiled and then written into Flash memory, and your program takes 0K of code and 0K of data space.
Next the program will be executed, as evidenced by the output of "Hi Mom" to the console.
ARMexpress also reports back how long the program executed,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

On to the next Step

Step 3: Writing your first Program with BASICtools

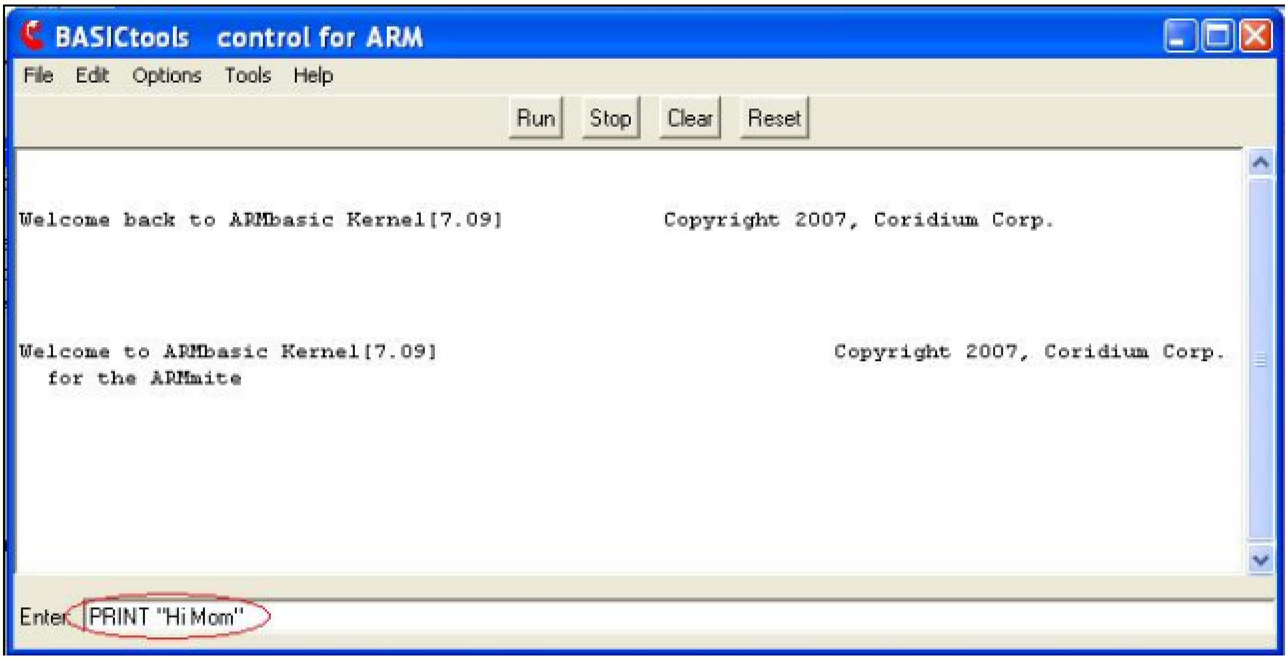
Start the BASICtools from the StartMenu or from the Desktop Icon.   You should see a welcome message which has been sent from the ARMmite or ARMexpress-



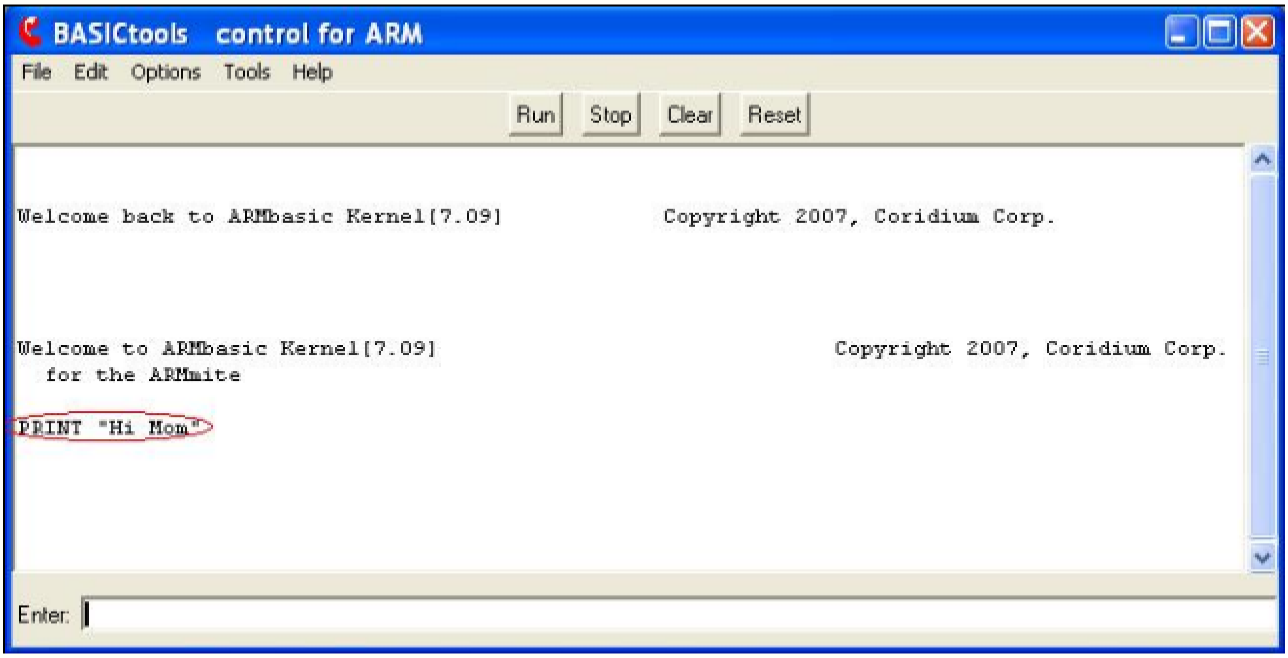
If you do not see this welcome, even after pushing the RESET button, then communication has not been established.

- 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**

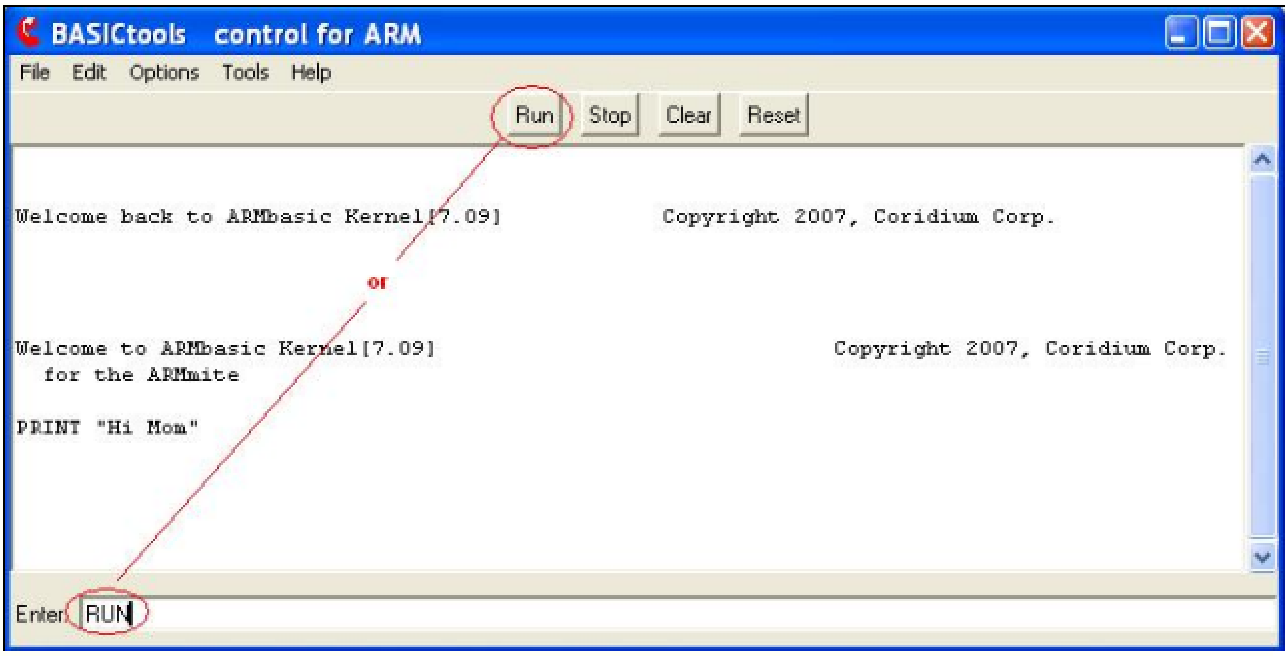
The traditional "Hi Mom" program



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

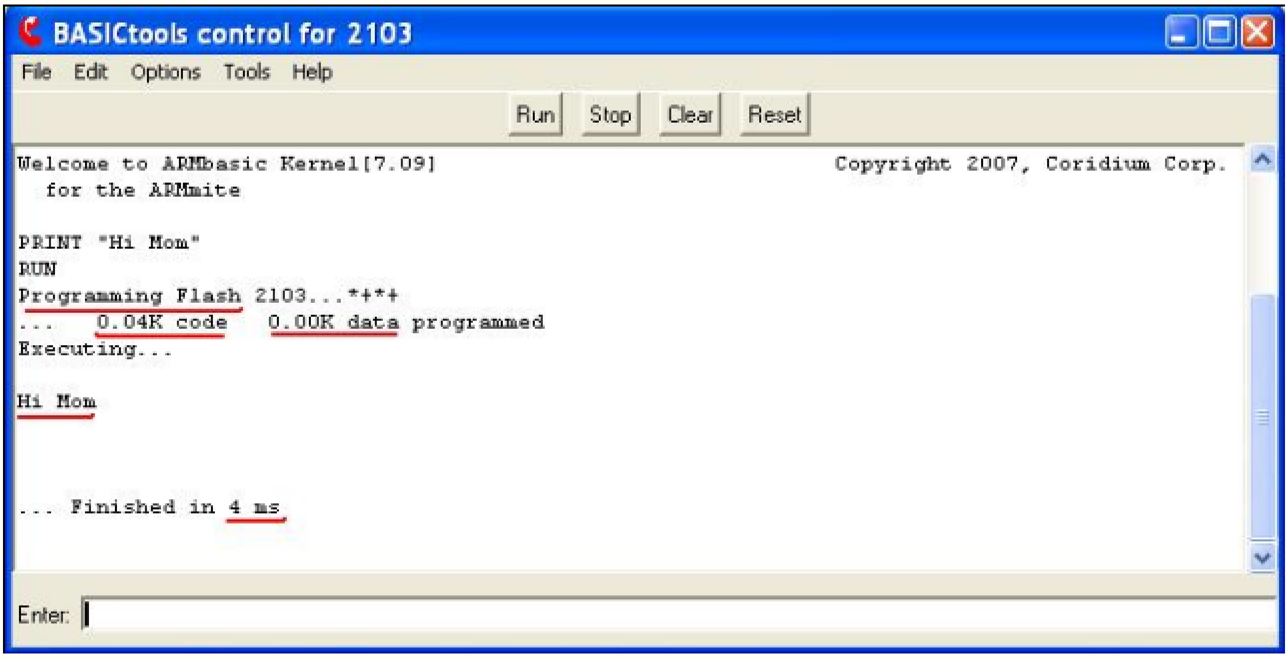


Now RUN the program



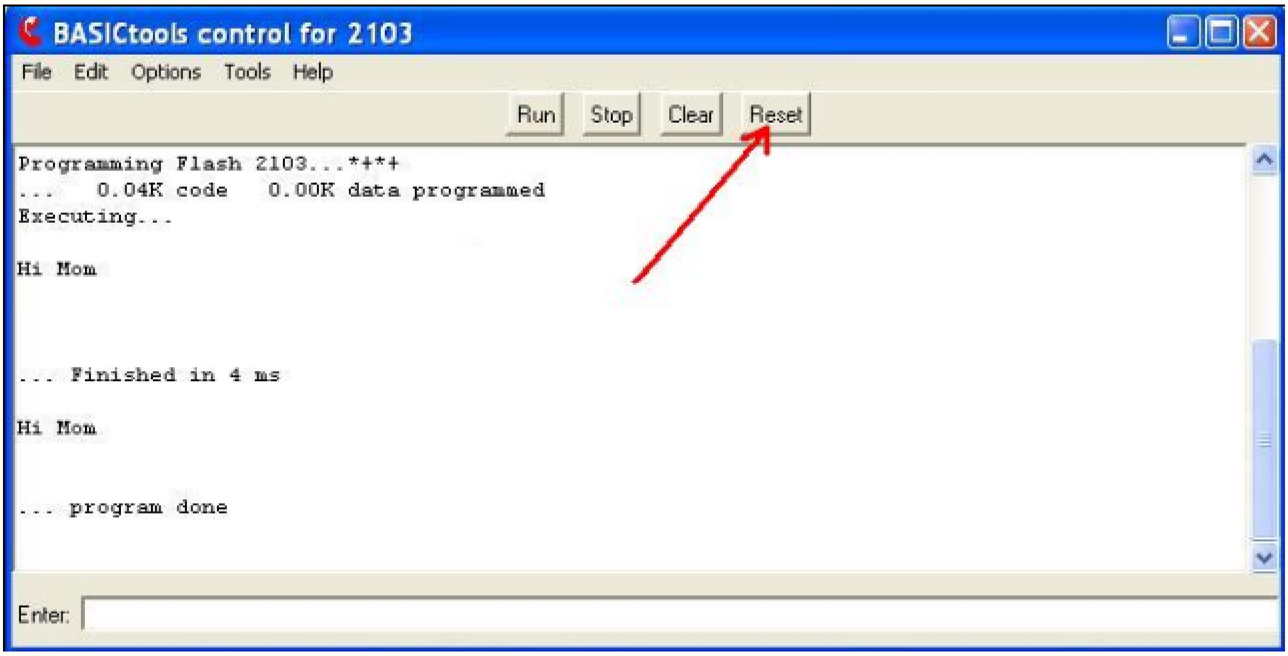
Which you can do by either typing RUN or hitting the RUN button at the top of the screen.

<u>And see the results</u>



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 Mom" to the console. ARMexpress also reports back how long the program executed, in this case 4 msec, which is mostly startup time.

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


BASICtools control for 2103
File Edit Options Tools Help
Run Stop Clear Reset
Programming Flash 2103...*+*+
... 0.04K code 0.00K data programmed
Executing...
Hi Mom
... Finished in 4 ms
Hi Mom
... program done
Enter:

On to Step 4

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.

Wireless ARMmite Getting Started

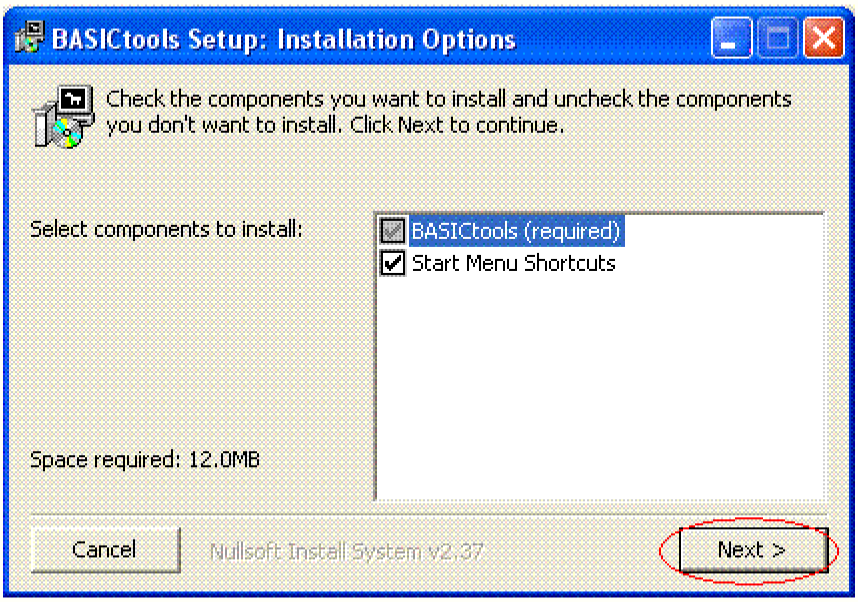
Step 0: Have a wired alternative

Because a wireless link can be an additional unknown, we **STRONGLY** suggest you have a wired connection handy, either a **SparkFun USB breakout** board and **connector** or something you have that is homebuilt. At less than $20 this will give you visibility into what is going on between the PC and the wireless ARMmite. You can also use this connection to monitor the data from the ARMmite or the wireless modem (do this by jumpering the RXD pin on breakout board to either RXD or TXD, also remember a GND connection, do NOT connect TXD when monitoring in parallel with the modem).

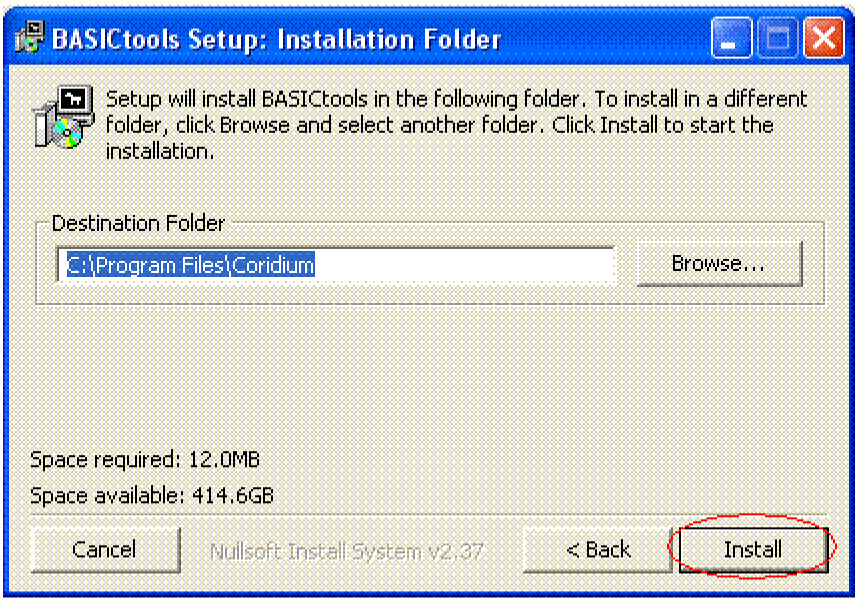
Step 1: Install Software

The **ARMexpress** family 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 ARMexpress and ARMmite. Also, a number of help files and documents about the ARMexpress will be installed on the machine at this time. This installer is meant for WIndows either 98, NT, XP or XPx64 and Vista.

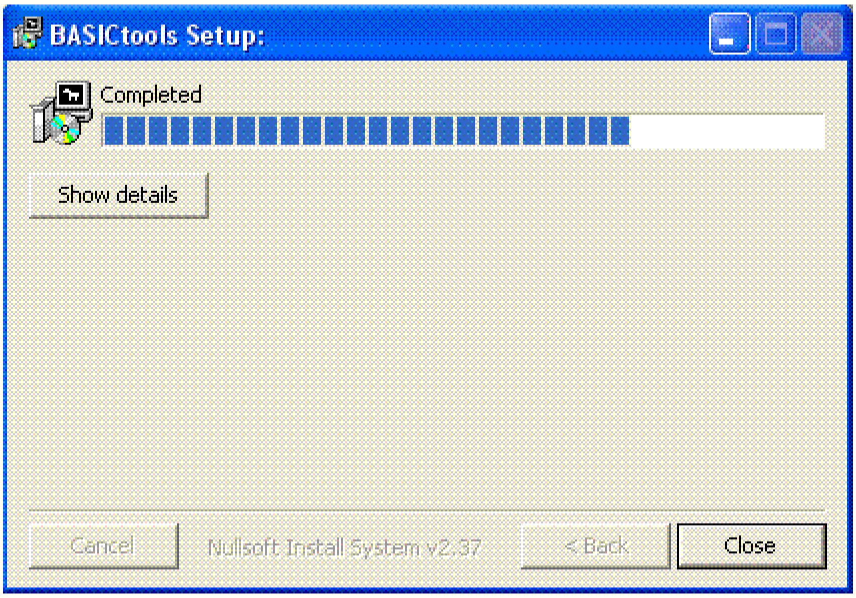
If you are installing from the CD, then it will automatically run the install program when the CD is inserted. If downloading from the web, run the SETUP program to start the installation.



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** .

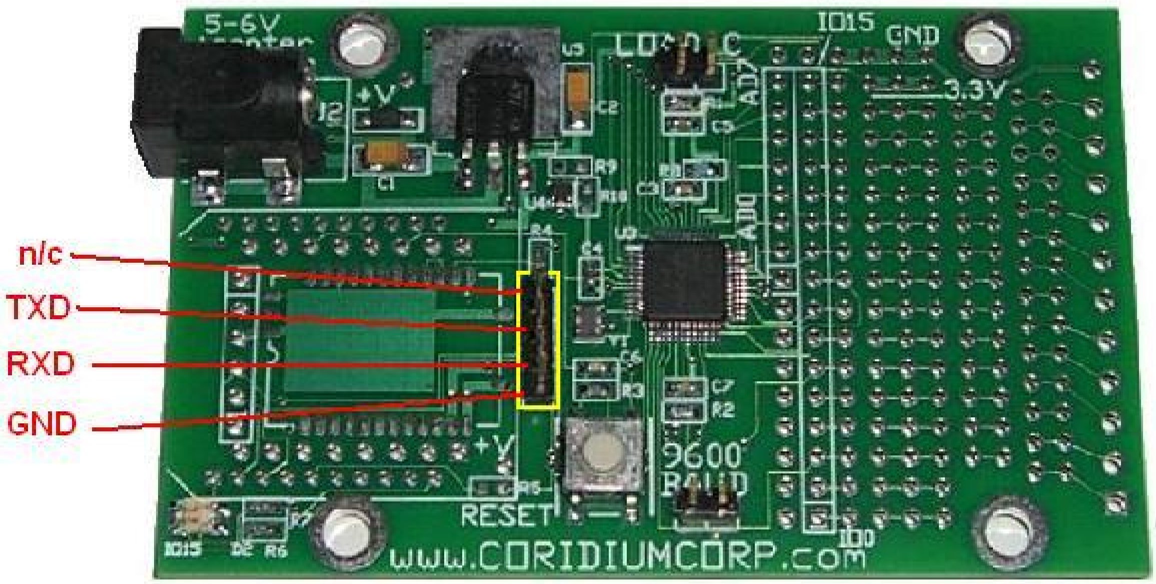
And its as easy as that.

On to Step 2

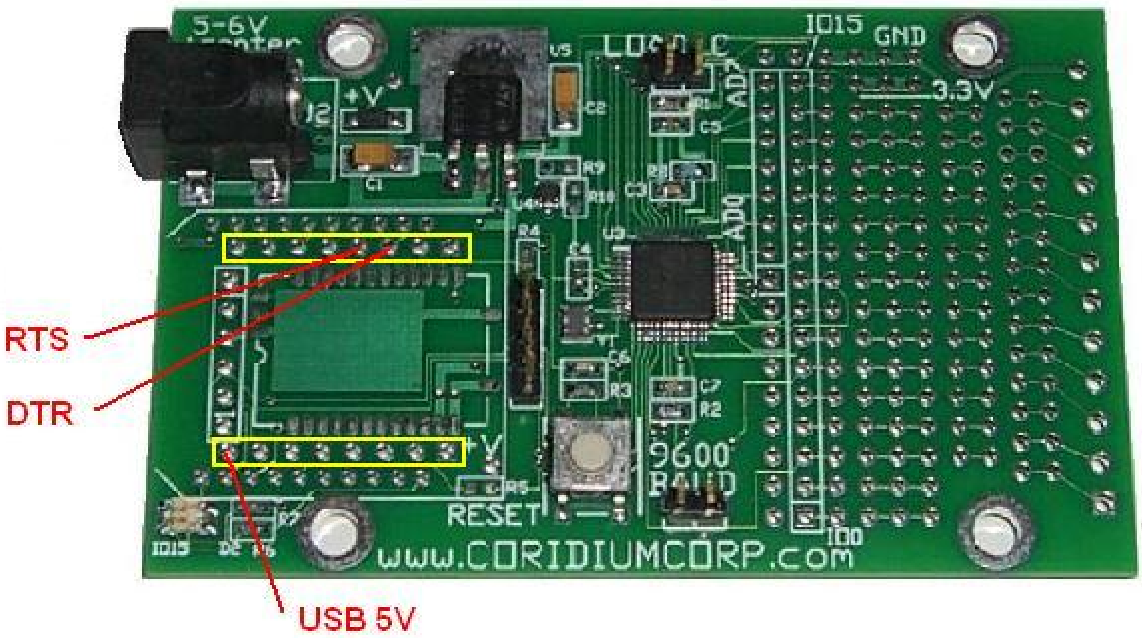
Step 2: Make USB connections

The Wireless **ARMmite** can be connected to **SparkFun's USB breakout board**. The minimal connection uses a 4 pin 0.1" header. This connection gives a hardwired serial connection for configuration and debugging, which can be useful during the initial setup of the tools and software, or for monitoring serial traffic during program debugging.

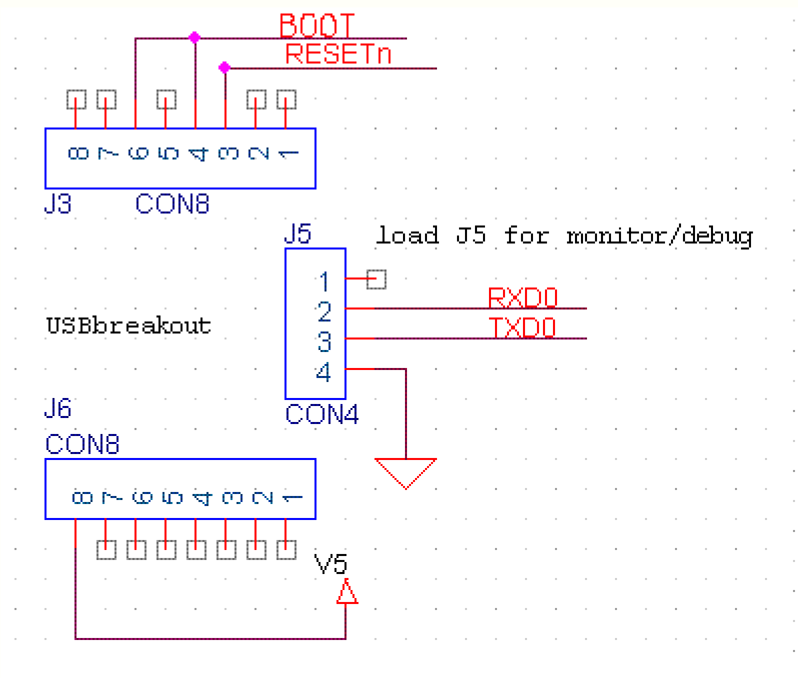
Minimal USB connections. The pin diagram is shown below (pin names to match the USB breakout board)



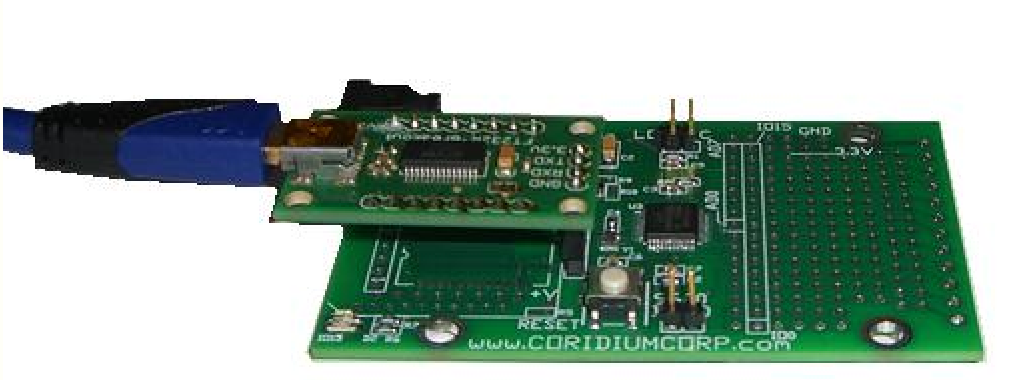
Additional pins may be wired up to the USB breakout board, so that it will work identically with the original ARMmite. In this case 5V from the USB will power the board. **But when that connection is made a power supply should NOT be connected**.



Below is the schematic with the names representing the perspective of the ARM processor (RXD0 on the ARM connnects to TXD on the USB breakout board).

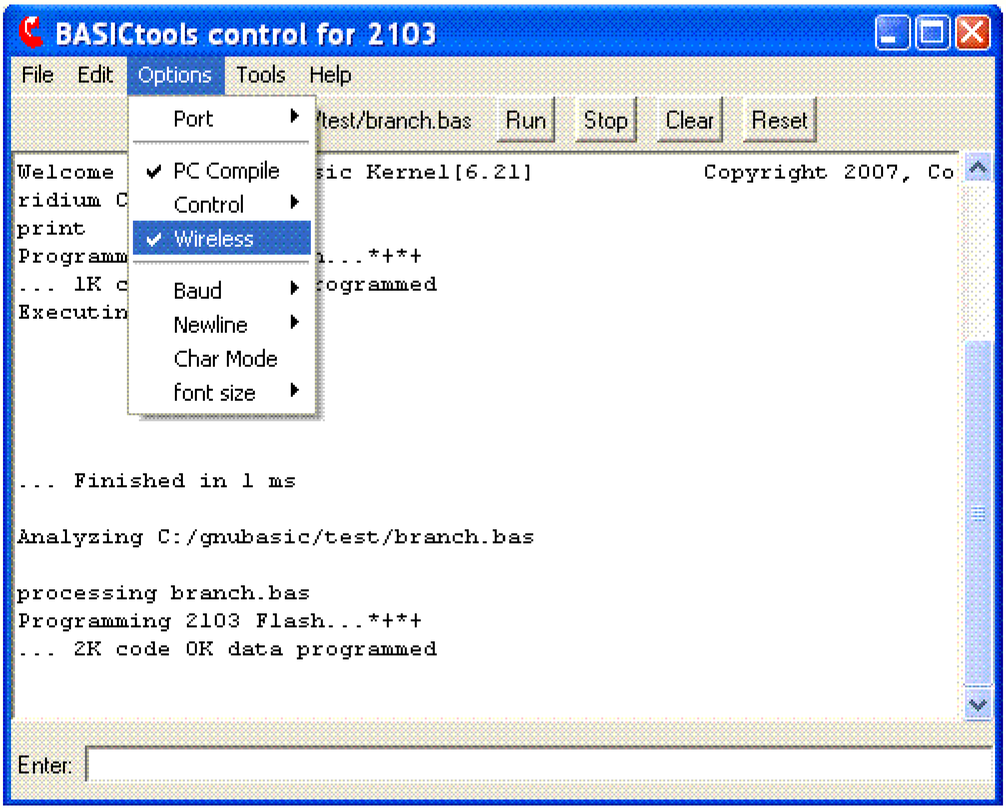


Shown below is the orientation with the USB breakout board mounted on the ARMmite, using a 4 pin receptacle soldered into the breakout board-



BASICtools Configuration

While you are not using a wireless connection, if you are using just the 4 pin connection to the USB breakout board, the Wireless ARMmite is functioning in "Wireless" mode as there is no control from the PC for reset. So for BASICtools to function correctly you must enable the Wireless option shown below.

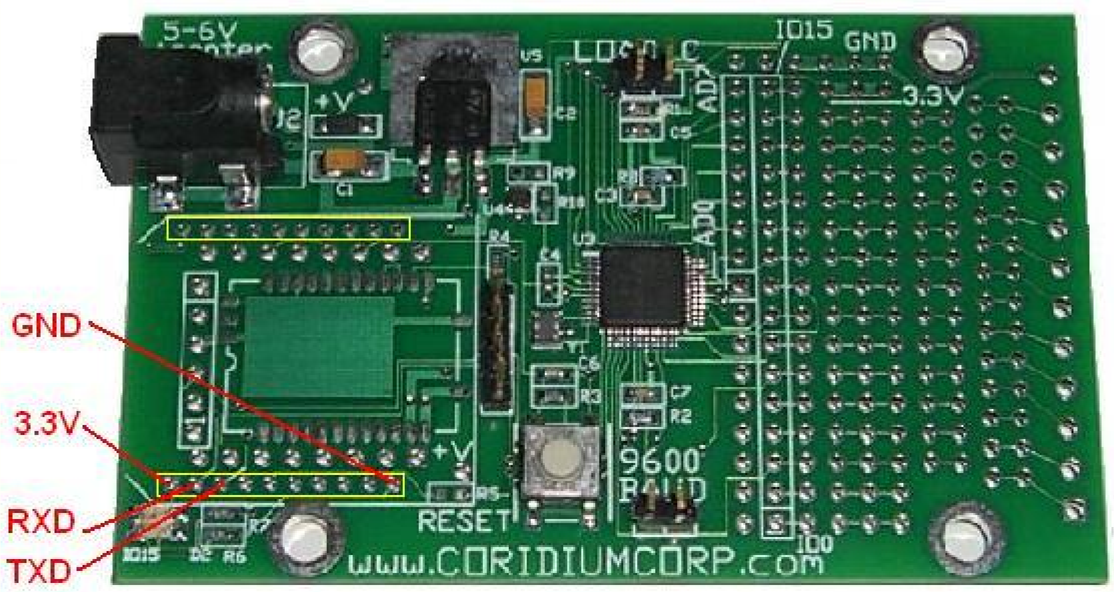

BASICtools control for 2103
File
Edit
Options
Tools
Help
Port
PC Compile
Control
Wireless
Baud
Newline
Char Mode
font size
/test/branch.bas
Run
Stop
Clear
Reset
Welcome
ridium C
print
Programm
... 1K c
Executin
sic Kernel[6.21]
Copyright 2007, Co
n...*+*+
rogrammed
... Finished in 1 ms
Analyzing C:/gnubasic/test/branch.bas
processing branch.bas
Programming 2103 Flash...*+*+
... 2K code 0K data programmed
Enter:

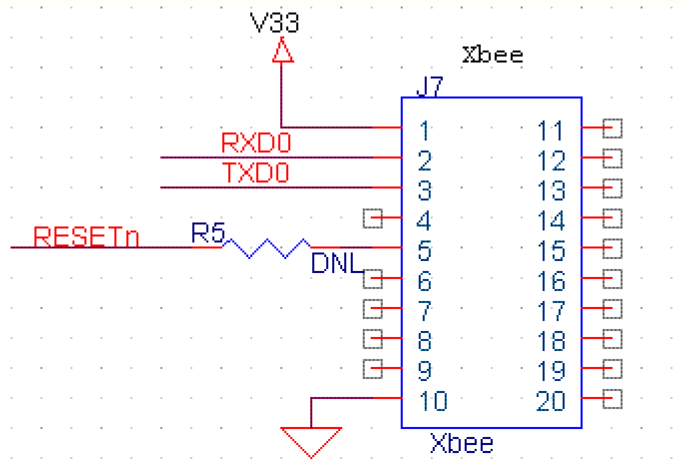
On to Step 2

Step 3: Make Zigbee connections

The Wireless **ARMmite** can be connected to **Maxstream Zigbee Xbee and Xbee PRO modules** (available at **Newark** or **Digikey**). The connection uses **two 10 pin 2mm receptacles** .

Xbee connections. The pin diagram is shown below-





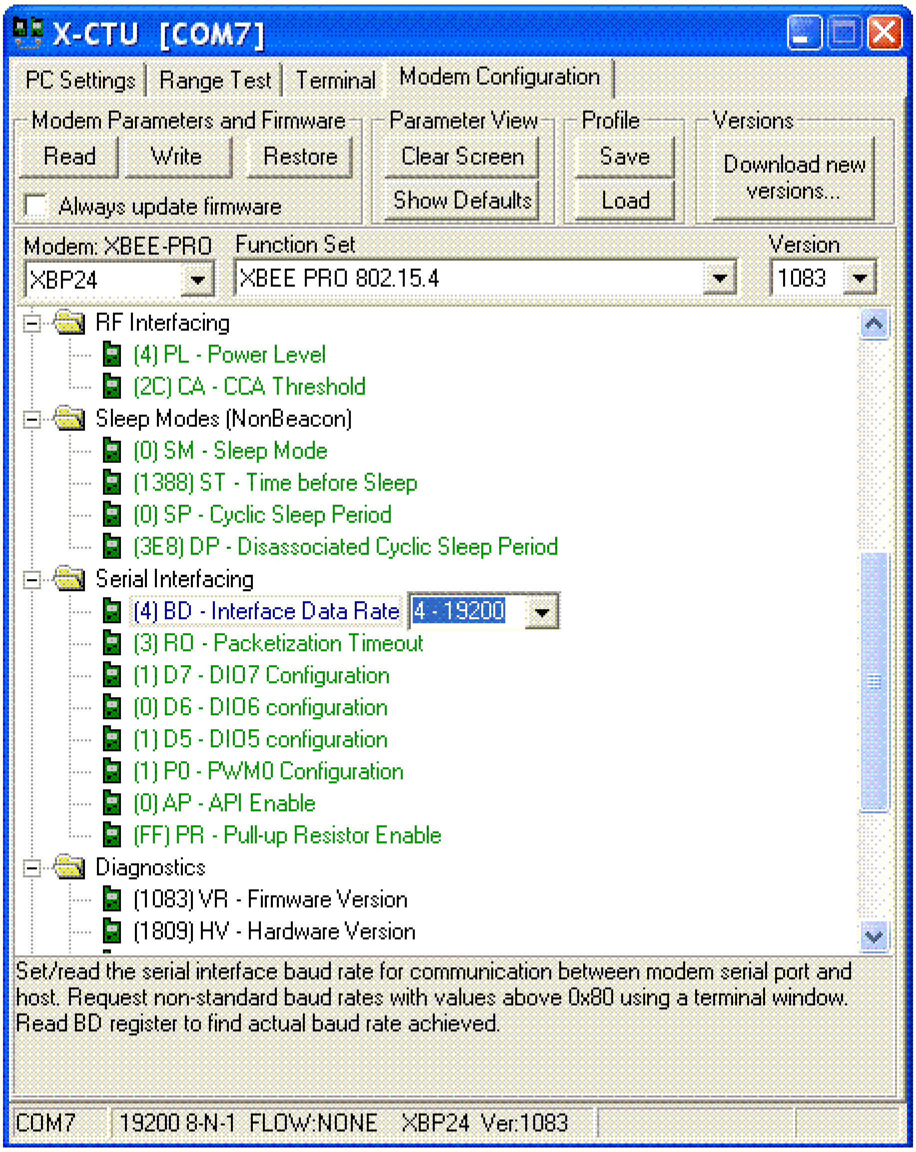
Shown below is the orientation with the Xbee module mounted on the ARMmite, using **two 10 pin receptacles** soldered into the ARMmite-

PC side connection

The other end of the Zigbee wireless connection can use a **Maxstream Development Kit** or a **USB Maxstream dongle adapter**. At present while Zigbee is a standard, modules are only compatable with each other if they are based on the same firmware which often means both ends are from the same vendor (Maxstream in this case). Follow directions supplied with that unit for installation on the PC. The **Maxstream X-CTU utility** can be used to configure and test the setup.

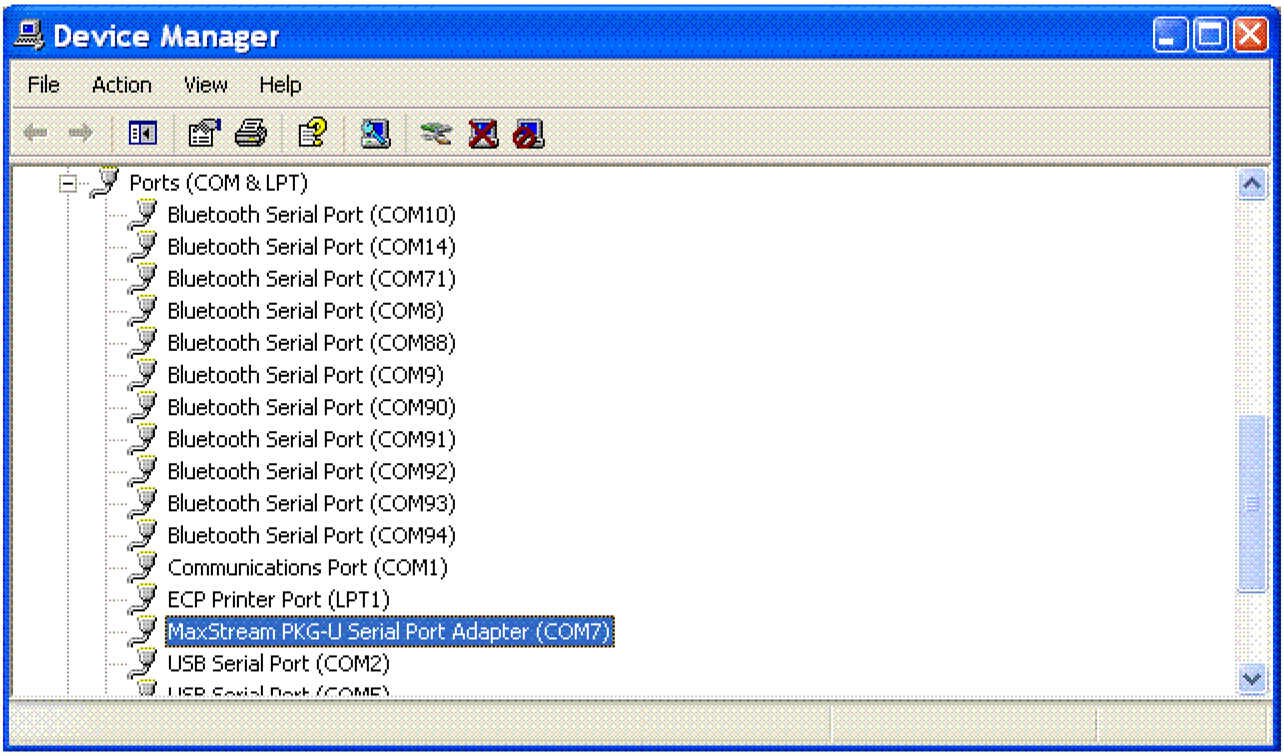
Setting the Baud rate

The default baud rate for the Xbee module is 9600 baud. This can be changed to 19.2Kb with the X-CTU utility or it can be lett there. To run the ARMmite at 9600 baud, install a jumper on the 9600 BAUD location.

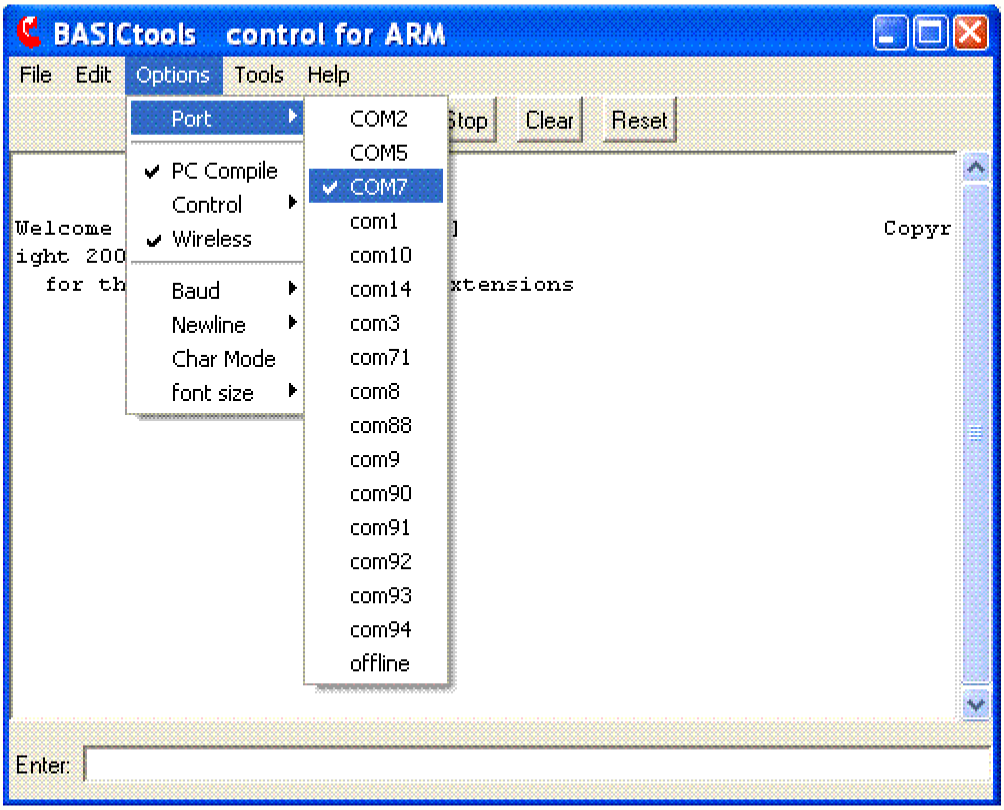


Setting the COM port

The port for the USB connection can be set with the Control Panel.



At this point BASICtools will work normally, (make sure you check the Wireless option, and note that the serial port will be one identified as a USB serial - in capital letters, assuming you're using the USB adapter)-

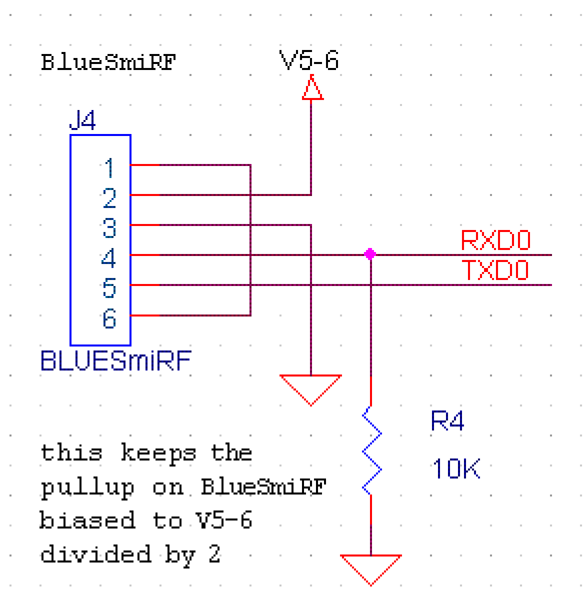


On to Step 2

Step 3: Make BlueSMiRF connections

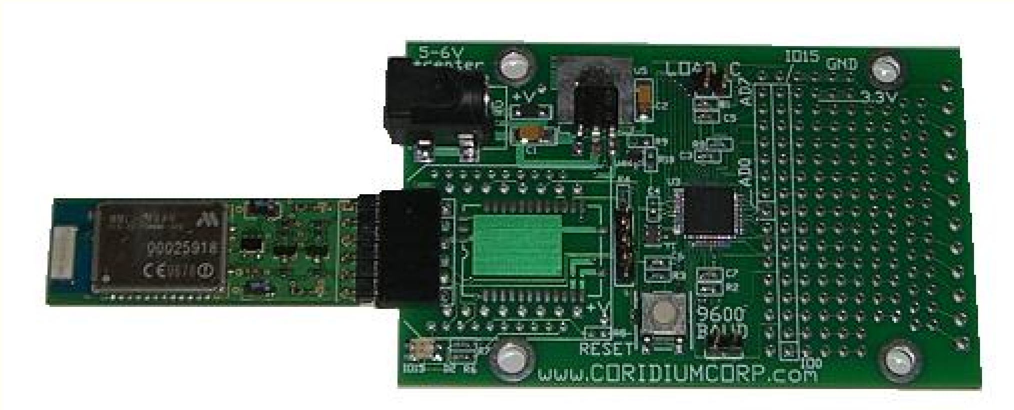
The Wireless **ARMmite** can be connected to **SparkFun's BlueSMiRF module** . **When using the BlueSMiRF, the power from the wall adapter is applied directly to the BlueSMiRF and it must be limited to 6V or less.** **We recommend using a regulated 5V supply such as carried by SparkFun** .

The connection can be made with a right angle 0.1" receptacle or by soldering the 2 boards together directly. The pin diagram is shown below-



Shown below is one orientation with the BlueSMiRF mounted below the ARMmite-

And this orientation is also proper-

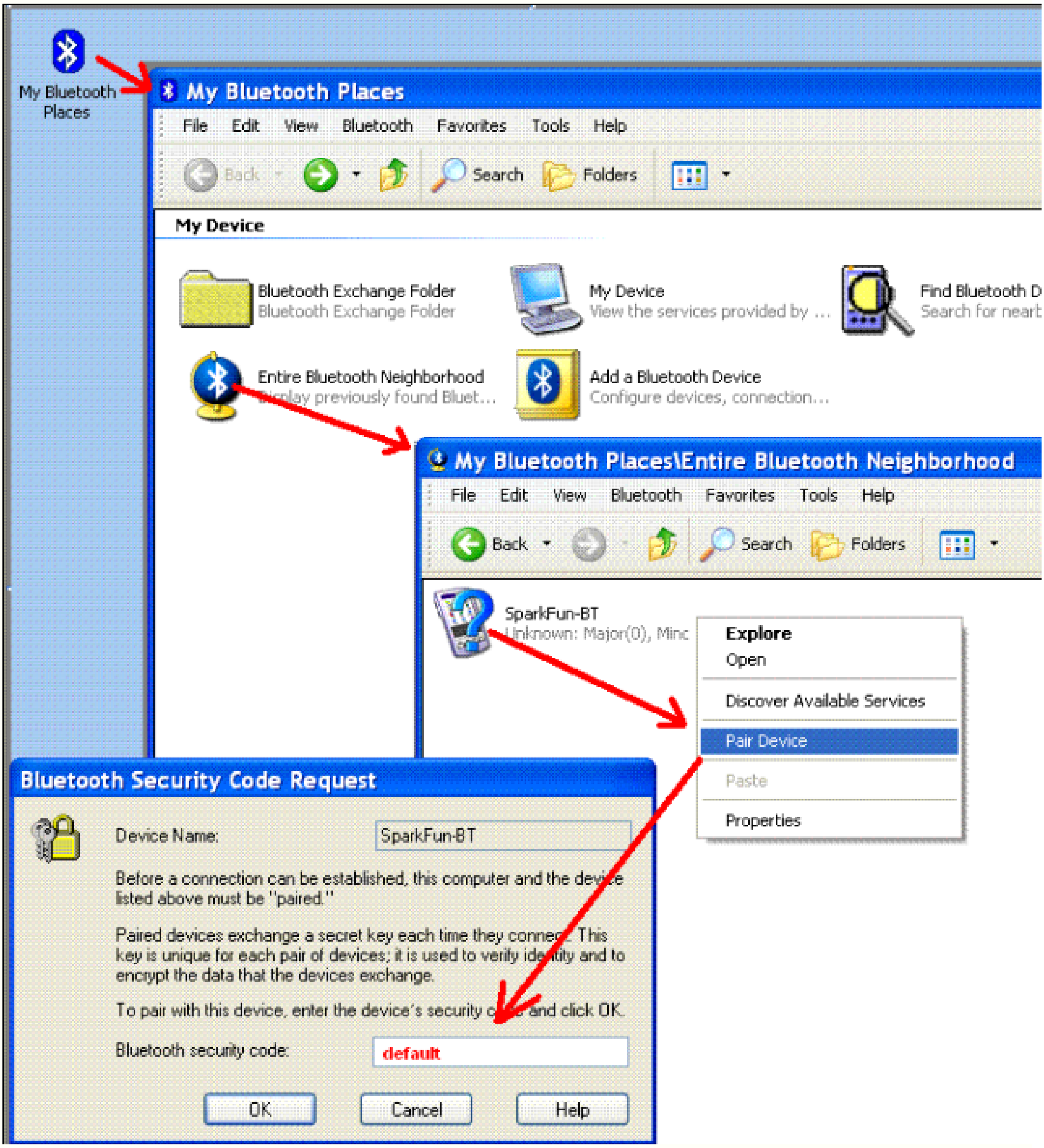


PC side connection

The other end of the Bluetooth wireless connection can use a **Bluetooth USB dongle** . Follow directions supplied with that unit for installation on the PC. Do not try to install more than 1 Bluetooth USB dongle on a PC, as the drivers will probably conflict. Also the Bluetooth software will assign a number of serial ports of which 2 may be used to emulate a serial connection that can be used with the BASICtools.

WIDCOMM tools

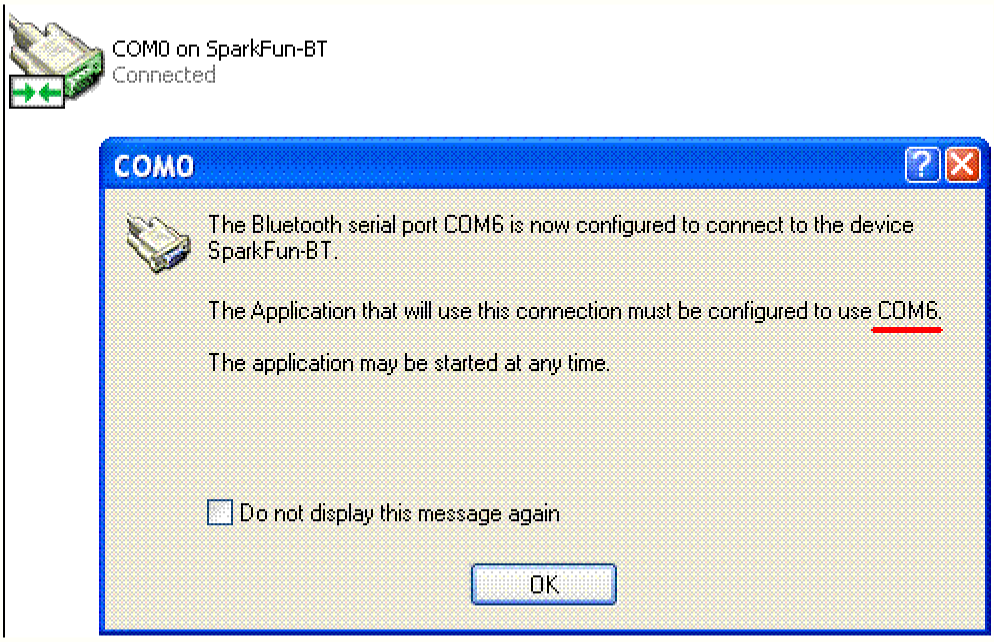
After you install the tools (the latest from SparkFun are the WIDCOMM utilities), you will see a BlueTooth icon on the desktop. When connecting for the first time open this to "pair" the PC to the BlueSmiRF--



The Bluetooth security code for the SparkFun BlueSmiRF is "default". In some versions of the BlueSmiRF this pairing must occur within 60 seconds of the BlueSmiRF powering on. So it may be necessary to cycle power before the pairing. The symptom for not pairing is that no services will be available for the BlueSmiRF.

Now paired, if you double click on the SparkFun-BT it should display

At this point you should be able to connect the serial port by double clicking or right clicking on this icon. If the connection is made, the flashing green LED will go off on the BlueSmiRF and its red LED should be continously on. The icon should now show-



Now you know where the com port has been located, as COM6 in the above example.

You can now start the BASICtools and use COM6. Warning, the drivers will often at this point get confused, and you may not be able to make the connection, but at least at this point everything is configured correctly. One indication is that the red LED on the BlueSmiRF will go off, and it will return to flashing green. The best course is to reboot Windows at this point, start BASICtools, set the com port and baud rate and then make the Bluetooth connection.

Setting the Baud rate

The default baud rate for the BlueSMiRF is 9600 baud. Once communication is established it can be changed to 19.2Kb or it can be left at 9600. To run the ARMmite at 9600 baud, install a jumper on the 9600 BAUD location.

The command to change the baud rate is done with an AT command, specifically ATSW20,79,0,0,1<cr> which can be done with a short BASIC program

version 7

```
PRINT "ATSW20,79,0,0,1"
```

version 6

```
SEROUT 16,9600,["ATSW20,79,0,0,1",13]
```

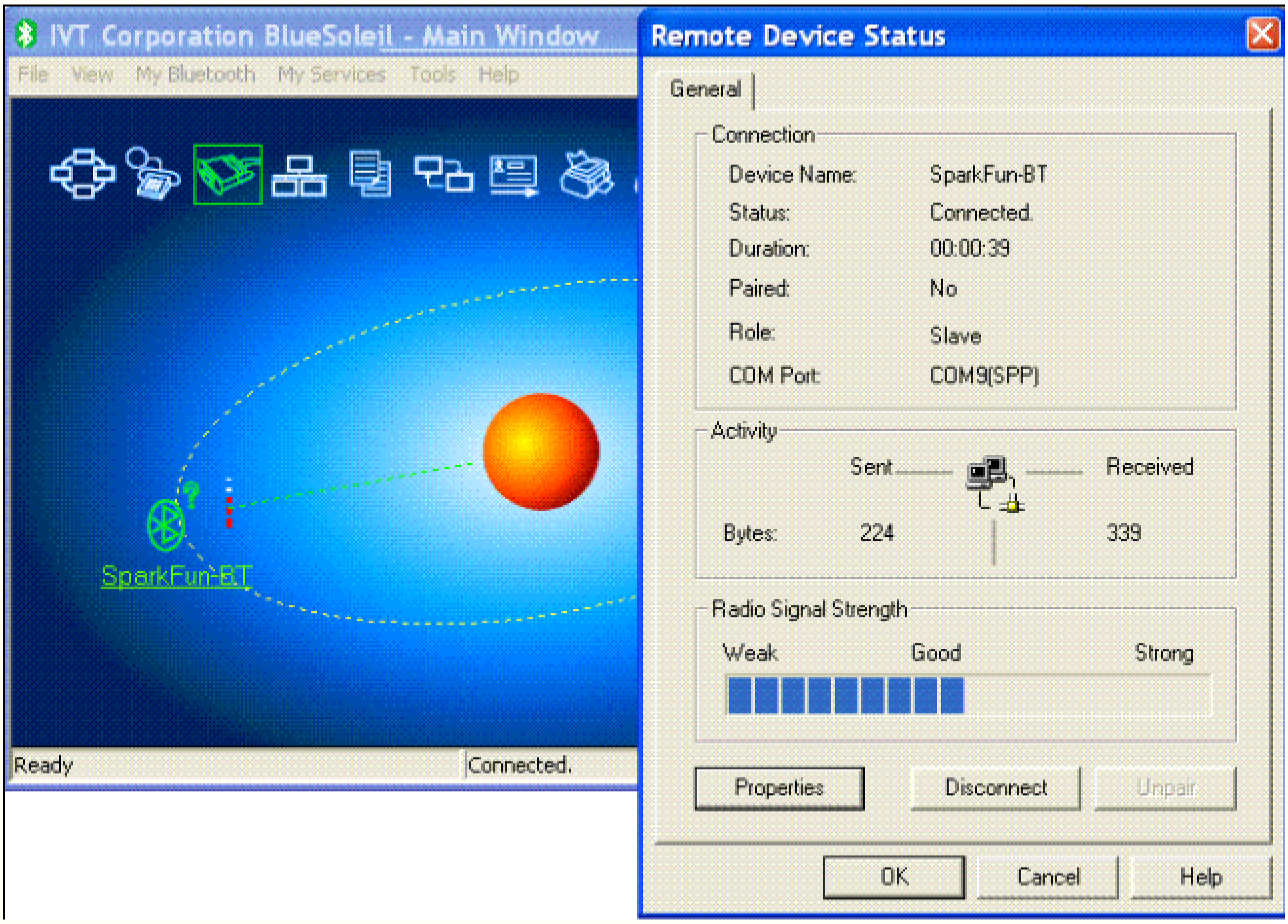
To return to 9600 baud

```
PRINT "ATSW20,39,0,0,1"
```

BlueSoleil connection

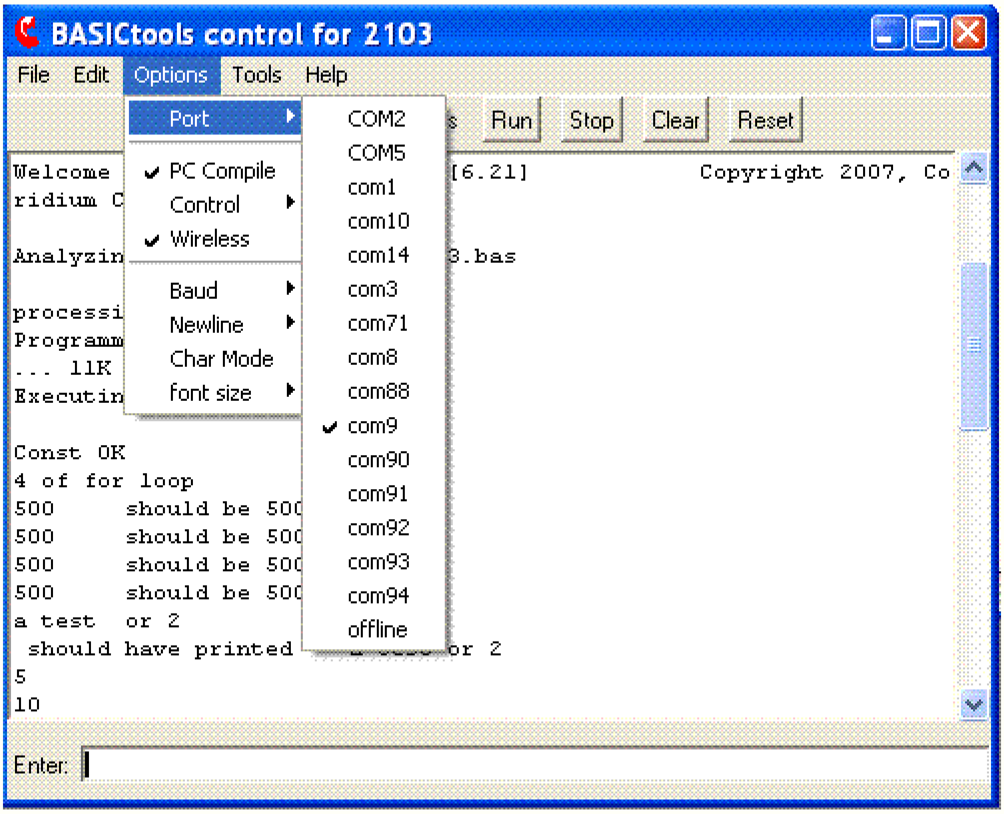
In IVT's BlueSoleil, this is not a trivial excercise. And it seems to be a bit hit or miss. The listing of ports in the Control panel also seems a bit arbitrary and the services option of BlueSoleil seems to misreport which COM port will be assigned. But once a connection is made on the proper port, it does seem to stay there through reboot.

To connect the serial port, you may need to Refresh Devices, Refresh Services, then Connect. When all is working well you can identify the port being used in the Status window-



At this point BASICtools will work normally, (make sure you check the Wireless option, and note that the serial port will not be one identified as a USB serial - in capital letters)-

Also if you disconnect the service in the BlueSoleil utility, you will need to exit the program, restart it, refresh devices, refresh services and then connect.

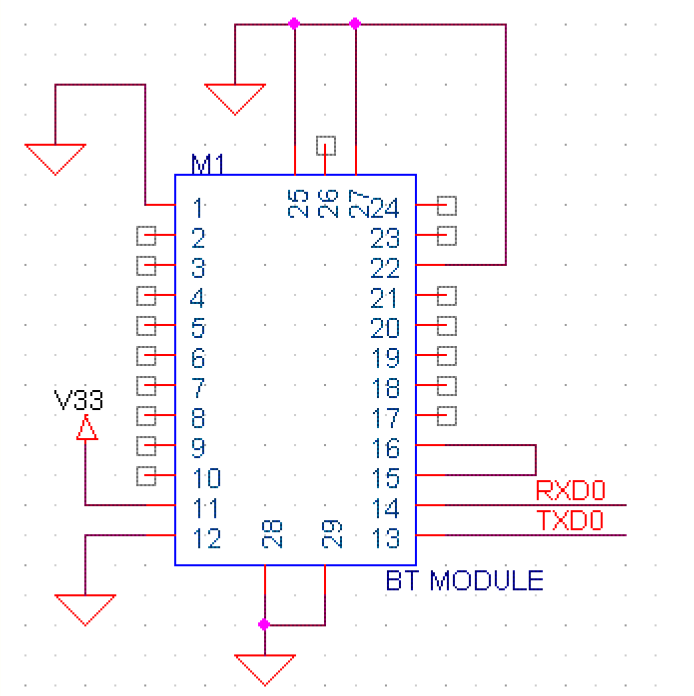

BASICtools control for 2103
File Edit Options Tools Help
Port
PC Compile
Control
Wireless
Baud
Newline
Char Mode
font size
COM2
COM5
com1
com10
com14
com3
com71
com8
com88
com9
com90
com91
com92
com93
com94
offline
Run Stop Clear Reset
Welcome
ridium C
[6.21] Copyright 2007, Co
Analyzin
3.bas
processi
Programm
... 11K
Executin
Const OK
4 of for loop
500 should be 500
500 should be 500
500 should be 500
500 should be 500
a test or 2
should have printed
or 2
5
10
Enter:

On to Step 2

Step 3: Make BlueTooth SMD module connections

The Wireless **ARMmite** can be connected to **SparkFun's BlueTooth v2.0 SMD module** .

Under development

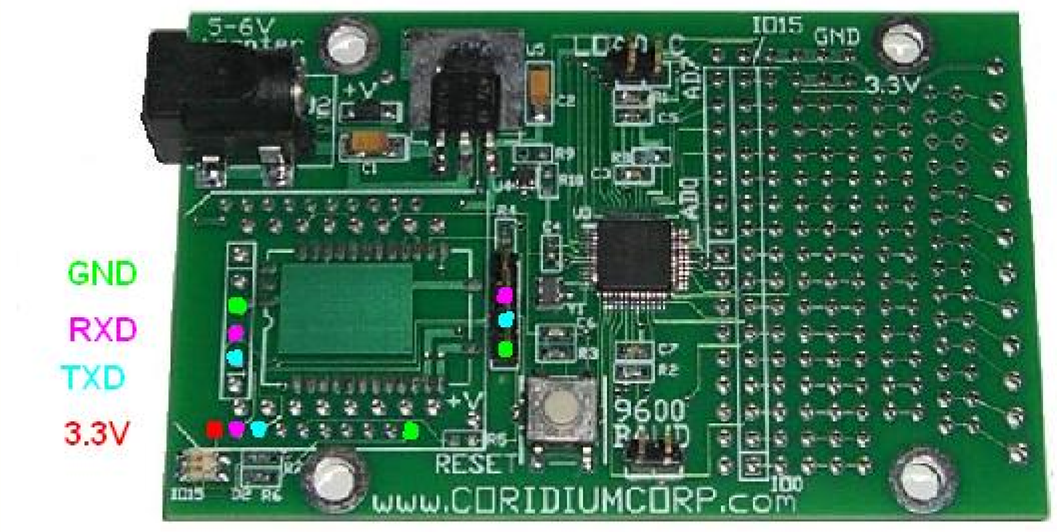


So far there has not been enough customer interest to complete this board.

Step 3: Custom Serial connections

The Wireless **ARMmite** can be used with the USB breakout board to download BASIC code, but then use the download/debug connection to communicate with some other serial device..

Available serial connections. The pin diagram is shown below (pin named for perspective of the ARM CPU, RXD is an input to the ARM).

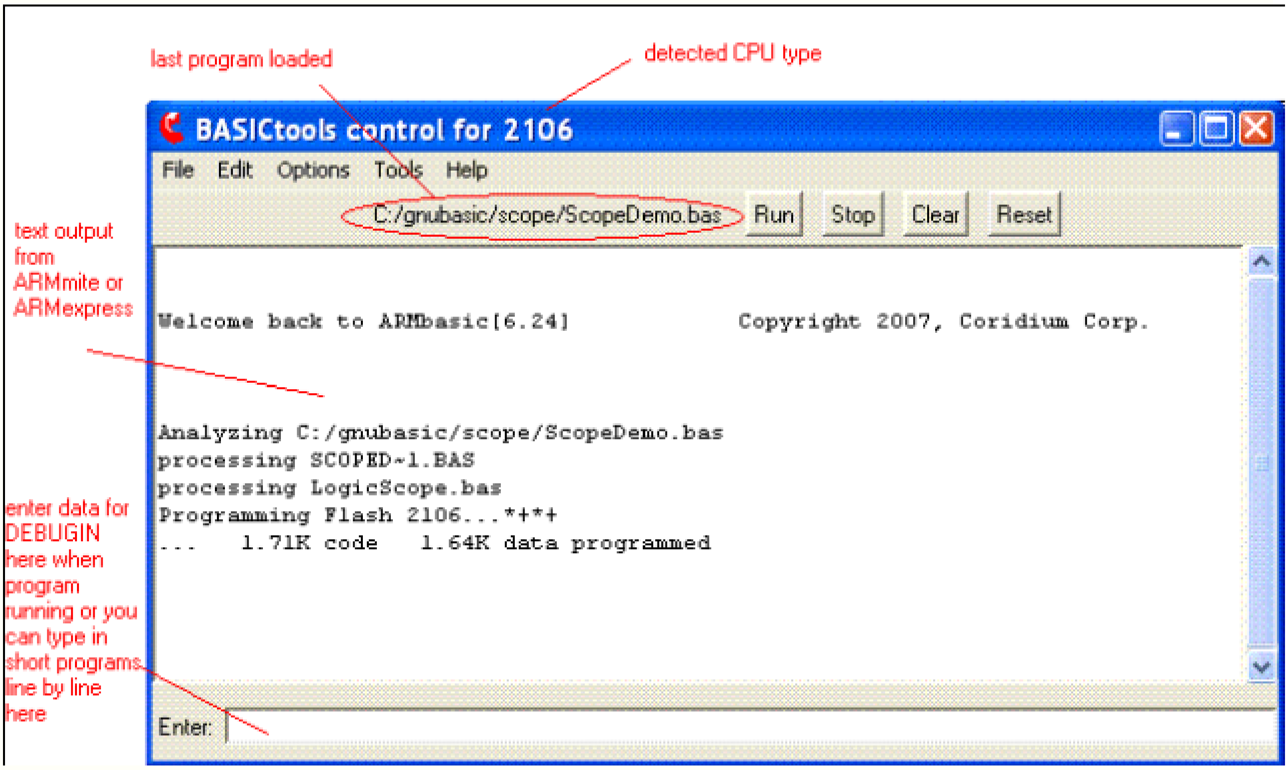


BASICtools Features

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, close BASICtools and then restart it (you may need to use the Task Manager to exit). It will STOP your spewing program.

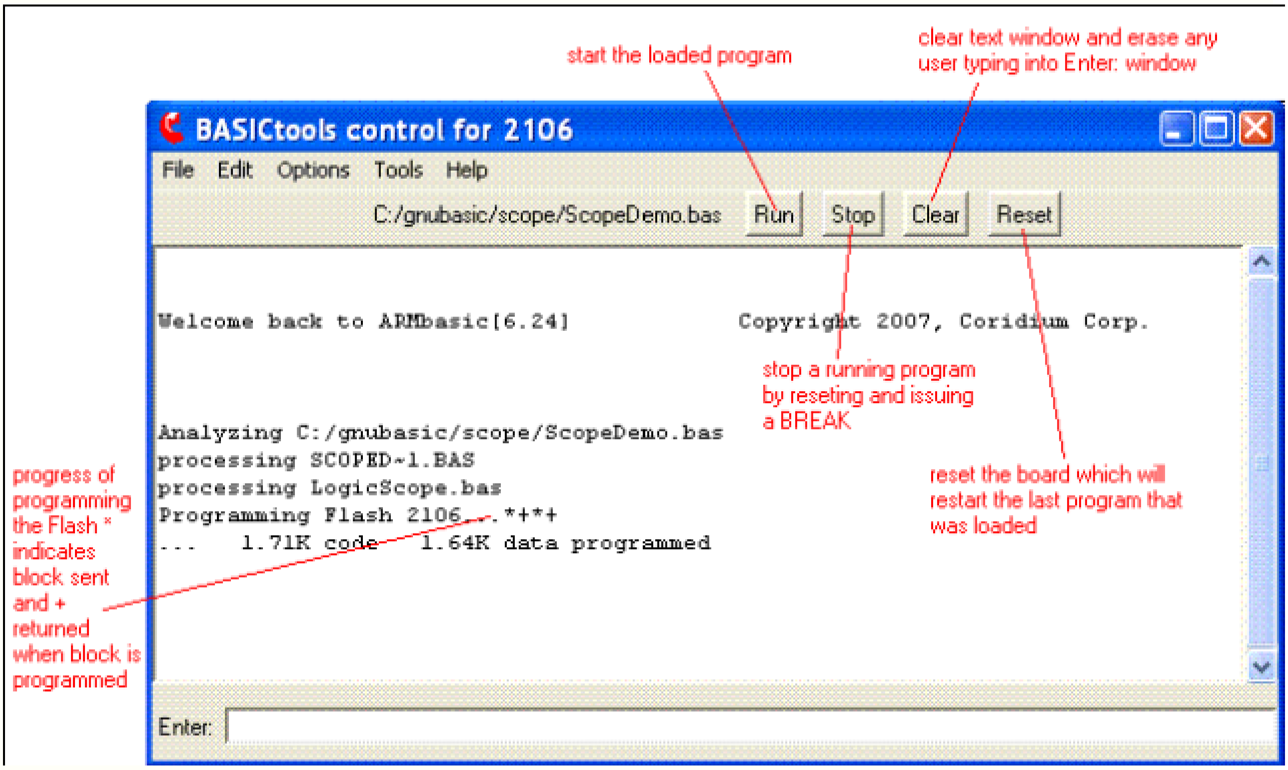
BASICtools Layout



keyw ords: enter line debugin type BASIC commands

Buttons

..



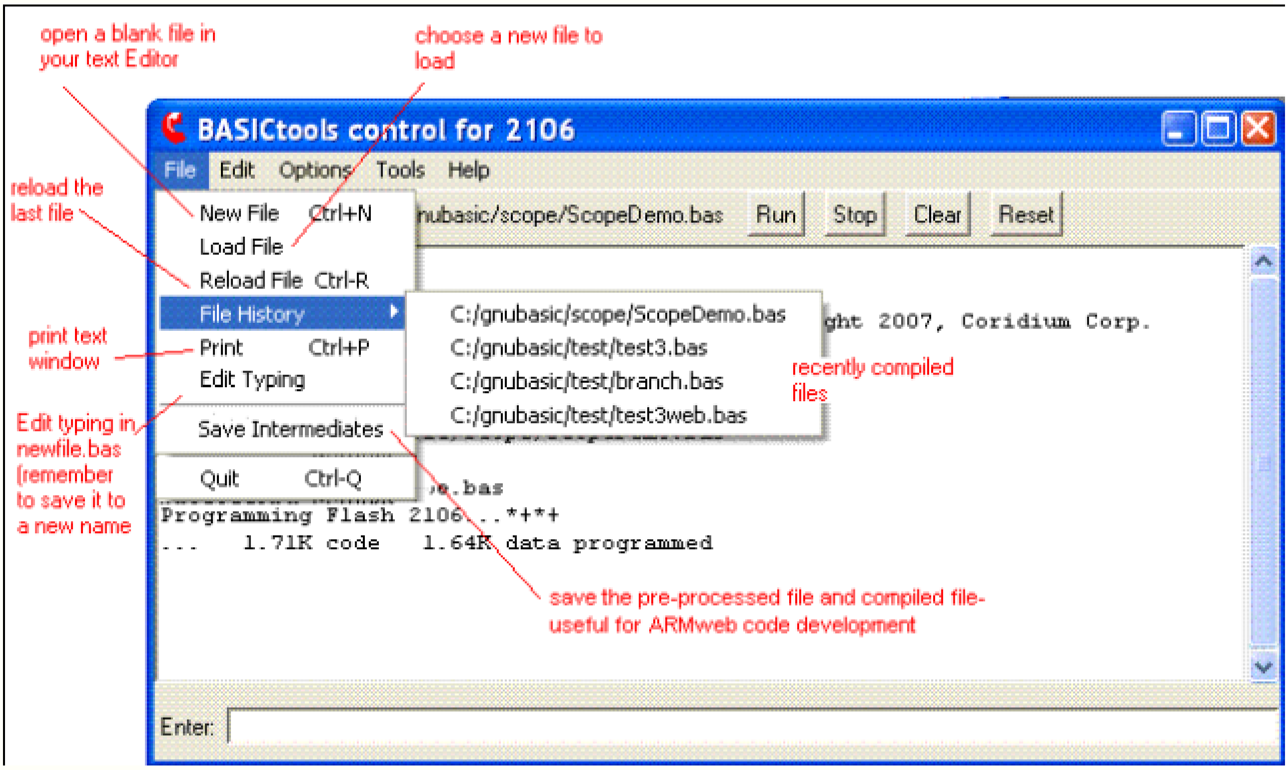
The CLEAR button only erases the display screen and the buffer on the PC of statements you have typed into the Enter window.

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

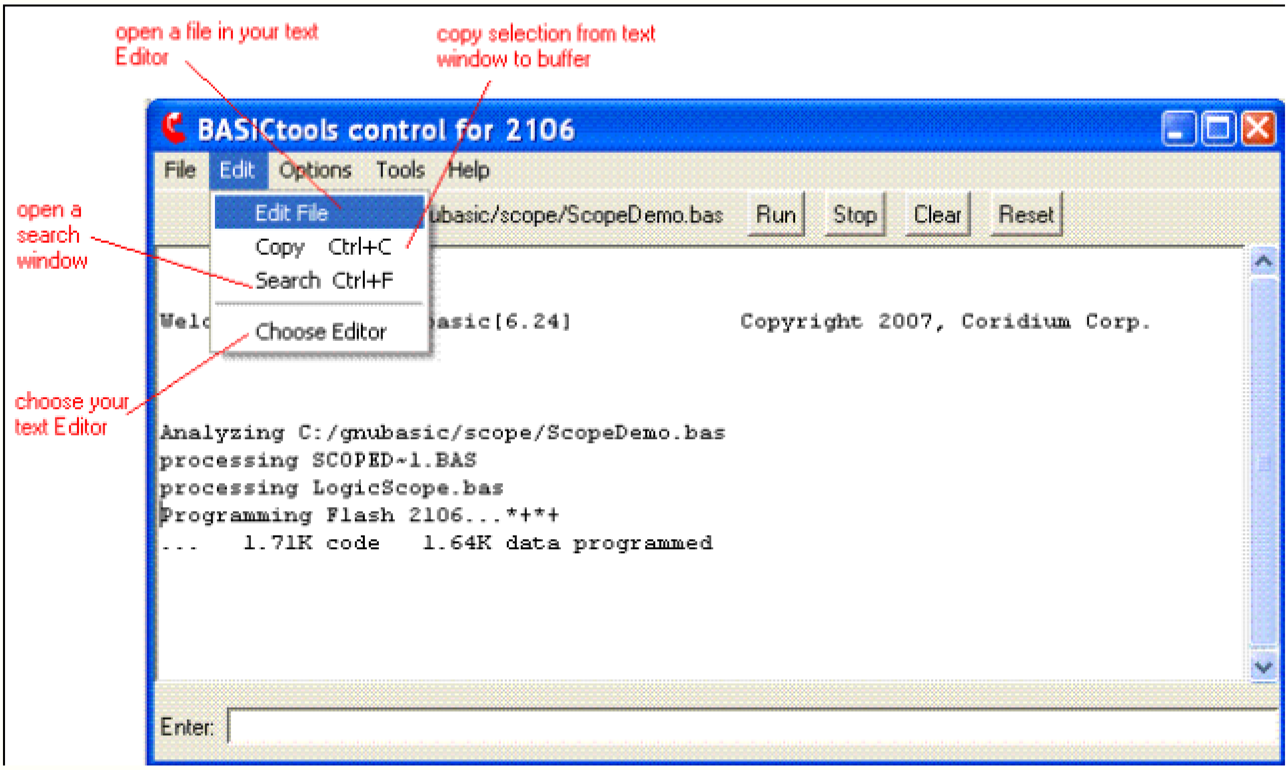
..



file load file reload file print save file quit

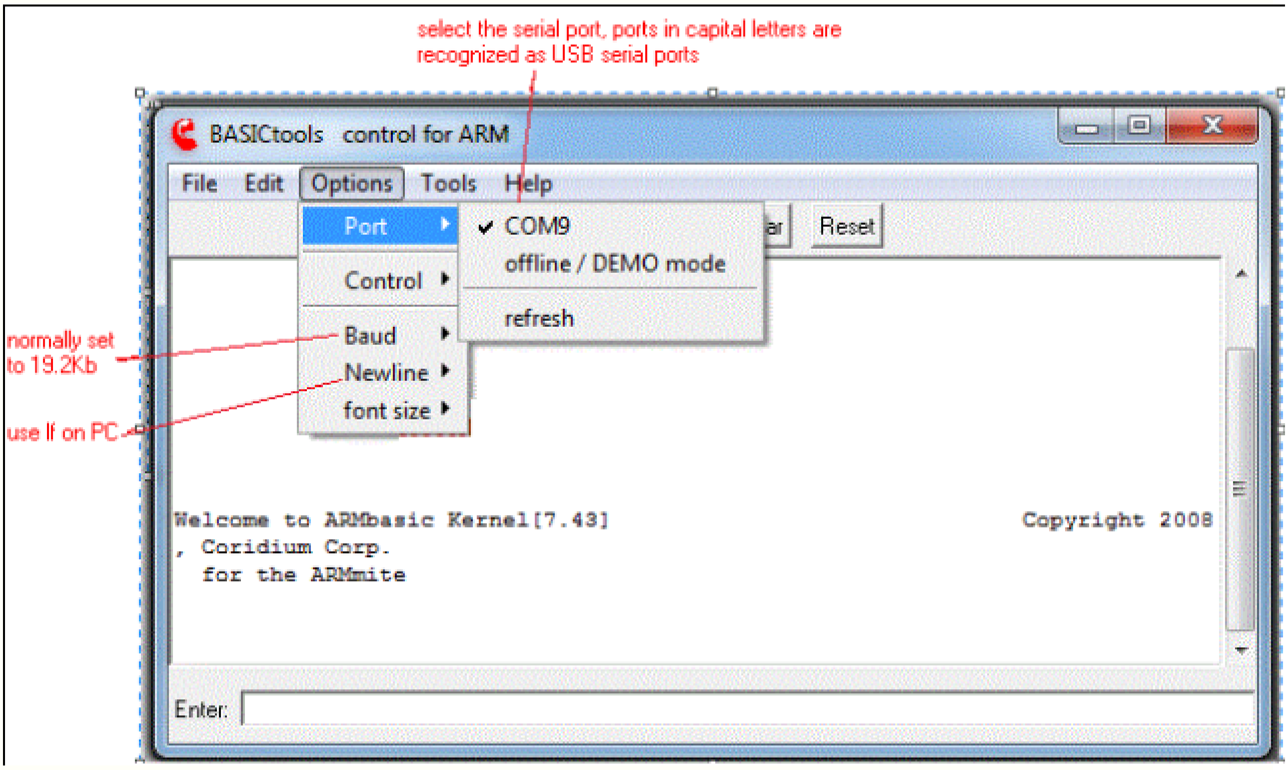
Edit Menu

..



keyw ords: edit choose editor

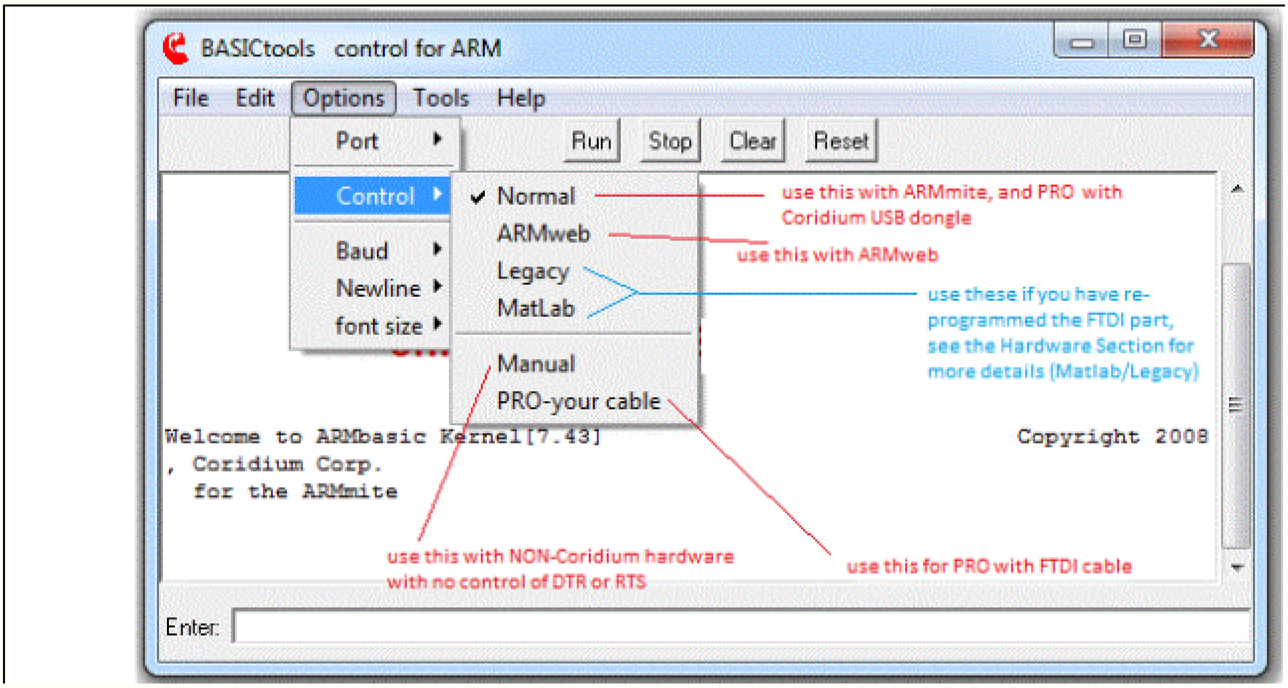
Options Menu



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

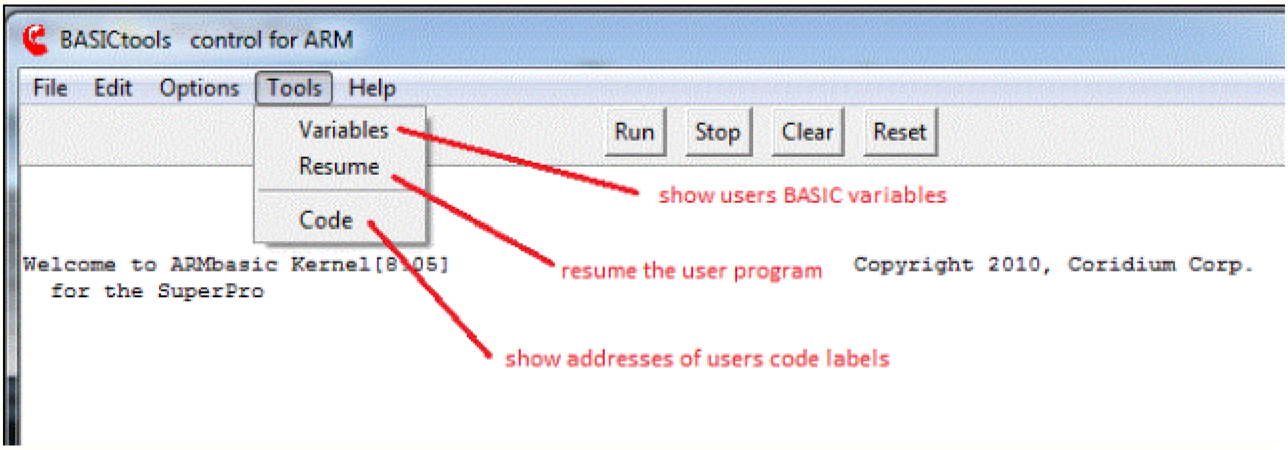
Control Menu



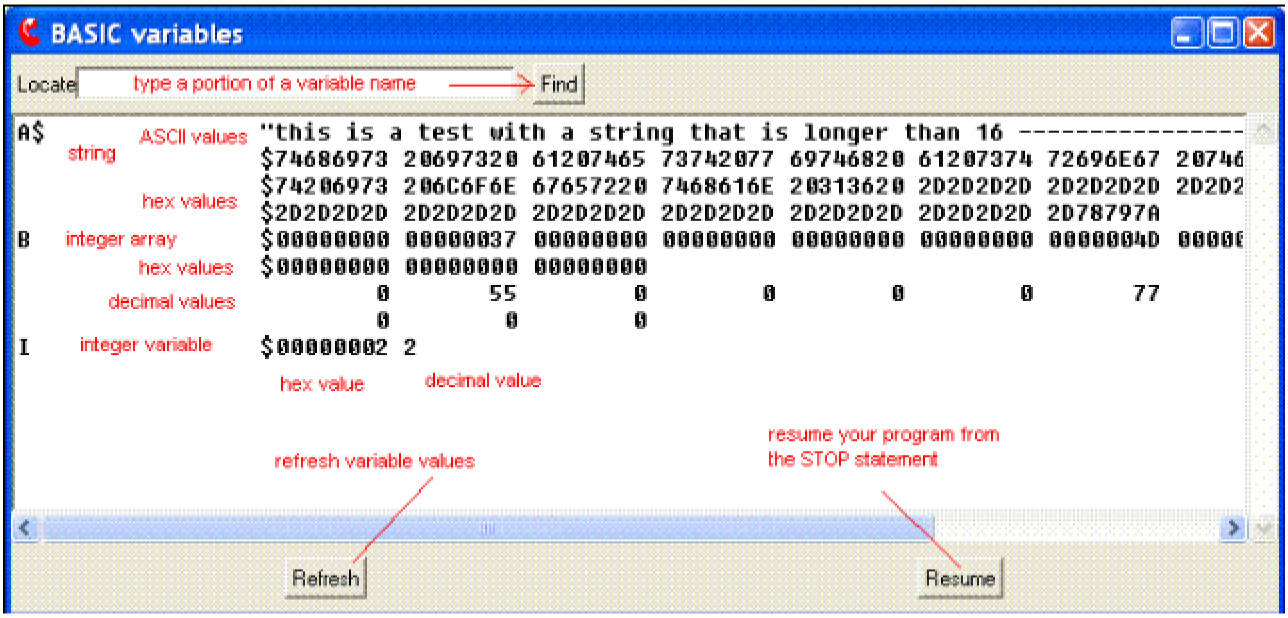
keyw ords: options port baud new line char mode PC compile control throttle

BASIC variable viewer

Open this window from the Tools Menu (variables)

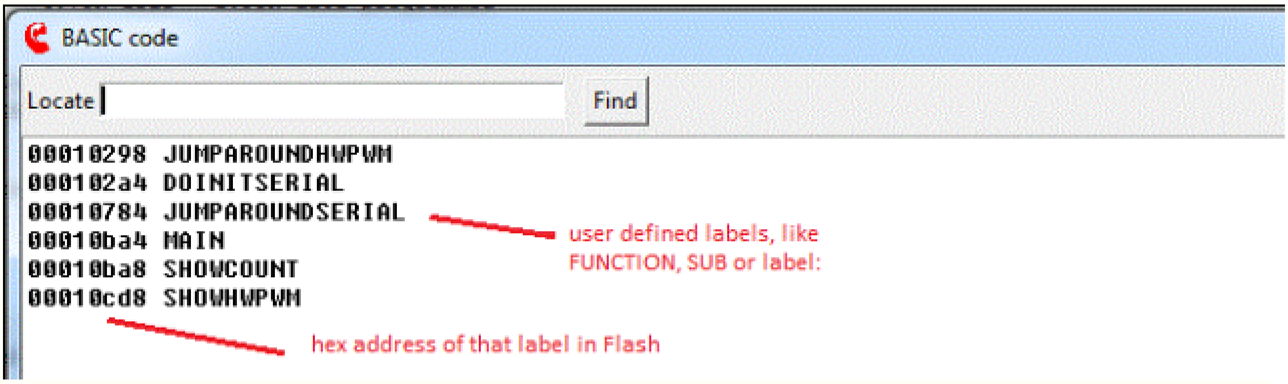


variables window



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

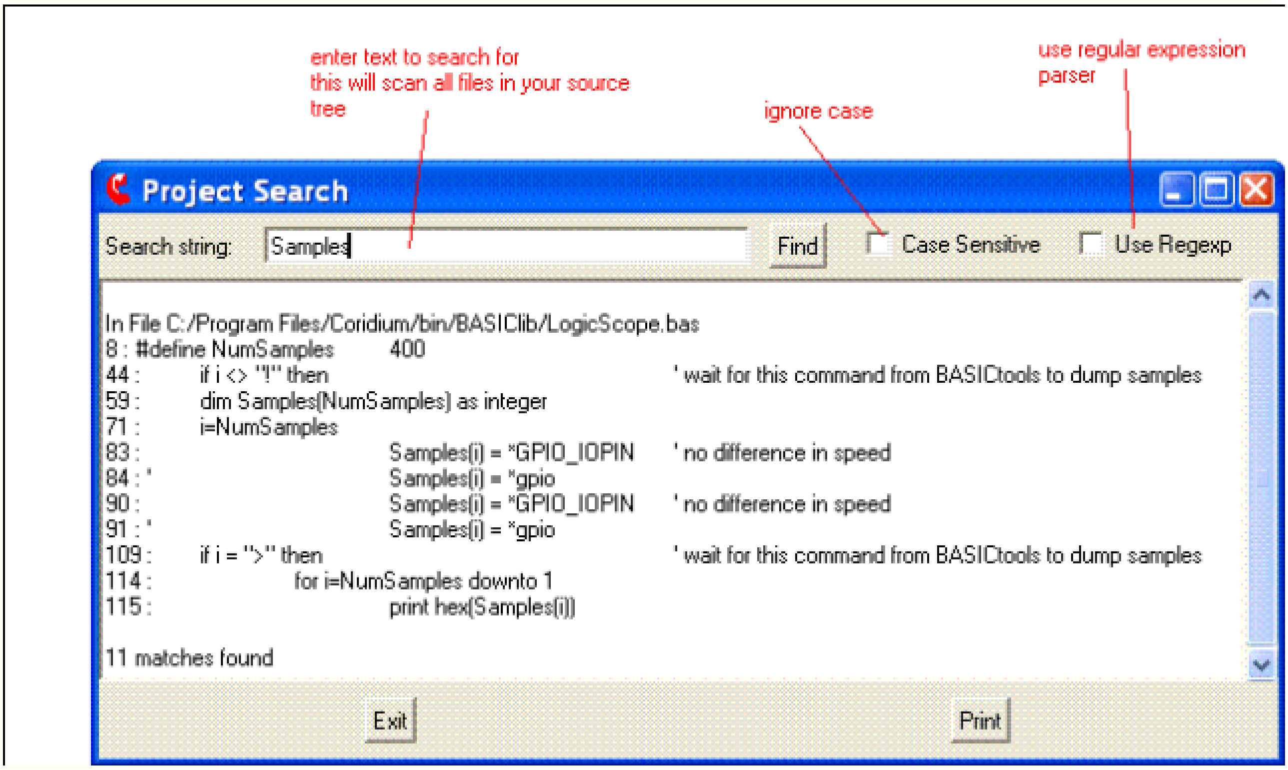
code window



keyw ords: variable dump breakpoint STOP view memory

Search Window

Open this window from the Edit Menu



keywords: search

Logic Scope Window

This module must be included in your BASIC program. It will monitor the pins for a period of time when called from your program.

See the example program ScopeDemo.bas and details in the **Logic Scope Section** .

keyw ords: oscilloscope logic analyzer logic scope

Win98 Setup

The BASICtools.exe is not compatable with Win98. BASICtools.exe is generated by the Freewrap utility which turns a Tcl program into a standalone executeable requiring no other .DLLs or support programs.

There seems to be a bug in the Freewrap that does not support the calls to batch (.BAT) files that BASICtools uses to run the pre-processor.

When Tcl is installed on a Win98 system, and the Tcl source is run that way, it all functions normally. So that is the current work-around for Win98 systems. So to run BASICtools on Win98, you will need to install some version of TclTk. We like the MinGW version as it is pretty simple and requires only a few .DLLs. This is available at SourceForge.net, and we also have a copy on our server-

TclTk 8.4.1 installation

This is a self-extracting executeable that will install TclTk.

Once this is installed copy the BASICtools.tcl file to the \Program Files\Coridium\ directory

BASICtools.tcl

Change the Desktop shortcut for BASICtools from

C:\Program FIles\Coridium\BASICtools.exe to C:\Program FIles\Coridium\BASICtools.tcl

This will launch the Tcl source version which runs on Win98.

While this is not optimal, it does work, and will probably be required for Win98 to be used.

This section does NOT apply to Coridium Hardware Products, it is for installing BASIC on boards from other vendors.

The **ARMbasic** compiler runs on the PC, in combination with a BASIC support library that is installed on the ARM.

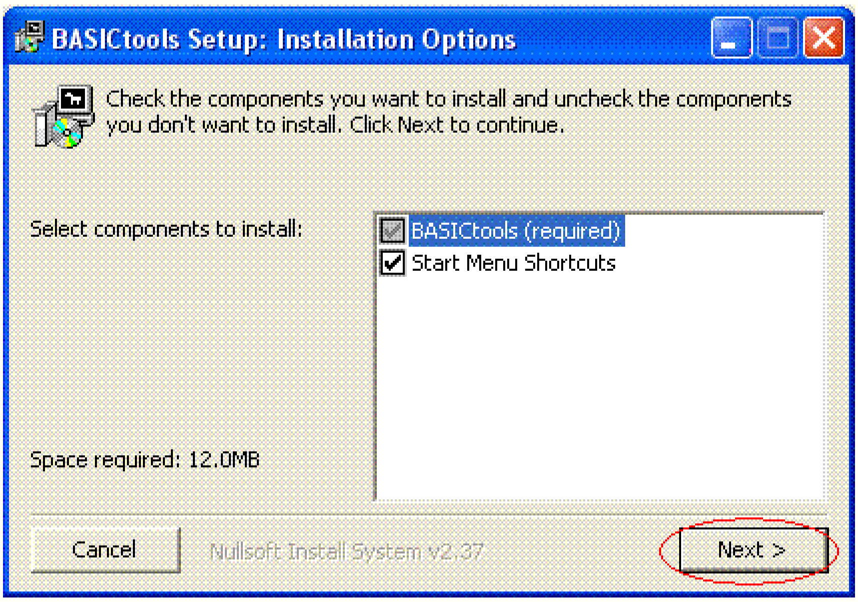
Getting Started Install Software Install DEMO Firmware Unlocking the firmware installer	Writing your first program Programming the IO More complex programs Trouble Shooting BASICtools Features

This section does NOT apply to Coridium Products, it is for installing BASIC on boards from other vendors.

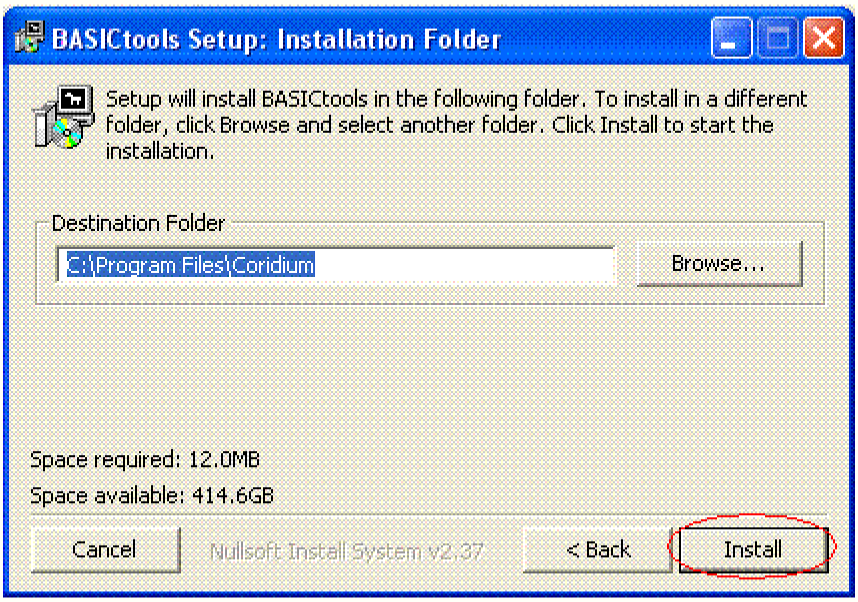
Step 1: Install Software

The **ARMbasic** compiler runs on the PC, in combination with a BASIC support library that is installed on the ARM. Coridium supplies BASICtools which includes a terminal emulator and IDE that is specifically designed to run BASIC on an ARM processor. 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 32 bit WIndows either NT, XP or XPx64 and Vista.

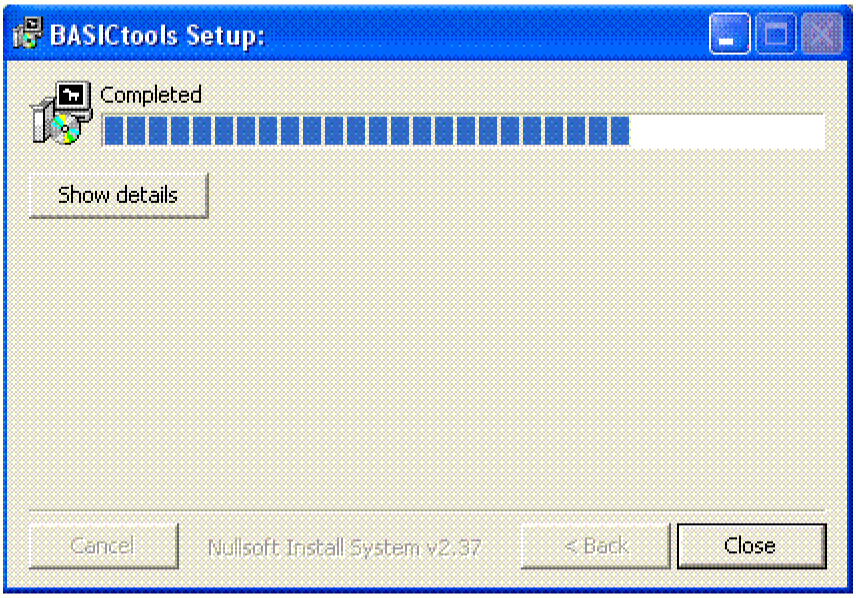
The software is downloaded from the web, and run as an installer SETUP program.



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.

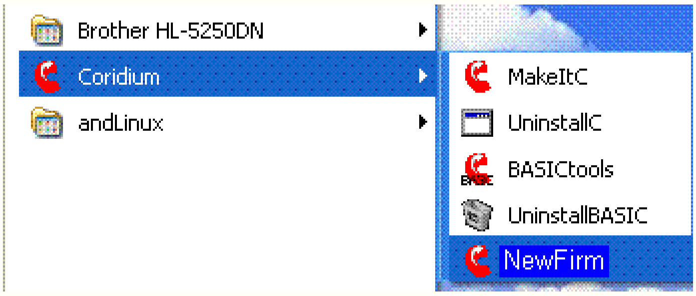
On to Step 2

This section does NOT apply to Coridium Products, it is for installing BASIC on boards from other vendors.

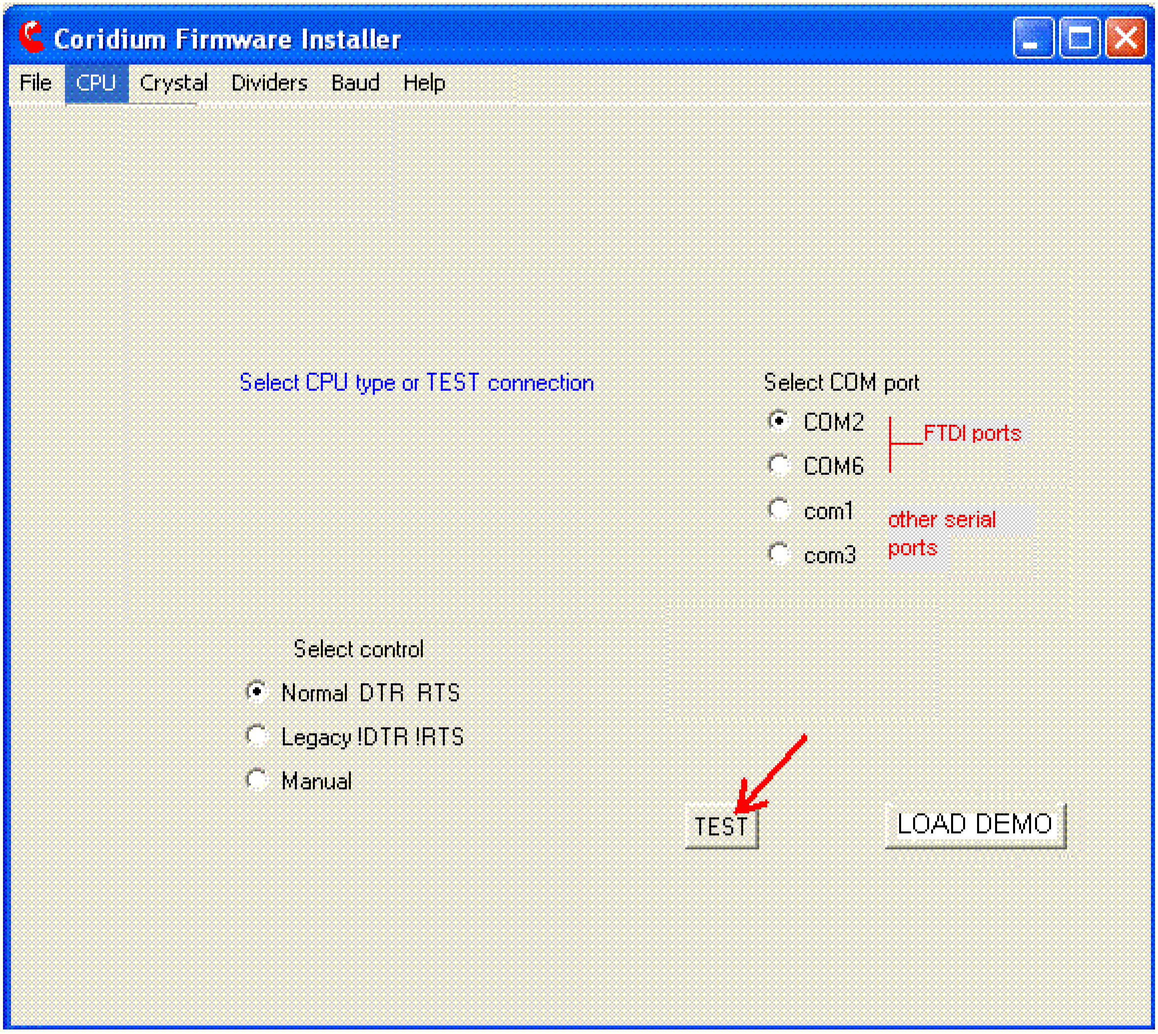
Step 2: Install DEMO Firmware, if you have purchased the compiler skip to step 3

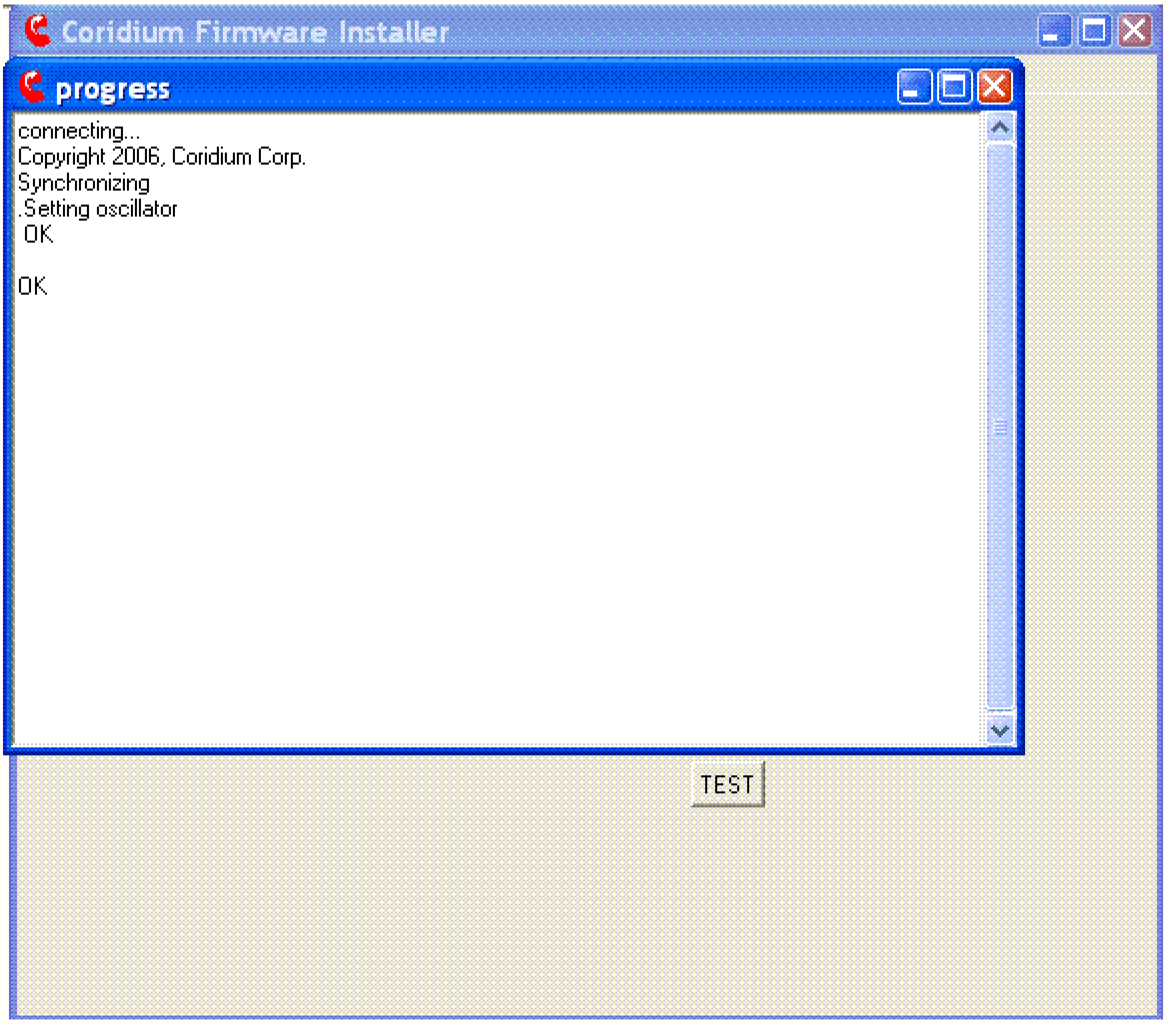
The **ARMbasic** compiler is freely downloaded, but the utility to install BASIC support libraries is locked to a PC. But we do support a DEMO mode that limits variables to 100 words and 4K of code. To install this firmware follow these steps.

The software installed in the previous step is NewFirm for the standalone **ARMbasic** compiler.



NewFirm 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, For Coridium style designed boards which use DTR for reset and RTS for boot, this can be selected by the Normal checkbox. For boards without those connections, you must Manually get the board into a ROM boot configuration. This is done by holding P0.14 low while asserting RESET. For instance on Olimex boards this is done by shorting the BSL jumper while pushing RST. On Futurelec boards, hold the LOAD button while pressing and releasing RESET.


Coridium Firmware Installer
File
CPU
Crystal
Dividers
Baud
Help
Select CPU type or TEST connection
Select COM port
COM2
COM6
com1
com3
FTDI ports
other serial ports
Select control
Normal DTR RTS
Legacy !DTR !RTS
Manual
TEST
LOAD DEMO



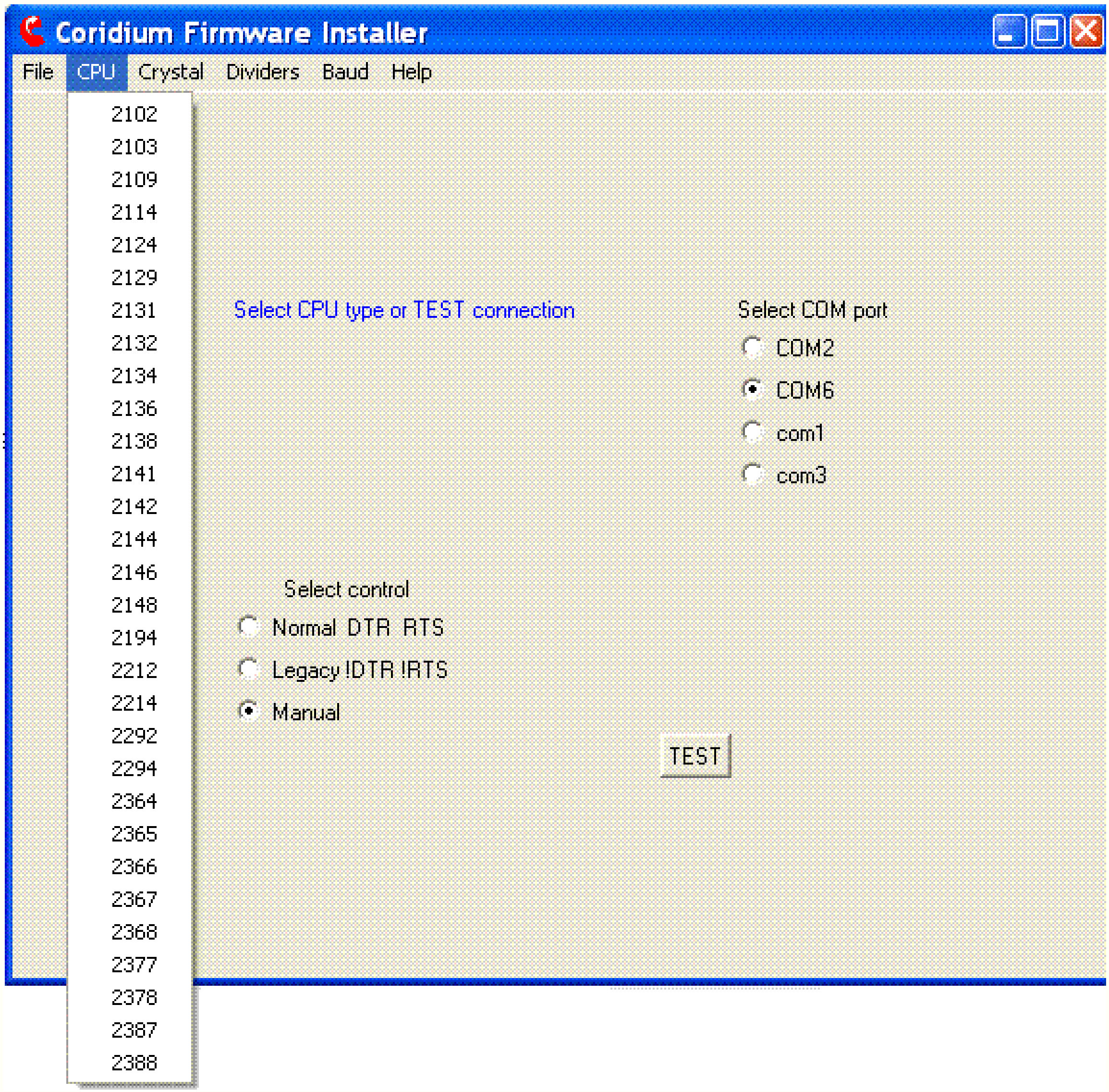
If this does not pass, then you **cannot** go on to the next step. You must verify your connections, choice of COM port, and whether you are driving P0.14 low while driving RST on the LCP2xxx low, and then releasing it. These would be the same steps you use to program any hex file with a program like FlashMagic. Refer to the documents that came with your PCB.
Once the TEST passes, you can load the DEMO code. Set the CPU and Crystal values. Then you can LOAD DEMO firmware.

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.

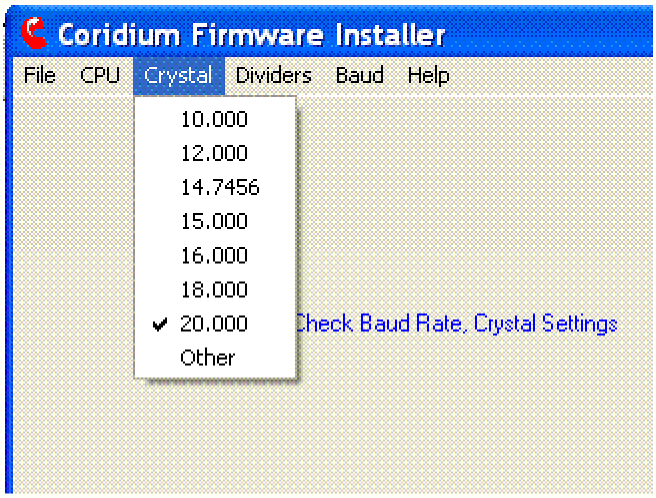
The NewFirm utility is also used to accomplish this. The first time you run this portion of the installation, a key will be required. This process is not yet automated, and requires you to get a key from Coridium. For details on that look at the **unlock pages** .

After passing the communication TEST, choose the CPU type -

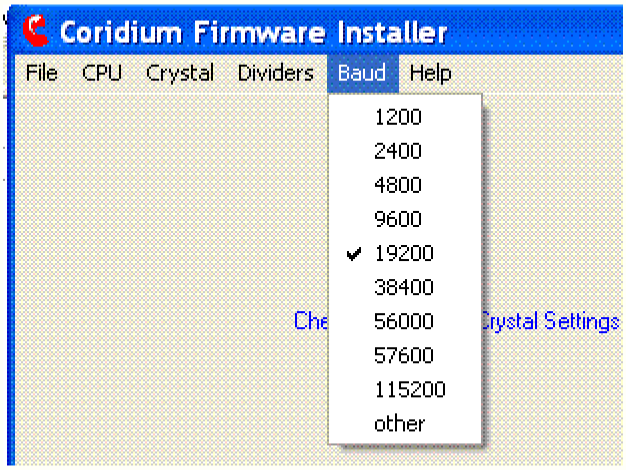


Once the CPU is chosen the TEST button will become an UPDATE.

Before doing the UPDATE, check the Crystal setting, for instance the Olimex board uses a 14.7456 MHz crystal.



You can also choose the default baud rate.

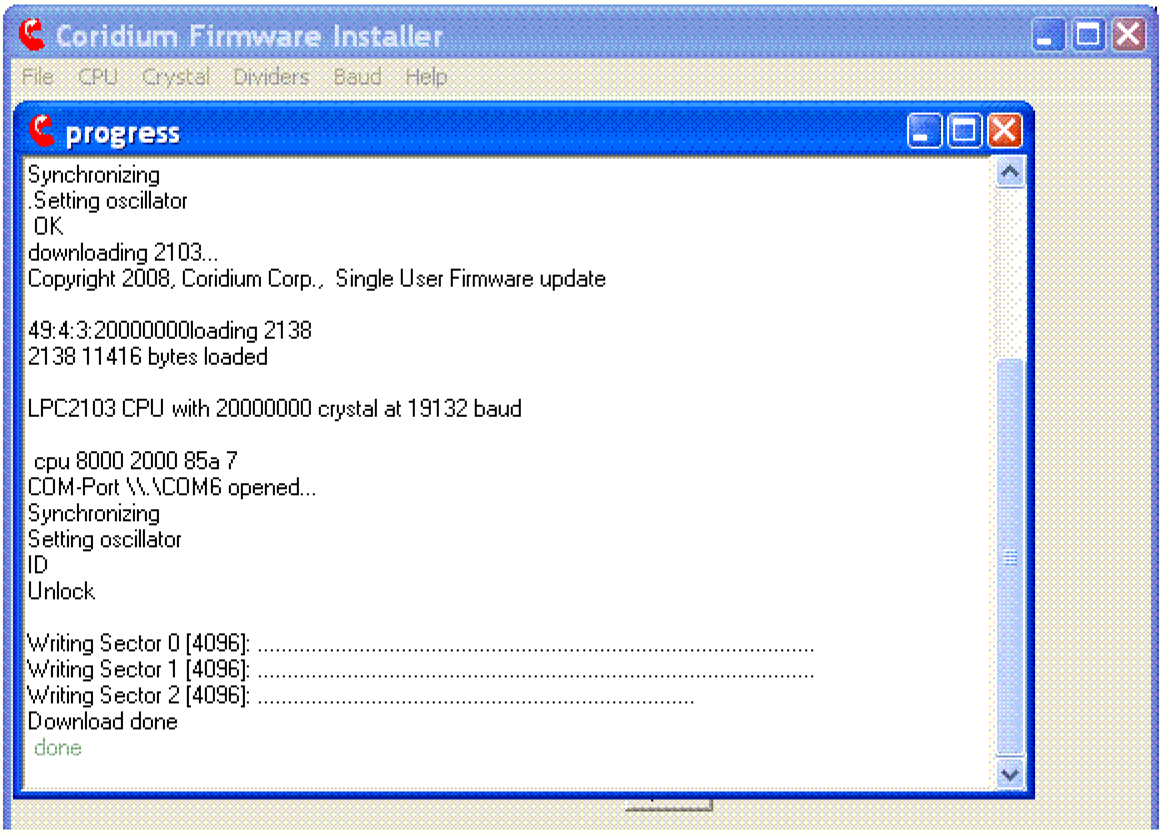


Remember, that if you change the baud rate here, you will need to set the baud rate in BASICtools, the default is 19.2Kb.

Now you are ready to place the Firmware on your PCB.

You can use the UPDATE option if you have purchased an ARMbasic firmware license, if not you can install the demo code.

Click UPDATE or click LOAD DEMO



Assuming all was connected correctly, you will see something like above, and you are now ready to start writing **ARMbasic** programs. This is the last time you will need to run the NewFirm program, as the portion of the Flash that contains your program will be maintained by the BASIC program.

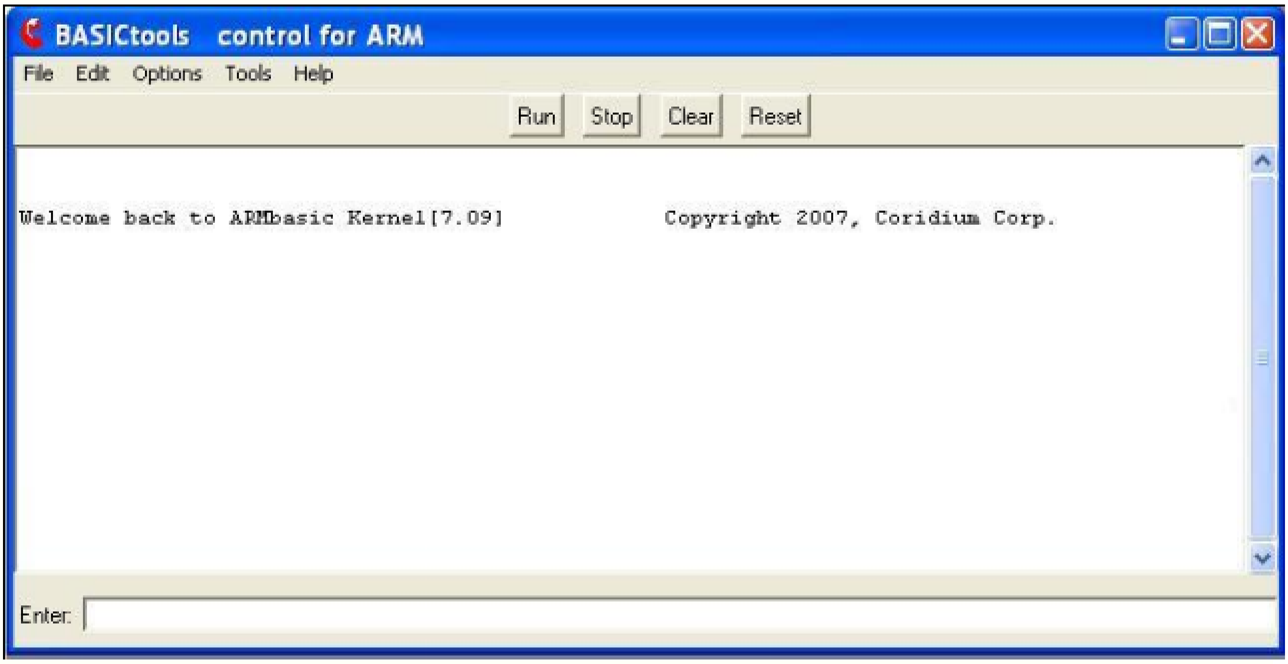
Remove any BOOT jumpers, and press RESET, which will now launch the **ARMbasic** runtime monitor running on your PCB.

If you bought an ARMbasic compiler, continue to installing full firmware.

If you are just running the demo code, continue to write your first program.

Step 3: Writing your first Program with BASICtools

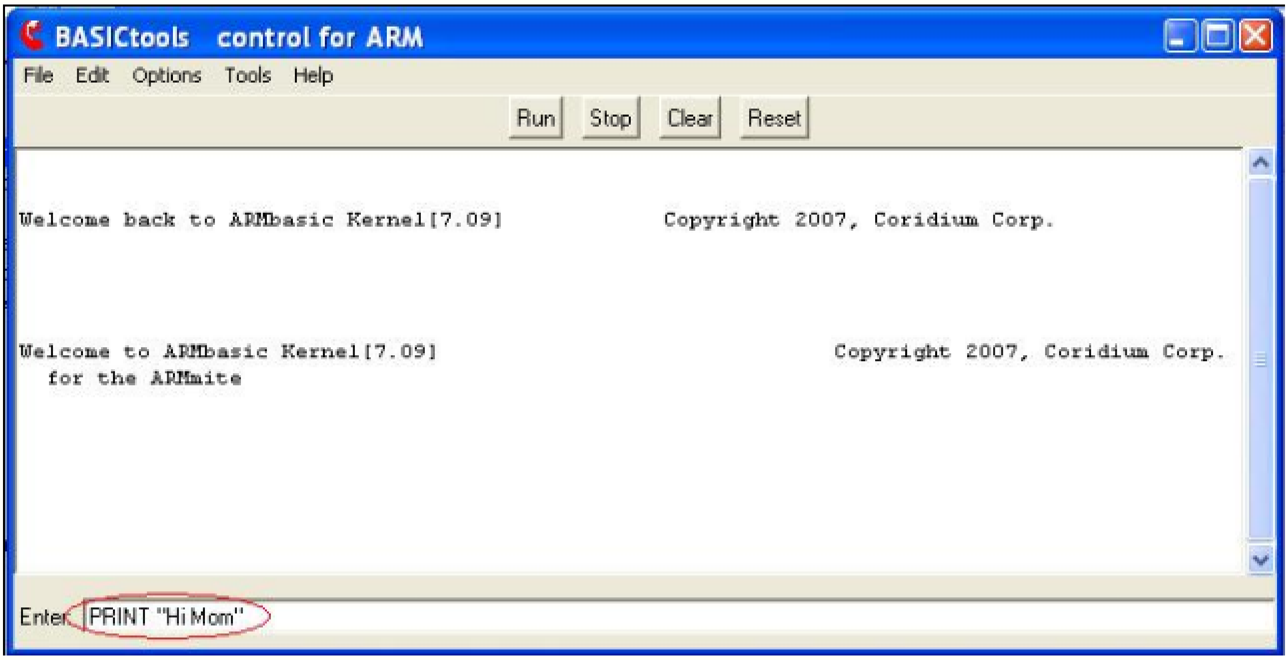
Start the BASICtools from the StartMenu or from the Desktop Icon. You should see a welcome message which has been sent from the ARMmite or ARMexpress-



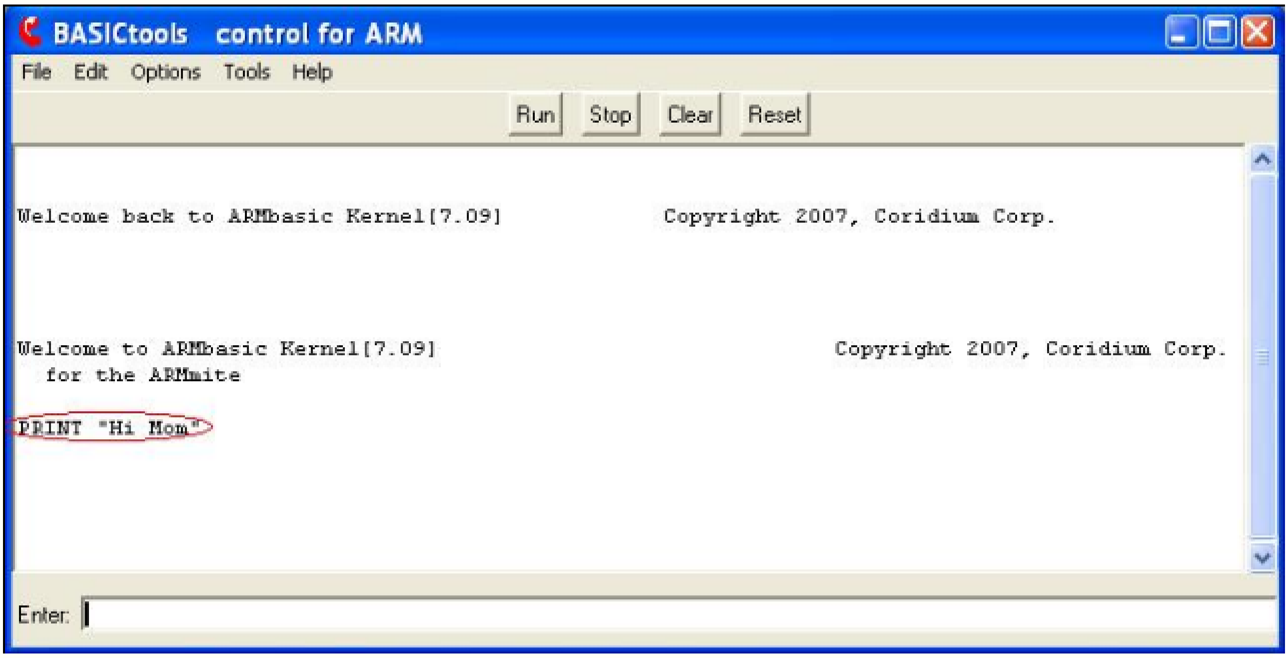
If you do not see this welcome, even after pushing the RESET button, then communication has not been established.

- 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**

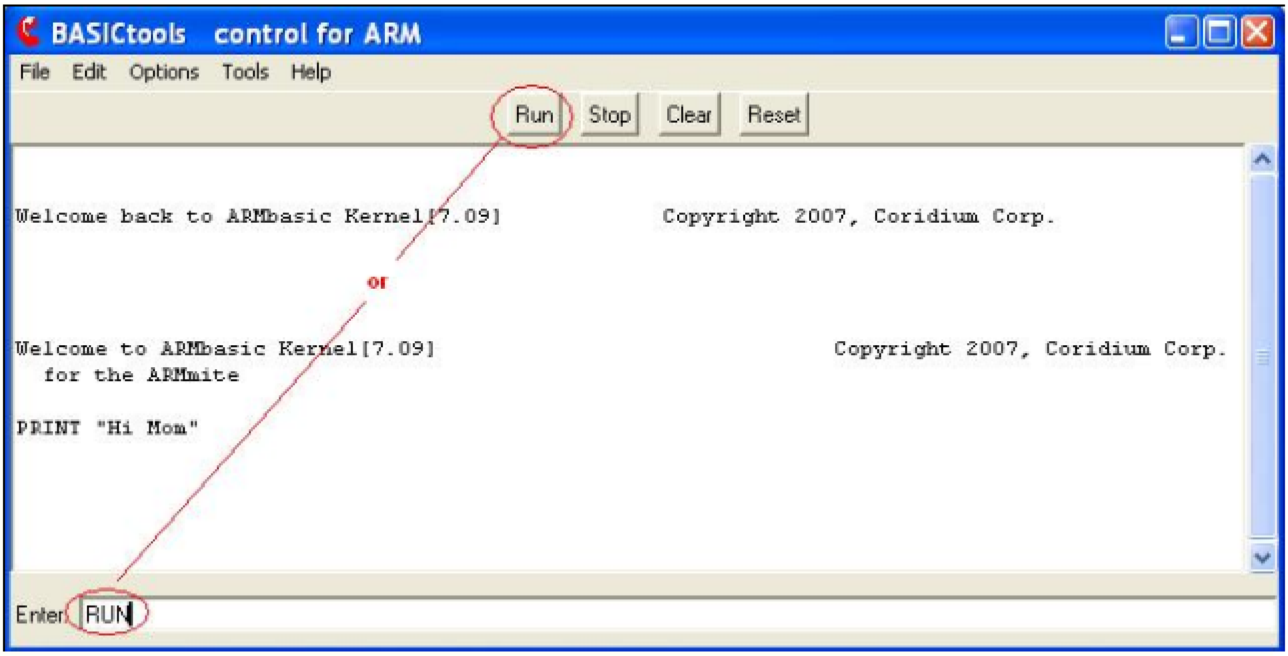
<u>The traditional "Hi Mom" program</u>



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

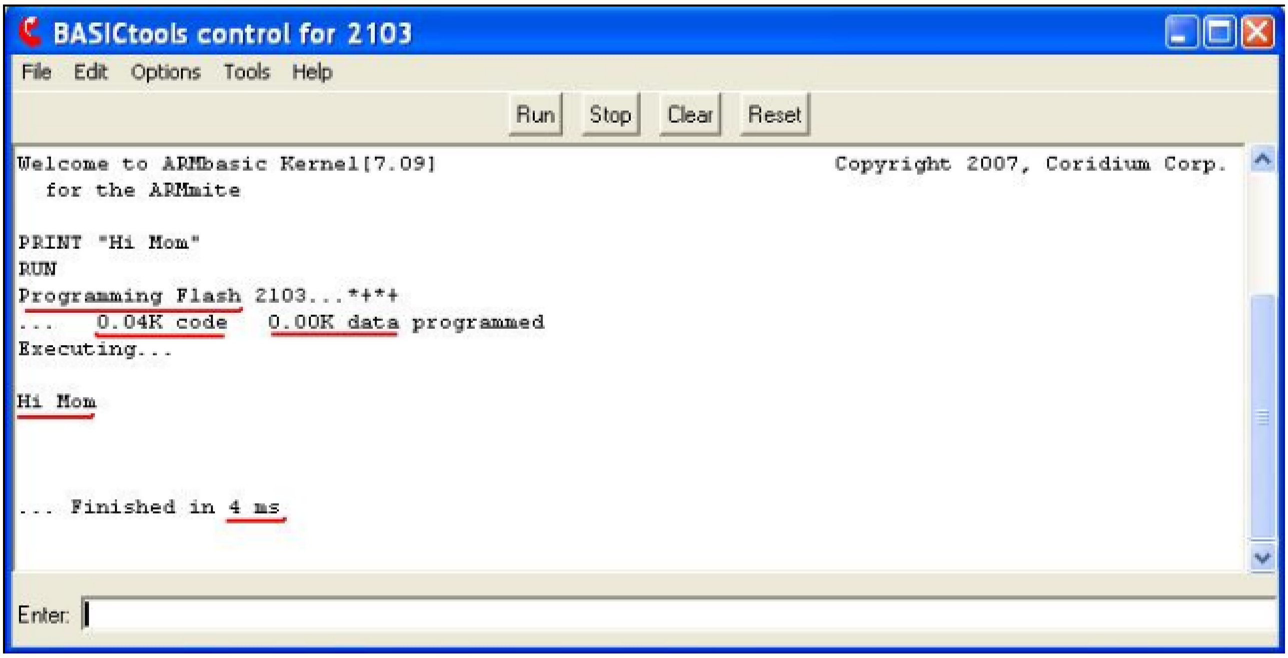


Now RUN the program



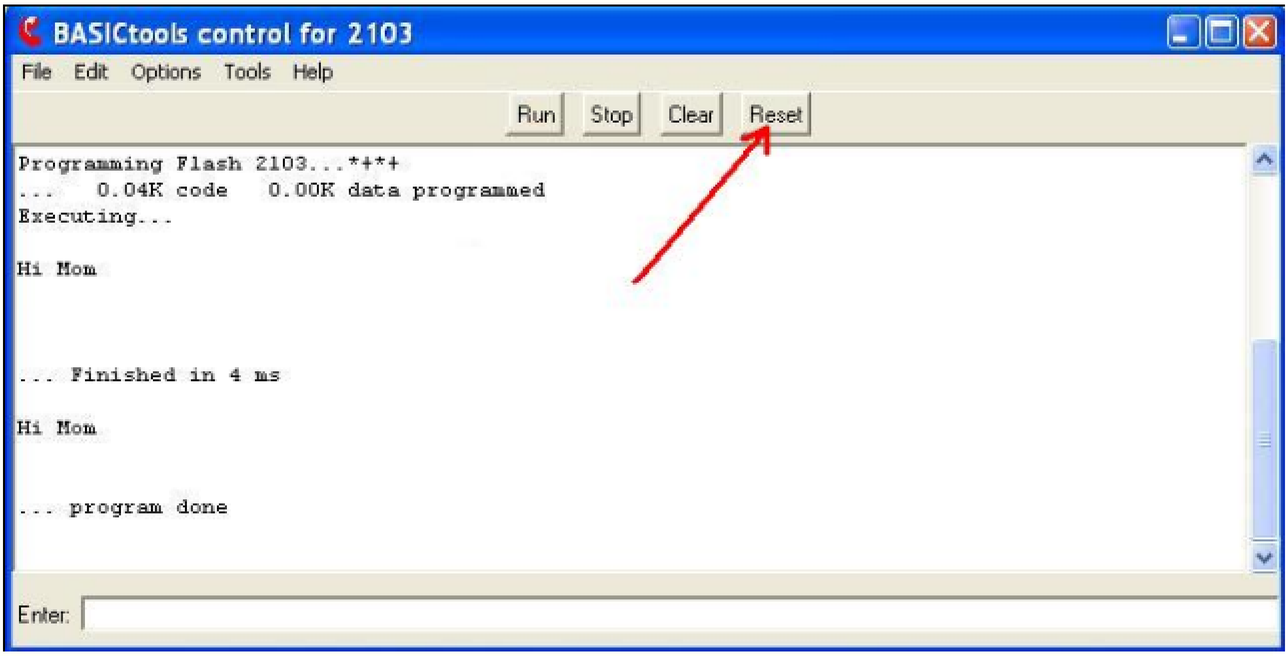
Which you can do by either typing RUN or hitting the RUN button at the top of the screen.

<u>And see the results</u>



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 Mom" to the console. ARMexpress also reports back how long the program executed, in this case 4 msec, which is mostly startup time.

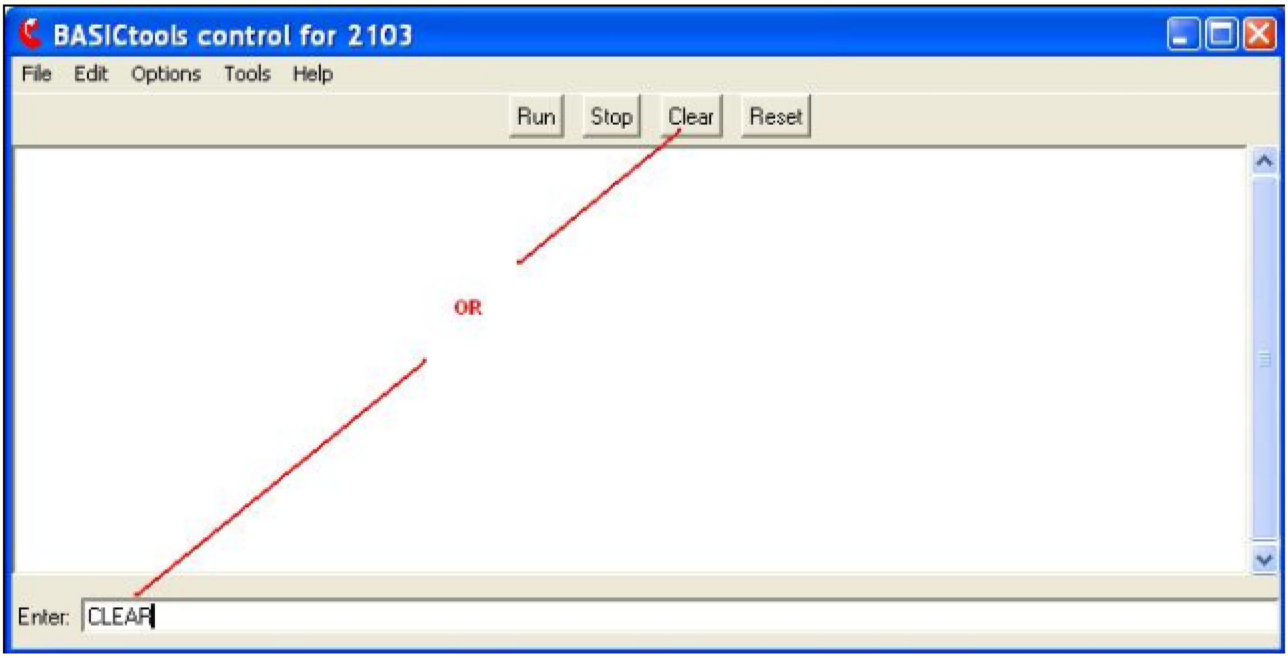
Also your program is now saved in the ARMmite/express Flash memory. And it will be executed the next time the board is RESET. So try that...


BASICtools control for 2103
File Edit Options Tools Help
Run
Stop
Clear
Reset
Programming Flash 2103...*+*+
... 0.04K code 0.00K data programmed
Executing...
Hi Mom
... Finished in 4 ms
Hi Mom
... program done
Enter:

On to Step 4

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

Type the following program in the console window. (below -- assuming Olimex 2106 proto board, an LED is connected to IO(12) on the Olimex, IO(15) on many Coridium boards).

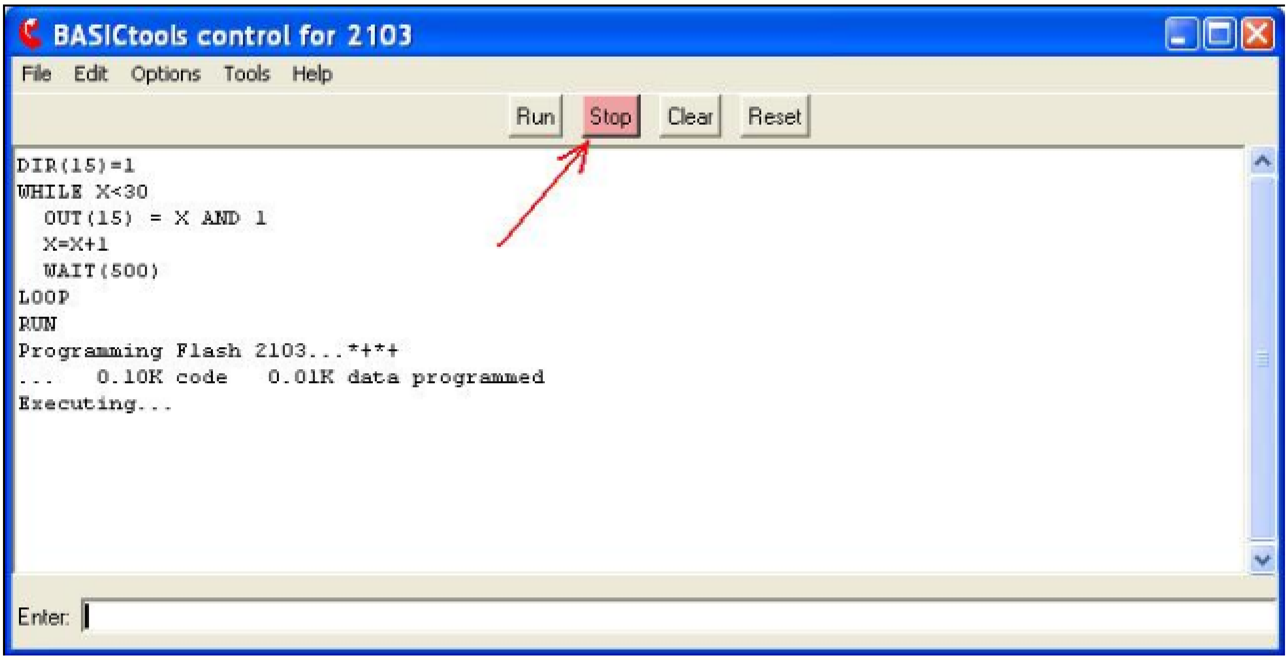
```
DIR(12)= 1                ' enable pin 12 as an output
WHILE X<30
  OUT(12) = X AND 1       ' drive pin 15 high when x is odd, low when x is even
  X=X+1
  WAIT(500)
LOOP
```

Now RUN the program

The LED on the PCB should pulse 15 times.

You can allow the program to finish or --

<u>Stop the program</u>



To stop a running program simply press the Stop button.

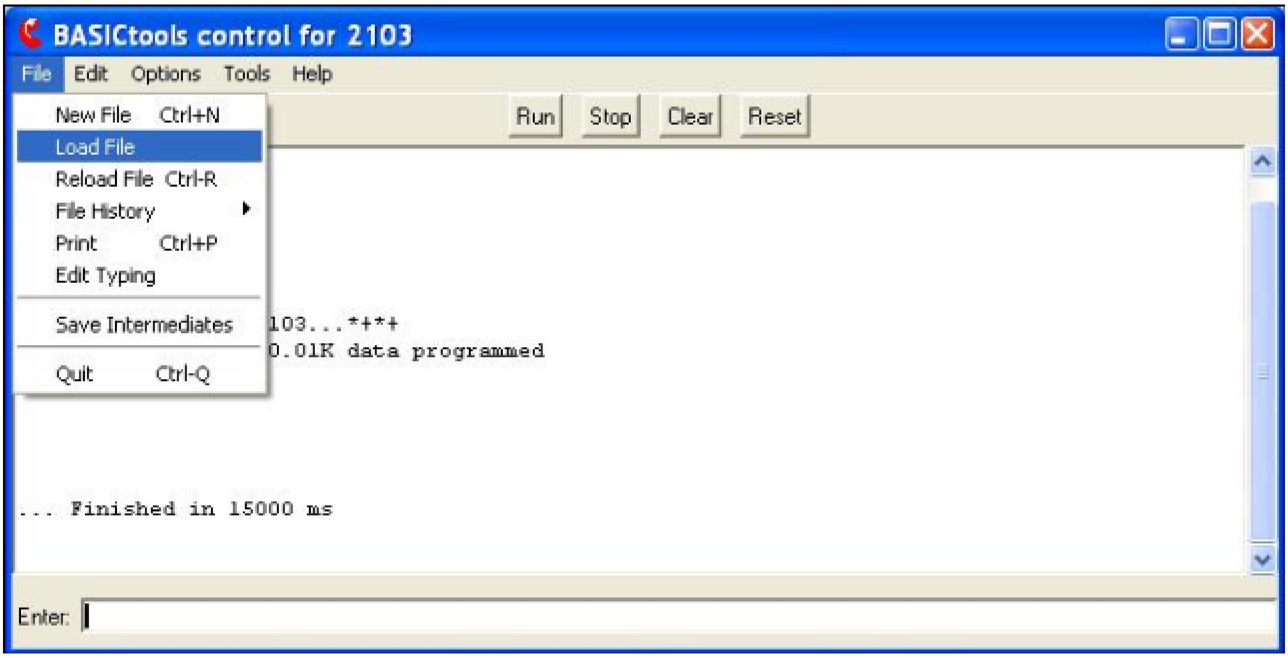
On to Step 5

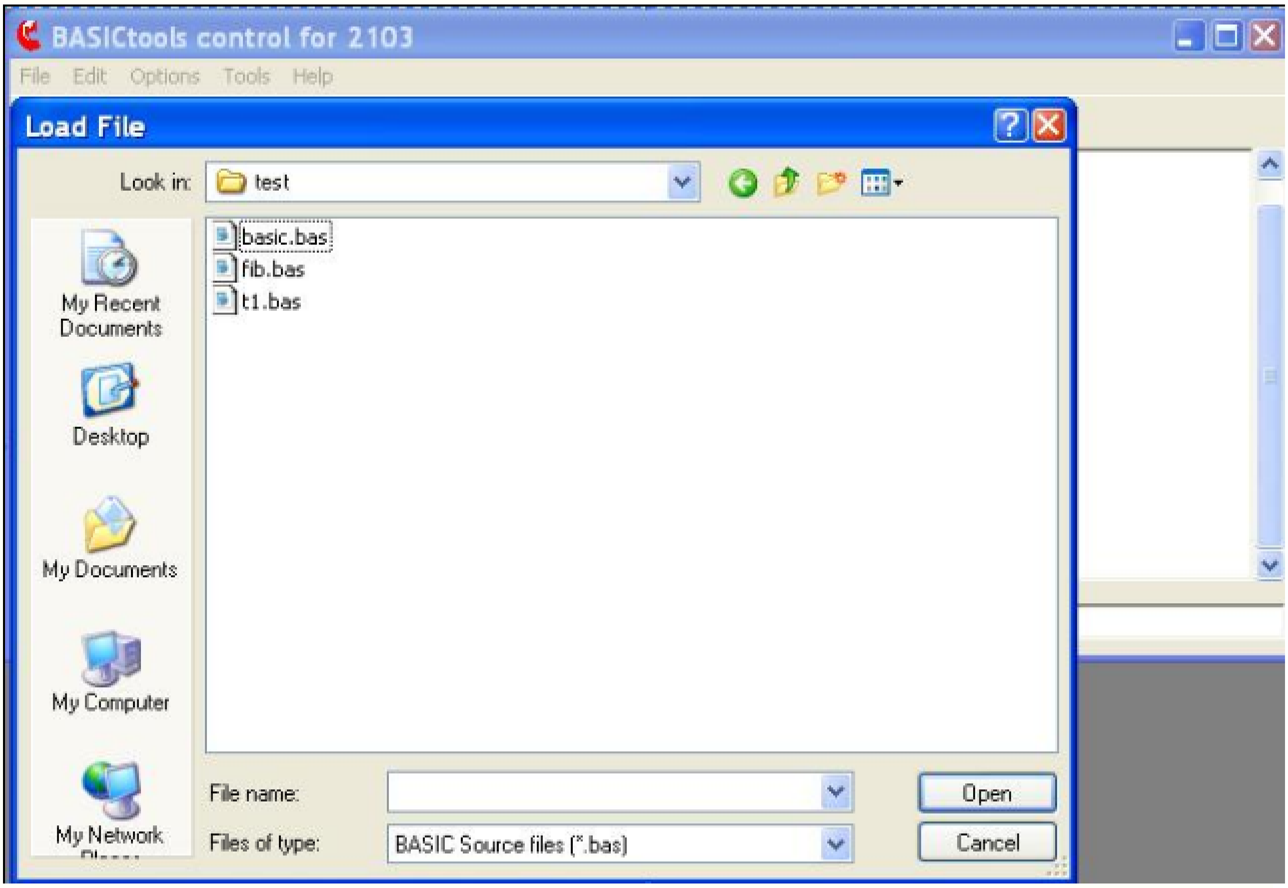
Step 5: More Complex Programming

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 **Yahoo Forum** for files and help from other users of ARMbasic products. There are also examples on the **Coridium Website Programming pages.**

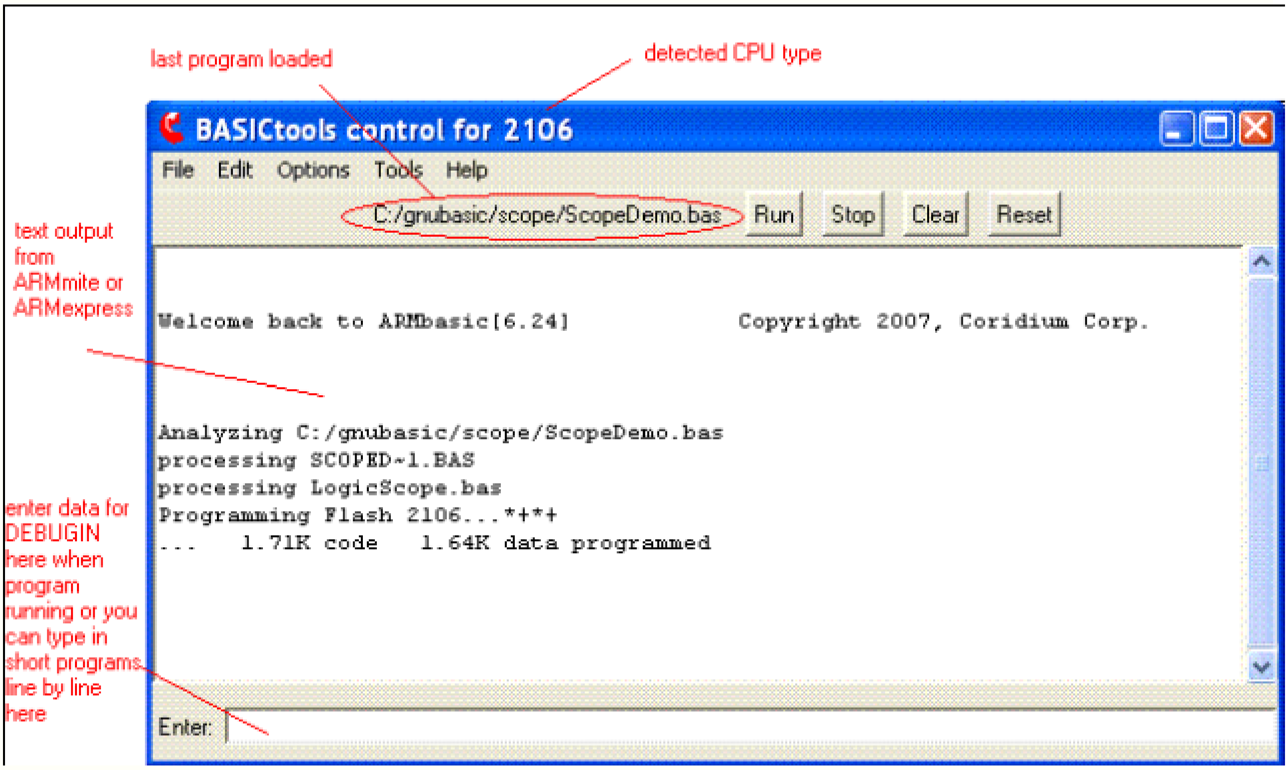
For more details on the BASICtools IDE check the **next page**.

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, close BASICtools and then restart it (you may need to use the Task Manager to exit). It will STOP your spewing program.

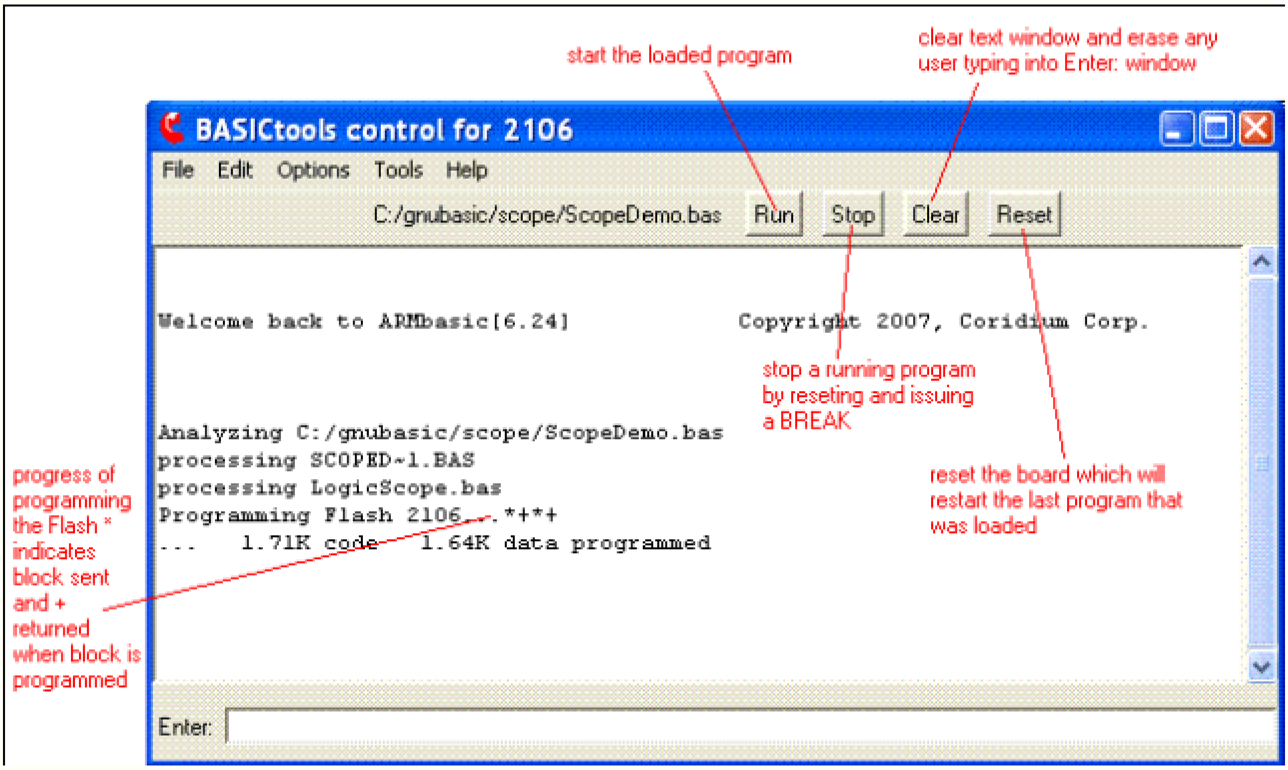
BASICtools Layout



keyw ords: enter line debugin type BASIC commands

Buttons

..



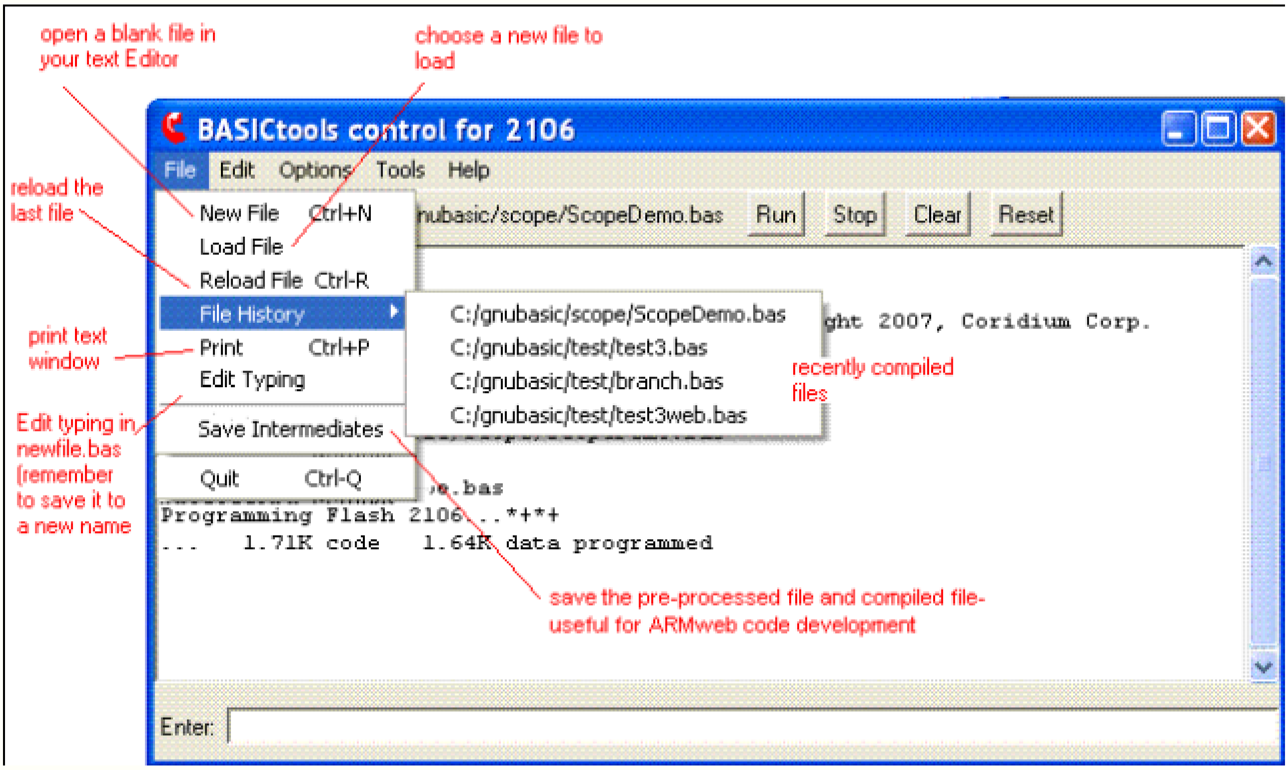
The CLEAR button only erases the display screen and the buffer on the PC of statements you have typed into the Enter window.

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

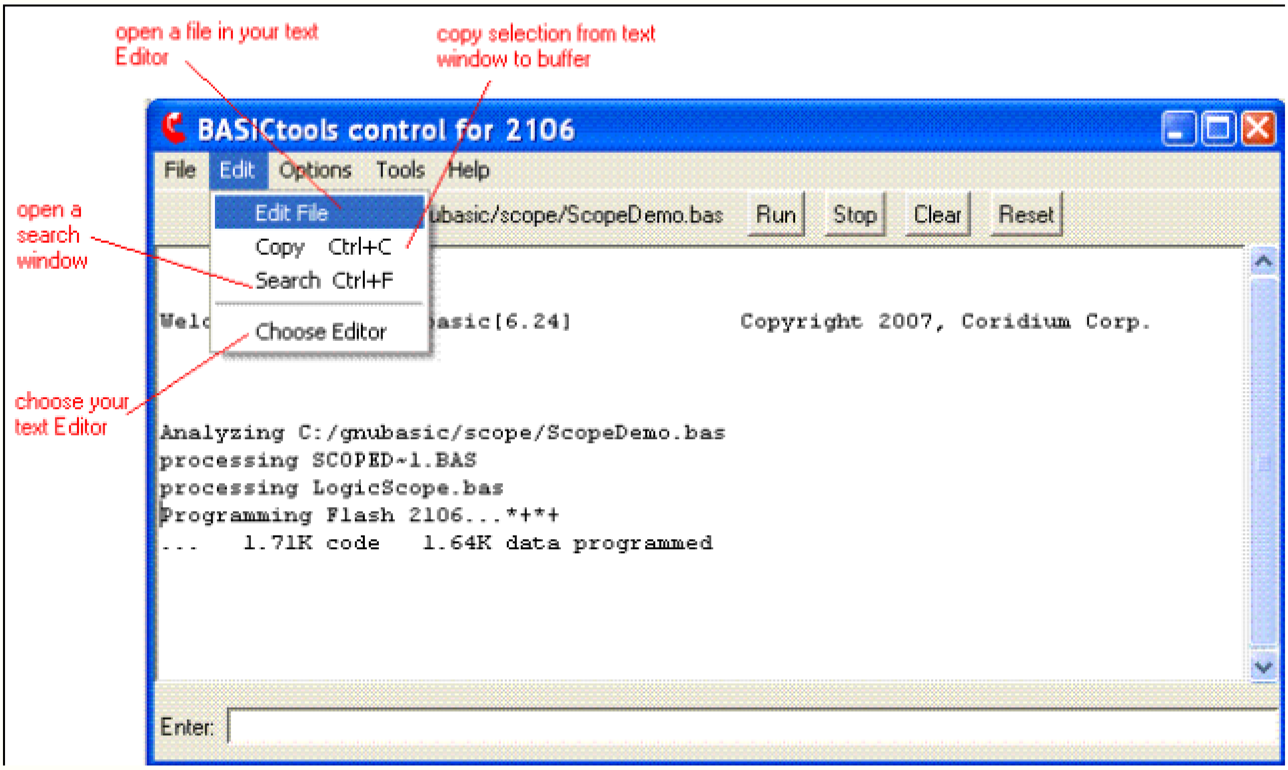
..



file load file reload file print save file quit

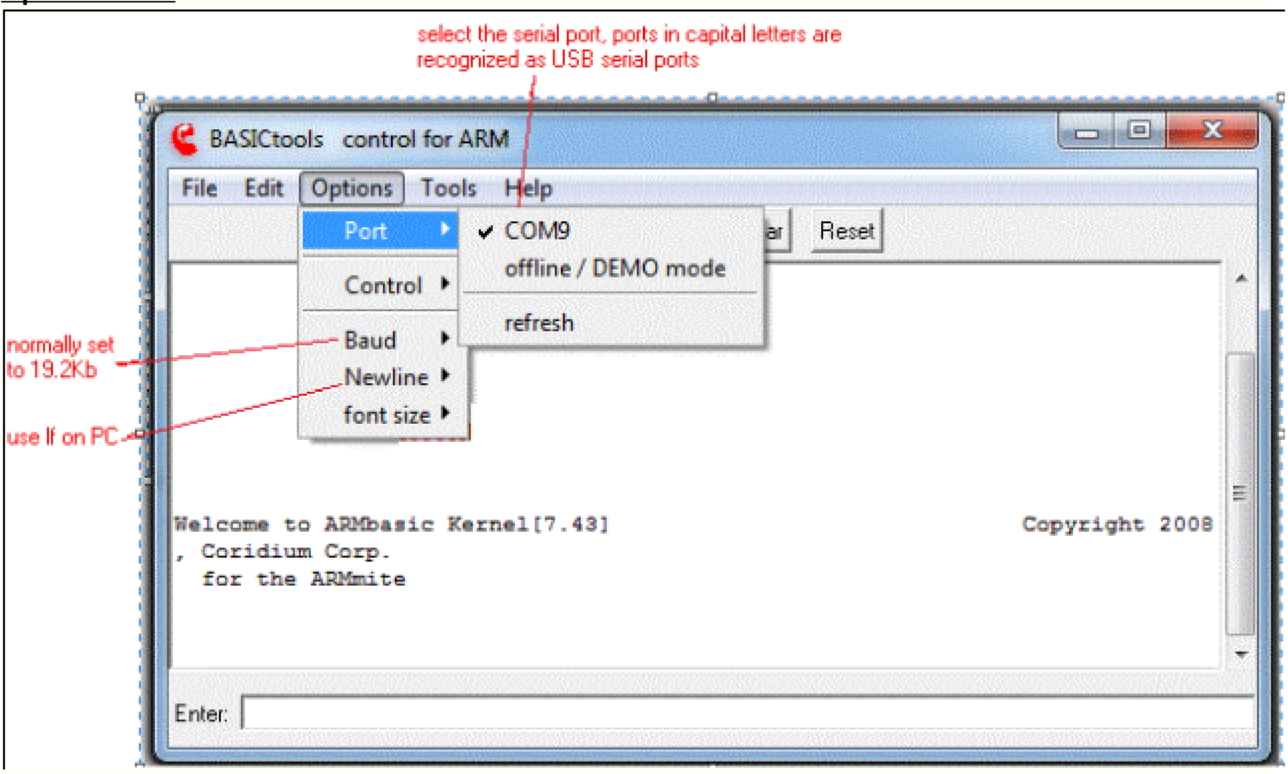
Edit Menu

..



keyw ords: edit choose editor

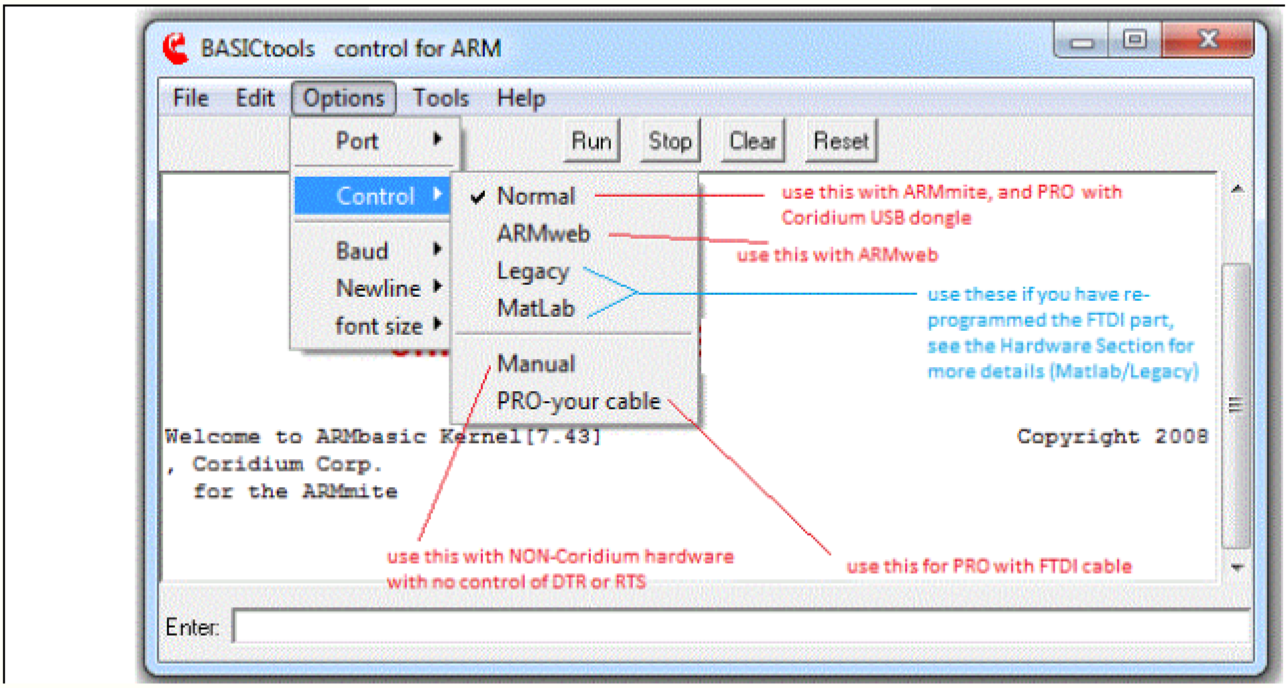
Options Menu



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

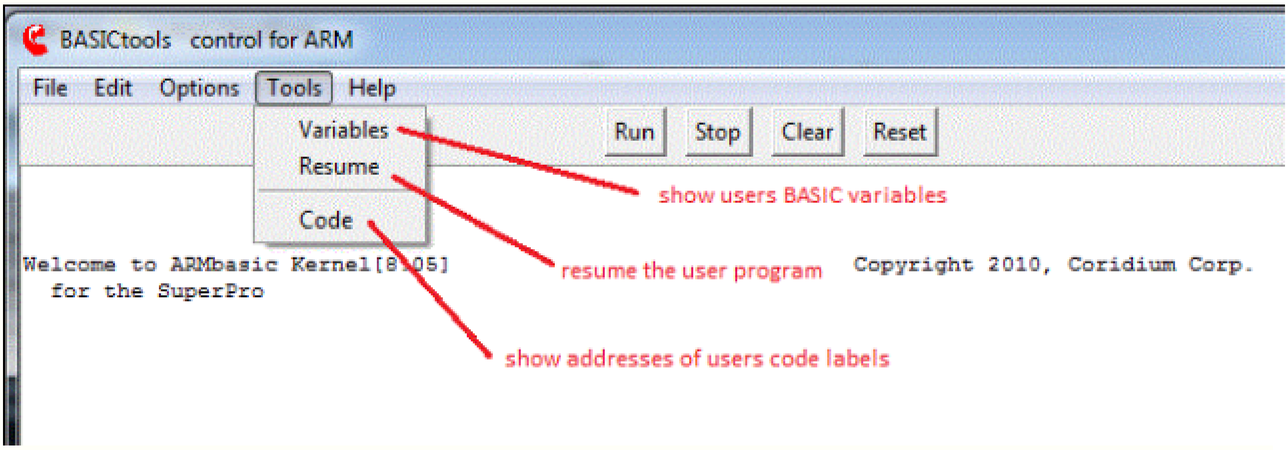
Control Menu



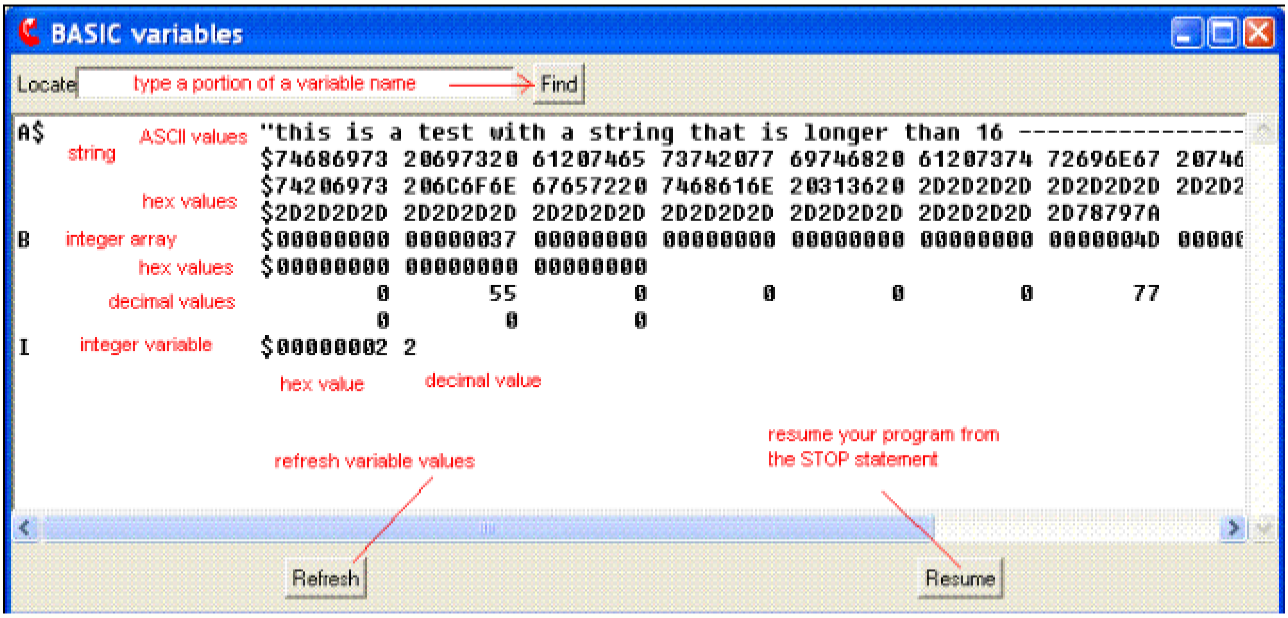
keyw ords: options port baud new line char mode PC compile control throttle

BASIC variable viewer

Open this window from the Tools Menu (variables)

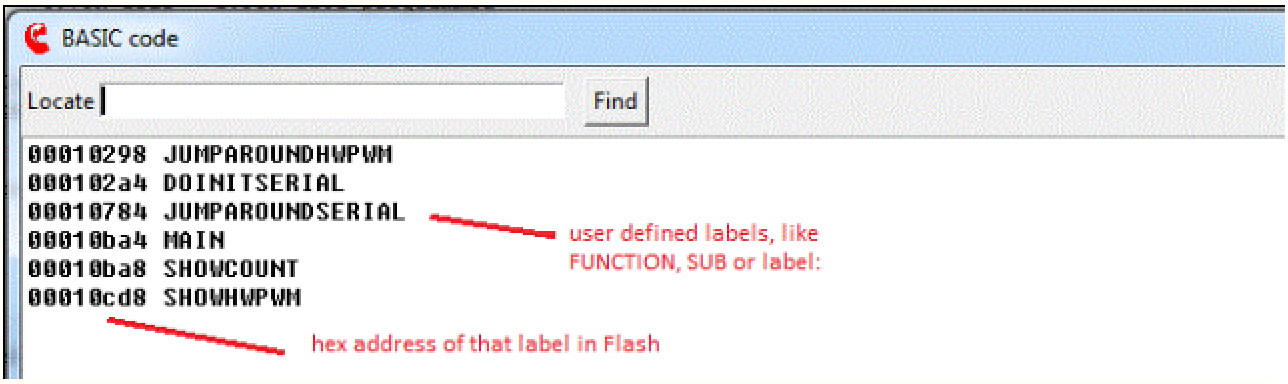


variables window



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

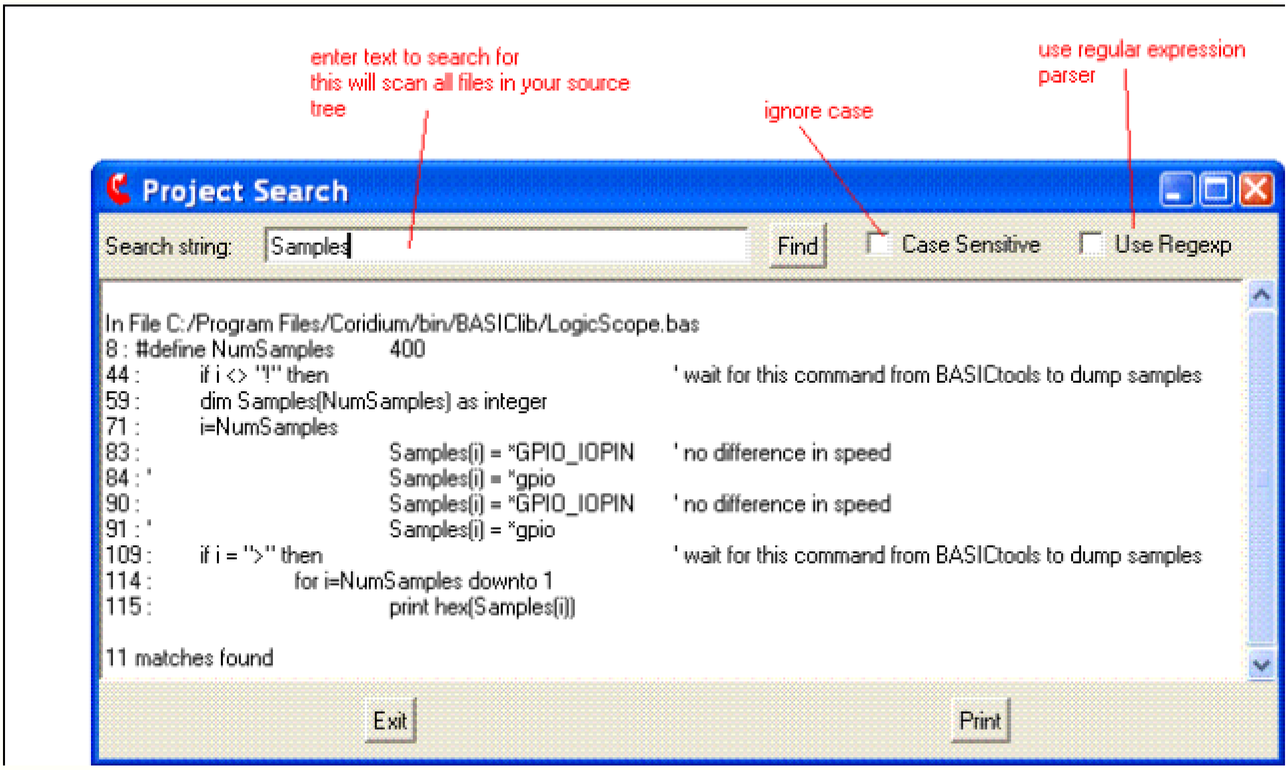
code window



keyw ords: variable dump breakpoint STOP view memory

Search Window

Open this window from the Edit Menu



keyw ords: search

Logic Scope Window

This module must be included in your BASIC program. It will monitor the pins for a period of time when called from your program.

See the example program ScopeDemo.bas and details in the **Logic Scope Section** .

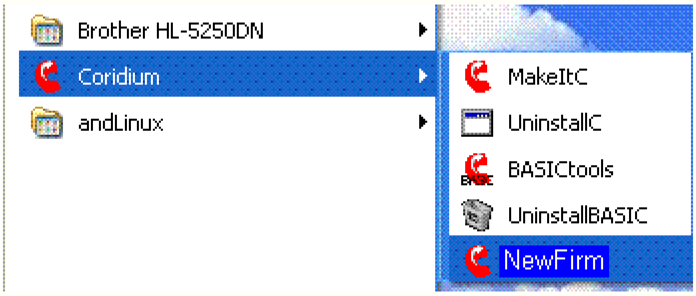
keyw ords: oscilloscope logic analyzer logic scope

This section does NOT apply to Coridium Hardware Products, it is for installing BASIC on boards from other vendors.

Writing ARMbasic Firmware

The **ARMbasic** compiler is freely downloaded and there is a demo version of firmware freely available, but the to install the full BASIC a special NewFirm utility has to be purchased from Coridium.

The software installed in the previous step is NewFirm for the standalone **ARMbasic** compiler.

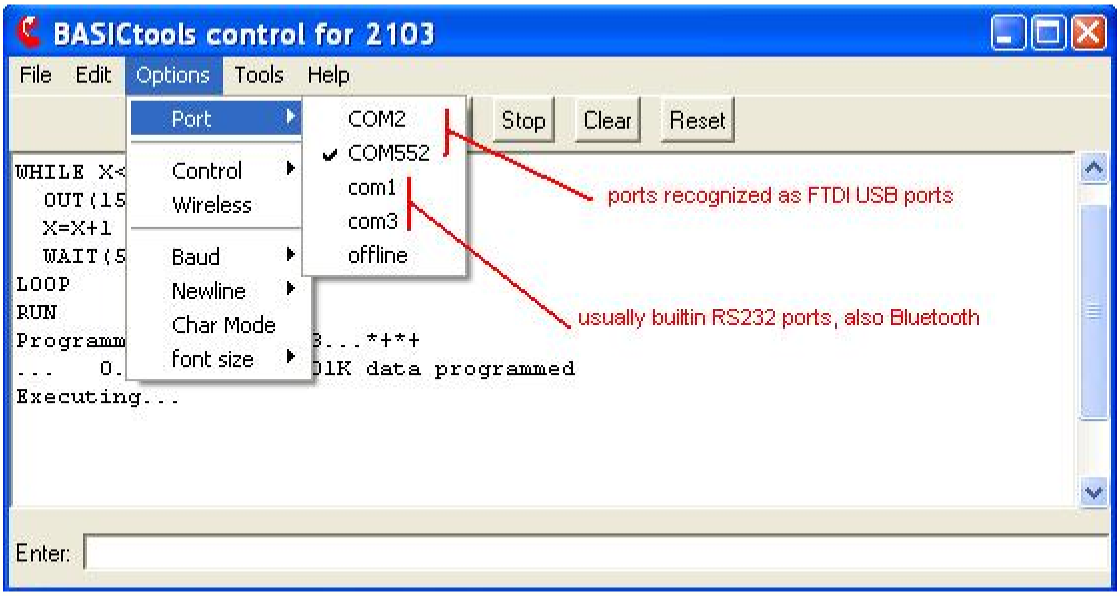


A specific version of the NewFirm has been built for you. This utilty does require a network connection, and it is limited to 5 installs for the single user license, and 100 installations for the commercial license. Larger licenses are available, contact the Coridium Sales Department.

On to Step 3

Trouble Shooting

Reset Target PCB shows no message

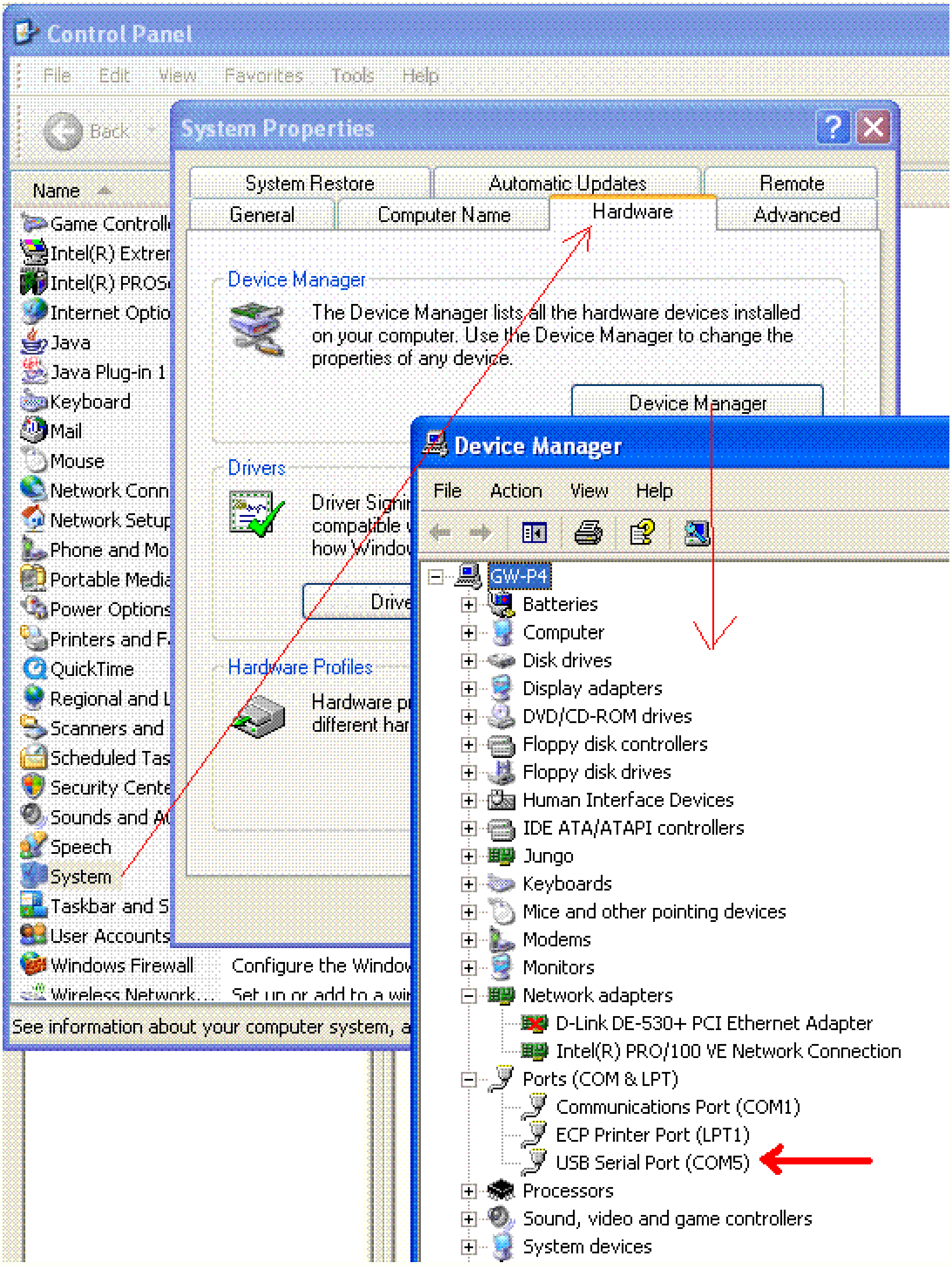


Most PCs have a number of COM ports, you might not have the correct port selected, you can change that in the Options>Port Menu This window lists all the available ports, those in capital letters are recognized as FTDI USB serial ports and are usually the location of the ARMexpress Eval PCB or the ARMmite.

One other reason that communication could be lost, is that the driver can lose sync with the card if it is disconnected and reconnected with the USB, especially when BASICtools or TclTerm (under MakeItC) is running and connected to the card. When this happens it is often necessary to restart the PC. Because the serial port is being emulated, and the Windows enumerator gets involved, when the USB is disconnected, the various pieces of software can get confused if the port is open. If you are using the original hardware serial port, normally with COM1 this is not an issue.

Determining which COM port should be used

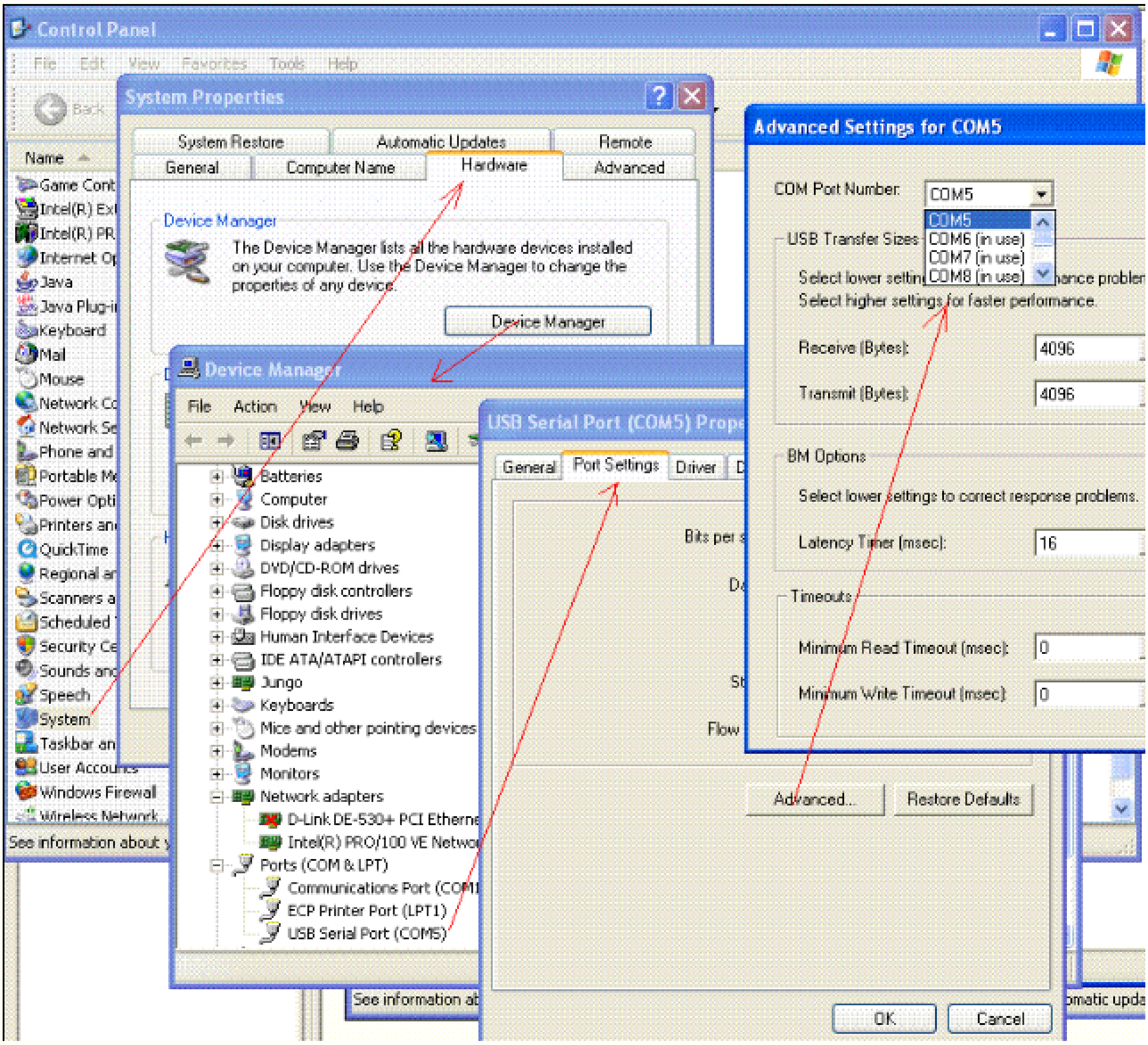
This can be found in the Control Panel>System>Device Manager



COM port conflicts

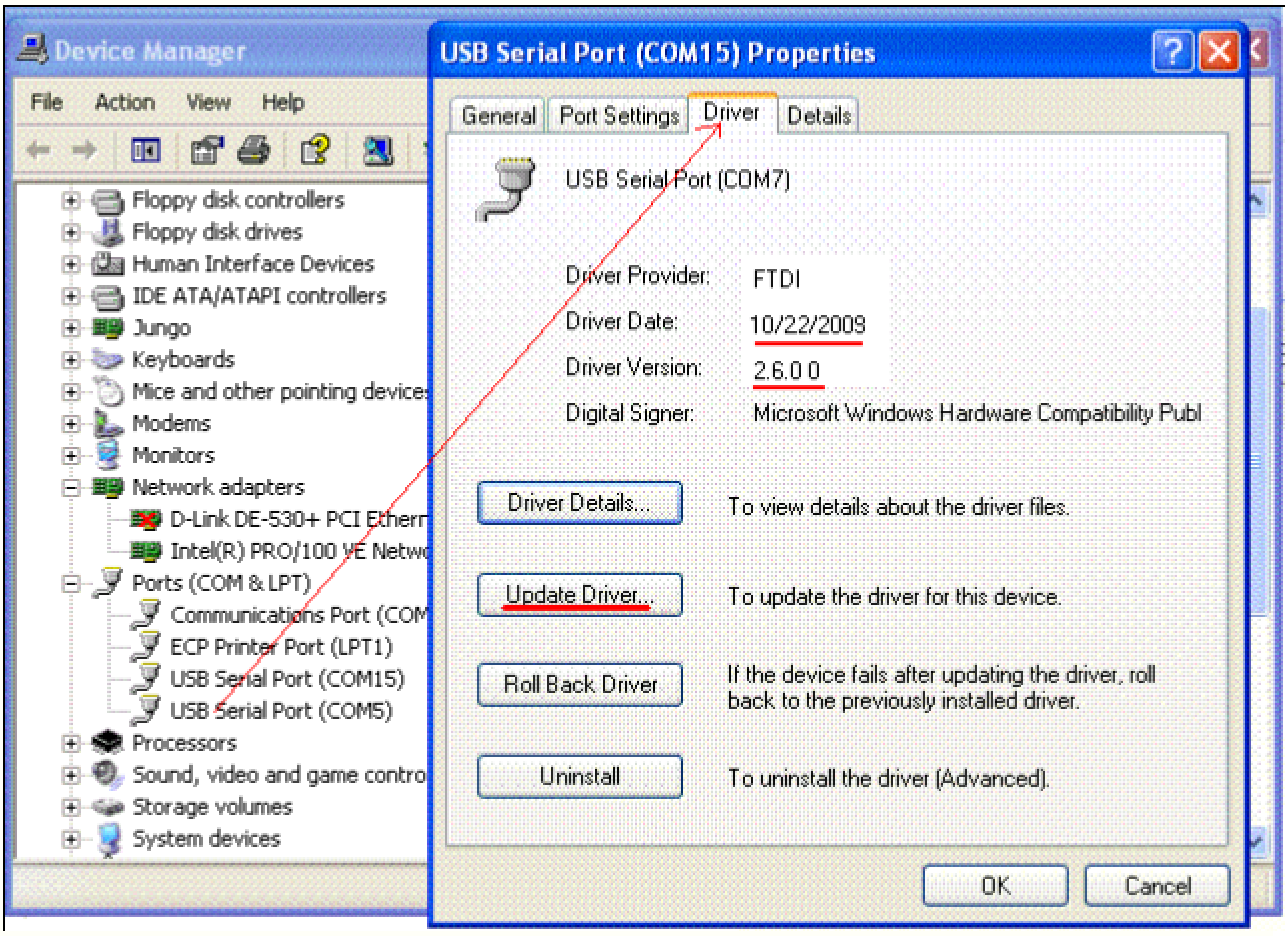
While rare there are systems out there with non-plug and play serial ports, or its possible for 2 com ports to have the same address. The address can be changed from the Control Panel.

Control Panel> System> Hardware> Device Manager> Ports> Port Settings> Advanced



Check the USB Driver version

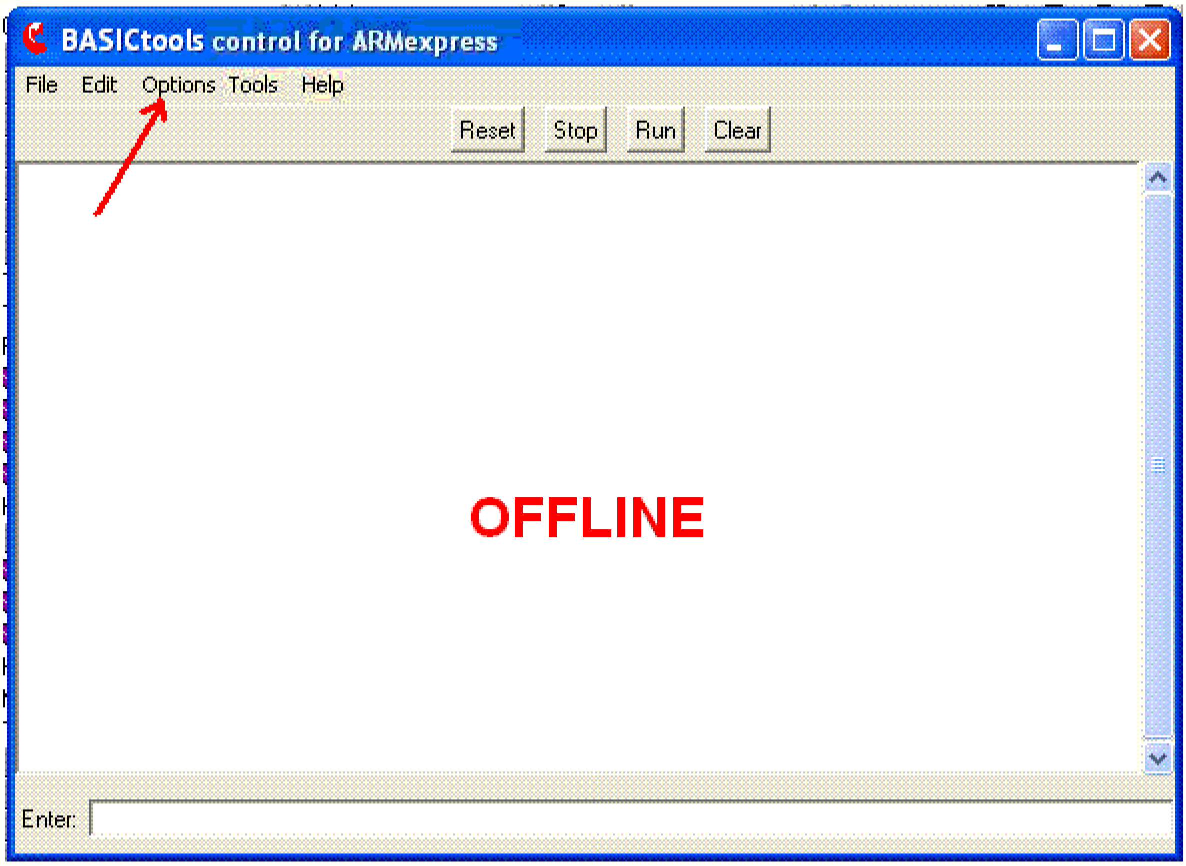
Our software does not reinstall the USB drivers if they already existed. We expect to be running version 2.4.6.0 dated 3/13/2008 (for XP). Find this in the Control panel>Driver properties



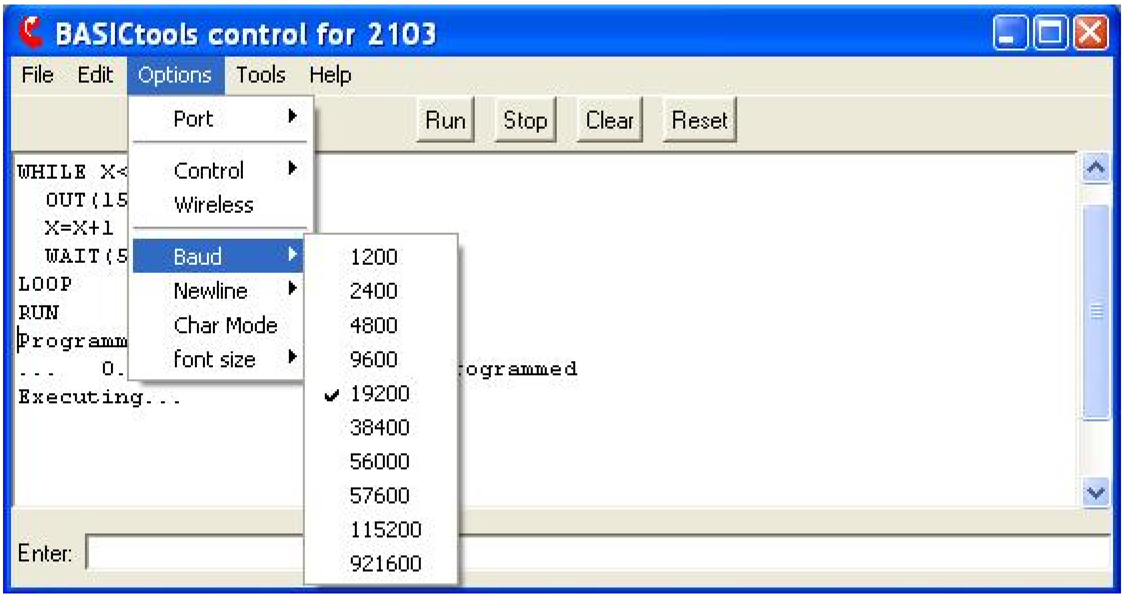
If this does not match, then you have an older driver and it should be updated...

<u>Offline indicator</u>

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 ARMmite or ARMexpress.



Check Baud Rate

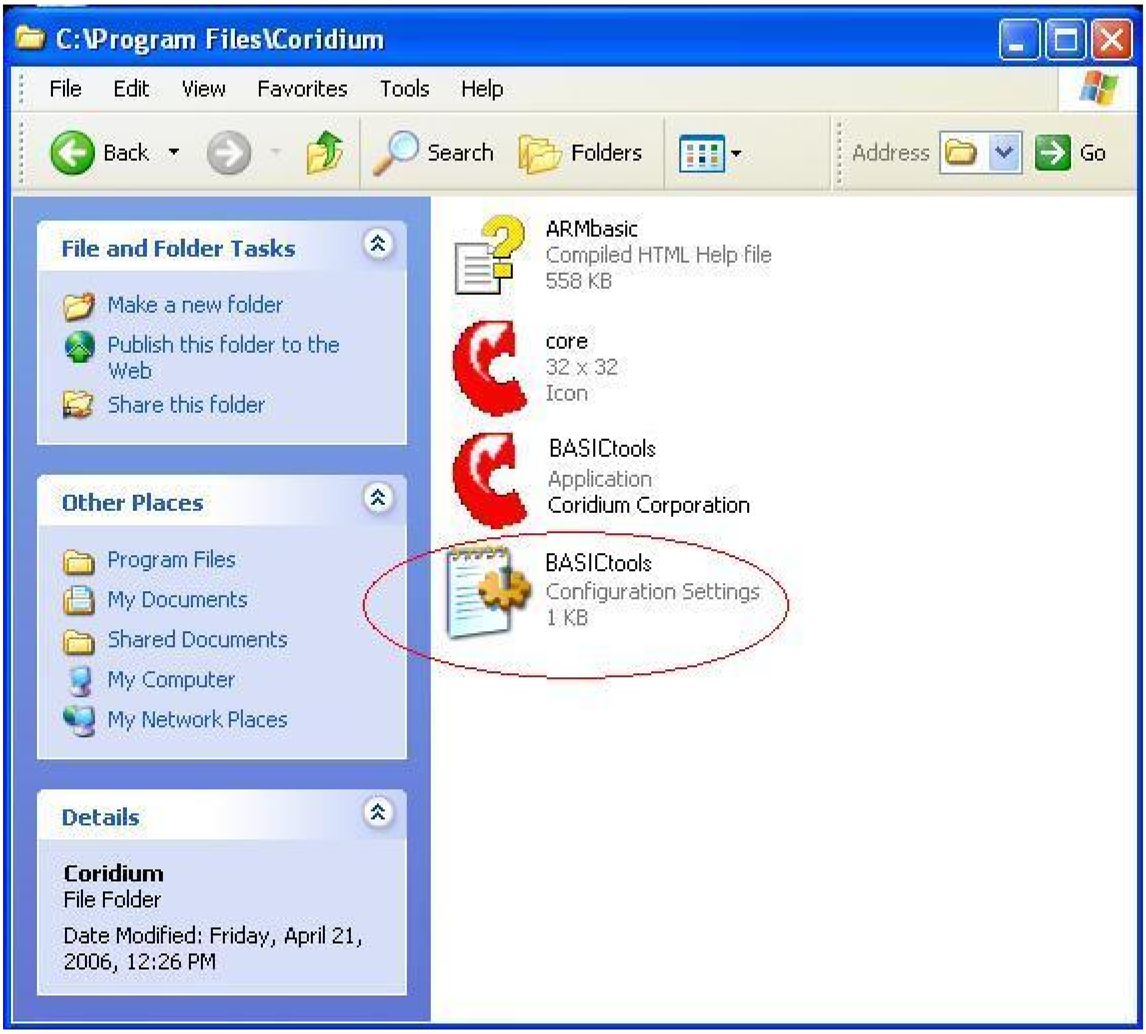


Or you might not have the correct baud rate selected.

Check your cables

Check the serial connection to your PCB.

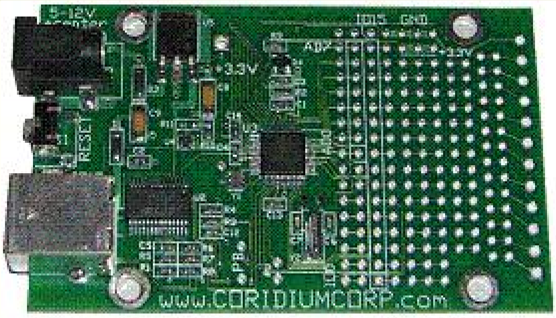
Odd behavior following Windows Update



In rare cases, when the Windows Update has automatically rebooted while BASICtools was running, the serial port settings of BASICtools have been corrupted. To correct this, reboot the system, and if the problem persists delete the BASICtools configuration settings (BASICtools.ini, it will be regenerated when you run BASICtools). This file is located in the %AppData%\Coridium directory.

Have Fun!!

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 such as that found in the ARMexpress PCB, which is pin compatable with other DIP24 modules such as the Parallax BASICstamp.

ARMbasic is simple to use, and runs totally on the ARMexpress or from the PC for the ARMmite, and both can programmed from a serial port. 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 and PBASIC, **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 Compatability

- **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
- HIGH and LOW control pin output values
- I2C on any of the 15 pin pairs
- SPI using any group of 2/3 pins
- HWPWM on ARMmite/ARMweb
- Software PWM on any pin with 256 levels
- FREQOUT on any pin upto 50 KHz
- 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 upto 115Kbaud
- RCTIME will measure a capacitive delay

Support for 32-bit variables and Strings

- Integer: (32-bit math)
- 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 ARMexpress 48KB is available for user programs, and an additional 8KB is available for DATA constants and constant strings. 4KB of this space (overlays DATA area) can be written into Flash and functions as non-volatile memory. Note that Flash may be written a minimum of 100K times.
- On the ARMmite 19KB is available for user programs, and 1KB shared for DATA constants (256 max) and constant strings.

Direct Hardware Access

- Uses the same syntax as C-pointers

Debugging support

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

Included with any module

- The **ARMmite** and **ARMexpress** compile their programs on the PC and they are downloaded using BASICtools, that compiler is part of the utilities available on CD or download from Coridium

Requirements

All versions

- **ARMbasic** for the ARMmite, Wireless and ARMexpressLITE runs on Windows and is controlled by a USB port..
- The **ARMbasic** compiler runs on the ARMexpress hardware platform and only requires a terminal emulator connection through either a USB or serial port, but to get pre-processor functions the compiler needs to run on Windows.
- The **ARMbasic** compiler runs on the ARMweb hardware platform and only requires a browser for programming.
- 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 and HTML.

Installing

Window 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 can be run on either the PC or the ARMexpress. New debug features of BASICtools do rely on the compiler being run 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 ARMexpress, 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
- an alternative implementation exists at **http://www.devscott.org/projects/bside/**

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 some object oriented extensions.
- In ARMbasic the default data type is 32 bits (SIGNED INTEGER), and also supports arrays of SIGNED INTEGERS and STRINGs.

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

Although version 6 of**ARMbasic** has an extremely similar syntax to PBASIC, there are subtle differences.

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 seperate 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 bits, 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

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 utilites.

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 nor available when using the builtin 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

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 90+ page document** from the Free Software Foundation.

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 c:/Program Files/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 c:/Program Files/Coridium/temp 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 ARMexpress or ARMmite.

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 ARMexpress and ARMweb products or on the PC for the ARMmite.

Aside from having a syntax generally compatible with Visual BASIC, **ARMbasic** introduces several features of the popular PBASIC, including I2C, SERIAL, PWM, IN, OUT and FREQOUT.

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 builtin 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 4 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.

What differences are there between ARMbasic and PBASIC?

See **Differences between ARMbasic and PBASIC**.

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

- All 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. At present this only requires a change to the compiler running on the PC not firmware on the ARMexpress/mite.

Floating Point Math

- ARMbasic uses 32 bit math for all numeric operations. There is no plan to add floating point at this point. Floats are available in C for the ARMexpress and ARMmite.

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 existance. But the intention was always to have an ARMweb product, and for that product to support adding ARMbasic statements into a webpage 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 onchip, 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

Getting Started with ARMbasic questions

Advanced ARMbasic

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:

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 compatable) 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 upto 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 ctlC 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 compatable to 6.17 and later firmware versions.

Firmware changes: only look for ESC/ctlC 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 builtin 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 optiimized.

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 VBstyle 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.

Notices

NO WARRANTY

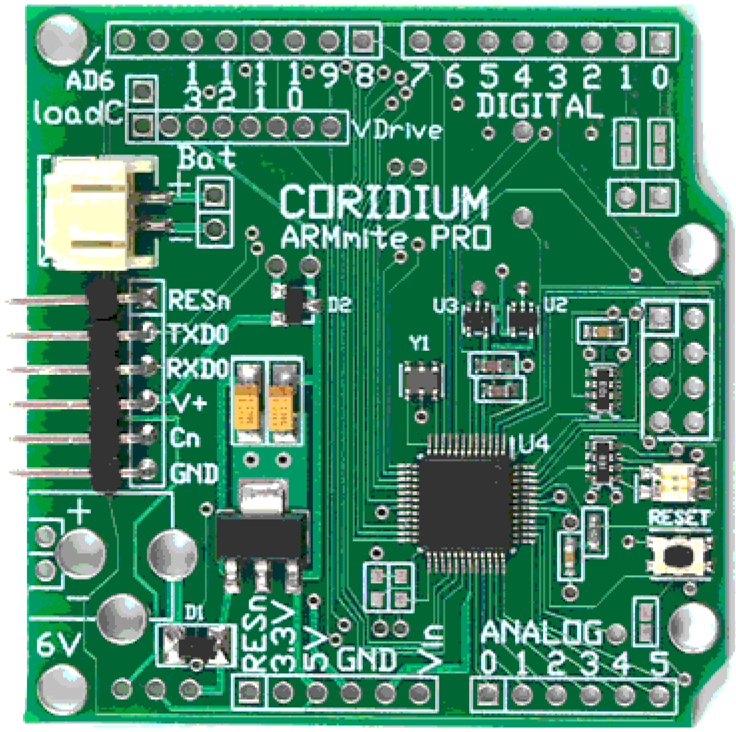
```
  1. THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY
APPLICABLE LAW.  CORIDIUM PROVIDES THE PROGRAM "AS IS" WITHOUT WARRANTY
OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO,
THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
PURPOSE.  THE 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 **ARMbasic©** compiler is distributed as part of hardware sold by Coridium Corp. such as the ARMexpress module. All rights to the compiler are reserved under copyright to Coridium Corp. It may not be copied or reverse engineered..

- Windows® is a registered trademark of Microsoft Corporation.
- VisualBASIC® is a registered trademark of Microsoft Corporation.
- BASIC Stamp® is a registered trademark of Parallax, Inc.
- PBASIC™ is a trademark of Parallax, Inc.
- I²C® is a registered trademark of Philips Corporation.
- 1-Wire® is a registered trademark of Maxim/Dallas Semiconductor.
- SPI™ is a trademark of Motorola

This documentation is released under the GFDL license.

The Language



The Language

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 utilites.

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 nor available when using the builtin 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

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 90+ page document** from the Free Software Foundation.

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 c:/Program Files/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 c:/Program Files/Coridium/temp 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 ARMexpress or ARMmite.

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
```

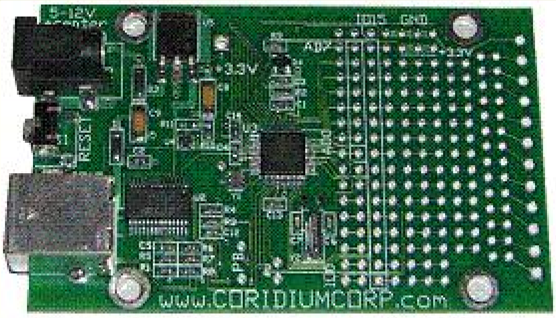
Differences from other BASICs

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

See also

- **#ifdef**

Simple Statements



Simple Statements

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

IN(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

or

CALL label

[CALL] function/sub

CALL (expr)

Description

GOSUB is supported for backward compatibility, now **FUNCTIONs and SUBs** and their implied CALL would be a 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 label: or as a SUB or FUNCTION

CALL for a FUNCTION or SUB is optional. When CALLing a FUNCTION the return value is discarded.

CALL (expr) was added in 7.40 compiler which allows calls to a pointer to a function. The parenthesis are required. Parameter passing to this type of call is not supported.

Example

```
GOSUB message
END

message:
PRINT "Welcome!
return

sub print1111
  print 1111
endsub

main:
  fpointer = ADDRESSOF print1111

  call ( fpointer )
```

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 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

or

CALL label

[CALL] function/sub

CALL (expr)

Description

GOSUB is supported for backward compatibility, now **FUNCTIONs and SUBs** and their implied CALL would be a 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 label: or as a SUB or FUNCTION

CALL for a FUNCTION or SUB is optional. When CALLing a FUNCTION the return value is discarded.

CALL (expr) was added in 7.40 compiler which allows calls to a pointer to a function. The parenthesis are required. Parameter passing to this type of call is not supported.

Example

```
GOSUB message
END

message:
PRINT "Welcome!
return

sub print1111
  print 1111
endsub

main:
  fpointer = ADDRESSOF print1111

  call ( fpointer )
```

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

Jumps code execution to a line label.

Goto's should be avoided for more modern structures such as **DO...LOOP**, **FOR...NEXT**, and **WHILE...LOOP**
.

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 ARMexpress/ARMmite are running stand-alone and without direct human input. However, during the bringup 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.

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

- ARMexpress DEBUGIN can take numbers in hexadecimal, binary or decimal format by using $hex %bin

- PBASIC is taylored for more interaction and allows more complex DEBUGIN
- other BASICs calls this function INPUT

See also

PRINT

Syntax

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

Description

Prints *expressionlist* to screen.

Expressionlist can be constant string, constant numbers, variables, string variables or expressions consisting ov variables and numbers. Seperated 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 16 byte FIFO in the serial port, once that is filled BASIC will wait for space to be available.

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**

READ

Syntax

READ *{constant,} variable_list*

variable_list = *variable* | *array_element* | *string_element {, variable_list }*

Description

Reads data stored by the BASIC application with the **DATA** command.

The elements of the *variable_list* must be integer variables, elements of a string, or elements of arrays. Each element read, will be filled from a 32bit value in the 4K space used to store the DATA statements. All the DATA statements in the program behave as a single list.

After the last element of a DATA is read, the first element of the following DATA will be read.

The **RESTORE** statement resets the next-element pointer to the start of the DATA. This allows the user to alter the order in which the DATA are READ.

If the READ is followed by a *constant,* then the element will be filled from the nth DATA element where n = *constant*.

Example

```
' Create an array of 5 integers.
DIM h(4)

' Set up to loop 5 times (for 5 numbers... check the data)
FOR read_data = 0 TO 4

  ' Read in an integer.
  READ h(read_data)

  ' Display it.
  PRINT "Number"; read_data;" = "; h(read_data)

NEXT

END


' Block of data.

DATA 3, 234, 4354, 23433, 87643
```

Differences from other BASICs

- Most classic BASICs contain this construct
- Does not exist in Visual BASIC
- PBASIC allows modifiers for size. In PBASIC the first element always sets the offset into the data array. This is the case in ARMbasic only if the first element is a constant.

See also

- **DATA**
- **RESTORE**

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 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

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.

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 interchangeablilty 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.

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

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 compaired 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 wll 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 $41,   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

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 (seperate words) are used in Visual BASIC and is allowed in **ARMbasic**

See also

- **IF...THEN**

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.

Example

```
WHILE x = 0
      x = 1
LOOP
```

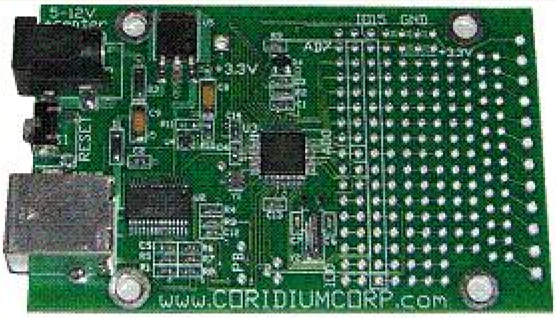
Differences from other BASICs

- 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



Other Statements

- CLEAR
- CONST
- DATA
- DIM
- END
- label:
- MAIN
- ON
- RESTORE
- RUN
- STOP

CONST

Syntax

CONST *symbolname* = *value*

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-

Constants do not take up any program space on the ARMmite or when using the PC Compile option on the ARMexpress. In this case the constants are used by the compiler running on the PC and compiled into code when used. When using the ARMexpress compiler, constants do take up space in the symbol table.

Constants can be 32 bit values using the PC ARMbasic compiler, butconstants are limited to 16bit values for the onchip ARMweb compiler.

Example

```
CONST reps = 5

FOR I = 1 TO reps
  PRINT I
NEXT I

  --  will print out
1
2
3
4
5
```

Differences from other BASICs

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

See also

- **Preprocessor**

DATA

Syntax

DATA *constant1* [,*constant2*]...

Description

DATA statements are used to build up a list of elements in Flash. The compiler processes them in order of appearance in the progam, NOT in order of execution. DATA statements are evaluated at compile time, so they should contain constant integers. DATA statements may not be located within complex statements (such as FOR..NEXT, SUB..ENDSUB ...)

RESTORE resets the READ data pointer to the first DATA element defined.

DATA is normally used to initialize variables.

On the ARMmite, DATA statements are stored above the code space. So using DATA will reduce the space available for code by 1K. DATA space is shared with constant strings on the ARMmite, so the combined space allowable is 1K.

The space between the end of your code and the start of DATA statements can be written and read with **FREAD** and **WRITE** commands, see the **memory map** for details.

Example

```
' Create an array of 5 integers and a string to hold the data.
DIM h(5)
' Set up to loop 5 times (for 5 numbers... check the data)
FOR read_data = 0 TO 4

  ' Read in an integer.
  READ h(read_data)

  ' Display it.
  PRINT "Number"; read_data;" = "; h(read_data)

NEXT


DATA 3, 234, 435, 23, 87643
```

Differences from QB

- common to earlier BASICs
- no equivalent in Visual BASIC
- similar to PBASIC

See also

- **READ**
- **RESTORE**
- **WRITE**

DIM

Syntax

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

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

Declaring Integers:
  DIM *symbolname* AS INTEGER

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 assign 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 compatibilty strings may have the last character the dollar sign $.

Only one symbolname may be declared with each DIM statement.

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 alligned, you must allow enough space to accomodate the expected maximum size of the string plus 1 byte for a termination (0) character. String operators rely on the terminator.

Simple variable 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.

SUB procedures also use DIM between SUB .. ENDSUB. 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.

In version 7.05, AS STRING arrays are no longer limited to 255 bytes, so that they may be used for larger arrays of bytes. However, string operations and functions ARE limited to 255 bytes.

Example

```
DIM a$ (10)
DIM b$ (20)
DIM c$ (30)


a$  = "Hello World"
b$  = "... from ARMbasic!"
c$ = a$ + b$

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 compatable 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

....

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.

Example

```
SUB1:
PRINT "Hello from sub1"
RETURN

MAIN:
GOSUB SUB1
END
```

Differences from other BASICs

- none from Visual BASIC
- none from PBASIC

See also

- EXIT

ON (version 7.30 and later on ARM7 parts)

For PROplus and SuperPRO see INTERRUPT SUB

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. This pin is available on the wireless ARMmite, but if you intend to use it, make sure it is pulled high normally, otherwise when the board is reset it will go into the download C mode and will not run your BASIC program. 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

```
#defineVICIntEnClear  *$FFFFF014

#define TIMERoff    VICIntEnClear = $20
#define EINT0off    VICIntEnClear = $4000
#define EINT1off    VICIntEnClear = $8000
#define EINT2off    VICIntEnClear = $10000
```

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

- VB ???
- no equivalent in PBASIC

See also

- GOTO
- RETURN

RESTORE

Syntax

RESTORE

Description

Sets the next-data-to-read pointer to the first element of the first **DATA** statement.

Example

```
' Create an 2 arrays of integers and a 2 strings to hold the data.
DIM h(4)
DIM h2(4)


' Set up to loop 5 times (for 5 numbers... check the data)
FOR read_data1 = 0 TO 4

  ' Read in an integer.
  READ h(read_data1)

  ' Display it.
  PRINT "Bloc 1, number"; read_data1;" = "; h(read_data1)

NEXT

' Set the data read to the beginning
RESTORE

' Print it.
PRINT  "Bloc 1 string = " + hs

' Spacers.
PRINT
Print


' Set the data read to the beginning
RESTORE

' Set up to loop 5 times (for 5 numbers... check the data)
FOR read_data2 = 0 TO 4

  ' Read in an integer.
  READ h2(read_data2)

  ' Display it.
  PRINT "Bloc 2, number"; read_data2;" = "; h2(read_data2)

NEXT



DATA 3, 234, 4354, 23433, 87643


DATA 546, 7894, 4589, 64657, 34554
```

Differences from QB

- common to many earlier BASICs
- no equivalent in Visual BASIC
- none from PBASIC

See also

- DATA
- READ

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 excecution, but allows BASICtools to dump variable values. Also in BASICtools RUN will resume execution at the statement following STOP.

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 varialble 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.

It is also possible to execute a simple statement immediately. This can be very useful when interfacing to a serial device, as you can step through operations manually, to test a program.

There are a number of operations that aid during the debug phase of programming an ARMexpress.

Debugging Functions

- >
- .
- @
- CLEAR
- DEC
- HEX
- RUN

@ (dump memory)

Syntax

@ [expression]

Description

This command will dump ARM memory starting at *expression*. It is useful for debugging direct control of the ARM peripherals. If *expression* is omitted, then the next page of memory will be displayed. Normally @ expression will be used first, with following pages displayed by typing @ without the *expression*.

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

Example

The following example displays the area of ARM memory corresponding to the PWM registers. Memory address on the left, followed by 4 words of memory displayed in hex and then displayed as printable ASCII characters.

```
@e0014000

00000000 00000001 04BFE6BB 0000E663 E0014010: 0000A516 00000000 00000000
00000000
```

Differences from other BASICs

- non-existant function 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 words of memory displayed in hex and then displayed as printable ASCII characters.

```
@e0014000

00000000 00000001 04BFE6BB 0000E663 E0014010: 0000A516 00000000 00000000
00000000

!e0014000 1234567

@e0014000

01234567 00000001 04BFE6BB 0000E663 E0014010: 0000A516 00000000 00000000
00000000
```

Differences from other BASICs

- non-existant function in Visual BASIC or PBASIC

See also

- @ (dump memory)

CLEAR

Syntax

CLEAR

Description

This is a compile time command that erases the current BASIC program in memory.

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 ARMexpress/ARMmite are running stand-alone and without direct human input. However, during the bringup 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.

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

- ARMexpress DEBUGIN can take numbers in hexadecimal, binary or decimal format by using $hex %bin

- PBASIC is taylored for more interaction and allows more complex DEBUGIN
- other BASICs calls this function INPUT

See also

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 using the Save As under files in BASICtools.

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
```

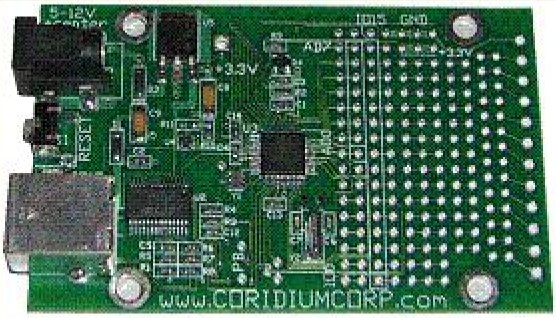
Differences from other BASICs

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

See also

- CLEAR

FUNCTIONs and SUBroutines



Sub Programs

- FUNCTION
- SUB
- ENDFUNCTION
- ENDSUB

FUNCTION *name* (optional parameters)

Syntax

```
FUNCTION name  [AS INTEGER |  AS STRING]

   or

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

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 accessable from within that FUNCTION procedure (the local scope).

Parameters are assumed to be called BYVAL if not specified. In BYVAL calls, a copy of the parameter is passed to the Function. 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.

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 .

An implied RETURN is compiled at the ENDFUNCTION , but code should also return to the caller with RETURN <expression>. A FUNCTION may also be called with a GOSUB, but the returned value is ignored.

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) - $20
  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**
- **ENDSUB**
- **MAIN:**

SUB *name* (optional parameters)

Syntax

SUB *name*

or

```
SUB name (parameter list)
  parameter list = parameter [, parameter list]
  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 accessable from within that SUB procedure (the local scope).

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.

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.

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
- 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) - $20
  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
- AND (Conjunction)
- COS
- MOD (Integer modulo)
- NOT (Bit-wise complement)
- OR (Disjunction, Inclusive Or)
- << (Shift-left)
- >> (Shift-right)
- REV
- SIN
- 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$(20)
DIM C$(20)
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 negitive 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**

/ (Division)

Syntax

argument1 / *argument2*

Description

The division operator is used to divide (or to find the ratio of) two numbers and return an integer result. Division is the inverse of multiplication, *. The arguments *argument1* and *argument2* can be any valid numerical expression. If either argument is an uninitialized variable, that argument will be evaluated as zero. If *argument2* is zero, a division by zero will be raised.

Example

```
PRINT n / 5
n = 600000 / 23
PRINT n
SLEEP
```

The output would look like:

```
0
 26086
```

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)

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**

>= (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

number AND *number*

Description

And, 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 both bits are true, and false for any other combination. The truth table below demonstrates all combinations of a boolean and operation:

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

This holds true for conditional expressions in **ARMbasic** . When using "And" encased in an If block, While loop, or Do loop, the output will behave quite literally:

```
IF condition1 AND condition2 THEN expression1
```

Is translated as:

```
IF condition1 IS true, AND condition2 IS true, THEN perform expression1
```

When given two expressions, numbers, or variables that return a number that is more than a single bit, AND is performed "bitwise". A bitwise operation compares each bit of one number, with each bit of another number, performing a logic operation for every bit.
The boolean math expression below describes this:

```
00001111 AND
00011110
--------  equals
00001110
```

Notice how in the resulting number of the operation, reflects an AND operation performed on each bit of the top operand, with each corresponding bit of the bottom operand. The same logic is also used when working with conditions.

Example

```
' Using the AND operator on two numeric values
numeric_value1 = 15 '00001111
numeric_value2 = 30 '00011110

'Result =  14  =     00001110
PRINT numeric_value1 AND numeric_value2
END
```

```
' Using the AND operator on two conditional expressions
numeric_value1 = 15
numeric_value2 = 25

IF numeric_value1 > 10 AND numeric_value1 < 20 THEN PRINT "Numeric_Value1 is between 10 and 20"
IF numeric_value2 > 10 AND numeric_value2 < 20 THEN PRINT "Numeric_Value2 is between 10 and 20"
END

' This will output "Numeric_Value1 is between 10 and 20" because
' both conditions of the IF statement is true
' It will not output the result of the second IF statement because the first
' condition is true and the second is false.
```

Differences from other BASICs

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

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 "bitwise". A bitwise 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  ...
'-------- eqivalent to
if x <= 5 then ...
```

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

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

numeric_value1 = 15
numeric_value2 = 25

IF NOT numeric_value1 = 10 THEN PRINT "Numeric_Value1 is not equal to 10"
IF NOT numeric_value2 = 25 THEN PRINT "Numeric_Value2 is not equal to 25"
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

Or, 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 either bit is true, and false if both bits are false. The truth table below demonstrates all combinations of a boolean or operation:

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

This holds true for conditional expressions in ARMbasic. When using "Or" encased in an If block, While loop, or Do loop, the output will behave quite literally:

```
IF condition1 OR condition2 THEN expression1
```

Is translated as:

```
IF condition1 IS true, OR condition2 IS true, THEN perform expression1
```

When given two expressions, numbers, or variables that return a number that is more than a single bit, Or is performed "bitwise". A bitwise operation compares each bit of one number, with each bit of another number, performing a logic operation for every bit.
The boolean math expression below describes this:

```
00001111 OR
00011110
-------- equals
00011111
```

Notice how in the resulting number of the operation, reflects an OR operation performed on each bit of the top operand, with each corresponding bit of the bottom operand. The same logic is also used when working with conditions.

Example

```
numeric_value1 = 15 '00001111
numeric_value2 = 30 '00011110

'Result =  31  =    00011111
PRINT numeric_value1 OR numeric_value2
END
```

```
' Using the OR operator on two conditional expressions
numeric_value = 10

IF numeric_value = 5 OR numeric_value = 10 THEN PRINT "Numeric_Value equals 5 or 10"
END

' This will output "Numeric_Value equals 5 or 10" because
' while the first condition of the first IF statement is false, the second is true
```

Differences from PBASIC

- PBASIC OR is always logical, and | is bitwise

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.

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.

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

- <<

REV

Syntax

(*value*) REV (*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

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

This holds true for conditional expressions in ARMbasic. When using "Xor" encased in an If block, While loop, or Do loop, the output will behave quite literally:

```
IF condition1 XOR condition2 THEN expression1
```

Is translated as:

```
IF condition1 IS only true, OR only condition2 IS true, THEN perform expression1
```

When given two expressions, numbers, or variables that return a number that is more than a single bit, Xor is performed "bitwise". A bitwise operation compares each bit of one number, with each bit of another number, performing a logic operation for every bit.
The boolean math expression below describes this:

```
00001111 XOR
00011110
-------- equals
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. The same logic is also used when working with conditions.

Example

```
' Using the XOR operator on two numeric values

numeric_value1 = 15 '00001111
numeric_value2 = 30 '00011110

'Result =  17  =     00010001
PRINT numeric_value1 AND numeric_value2
END
```

```
' Using the XOR operator on two conditional expressions

numeric_value1 = 10
numeric_value2 = 15

IF numeric_value1 = 10 XOR numeric_value2 = 20 THEN PRINT "Numeric_Value1 equals 10 or
Numeric_Value2 equals 20"
END

' This will output "Numeric_Value1 equals 10 or Numeric_Value2 equals 20"
' because only the first condition of the IF statement is true
```

Differences from PBASIC

- PBASIC XOR is always logical, and ^ is bitwise

See also

- **AND**
- **OR**
- **NOT**

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.

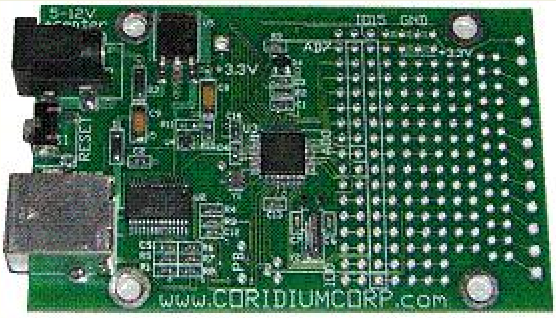
Parentheses can be used to override operator precedence. Operations within parentheses are performed before other operation. However, within the parentheses operator precedence is used.

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

See also

- Operator List

Data Types



Data Types

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

Example

```
CONST FirstNumber = 1
CONST SecondNumber = - 2


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

See also

- **CONST**

Variables

Syntax

symbolname = *expression*   ' automatic declaration

or

DIM *symbolname* AS INTEGER

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.

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

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 syntac 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 tolook 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**.

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

Description

A STRING is an array of characters, and is limited to 256 characters. Larger strings may be allocated, but string operations should be limited to the first 256 characters (no runtime check).

Despite the use of the *maxlength* , an implicit **CHR** (0) 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 as a portion of a string. Byte arrays can be used using an allocation as a string, 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-

```
a$(0) = 0
a$(1) = 1
a$(2) = 2
a$(3) = 3
```

But the following will **NOT** work-

```
a$ = chr(0) + chr(1) + chr(2) + chr(3)   ' fails as the first 0 terminates this string operation
```

But it can be done as-

```
a$ = chr(1) + chr(1) + chr(2) + chr(3)
a$(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 a$(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 a$ (20)
a$ = "Hello"

a$ = a$+chr(32)+ "World"


PRINT a$       '   = "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 = * ($E0024034)    ' read the real time clock day of week register

* ($E0024034) = 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

AdressOf

Syntax

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

or

... = ADRESSOF variable/string

Description

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

Example

```
xx = 0

sub doit
  xx = xx+1
end sub

VICVectAddr3 =  ADRESSOF doit    ' setup the 3rd interrupt to execute doit
```

Differences from other BASICs

- no equivalent in Visual BASIC
- no equivalent in PBASIC

See also

-

Converting Data Types

<u>To/From Strings</u>

- ASC implied
- CHR
- HEX
- STR
- VAL

ASC -- implied function

Syntax

In ARMbasic this is an automatic type conversion

But if you want to do it explicity, in your code add the following do-nothing #define

```
#define ASC(x)     x
```

Description

ARMbasic allows individual elements of a string to be accessed, and when they are assigned or compared to integer variable/constants, the ASCII value will be used.

Example

```
PRINT "the character represented by the ASCII code of 97 is:"; CHR(97)    ' will print   a

DIM astr(10) as string              ' examples of automatic type conversion complimentary to CHR

PRINT astr(0), chr(astr(0))                  ' will print    97     a
x = astr(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

- ASCII table
- HEX
- 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$(10)             ' examples of automatic type conversion complimentary to CHR
a$="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**
- **[ASC]**

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.

This may also be used during debuging to change the default base to Hexadecimal, do this by typing just HEX on the line, opposite of DEC when used this way.

Example

```
DIM text$(10)

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

STR

Syntax

STR(*expression*)

Description

STR will convert a *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.

b$ = 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.

Example

```
DIM b$ (10)
a = 8421
b$ = 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 charater it finds.

Incidentally, this function is the opposite of **STR** , which converts a number to a string.

Example

```
DIM a$(20)
a$ = "20xa211"
b = VAL(a$)
PRINT a$, b
```

```
20xa211   20
```

Differences from other BASICs

- None from Visual BASIC
- similar to formatting directives DEC, HEX in PBASIC

See also

- **STR**
- **HEX**
- **CHR**

Alphabetical Keyword List

With Version 7, most of the builtin firmware hardware routines have been replaced by ARMbasic routines that can be accessed by **#include <filename>.** Version 7 frees up space for more user code (20K vs 12K in the ARMmite). Version 7 is more Visual BASIC like.

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.

Version 7 Firmware Keywords

OPERATORS

- See Operator List

A

- ABS
- AD
- AND
- [ASC]
- AS

B

- BAUD
- BAUD0
- BAUD1
- BYREF
- BYVAL

C

- CALL
- CASE
- CHR
- CLEAR
- CONST

D

- DATA
- DEBUGIN
- DIM

M

- MAIL
- MAIN
- MOD

N

- NEXT
- NOT

O

- ON
- OR
- OUT
- OUTPUT

P

- PRINT

R

- READ
- RESTORE
- RETURN
- REV
- RIGHT
- RND
- RUN
- RXD
- RXD0

- LIST - LOOP - LOW	- * pointer

* (ARM peripheral access)

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.

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.

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 = 0x400    ' 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

- Hardware Library Functions

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.

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

AD

Syntax

FUNCTION AD (*expression*)

Description --- not available on the original ARMexpress

ARMmite and ARMmite PRO version

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 builtin 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.

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)

Stand-Alone Compilers

Because the hardware is not compatible between LPC types, this must be implemented as a FUNCTION in BASIC and is not part of the firmware.

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

ADDRESSOF

Syntax

ADDRESSOF variable_name

or

ADDRESSOF subroutine_name

Description

ADDRESSOF will return the address of a variable or subroutine.

Example

```
sub print1111
  print 1111
endsub

main:
  fpointer = ADDRESSOF print1111

  call ( fpointer )
```

Differences from other BASICs

- similar to VB
- no equivalent in PBASIC

See also

- CALL

AND

Syntax

number AND *number*

Description

And, 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 both bits are true, and false for any other combination. The truth table below demonstrates all combinations of a boolean and operation:

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

This holds true for conditional expressions in **ARMbasic** . When using "And" encased in an If block, While loop, or Do loop, the output will behave quite literally:

```
IF condition1 AND condition2 THEN expression1
```

Is translated as:

```
IF condition1 IS true, AND condition2 IS true, THEN perform expression1
```

When given two expressions, numbers, or variables that return a number that is more than a single bit, AND is performed "bitwise". A bitwise operation compares each bit of one number, with each bit of another number, performing a logic operation for every bit.
The boolean math expression below describes this:

```
00001111 AND
00011110
-------- equals
00001110
```

Notice how in the resulting number of the operation, reflects an AND operation performed on each bit of the top operand, with each corresponding bit of the bottom operand. The same logic is also used when working with conditions.

Example

```
' Using the AND operator on two numeric values
numeric_value1 = 15 '00001111
numeric_value2 = 30 '00011110

'Result =  14  =     00001110
PRINT numeric_value1 AND numeric_value2
END
```

```
' Using the AND operator on two conditional expressions
numeric_value1 = 15
numeric_value2 = 25

IF numeric_value1 > 10 AND numeric_value1 < 20 THEN PRINT "Numeric_Value1 is between 10 and 20"
IF numeric_value2 > 10 AND numeric_value2 < 20 THEN PRINT "Numeric_Value2 is between 10 and 20"
END

' This will output "Numeric_Value1 is between 10 and 20" because
' both conditions of the IF statement is true
' It will not output the result of the second IF statement because the first
' condition is true and the second is false.
```

Differences from other BASICs

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

See also

- OR
- XOR
- NOT

AS

Syntax

FUNCTION *name* [AS INTEGER | AS STRING]

or

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

or

DIM *symbolname* (size) AS STRING

DIM *symbolname* AS INTEGER

Description

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

Differences from other BASICs

- simplification of Visual BASIC
- no equivalent in PBASIC

See also

- FUNCTION
- SUB
- DIM

ASC -- implied function

Syntax

In ARMbasic this is an automatic type conversion

But if you want to do it explicity, in your code add the following do-nothing #define

```
#define ASC(x)     x
```

Description

ARMbasic allows individual elements of a string to be accessed, and when they are assigned or compared to integer variable/constants, the ASCII value will be used.

Example

```
PRINT "the character represented by the ASCII code of 97 is:"; CHR(97)    ' will print   a

DIM astr(10) as string              ' examples of automatic type conversion complimentary to CHR

PRINT astr(0), chr(astr(0))                  ' will print    97     a
x = astr(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

- ASCII table
- HEX
- VAL

BYREF

Syntax

FUNCTION *name* [AS INTEGER | AS STRING]

  or

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

Description

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

When used a pointer to the parameter will be used in the FUNCTION or SUB.   This allows a function to read AND write the original source parameter.

An advantage in 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.

Differences from other BASICs

- simplification of Visual BASIC
- no equivalent in PBASIC

See also

- FUNCTION
- SUB

BYTEBUS (ARMweb only)

Syntax

BYTEBUS (control)

Description

BYTEBUS reads or writes the 8 bit + 2 control lines on Port1 of the LPC2138. The control field sets the state of the 2 control lines, with the intention of line 0 being used as a R/W line and line 1 being used as a CS line-

0 -- set control line 0 low, and pulse line 1 low
1 -- set control line 0 high, and pulse line 1 low
2 -- set control line 0 low, and pulse line 1 high
3 -- set control line 0 high, and pulse line 1 high

4 -- use the 10 lines as a block of inputs or outputs (added in version 7 firmware)

For 0-3:

The pulsewidth on line 1 is 250 nsec for write, and 550 nsec for read.

Back to back operations occur 2.4 usec apart for writes, 2 usec for read.

None of these lines are driven on reset, and should be biased with resistors if devices connected to this bus require it.

Example

```
'write to byte bus - negative true CS and W
BYTEBUS(0)  = $A5

'read from byte bus - negative true CS, R-notW line
x = BYTEBUS(1)
```

block control added in version 7 firmware-

```
'write to 10 pins as a block
BYTEBUS(4)  = $2A5

'read from 10 pins as a block
x = BYTEBUS(4)
```

Differences from other BASICs

- no equivalent in Visual BASIC
- no equivalent in PBASIC

See also

- HIGH

BYVAL

Syntax

FUNCTION *name* [AS INTEGER | AS STRING]

or

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

Description

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

When 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.

Differences from other BASICs

- simplification of Visual BASIC
- no equivalent in PBASIC

See also

- FUNCTION
- SUB

GOSUB CALL

Syntax

GOSUB label

or

CALL label

[CALL] function/sub

CALL (expr)

Description

GOSUB is supported for backward compatibility, now **FUNCTIONs and SUBs** and their implied CALL would be a 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 label: or as a SUB or FUNCTION

CALL for a FUNCTION or SUB is optional. When CALLing a FUNCTION the return value is discarded.

CALL (expr) was added in 7.40 compiler which allows calls to a pointer to a function. The parenthesis are required. Parameter passing to this type of call is not supported.

Example

```
GOSUB message
END

message:
PRINT "Welcome!
return

sub print1111
  print 1111
endsub

main:
  fpointer = ADDRESSOF print1111

  call ( fpointer )
```

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**

CASE

Syntax

CASE expression

Description

CASE is used in a SELECT CASE statement to determine conditions for running a branch of code.

See **SELECT CASE**.

See also

- **SELECT CASE**

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$(10)              ' examples of automatic type conversion complimentary to CHR
a$="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**
- **[ASC]**

CLEAR

Syntax

CLEAR

Description

This is a compile time command that erases the current BASIC program in memory.

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

CONST

Syntax

CONST *symbolname* = *value*

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-

Constants do not take up any program space on the ARMmite or when using the PC Compile option on the ARMexpress. In this case the constants are used by the compiler running on the PC and compiled into code when used. When using the ARMexpress compiler, constants do take up space in the symbol table.

Constants can be 32 bit values using the PC ARMbasic compiler, butconstants are limited to 16bit values for the onchip ARMweb compiler.

Example

```
CONST reps = 5

FOR I = 1 TO reps
  PRINT I
NEXT I

  --  will print out
1
2
3
4
5
```

Differences from other BASICs

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

See also

- **Preprocessor**

DATA

Syntax

DATA *constant1* [,*constant2*]...

Description

DATA statements are used to build up a list of elements in Flash. The compiler processes them in order of appearance in the progam, NOT in order of execution. DATA statements are evaluated at compile time, so they should contain constant integers. DATA statements may not be located within complex statements (such as FOR..NEXT, SUB..ENDSUB ...)

RESTORE resets the READ data pointer to the first DATA element defined.

DATA is normally used to initialize variables.

On the ARMmite, DATA statements are stored above the code space. So using DATA will reduce the space available for code by 1K. DATA space is shared with constant strings on the ARMmite, so the combined space allowable is 1K.

The space between the end of your code and the start of DATA statements can be written and read with **FREAD** and **WRITE** commands, see the **memory map** for details.

Example

```
' Create an array of 5 integers and a string to hold the data.
DIM h(5)
' Set up to loop 5 times (for 5 numbers... check the data)
FOR read_data = 0 TO 4

  ' Read in an integer.
  READ h(read_data)

  ' Display it.
  PRINT "Number"; read_data;" = "; h(read_data)

NEXT


DATA 3, 234, 435, 23, 87643
```

Differences from QB

- common to earlier BASICs
- no equivalent in Visual BASIC
- similar to PBASIC

See also

- **READ**
- **RESTORE**
- **WRITE**

DEBUGIN *variable*

Syntax

DEBUGIN *variable* | *string*

Description

Normally the programs running on an ARMexpress/ARMmite are running stand-alone and without direct human input. However, during the bringup 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.

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

- ARMexpress DEBUGIN can take numbers in hexadecimal, binary or decimal format by using $hex %bin

- PBASIC is taylored for more interaction and allows more complex DEBUGIN
- other BASICs calls this function INPUT

See also

DIM

Syntax

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

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

Declaring Integers:
   DIM *symbolname* AS INTEGER

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 assign 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 compatibilty strings may have the last character the dollar sign $.

Only one symbolname may be declared with each DIM statement.

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 alligned, you must allow enough space to accomodate the expected maximum size of the string plus 1 byte for a termination (0) character. String operators rely on the terminator.

Simple variable 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.

SUB procedures also use DIM between SUB .. ENDSUB. 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.

In version 7.05, AS STRING arrays are no longer limited to 255 bytes, so that they may be used for larger arrays of bytes. However, string operations and functions ARE limited to 255 bytes.

Example

```
DIM a$ (10)
DIM b$ (20)
DIM c$ (30)


a$  = "Hello World"
b$  = "... from ARMbasic!"
c$ = a$ + b$

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 compatable with earlier BASICs which indexed from 1 to *size* .

- PBASIC uses the syntax *symbolname* VAR WORD | BYTE [(*size*)]

See also

DIR

Syntax

DIR (*expression*)

Description

DIR (expression) can be used to set or read the direction of the 16 configurable pins. If DIR (expression) is 1 then the corresponding pin is an output. If the value is 0 then that pin is an input.

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, ARMmite PRO, ARMexpress and ARMexpress LITE these pin numbers correspond to the pin numbers shown in the **Hardware Section** . For the ARMweb, DINkit, SuperPRO these pin numbers correspond only to the Port 0 assigned by NXP, for instance DIR 3 corresponds to P0.3

For port pins after Port 0, use the **P1 .. P4 commands**, or a #define FIO0DIR.

Example

```
' Set pin 4 as an input
DIR(4) = 0

' Set pin 12 as an output
DIR(12) = 1
```

Differences from other BASICs

- no equivalent in Visual BASIC
- equivalent to DIR0..15 in PBASIC

See also

- **INPUT**
- **OUTPUT**

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.

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 interchangeablilty of UNTIL as the equivalent of NOT WHILE

See also

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

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
```

ELSE

Syntax

if [condition] then [action] ELSE [action]

Description

see **IF...THEN**.
Example

```
IF 1 THEN
 PRINT "One!"
ELSE
 PRINT "Nope!"
ENDIF
```

Differences from QB

- none from Visual BASIC
- none from PBASIC

See also

- **IF THEN**

ELSEIF

Syntax

if [condition] then [action] ELSEIF [condition] then [action]

Description

see **IF...THEN**.

Example

```
IF A = 1 THEN
  PRINT "ONE!"
ELSEIF A = 2 THEN
  PRINT "TWO!"
ENDIF
```

Differences from other BASICs

- None from PBASIC
- Visual BASIC uses a two word END IF, rather than the ARMbasic ENDIF

See also

- **IF...THEN**

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

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) - $20
  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:

ENDIF | END IF

Syntax

if [statement] then
[action]
ENDIF

Description

ENDIF is used to denote the end of a block IF statement.

Version 7.00 allows ENDIF or END IF syntax

Example

```
IF a = 1 THEN
 PRINT "A is equal to one!"
ENDIF
```

See also

- IF...THEN

ENDSELECT | END SELECT

Syntax

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

ENDSELECT or END SELECT syntax are allowed

Description

ENDSELECT is used to terminate the SELECT..CASE statement.

Example

```
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

- ENDSELECT is used to terminate the SELECT in PBASIC
- END SELECT used in Visual BASIC

See also

- IF...THEN
- SELECT CASE

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:

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**

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.

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

FREAD

Syntax

SUB FREAD (*FlashAddr*, *Destination*, *size*)

Destination = *arrayname* | *stringname*

size in bytes

Description -- added version 7.13

The builtin 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

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

FUNCTION *name* (optional parameters)

Syntax

FUNCTION *name* [AS INTEGER | AS STRING]

or

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

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 accessable from within that FUNCTION procedure (the local scope).

Parameters are assumed to be called BYVAL if not specified. In BYVAL calls, a copy of the parameter is passed to the Function. 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.

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 .

An implied RETURN is compiled at the ENDFUNCTION , but code should also return to the caller with RETURN <expression>. A FUNCTION may also be called with a GOSUB, but the returned value is ignored.

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) - $20
  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**
- **ENDSUB**
- **MAIN:**

GOSUB CALL

Syntax

GOSUB label

or

CALL label

[CALL] function/sub

CALL (expr)

Description

GOSUB is supported for backward compatibility, now **FUNCTIONs and SUBs** and their implied CALL would be a 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 label: or as a SUB or FUNCTION

CALL for a FUNCTION or SUB is optional. When CALLing a FUNCTION the return value is discarded.

CALL (expr) was added in 7.40 compiler which allows calls to a pointer to a function. The parenthesis are required. Parameter passing to this type of call is not supported.

Example

```
GOSUB message
END

message:
PRINT "Welcome!
return

sub print1111
  print 1111
endsub

main:
  fpointer = ADDRESSOF print1111

  call ( fpointer )
```

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

Jumps code execution to a line label.

Goto's should be avoided for more modern structures such as **DO...LOOP**, **FOR...NEXT**, and **WHILE...LOOP**
.

Example

```
GOTO message


message:
PRINT "Welcome!
```

Differences from other BASICs

- none from Visual BASIC
- none from PBASIC

See also

- **GOSUB**

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.

This may also be used during debuging to change the default base to Hexadecimal, do this by typing just HEX on the line, opposite of DEC when used this way.

Example

```
DIM text$(10)

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

HIGH

Syntax

HIGH *expression*

Description

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 compatablity.

For the ARMmite, ARMmite PRO, ARMexpress and ARMexpress LITE these pin numbers correspond to the pin numbers shown in the**Hardware Section** . For the ARMweb, DINkit, SuperPRO these pin numbers correspond only to the Port 0 assigned by NXP, for instance HIGH 3 corresponds to P0.3

For port pins after Port 0, use the **P1 .. P4 commands**.

Example

```
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

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**

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

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 bitwise 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, ARMmite PRO, ARMexpress and ARMexpress LITE these pin numbers correspond to the pin numbers shown in the**Hardware Section** . For the ARMweb, DINkit, SuperPRO these pin numbers correspond to the port assigned by NXP, for instance IN(3) corresponds to P0.3

For port pins after port 0, 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 and OUTPUT were added for PBASIC compatability, 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, ARMmite PRO, ARMexpress and ARMexpress LITE these pin numbers correspond to the pin numbers shown in the **Hardware Section** . For the ARMweb, DINkit, SuperPRO these pin numbers correspond only to the Port 0 assigned by NXP, for instance INPUT 3 corresponds to P0.3

For port pins after Port 0, use the **P1 .. P4 commands**, or a #define FIO0DIR.

Example

```
INPUT (0)      ' this will make pin 0 an input
```

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**

INTEGER

Syntax

FUNCTION *name* [AS INTEGER | AS STRING]

or

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

or

DIM *symbolname* (size) AS STRING

DIM *symbolname* AS INTEGER

Description

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

Differences from other BASICs

- simplification of Visual BASIC
- no equivalent in PBASIC

See also

- FUNCTION
- SUB
- DIM

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.

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
  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

IO

Syntax

IO (*expression*)

Description

IO is a more complex way to access or control the pins. When IO (*expression*) is read, the pin corresponding to *expression* is converted to an input and the value on that pin is returned.

When assiging a value to IO(*expression*), 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 bitwise 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, HIGH or LOW.

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, ARMmite PRO, ARMexpress and ARMexpress LITE these pin numbers correspond to the pin numbers shown in the **Hardware Section** . For the ARMweb, DINkit, SuperPRO these pin numbers correspond only to the Port 0 assigned by NXP, for instance IO(3) corresponds to P0.3

For port pins after Port 0, use the **P1 .. P4 commands**, or a #define FIO0DIR.

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**

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.

A$ = LEFT("this is a test",5) + RIGHT(B$,3)   ' valid string operation

A$ = LEFT("this "+b$,5)   ' NOT ALLOWED nested operation

Example

```
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

LEN will return the length of a *string* in characters.

Example

```
PRINT LEN("hello world") 'returns "11"
```

Differences from PBASIC

- This function does not exist in PBASIC.

See also

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 using the Save As under files in BASICtools.

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
```

LOOP

Description

Part of Do [...] Loop.
See **DO...LOOP**.

LOW

Syntax

LOW *expression*

Description

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 compatablity.

For the ARMmite, ARMmite PRO, ARMexpress and ARMexpress LITE these pin numbers correspond to the pin numbers shown in the **Hardware Section** . For the ARMweb, DINkit, SuperPRO these pin numbers correspond only to the Port 0 assigned by NXP, for instance LOW 3 corresponds to P0.3

For port pins after Port 0, use the **P1 .. P4 commands**, or a #define FIO0DIR.

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 hign 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**

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.

Example

```
SUB1:
PRINT "Hello from sub1"
RETURN

MAIN:
GOSUB SUB1
END
```

Differences from other BASICs

- none from Visual BASIC
- none from PBASIC

See also

- EXIT

MOD

Syntax

argument1 MOD *argument2*

Description

MOD is the modulus or "remainder" arthimetic operator. The result of MOD is the integer remainder of *argument1* divided by *argument2*.

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

NEXT

Syntax

NEXT [*identifier_list*]

Description

Indicates the end of a statement block associated with a matching **FOR** statement. *identifier_list*, if given, must match the identifiers used in the associated FOR statements in reverse order.

There should be exactly one NEXT statement (or one item in the identifier list) for every FOR statement.

Example

```
FOR i=1 TO 10
FOR j=1 TO 2
   ...
NEXT
next
```

```
FOR i=1 TO 10
FOR j=1 TO 2
   ...
NEXT j
NEXT i
```

```
FOR i=1 TO 10
FOR j=1 TO 2
   ...
NEXT j,i
```

See also

- **FOR** statement

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 "bitwise". A bitwise 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  ...
'-------- eqivalent to
if x <= 5 then ...
```

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

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

numeric_value1 = 15
numeric_value2 = 25

IF NOT numeric_value1 = 10 THEN PRINT "Numeric_Value1 is not equal to 10"
IF NOT numeric_value2 = 25 THEN PRINT "Numeric_Value2 is not equal to 25"
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

ON (version 7.30 and later on ARM7 parts)

For PROplus and SuperPRO see INTERRUPT SUB

_
_
_

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. This pin is available on the wireless ARMmite, but if you intend to use it, make sure it is pulled high normally, otherwise when the board is reset it will go into the download C mode and will not run your BASIC program. 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

```
#defineVICIntEnClear  *$FFFFF014

#define TIMERoff    VICIntEnClear = $20
#define EINT0off    VICIntEnClear = $4000
#define EINT1off    VICIntEnClear = $8000
#define EINT2off    VICIntEnClear = $10000
```

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

- VB ???
- no equivalent in PBASIC

See also

- GOTO
- RETURN

OR

Syntax

number OR *number*

Description

Or, 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 either bit is true, and false if both bits are false. The truth table below demonstrates all combinations of a boolean or operation:

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

This holds true for conditional expressions in ARMbasic. When using "Or" encased in an If block, While loop, or Do loop, the output will behave quite literally:

```
IF condition1 OR condition2 THEN expression1
```

Is translated as:

```
IF condition1 IS true, OR condition2 IS true, THEN perform expression1
```

When given two expressions, numbers, or variables that return a number that is more than a single bit, Or is performed "bitwise". A bitwise operation compares each bit of one number, with each bit of another number, performing a logic operation for every bit.
The boolean math expression below describes this:

```
00001111 OR
00011110
-------- equals
00011111
```

Notice how in the resulting number of the operation, reflects an OR operation performed on each bit of the top operand, with each corresponding bit of the bottom operand. The same logic is also used when working with conditions.

Example

```
numeric_value1 = 15 '00001111
numeric_value2 = 30 '00011110

'Result =  31  =     00011111
PRINT numeric_value1 OR numeric_value2
END
```

```
' Using the OR operator on two conditional expressions
numeric_value = 10

IF numeric_value = 5 OR numeric_value = 10 THEN PRINT "Numeric_Value equals 5 or 10"
END

' This will output "Numeric_Value equals 5 or 10" because
' while the first condition of the first IF statement is false, the second is true
```

Differences from PBASIC

- PBASIC OR is always logical, and | is bitwise

See also

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

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, ARMmite PRO, ARMexpress and ARMexpress LITE these pin numbers correspond to the pin numbers shown in the **Hardware Section** . For the ARMweb, DINkit, SuperPRO these pin numbers correspond only to the Port 0 assigned by NXP, for instance OUT(3) corresponds to P0.3

For port pins after Port 0, use the **P1 .. P4 commands**, or a #define FIO0DIR.

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.

INPUT and OUTPUT were added for PBASIC compatability, 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.

For the ARMmite, ARMmite PRO, ARMexpress and ARMexpress LITE these pin numbers correspond to the pin numbers shown in the **Hardware Section** . For the ARMweb, DINkit, SuperPRO these pin numbers correspond only to the Port 0 assigned by NXP, for instance OUTPUT 3 corresponds to P0.3

For port pins after Port 0, use the **P1 .. P4 commands**, or a #define FIO0DIR.

Example

```
' Set pin 9 as an output
OUTPUT (9)
```

Differences from other BASICs

- no equivalent in Visual BASIC
- none from PBASIC

See also

- DIR
- INPUT

PRINT

Syntax

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

Description

Prints *expressionlist* to screen.

Expressionlist can be constant string, constant numbers, variables, string variables or expressions consisting ov variables and numbers. Seperated 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 16 byte FIFO in the serial port, once that is filled BASIC will wait for space to be available.

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**

READ

Syntax

READ *{constant,} variable_list*

variable_list = *variable* | *array_element* | *string_element {, variable_list }*

Description

Reads data stored by the BASIC application with the **DATA** command.

The elements of the *variable_list* must be integer variables, elements of a string, or elements of arrays. Each element read, will be filled from a 32bit value in the 4K space used to store the DATA statements. All the DATA statements in the program behave as a single list.

After the last element of a DATA is read, the first element of the following DATA will be read.

The **RESTORE** statement resets the next-element pointer to the start of the DATA. This allows the user to alter the order in which the DATA are READ.

If the READ is followed by a *constant,* then the element will be filled from the nth DATA element where n = *constant*.

Example

```
' Create an array of 5 integers.
DIM h(4)

' Set up to loop 5 times (for 5 numbers... check the data)
FOR read_data = 0 TO 4

  ' Read in an integer.
  READ h(read_data)

  ' Display it.
  PRINT "Number"; read_data;" = "; h(read_data)

NEXT

END


' Block of data.

DATA 3, 234, 4354, 23433, 87643
```

Differences from other BASICs

- Most classic BASICs contain this construct
- Does not exist in Visual BASIC
- PBASIC allows modifiers for size. In PBASIC the first element always sets the offset into the data array. This is the case in ARMbasic only if the first element is a constant.

See also

- **DATA**
- **RESTORE**

RESTORE

Syntax

RESTORE

Description

Sets the next-data-to-read pointer to the first element of the first **DATA** statement.

Example

```
' Create an 2 arrays of integers and a 2 strings to hold the data.
DIM h(4)
DIM h2(4)


' Set up to loop 5 times (for 5 numbers... check the data)
FOR read_data1 = 0 TO 4

  ' Read in an integer.
  READ h(read_data1)

  ' Display it.
  PRINT "Bloc 1, number"; read_data1;" = "; h(read_data1)

NEXT

' Set the data read to the beginning
RESTORE

' Print it.
PRINT  "Bloc 1 string = " + hs

' Spacers.
PRINT
Print


' Set the data read to the beginning
RESTORE

' Set up to loop 5 times (for 5 numbers... check the data)
FOR read_data2 = 0 TO 4

  ' Read in an integer.
  READ h2(read_data2)

  ' Display it.
  PRINT "Bloc 2, number"; read_data2;" = "; h2(read_data2)

NEXT




DATA 3, 234, 4354, 23433, 87643


DATA 546, 7894, 4589, 64657, 34554
```

Differences from QB

- common to many earlier BASICs
- no equivalent in Visual BASIC
- none from PBASIC

See also

- DATA
- READ

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 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**.

REV

Syntax

(*value*) REV (*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

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

RND

Syntax

RND (*number*)

Description -- added in version 7

This is an LCG random number generator, that takes *number* in as a seed and produces a 32 bit integer pseudo-random number.

Example

```
PRINT RND (33 )
PRINT RND (33 )
PRINT RND ( 55 )

N = 69

PRINT RND ( N )
```

The output would look like:

```
632584417
632584417
-1809004169
2103579653
```

Differences from other BASICs

- none from Visual BASIC
- none from PBASIC

See also

- OR
- XOR
- NOT

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

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 compaired 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 wll 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 $41,   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

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 (seperate words) are used in Visual BASIC and is allowed in **ARMbasic**

See also

- **IF...THEN**

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.

Example

```
FOR I=10 TO 1 STEP -1
```

See also

- **FOR**

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 excecution, but allows BASICtools to dump variable values. Also in BASICtools RUN will resume execution at the statement following STOP.

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

STR

Syntax

STR(*expression*)

Description

STR will convert a *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.

b$ = 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.

Example

```
DIM b$ (10)
a = 8421
b$ = 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**

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)

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

STRING

Syntax

FUNCTION *name* [AS INTEGER | AS STRING]

or

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

or

DIM *symbolname* (size) AS STRING

DIM *symbolname* AS INTEGER

Description

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

Differences from other BASICs

- simplification of Visual BASIC
- no equivalent in PBASIC

See also

- FUNCTION
- SUB
- DIM

SUB *name* (optional parameters)

Syntax

SUB *name*

or

SUB *name* (parameter list)
  parameter list = parameter [, parameter list]
  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 accessable from within that SUB procedure (the local scope).

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.

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.

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**
- **GOSUB**
- **ENDSUB**
- **MAIN:**

THEN

Description

A component of an IF [...] Then decision statement.
See **IF...THEN**.

TIMER

Syntax

TIMER

Description

TIMER is a free running timer that increments every microsecond. Its it readable and writeable using this keyword.

Operations that require more precise timing should use the dedicated hardware routines, as interupts that are occuring for other time functions and serial input may make times using TIMER look longer than actual.

Example

```
START = TIMER< /EM >
WHILE (TIMER-START < WAIT_MICROSECONDS)
LOOP
```

Differences from other BASICs

- *no equivalent in PBASIC*
- *no equivalent in Visual BASIC*

See also

- ***MINUTE***
- ***HOUR***
- ***DAY***
- ***MONTH***
- ***YEAR***
- ***WEEKDAY***

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**

UNTIL

Syntax

See **DO..UNTIL**

Description

UNTIL is used with the **DO...LOOP** structure. See it for more info.

Example

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

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

Differences from other BASICs

- LOOP is required with UNTIL in Visual BASIC
- LOOP is optional in **ARMbasic**

See also

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 charater it finds.

Incidentally, this function is the opposite of **STR** , which converts a number to a string.

Example

```
DIM a$(20)
a$ = "20xa211"
b = VAL(a$)
PRINT a$, b
```

```
20xa211   20
```

Differences from other BASICs

- None from Visual BASIC
- similar to formatting directives DEC, HEX in PBASIC

See also

- **STR**
- **HEX**
- **CHR**

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

- SLEEP
- TIMER

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.

Example

```
WHILE x = 0
      x = 1
LOOP
```

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**

WRITE

Syntax

FUNCTION WRITE (*FlashAddr*, *Source*, *subblocksize*)

Source = *arrayname* | *stringname*

subblocksize = 0 | 256* | 512 | 1024 | 2048 | 4096 | 8192*

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).**

Writing consists of erasing the whole sector and then writing a *subblock* or all.

Erases will erase the entire sector.

subblocksize portions may be written (ARMexpress allows upto 8K but not 256). FlashAddr must be alligned to *subblocksize*.

Data is copied from a string or array to the Flash. Only fixed *subblocksize* sizes are allowed. This function does not look for 0 terminators when a string is the source.

To force a sector to be erased use a block size of 0. Once a portion is written after an erase, it can not be written again without being erased.

WRITE assumes that the sector is to be erased when first written, or when the same *subblock* as the last call to WRITE is being written. When different *subblocks* of the same sector are being written, an erase will only occur when WRITE is called with a subblocksize of 0. The WRITE routine only keeps track of which sector and sublock were last written, you must manage sectors

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

...

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
```

Differences from other BASICs

- Does not exist in Visual BASIC
- PBASIC has a similar function

See also

- **FREAD**
- **Memory Map**
- **CPU details**

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

This holds true for conditional expressions in ARMbasic. When using "Xor" encased in an If block, While loop, or Do loop, the output will behave quite literally:

```
IF condition1 XOR condition2 THEN expression1
```

Is translated as:

```
IF condition1 IS only true, OR only condition2 IS true, THEN perform expression1
```

When given two expressions, numbers, or variables that return a number that is more than a single bit, Xor is performed "bitwise". A bitwise operation compares each bit of one number, with each bit of another number, performing a logic operation for every bit.
The boolean math expression below describes this:

```
00001111 XOR
00011110
-------- equals
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. The same logic is also used when working with conditions.

Example

```
' Using the XOR operator on two numeric values

numeric_value1 = 15 '00001111
numeric_value2 = 30 '00011110

'Result =  17  =     00010001
PRINT numeric_value1 AND numeric_value2
END
```

```
' Using the XOR operator on two conditional expressions

numeric_value1 = 10
numeric_value2 = 15

IF numeric_value1 = 10 XOR numeric_value2 = 20 THEN PRINT "Numeric_Value1 equals 10 or
Numeric_Value2 equals 20"
END

' This will output "Numeric_Value1 equals 10 or Numeric_Value2 equals 20"
' because only the first condition of the IF statement is true
```

Differences from PBASIC

- PBASIC XOR is always logical, and ^ is bitwise

See also

- **AND**
- **OR**
- **NOT**

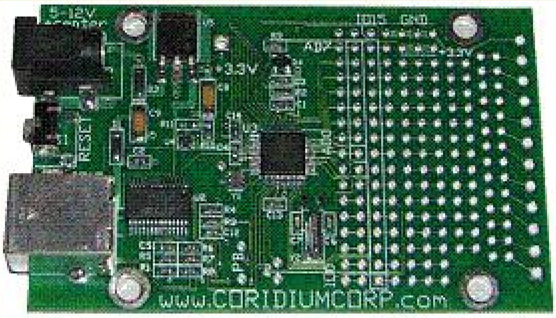
Additional Reserved Words

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.

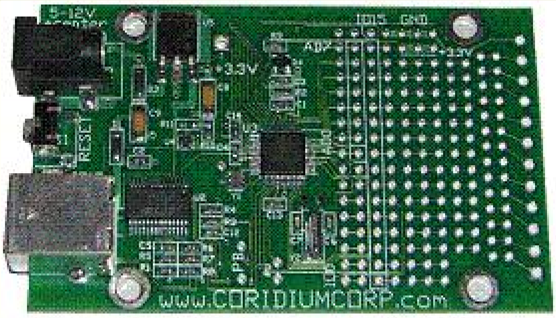
In order to maintain compatability with future ARMbasic instructions the following words have been reserved.

FLOAT QUIT QUITDUMP QUITNOW	READONLY WEB WEBGET



Runtime Library

Mathematical Functions



Mathematical Functions

- ABS
- MOD
- RND
- SIN, COS

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.

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" arthimetic operator. The result of MOD is the integer remainder of *argument1* divided by *argument2*.

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

RND

Syntax

RND (*number*)

Description -- added in version 7

This is an LCG random number generator, that takes *number* in as a seed and produces a 32 bit integer pseudo-random number.

Example

```
PRINT RND (33 )
PRINT RND (33 )
PRINT RND ( 55 )

N = 69

PRINT RND ( N )
```

The output would look like:

```
632584417
632584417
-1809004169
2103579653
```

Differences from other BASICs

- none from Visual BASIC
- none from PBASIC

See also

- OR
- XOR
- NOT

FREQOUT

Library

#include <FREQOUT.bas>

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-

```
initFREQOUT:
#include <FREQOUT.bas>
return

...

main:
   gosub initFREQOUT
```

C

- COS

F

- FREQOUT

S

- SIN

Interface

```
 #define SIN(x)   sin_tbl(x)
 #define COS(x)   sin_tbl(x-64)

' duration is in milliseconds
' freq1 and freq2 in Hz
SUB FREQOUT(pin, duration, freq1, freq2)
```

Internals

ARMbasic uses integers, but there may be a need for certain functions that normally use floating point calculations. One of these is the cosine function, which normally operates on degrees or radians. But for simplicity and the binary world, these values and the result value have been normalized to fit in a byte value. but in this case it is expressed as -127 to +127 or the cos() multiplied by 127.

These SIN and COS functions are identical to the PBASIC versions and are used by FREQOUT. Rather than degrees or radians there are 256 divisions (360/256 degrees) which returns a value of -127 to +127 which correspond to -1 to 1 for normal sine and cosine function.

The SIN function is implemented using string, and accessed a byte at a time to generate the 256 values. COS is the SIN function shifted 90 degrees or 64 places

Example

```
#include <FREQOUT.bas>
...
```

```
'Generate a soothing dual frequency tone on pin 4 for 8 seconds
'using frequncies 2500 and 6000

FREQOUT (4, 8000, 2500, 6000)
```

String Functions

Builtin 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 in the ARMmite 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.

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(a)) + "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 seperate entity. Also the str(len(a$)) 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

ASC -- implied function

Syntax

In ARMbasic this is an automatic type conversion

But if you want to do it explicity, in your code add the following do-nothing #define

```
#define ASC(x)     x
```

Description

ARMbasic allows individual elements of a string to be accessed, and when they are assigned or compared to integer variable/constants, the ASCII value will be used.

Example

```
PRINT "the character represented by the ASCII code of 97 is:"; CHR(97)    ' will print   a

DIM astr(10) as string              ' examples of automatic type conversion complimentary to CHR

PRINT astr(0), chr(astr(0))                 ' will print    97     a
x = astr(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

- ASCII table
- HEX
- 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$(10)             ' examples of automatic type conversion complimentary to CHR
a$="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**
- **[ASC]**

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.

This may also be used during debuging to change the default base to Hexadecimal, do this by typing just HEX on the line, opposite of DEC when used this way.

Example

```
DIM text$(10)

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/BASIClib
```

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.

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 consistant with the InStr VB function, but inconsistant 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/BASIClib

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.

A$ = LEFT("this is a test",5) + RIGHT(B$,3)   ' valid string operation

A$ = LEFT("this "+b$,5)   ' NOT ALLOWED nested operation

Example

```
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)

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 consistant with the MID VB function, but inconsistant with the VB *searchee*.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

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

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

STR

Syntax

STR(*expression*)

Description

STR will convert a *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.

b$ = 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.

Example

```
DIM b$ (10)
a = 8421
b$ = 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/BASIClib

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)

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/BASIClib

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 upshifts 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 upshifts 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 charater it finds.

Incidentally, this function is the opposite of **STR** , which converts a number to a string.

Example

```
DIM a$(20)
a$ = "20xa211"
b = VAL(a$)
PRINT a$, b
```

```
20xa211   20
```

Differences from other BASICs

- None from Visual BASIC
- similar to formatting directives DEC, HEX in PBASIC

See also

- **STR**
- **HEX**
- **CHR**

Version 7 Hardware Library

With Version 7, most of the builtin firmware hardware routines have been replaced by ARMbasic routines that can be accessed by **#include <filename>.**

Version 7 is more Visual BASIC like, and frees up space for more user code (20K vs 12K in the ARMmite).

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.

Hardware Library

* (ARM peripheral access)

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.

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.

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 = 0x400    ' 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

- Hardware Library Functions

Date and Time Functions

The ARMmite USB has a provision to add a battery to keep these time functions running when power is removed. This is not the case for the ARMexpress, ARMexpress LITE, or ARMmite Wireless.

Date and Time Functions

- DAY
- HOUR
- MINUTE
- MONTH
- SECOND
- SLEEP
- TIMER
- WAIT
- WEEKDAY
- YEAR

DAY

Syntax

#include <RTC.bas> ' source in /Program Files/Coridium/BASIClib

#include <RTC17.bas> ' for the PROplus and SuperPRO LPC175x

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

Syntax

```
#include <RTC.bas>                              ' source in /Program Files/Coridium/BASIClib

#include <RTC17.bas>                ' for the PROplus and SuperPRO LPC175x

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
```

MINUTE

Syntax

#include <RTC.bas>   ' source in /Program Files/Coridium/BASIClib

#include <RTC17.bas>   ' for the PROplus and SuperPRO LPC175x

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

Syntax

```
#include <RTC.bas>                   ' source in /Program Files/Coridium/BASIClib

#include <RTC17.bas>           ' for the PROplus and SuperPRO LPC175x

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

Syntax

```
#include <RTC.bas>                    ' source in /Program Files/Coridium/BASIClib

#include <RTC17.bas>          ' for the PROplus and SuperPRO LPC175x

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
```

SLEEP

Syntax

```
#include <RTC.bas>                    ' source in /Program Files/Coridium/BASIClib

#include <RTC17.bas>           ' for the PROplus and SuperPRO LPC175x
```

SLEEP (*seconds*)

Description

Delay program execution a number of seconds.

Example

```
#include <RTC.bas>

...

FOR I=0 TO 7
  OUTPUT I
  LOW I                ' set each pin as output and low
NEXT I

FOR I=0 TO 7
  SLEEP (1)
  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

- WAIT

TIMER

Syntax

TIMER

Description

TIMER is a free running timer that increments every microsecond. Its it readable and writeable using this keyword.

Operations that require more precise timing should use the dedicated hardware routines, as interupts that are occuring for other time functions and serial input may make times using TIMER look longer than actual.

Example

```
START = TIMER< /EM >
WHILE (TIMER-START < WAIT_MICROSECONDS)
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

- SLEEP
- TIMER

WEEKDAY

Syntax

#include <RTC.bas> ' source in /Program Files/Coridium/BASIClib

#include <RTC17.bas> ' for the PROplus and SuperPRO LPC175x

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

Syntax

#include <RTC.bas>   ' source in /Program Files/Coridium/BASIClib

#include <RTC17.bas>   ' for the PROplus and SuperPRO LPC175x

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.

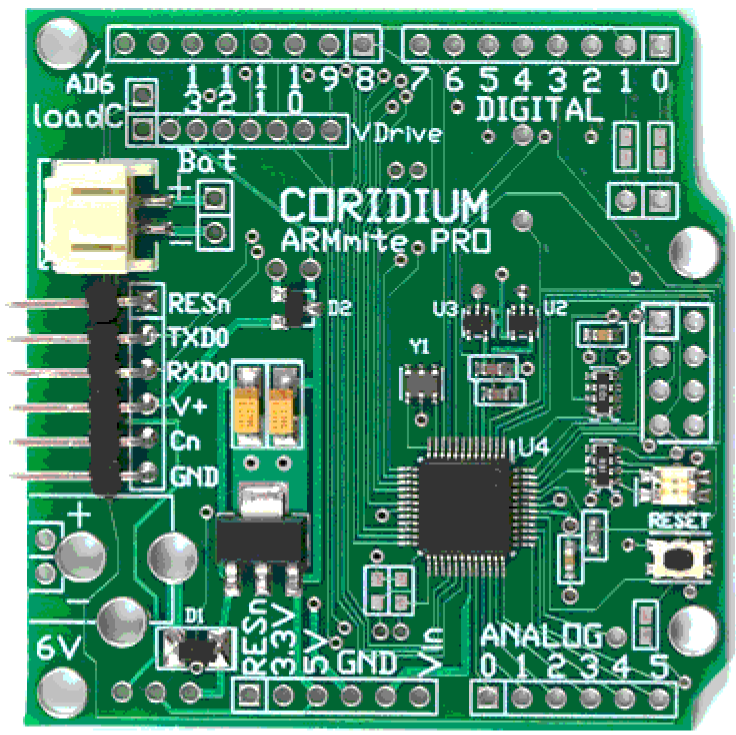
Example

```
#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

Description -- added version 7.13

The builtin 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

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

WRITE

Syntax

FUNCTION WRITE (*FlashAddr*, *Source*, *subblocksize*)

Source = *arrayname* | *stringname*

subblocksize = 0 | 256* | 512 | 1024 | 2048 | 4096 | 8192*

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).**

Writing consists of erasing the whole sector and then writing a *subblock* or all.

Erases will erase the entire sector.

subblocksize portions may be written (ARMexpress allows upto 8K but not 256). FlashAddr must be alligned to *subblocksize*.

Data is copied from a string or array to the Flash. Only fixed *subblocksize* sizes are allowed. This function does not look for 0 terminators when a string is the source.

To force a sector to be erased use a block size of 0. Once a portion is written after an erase, it can not be written again without being erased.

WRITE assumes that the sector is to be erased when first written, or when the same *subblock* as the last call to WRITE is being written. When different *subblocks* of the same sector are being written, an erase will only occur when WRITE is called with a subblocksize of 0. The WRITE routine only keeps track of which sector and sublock were last written, you must manage sectors

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

...

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
```

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_
- BAUD0_
- BAUD1

C

- COUNT

F

- FREQOUT
- FREAD

H

- HWPWM

I

- I2CIN
- I2COUT

O

- ON
- OWIN
- OWOUT

P

- PULSIN
- PULSOUT
- PWM

T

- RCTIME
- RXD
- RXD0
- RXD1

S

- SERIN
- SERINtimeout
- SEROUT
- SHIFTIN
- SHIFTOUT
- SPIBI
- SPIIN
- SPIOUT

T

- TXD
- TXD0
- TXD1

W

- WRITE

FREQOUT

Library

#include <FREQOUT.bas>

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-

```
initFREQOUT:
#include <FREQOUT.bas>
return

...

main:
   gosub initFREQOUT
```


Interface

```
 #define SIN(x)   sin_tbl(x)
 #define COS(x)   sin_tbl(x-64)

' duration is in milliseconds
' freq1 and freq2 in Hz
SUB FREQOUT(pin, duration, freq1, freq2)
```

Internals

ARMbasic uses integers, but there may be a need for certain functions that normally use floating point calculations. One of these is the cosine function, which normally operates on degrees or radians. But for simplicity and the binary world, these values and the result value have been normalized to fit in a byte value. but in this case it is expressed as -127 to +127 or the cos() multiplied by 127.

These SIN and COS functions are identical to the PBASIC versions and are used by FREQOUT. Rather than degrees or radians there are 256 divisions (360/256 degrees) which returns a value of -127 to +127 which correspond to -1 to 1 for normal sine and cosine function.

The SIN function is implemented using string, and accessed a byte at a time to generate the 256 values. COS is the SIN function shifted 90 degrees or 64 places

Example

```
#include <FREQOUT.bas>
...
```

```
'Generate a soothing dual frequency tone on pin 4 for 8 seconds
'using frequncies 2500 and 6000

FREQOUT (4, 8000, 2500, 6000)
```

COS

Syntax

#include <FREQOUT.bas>

FUNCTION COS (*expression*)   ' declared in FREQOUT.bas

Description

ARMbasic uses integers, but there may be a need for certain functions that normally use floating point calculations. One of these is the cosine function, which normally operates on degrees or radians. But for simplicity and the binary world, these values and the result value have been normalized to fit in a byte value. So rather than taking an argument of 0..359 or 0..2 p , the argument is 0-255 which is equal to the number of degrees times 0.7103 (256/360). The result would normally be between -1 and 1, but in this case it is expressed as -127 to +127 or the cos() multiplied by 127.

Example

```
#include <FREQOUT.bas>

PRINT "Please enter an angle in degrees: ";
DEBUGIN  a
r = a * 256 / 360       'Convert the degrees to "binary radians"
PRINT ""
PRINT "The cosine of a" ; a; " degree angle is"; COS ( r )
END
```

The output would look like:

```
Please enter an angle in degrees: 30

The cosine of a 30 degree angle IS 111
```

Differences from other BASICs

- Floating point routine in Visual BASIC
- The () around *expression* are enforced in ARMbasic, but not PBASIC

See also

- SIN

FREQOUT

Syntax

#include <FREQOUT.bas>

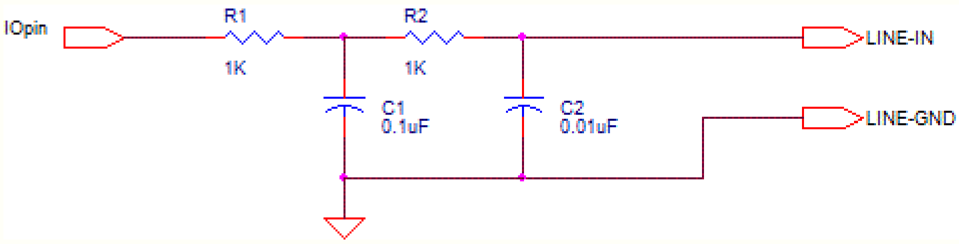
SUB FREQOUT (*pin, milliseconds, freq1, freq2) ' declared in FREQOUT.bas*

Description

Generate a sine-wave signal on *pin* for *milliseconds*.
A single frequency or mixed dual frequncy tone may be generated. Set *freq2* to 0 for a single frequency.
The IO direction of the pin will be set to output.
The output pin might be connected to a speaker or audio amplifier.

The sine wave signal is generated using **pulse width modulation**, for more details see that **link**.

A sample filter to make this signal compatible with an audio amp would be similar to that below



Example

```
#include <FREQOUT.bas>

'Generate a soothing dual frequency tone on pin 4 for 8 seconds
'using frequncies 2500 and 6000 Hz

FREQOUT (4, 8000, 2500, 6000)
```

Differences from other BASICs

- no equivalent in Visual BASIC
- none from PBASIC

See also

- **PWM**

SIN

Syntax

#include <FREQOUT.bas>   ' source in /Program Files/Coridium/BASIClib

FUNCTION SIN (*number*)   ' declared in FREQOUT.bas

Description

ARMbasic uses integers, but there may be a need for certain functions that normally use floating point calculations. One of these is the sine function, which normally operates on degrees or radians. But for simplicity and the binary world, these values and the result value have been normalized to fit in a byte value. So rather than taking an argument of 0..359 or 0..2 p , the argument is 0-255 which is equal to the number of degrees times 0.7103 (256/360).   The result would normally be between -1 and 1, but in this case it is expressed as -127 to +127 or the sin() multiplied by 127.

Example

```
PRINT "Please enter an angle in degrees: ";
DEBUGIN a
r = a * 256 / 360        'Convert the degrees to Radians
PRINT ""
PRINT "The sine of a" ; a; " degree angle is"; SIN ( r )
END
```

The output would look like:

```
Please enter an angle in degrees: 30

The sine of a 30 degree angle IS 64
```

Differences from otber BASICs

- SIN is a floating point routine in Visual BASIC
- () are enforced in **ARMbasic** not PBASIC

See also

- COS

HWPWM

This function is available on ARMmite, ARMmite Wireless, ARMexpress LITE and ARMmite PRO.

Library

#include <HWPWM.bas>

#include <HWPWM17.bas>   ' for the PROplus and SuperPRO LPC17xx based boards.

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.

If TIMER interrupts are used, then only 4 hardware PWM channels are available.

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	P2.0
channel2	P2.1
channel3	P2.2
channel4	P2.3
channel5	P2.4
channel6	P2.5

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>
...
```

```
'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)
```

HWPWM

Syntax

#include <HWPWM.bas>   ' source in /Program Files/Coridium/BASIClib

SUB HWPWM (channel, cycletime, hightime)

Description --- available on ARMmite and ARMexpress LITE but not on the original ARMexpress

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. If the value is -1, then that IO is left as a digital IO.

hightime1	IO(0)
hightime2	IO(1)
hightime3	IO(2)
hightime4	IO(3)
hightime5	IO(4)
hightime6	IO(9)
hightime7	IO(10)
hightime8	IO(11)

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. The format of the command uses 8 channel assignments, but 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. If the value is -1, then that IO is left as a digital IO.

hightime1	IO(5)
hightime2	IO(6)
hightime3	IO(3)
hightime4	not available
hightime5	IO(14)
hightime6	not available
hightime7	IO(13)
hightime8	IO(15)

PROplus SuperPRO LITE version

The PROplus/SuperPRO 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. Use the <HWPWM17.bas> include file.

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. If the value is -1, then that IO is left as a digital IO.

hightime1	P2(0)
hightime2	P2(1)
hightime3	P2(2)
hightime4	P2(3)
hightime5	P2(4)
hightime6	P2(5)

Example

```
#include <HWPWM.BAS>
...

'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)
```

Differences from other BASICs

- no equivalent in Visual BASIC
- no equivalent in PBASIC

See also

- *** peripheral access**
- **FREQOUT**
- **PWM**

I2C

Library

#include <I2C.bas>

#include <I2C17.bas>   ' use this for PROplus and SuperPRO

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 deviced 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 seperate 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 deviced 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 transfered 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>

```
#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 transfered 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 eqals 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)=$cc
outbytes(1)=$f
outbytes(2)=$6
outbytes(3)=$be
outbytes(4)=$41

present = owout (7 ,5, outbytes)
print present

outbytes(0)=$cc
outbytes(1)=$aa
outbytes(2)=$6

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)=$cc
outbytes(1)=$f
outbytes(2)=$6
outbytes(3)=$be
outbytes(4)=$41

present = owout (7 ,5, outbytes)
print present
```

Differences from other BASICs

- no equivalent in Visual BASIC
- simplified than PBASIC

See also

- OWIN

PULSE timing

Library

#include <PULSE.bas>

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 tristate (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

Syntax

#include <PULSE.bas>

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

Syntax

#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

Syntax

#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.
LOW 4
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

Syntax

#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 tristate (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**
- **FREQOUT**
- **PULSOUT**
- **Hardware Pulse Routines**

RCTIME

Syntax

#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>

INPUT 7

'... some procedure which 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

BIt Banged Serial

Library

#include <SERIAL.bas>

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

- BAUD

R

- RXD

S

- SERIN
- SERINtimeout
- SEROUT

T

- TXD

Interface

```
DIM BAUD(16)
SERINtimeout = 500000          ' timeout for bit-banged serial input in microseconds  -- this is the 0.5 second default value

FUNCTION RXD(pin)
SUB TXD(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. Interupts 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 upto 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 = RXD(pin)      ' read a character from pin as an asynchronous stream (BAUD must have been set before use)
```

RXD is a bit banged routine, so that the CPU will wait upto 0.5 seconds for a character to be received. The timeout can be changed with SERINtimeout.

```
TXD(pin, "A")       ' send an "A" to pin as an asynchronous serial stream
```

BAUD

Syntax

#include <SERIAL.bas>

DIM BAUD(*pin*)   ' declared inside SERIAL.bas

Description

BAUD (*pin*) will set the baudrate for the *pin* that will be later used by either RXD or TXD functions.

Baudrates can be upto 115.2 Kbaud for transmit, 57Kbaud for receive.

Example

```
BAUD(2) = 19200       ' set the baud rate for serial I/O on pin 2

BAUD(1) = BAUD(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

- TXD
- RXD

RXD

Syntax

#include <SERIAL.bas>

RXD (*pin*)

Description

RXD (*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 RXD, that is done using the BAUD(pin) function.

RXD will return 0-255 if there was data present. RXD will timeout after 0.5 seconds and return -1 ($FFFFFFFF) 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. Interupts 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 upto 57 Kbaud for all pins.

Example

```
#include <SERIAL.bas>

BAUD(1) = 9600      ' set the baud rate for serial I/O on pin 1

' Wait for serial input on pin 1
DO
   MyByte = RXD(1)
UNTIL MyByte >= 0
```

Differences from other BASICs

- no equivalent in Visual BASIC
- preferred alternate to SERIN of PBASIC

See also

- **BAUD**
- **TXD**
- **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 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. Interupts 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 upto 115.2 Kbaud for all pins on transmit. Receive rates to 57Kb

Example

```
' Read serial stream for 1 byte from pin 1 saving to MyByte, negative true
SERIN (1, 19200, 0, 1,MyByte)
PRINT HEX(MyByte)
' 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
io(15)=0 ' flag that we are sync'ing up
while 1
  serin (3,19200, 1, 1, a$)
  if a$(0) = 10 then exit
loop

io(15)=1 ' and that sync is complete
while 1
  serin (3,19200,1, 0, a$)
  print a$
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/BASIClib

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 upto 115.2 Kbaud for all pins

Example

```
#include <SERIAL.bas>

DIM a$(20)

a$ = "123"
SEROUT (3, 1200, 0, 3, a$)     ' sends out   123 at 1.2Kbaud, negative true
```

Differences from other BASICs

- no equivalent in Visual BASIC
- simplified from PBASIC

See also

- SERIN

TXD

Syntax

#include <SERIAL.bas>   ' source in /Program Files/Coridium/BASIClib

SUB TXD(pin, ch)

Description

TXD (*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 TXD, that is done using the BAUD(pin) array.

TXD 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. Interupts are also disabled during each byte for these operations.

Example

```
DIM A$(10)
BAUD(2) = 19200      ' set the baud rate for serial I/O on pin 2

...

A$ = "Hello World"
GOSUB PRINTSTR

...



' Send a string of characters serially out pin 2
PRINTSTR:
   I=0
   WHILE A$(I)
      TXD(2,A$(I))
      I=I+1
   LOOP

   RETURN
```

Differences from other BASICs

- no equivalent in Visual BASIC
- SEROUT in PBASIC

See also

- BAUD
- RXD
- SEROUT

Hardware Serial

UART0 and UART1 support is built into the BASIC compiler. UART1 and BAUDx support added in 7.13 firmware.

Interface

DEBUGIN *variable* | *string*

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

FUNCTION RXD0
FUNCTION RXD1

SUB TXD0 (*expr)*
SUB TXD1 (*expr)*

SUB BAUD0 (*expr)*
SUB BAUD1 (*expr)*

Description

DEBUGIN gives the programmer a way to accept strings or numbers from the USB serial port. In many BASICs this uses INPUT, but in ARMbasic INPUT is used to control the direction of one of the IO pins. So a simplified replacement of the normal BASIC INPUT has been added, called DEBUGIN.

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.

PRINT will send strings or numbers to the debug serial port (UART0), which may be displayed in BASICtools, or can be interpreted by a user program running on the PC. Simple formating is accomplished by seperating expressions with a comma (for TAB) or semicolon for no space seperation. A semicolon at the end of a PRINT suppresses carriage return.

RXD0 uses the hardware UART, so the CPU is not tied up. Also when RXD0 is read and no data is available -1 is returned immediately, RXD0 uses interupts and has a buffer of 256 characters that are filled by interrupt running in the background.

TXD0 uses the hardware serial port to send data out the USB debug port. Data is transfered into a 16 byte FIFO, when that FIFO is full the CPU will wait until space is available.

On the ARMexpress/ARMexpress LITE these routines are all limited to 19.2Kb due to the level translators for RS-232. If the connection to the ARMexpress/LITE is short (less than a couple inches), then higher baud rates can be used.

Added in version 7.13 --

BAUD0 will set the baud rate for TXD0, RXD0, the default is 19.2Kbaud.

BAUD1 will set the baud rate for TXD1, RXD1 and will enable that serial channel (in ARMmite these pins are general purpose IOs IO(0)switches to RXD1 and IO(1) switches to TXD1.

RXD1 uses the hardware UART, so the CPU is not tied up. Also when RXD1 is read and no data is available -1 is returned immediately, RXD1 uses interupts and has a buffer of 256 characters that are filled by interrupt running in the background.

TXD1 uses the hardware serial port to send data out the IO(1) on the ARMmite. Data is transfered into a 16 byte FIFO, when that FIFO is full the CPU will wait until space is available.

Example

```
'  simple example of serial write and read
BAUD0 (2400)          ' change the default baud rate
...
TXD0("X")

ch = RXD0
WHILE  ch < 0          ' wait for a character to come in
  ch = RXD0
LOOP
...
```

Differences from other BASICs

- Visual BASIC
- PBASIC has similar functions, DEBUGIN allows a string to be printed before input

BAUD0 BAUD1

Syntax

SUB BAUD0(*rate*)

SUB BAUD1(*rate*)

Description -- added in version 7.13

BAUD0 (*rate*) will set the baudrate for the SIN/SOUT pins, that will be later used by PRINT, DEBUGIN, RXD0 or TXD0 functions.

BAUD1 (*rate*) will set the baudrate for the IO(0) IO(1) pins on the ARMmite, that will be later used by either RXD1 or TXD1 functions. On reset these pins are configured as general purpose IOs, and a call to BAUD1 will configure them as UART1. The ARMexpressLITE uses pins IO5 and IO6 for UART1.

Baudrates for the LPC21xx and LPC23xx based boards are   15000/(n*16) in Kbaud

Baudrates for the LPC17xx based boards are   25000/(n*16) in Kbaud.   n is an integer

The ARMexpress/ARMexpressLITE is limited to 19.2 Kbaud by the level translators on SIN/SOUT when connecting to cables. Onboard connections for the ARMexpress/ARMexpressLITE may run faster.

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**Yahoo Forum** or the **NXP User Manuals** .

Example

```
BAUD1(19200)       ' set the baud rate and enable serial I/O on IO(0) IO(1)

BAUD0(9600)        ' set the baud rate for SIN and SOUT
```

Differences from other BASICs

- no equivalent in Visual BASIC
- no equivalent in PBASIC

See also

- **TXD0**
- **RXD0**
- **TXD1**
- **RXD1**

BAUD0 BAUD1

Syntax

SUB BAUD0(*rate*)

SUB BAUD1(*rate*)

Description -- added in version 7.13

BAUD0 (*rate*) will set the baudrate for the SIN/SOUT pins, that will be later used by PRINT, DEBUGIN, RXD0 or TXD0 functions.

BAUD1 (*rate*) will set the baudrate for the IO(0) IO(1) pins on the ARMmite, that will be later used by either RXD1 or TXD1 functions. On reset these pins are configured as general purpose IOs, and a call to BAUD1 will configure them as UART1. The ARMexpressLITE uses pins IO5 and IO6 for UART1.

Baudrates for the LPC21xx and LPC23xx based boards are 15000/(n*16) in Kbaud

Baudrates for the LPC17xx based boards are 25000/(n*16) in Kbaud. n is an integer

The ARMexpress/ARMexpressLITE is limited to 19.2 Kbaud by the level translators on SIN/SOUT when connecting to cables. Onboard connections for the ARMexpress/ARMexpressLITE may run faster.

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**Yahoo Forum** or the **NXP User Manuals** .

Example

```
BAUD1(19200)      ' set the baud rate and enable serial I/O on IO(0) IO(1)

BAUD0(9600)       ' set the baud rate for SIN and SOUT
```

Differences from other BASICs

- no equivalent in Visual BASIC
- no equivalent in PBASIC

See also

- **TXD0**
- **RXD0**
- **TXD1**
- **RXD1**

RXD0

Syntax

FUNCTION RXD0 as INTEGER

Description

RXD0 will receive a single byte of data that is shifted as an asynchronous serial stream. This function is similar to SERIN, but is a more efficient implementation.

RXD0 will return 0-255 if there was data present. or -1 ($FFFFFFFF) if there is no serial stream available on SIN. The hardware UART is used, so the CPU is not tied up, and bytes are buffered up to 256 bytes being received by an interrupt routine

ARMexpress and ARMexpressLITE-

Data is received on the SIN pin. SIN and SOUT are always negative true. UART0 of the LPC2103/06

SIN and SOUT are limited by the level translators.

ARMmite--

Pin labeled RXD0 on the schematic, UART0 of the LPC2103. Data is always positive true.

Baudrates can be upto 115.2 Kbaud.

Example

```
' Wait for serial input on pin UART0
DO
   MyByte = RXD0
UNTIL MyByte >= 0
```

Differences from other BASICs

- no equivalent in Visual BASIC
- preferred alternate to SERIN of PBASIC

See also

- **TXD0**
- **BAUD0**

RXD1

Syntax

FUNCTION RXD1 as INTEGER

Description -- added in Version 7.13

RXD1 will receive a single byte of data that is shifted as an asynchronous serial stream.

RXD1 will return 0-255 if there was data present. or -1 ($FFFFFFFF) if there is no serial stream available. The hardware UART is used, so the CPU is not tied up, and bytes are buffered up to 256 bytes being received by an interrupt routine.

ARMmite--

Pin labeled IO0 on the schematic, UART1 of the LPC2103.

ARMexpress LITE

Pins labeled IO5

Data is positive true. Baudrates can be upto 115.2 Kbaud.

Example

```
BAUD1 = 19200                    ' set baud rate and enable channel
...

' Wait for serial input on pin UART1
DO
   MyByte = RXD1
UNTIL MyByte >= 0
```

Differences from other BASICs

- no equivalent in Visual BASIC
- preferred alternate to SERIN of PBASIC

See also

- TXD1
- BAUD1

TXD0

Syntax

SUB TXD0 (*char*)

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)

The hardware serial port is used, so the CPU is not tied up. So when a byte is sent it is placed into the UART0 FIFO, but if the 16 byte FIFO is full then the CPU will wait until space is available.

The compiler is also backward compatable with the syntax -- TXD0 = char

Example

```
DIM A$(10)

...

A$ = "Hello World"
GOSUB PRINTSTR

...



' Send a string of characters serially out UART0
PRINTSTR:
   I=0
   WHILE A$(I)
      TXD0 ( A$(I) )
      I=I+1
   LOOP

   RETURN
```

Differences from other BASICs

- no equivalent in Visual BASIC
- preferred alternate to SEROUT of PBASIC

See also

- BAUD0
- RXD0
- Hardware Serial Support

TXD1

Syntax

SUB TXD1 (*char*)

Description -- added in version 7.13

The data is transmitted on the IO(1) pin on the ARMmite.

On the ARMexpress LITE data is transmited on pin labeled IO6

Data is positive true.

The hardware UART1 port is used, so the CPU is not tied up. So when a byte is sent it is placed into the FIFO, but if the 16 byte FIFO is full then the CPU will wait until space is available.

Example

```
DIM A$(10)


...
BAUD1 = 19200      ' set baud rate and enable channel

...

A$ = "Hello World"
GOSUB PRINTSTR1


' Send a string of characters serially out UART0
PRINTSTR1:
   I=0
   WHILE A$(I)
      TXD1 ( A$(I) )
      I=I+1
   LOOP

   RETURN
```

Differences from other BASICs

- no equivalent in Visual BASIC
- preferred alternate to SEROUT of PBASIC

See also

- BAUD1
- RXD0
- Hardware Serial Support

SHIFTIN, SHIFTOUT

Library

#include <SHIFT.bas>

S

- SHIFTIN
- SHIFTOUT

Inteface

DIM shiftValues(MAXshiftARRAY) ' values to be shifted in or 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 of each shiftValues defines the number of bits that will be shifted out (1 - 32). 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 of each shiftValues defines the number of bits that will be shifted out (1 - 32). If shiftCounts is 0, it is assumed to be 8.

- Mode = 0   data is shifted out MSB first
- Mode = 1   data is shifted out LSB first

NOTE* these shift modes are compatable 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

shiftValues(0) = 2
shiftCounts(0) = 3

shiftValues(1) = &H1b
shiftCounts(1) = 5

shiftValues(2) = y
shiftCounts(2) = 8

io(6)=0                    ' used asCS
shiftout (3,4,1,3)         ' set reg &H1B to y
io(6)=1

shiftValues(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 = shiftValues(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 transfered with SPIIN or SPIOUT. CS is manually controlled in this example (it is also positive true).

```
#include <SHIFT.bas>
#include <SPI.bas>
...

DIM inlist(20) as string
DIM outlist(20) as string

'  mixed SPI, SHIFT example for a 93LC46 connected to pins 11-14

high 14                            ' CS to 93LC46
shiftValues(0) = $260
shiftCounts(0) = 10
SHIFTOUT(12,13,0,1)                '  write enable
low 14

shiftValues(0) = $280              ' count still 10
outlist(0) = $41
high 14
SHIFTOUT(12,13,0,1)                ' set write to address 0
SPIOUT (-1, 13, 12, 0, 1, outlist) ' send a byte of data
low 14

wait(20)                           ' allow for write time

high 14
shiftValues(0) = $300
SHIFTOUT(12,13,0,1)                '   read addr 0
SPIIN (-1, 11, 13, 12, 0, -1, "", 10, inlist) ' read 10 bytes of data
low 14
```

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 2 arrays defined in the #include file

DIM shiftValues(MAXshiftARRAY) ' values to be shifted in or out

DIM shiftCounts(MAXshiftARRAY)

While most other hardware functions use bytes, SHIFTIN is oriented for bit control. The shiftCounts of each shiftValues defines the number of bits that will be shifted out (1 - 32). 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

shiftValues(0) = 2
shiftCounts(0) = 3

shiftValues(1) = $1b
shiftCounts(1) = 5

shiftValues(2) = y
shiftCounts(2) = 8

io(6)=0                 ' used asCS
shiftout (3,4,1,3)      ' set reg $1B to y
io(6)=1

shiftValues(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 = shiftValues(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 of each shiftValues defines the number of bits that will be shifted out (1 - 32). If shiftCounts is 0, it is assumed to be 8.

- Mode = 0 data is shifted out MSB first
- Mode = 1 data is shifted out LSB first

NOTE*** these shift modes are compatable 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 inlist(20) as string
DIM outlist(20) as string

'  mixed SPI, SHIFT example for a 93LC46 connected to pins 11-14

high 14                          ' CS to 93LC46
shiftValues(0) = $260
shiftCounts(0) = 10
SHIFTOUT(12,13,0,1)              '  write enable
low 14

shiftValues(0) = $280            ' count still 10
outlist(0) = $41
high 14
SHIFTOUT(12,13,0,1)              ' set write to address 0
SPIOUT (-1, 13, 12, 0, 1, outlist) ' send a byte of data
low 14

wait(20)                         ' allow for write time

high 14
shiftValues(0) = $300
SHIFTOUT(12,13,0,1)              '   read addr 0
SPIIN (-1, 11, 13, 12, 0, -1, "", 10, inlist) ' read 10 bytes of data
low 14
```

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(&HFF)+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 bidirectional 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/BASIClib

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/BASIClib

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>

FUNCTION Fpu_ReadWord
 Fpu_ReadDelay
 str$(0) = 0
 SPIIN(FpuCS, FpuIn, FpuClk, FpuOut, 0, 0, str$, 2, str$)
 return (str$(0)<<8) + str$(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>

SUB Fpu_Write(bval1)
 str$(0) = bval1
 SPIOUT(FpuCS, FpuClk, FpuOut, 0, 1, str$)
END SUB
```

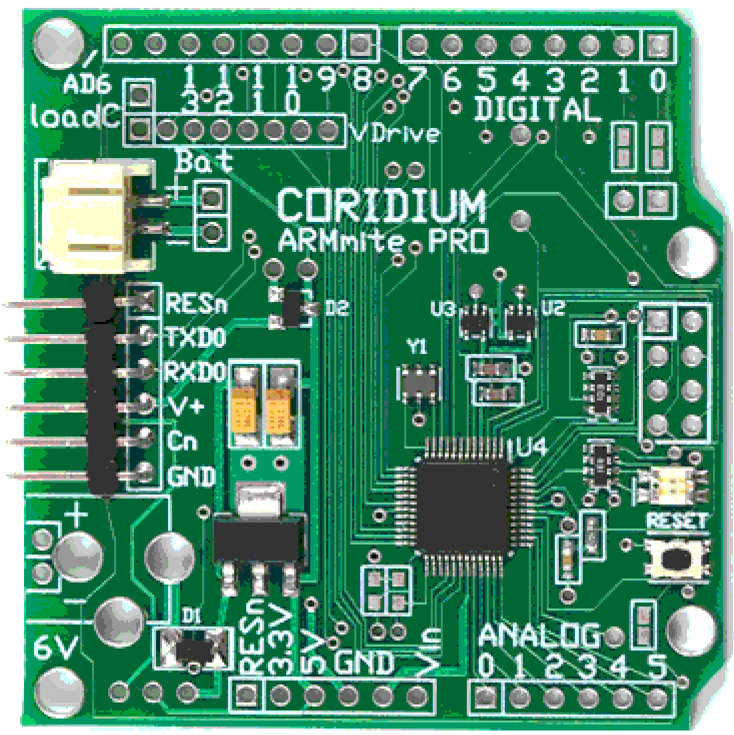
Differences from other BASICs

- 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 ON	

ADDRESSOF

Syntax

ADDRESSOF variable_name

or

ADDRESSOF subroutine_name

Description

ADDRESSOF will return the address of a variable or subroutine.

Example

```
sub print1111
  print 1111
endsub

main:
  fpointer = ADDRESSOF print1111

  call ( fpointer )
```

Differences from other BASICs

- similar to VB
- no equivalent in PBASIC

See also

- CALL

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.

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
  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.

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 function of 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
```

Differences from other BASICs

- no equivalent in Visual BASIC
- no equivalent in PBASIC

See also

- ON

ON (version 7.30 and later on ARM7 parts)

For PROplus and SuperPRO see INTERRUPT SUB

_
_
_

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. This pin is available on the wireless ARMmite, but if you intend to use it, make sure it is pulled high normally, otherwise when the board is reset it will go into the download C mode and will not run your BASIC program. 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

```
#defineVICIntEnClear  *$FFFFF014

#define TIMERoff   VICIntEnClear = $20
#define EINT0off   VICIntEnClear = $4000
#define EINT1off   VICIntEnClear = $8000
#define EINT2off   VICIntEnClear = $10000
```

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

- VB ???
- no equivalent in PBASIC

See also

- GOTO
- RETURN

Logic Scope

Logic Scope Timed Samples User Defined Sampling Stand Alone Analyzer	

Timed sampling with Logic Scope

Timing setup

The ARMexpress/mite can sample the upto 32 data lines at nearly 1 MHz rates in BASIC. The software library LogicScope.bas is used to coordinate this sampling. Other sample rates that are multiples of 40uSec 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 IOlines. 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

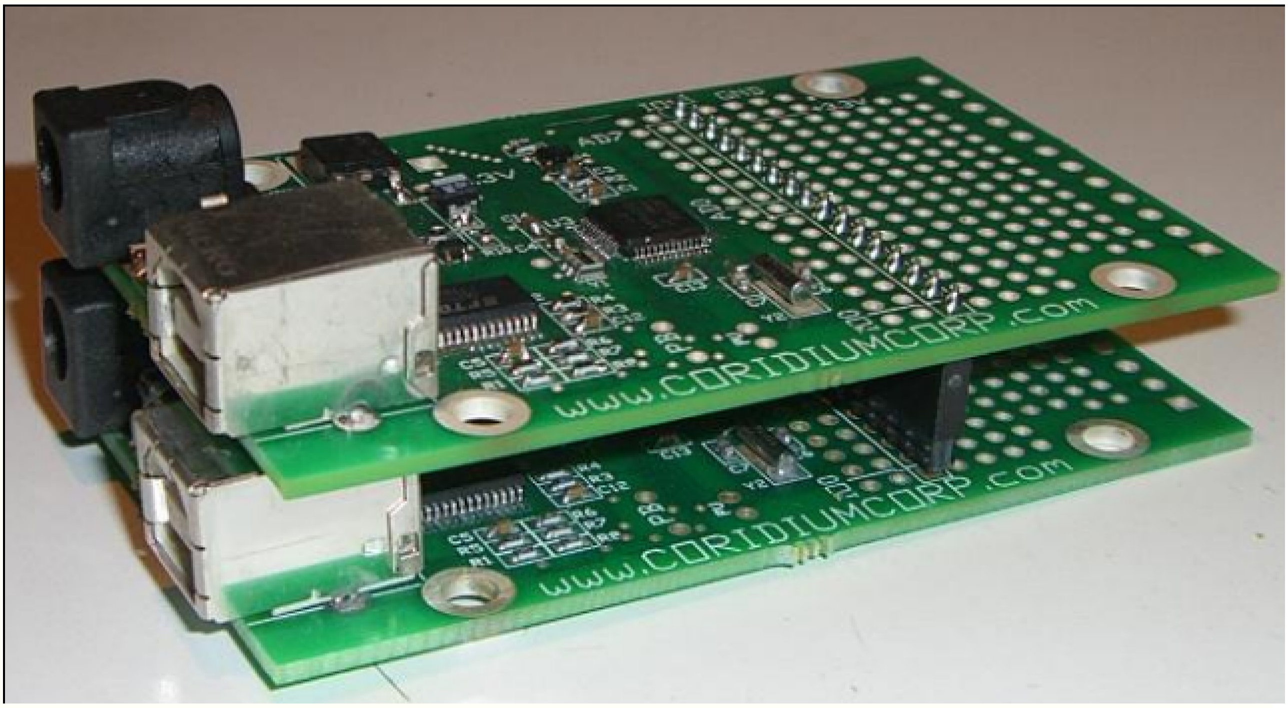
-

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

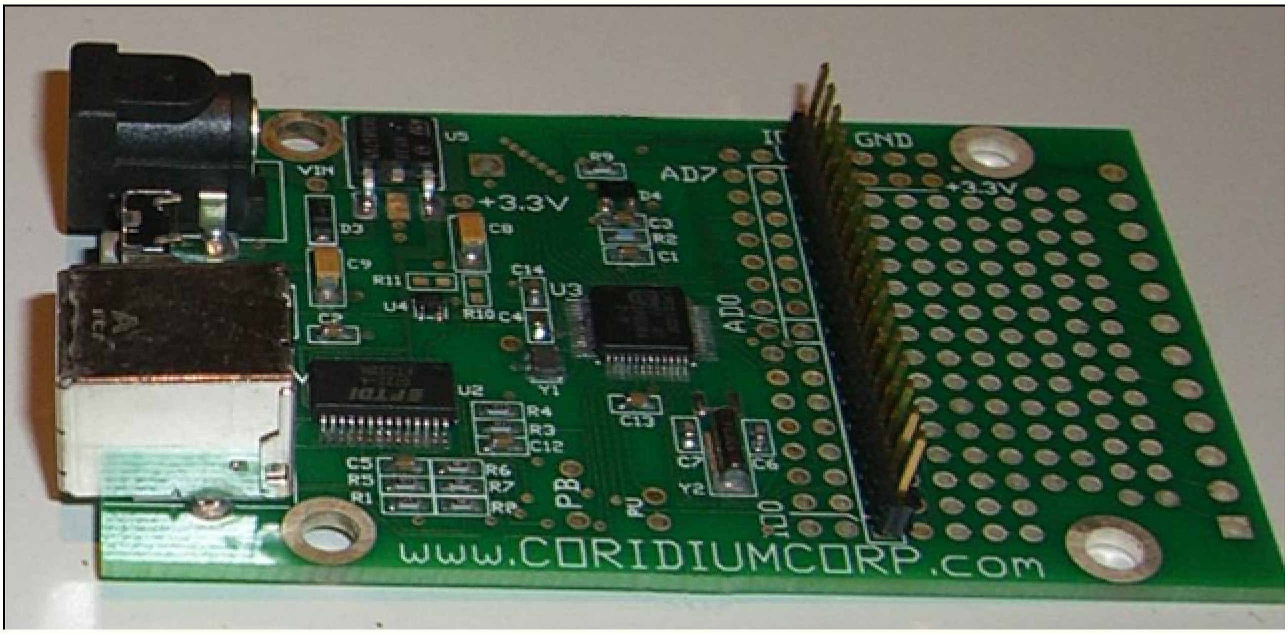
Stand Alone Logic Scope



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.

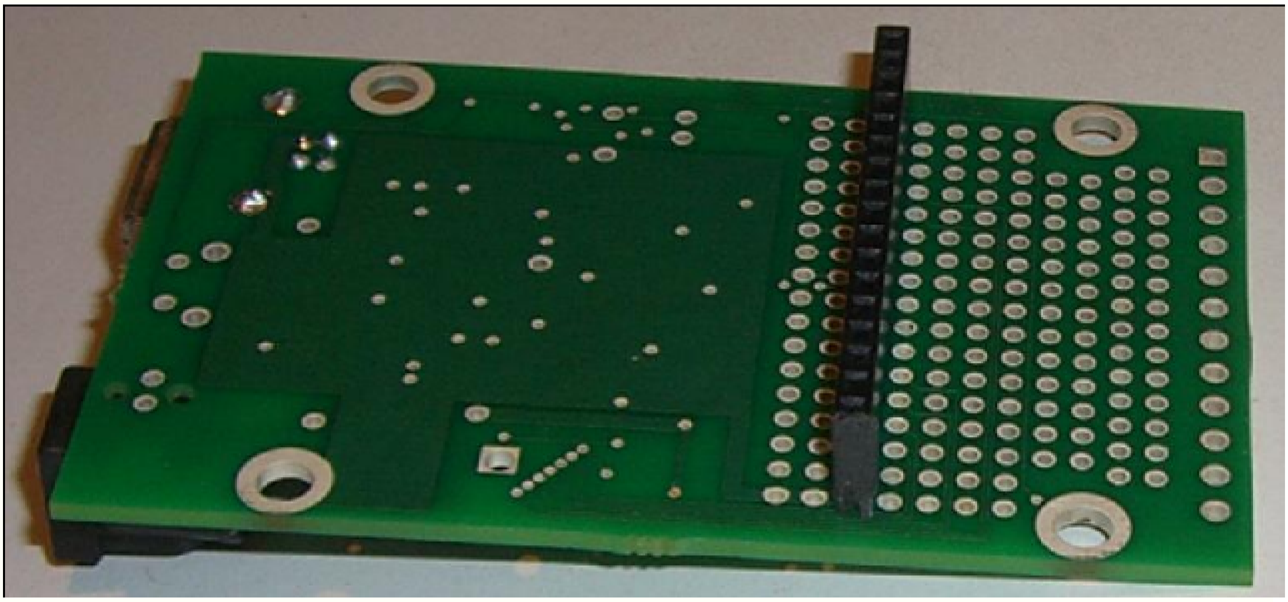
Board under test setup

..

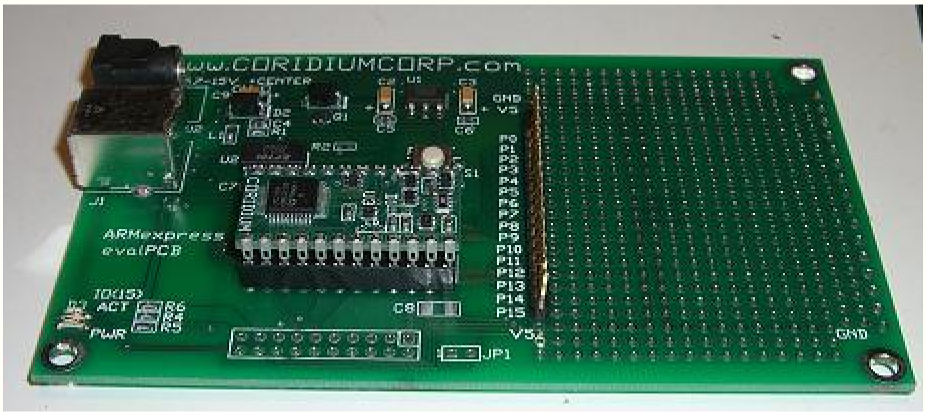


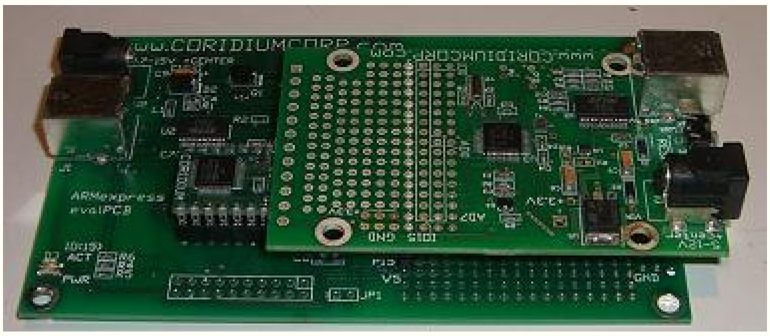
Analyzer board setup

..



ARMmite sampling data from ARMexpress/ ARMexpress LITE evaluation board

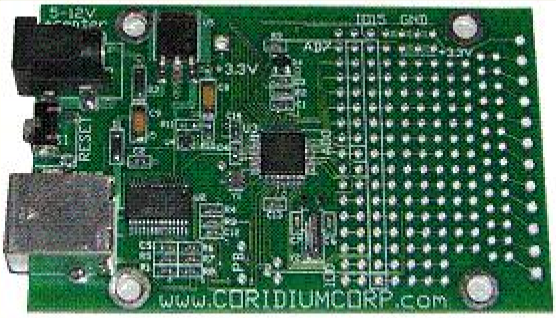




keyw ords: Logic Scope

–

Pin Control Functions



Pin Control Functions	
AD BYTEBUS -- ARMweb only DIR HIGH IN INPUT IO LOW OUT OUTPUT Port P0..P4	

AD

Syntax

FUNCTION AD (*expression*)

Description --- not available on the original ARMexpress

ARMmite and ARMmite PRO version

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 builtin 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.

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)

Stand-Alone Compilers

Because the hardware is not compatible between LPC types, this must be implemented as a FUNCTION in BASIC and is not part of the firmware.

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)

Syntax

BYTEBUS (control)

Description

BYTEBUS reads or writes the 8 bit + 2 control lines on Port1 of the LPC2138. The control field sets the state of the 2 control lines, with the intention of line 0 being used as a R/W line and line 1 being used as a CS line-

0 -- set control line 0 low, and pulse line 1 low
1 -- set control line 0 high, and pulse line 1 low
2 -- set control line 0 low, and pulse line 1 high
3 -- set control line 0 high, and pulse line 1 high

4 -- use the 10 lines as a block of inputs or outputs (added in version 7 firmware)

For 0-3:

The pulsewidth on line 1 is 250 nsec for write, and 550 nsec for read.

Back to back operations occur 2.4 usec apart for writes, 2 usec for read.

None of these lines are driven on reset, and should be biased with resistors if devices connected to this bus require it.

Example

```
'write to byte bus - negative true CS and W
BYTEBUS(0)  = $A5

'read from byte bus - negative true CS, R-notW line
x = BYTEBUS(1)
```

block control added in version 7 firmware-

```
'write to 10 pins as a block
BYTEBUS(4)  = $2A5

'read from 10 pins as a block
x = BYTEBUS(4)
```

Differences from other BASICs

- no equivalent in Visual BASIC
- no equivalent in PBASIC

See also

- HIGH

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 pullups
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

DIR (expression) can be used to set or read the direction of the 16 configurable pins. If DIR (expression) is 1 then the corresponding pin is an output. If the value is 0 then that pin is an input.

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, ARMmite PRO, ARMexpress and ARMexpress LITE these pin numbers correspond to the pin numbers shown in the **Hardware Section** . For the ARMweb, DINkit, SuperPRO these pin numbers correspond only to the Port 0 assigned by NXP, for instance DIR 3 corresponds to P0.3

For port pins after Port 0, use the **P1 .. P4 commands**, or a #define FIO0DIR.

Example

```
' Set pin 4 as an input
DIR(4) = 0

' Set pin 12 as an output
DIR(12) = 1
```

Differences from other BASICs

- no equivalent in Visual BASIC
- equivalent to DIR0..15 in PBASIC

See also

- **INPUT**
- **OUTPUT**

HIGH

Syntax

HIGH *expression*

Description

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 compatablity.

For the ARMmite, ARMmite PRO, ARMexpress and ARMexpress LITE these pin numbers correspond to the pin numbers shown in the**Hardware Section** . For the ARMweb, DINkit, SuperPRO these pin numbers correspond only to the Port 0 assigned by NXP, for instance HIGH 3 corresponds to P0.3

For port pins after Port 0, use the **P1 .. P4 commands**.

Example

```
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

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 bitwise 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, ARMmite PRO, ARMexpress and ARMexpress LITE these pin numbers correspond to the pin numbers shown in the**Hardware Section** . For the ARMweb, DINkit, SuperPRO these pin numbers correspond to the port assigned by NXP, for instance IN(3) corresponds to P0.3

For port pins after port 0, 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 and OUTPUT were added for PBASIC compatability, 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, ARMmite PRO, ARMexpress and ARMexpress LITE these pin numbers correspond to the pin numbers shown in the **Hardware Section** . For the ARMweb, DINkit, SuperPRO these pin numbers correspond only to the Port 0 assigned by NXP, for instance INPUT 3 corresponds to P0.3

For port pins after Port 0, use the **P1 .. P4 commands**, or a #define FIO0DIR.

Example

```
INPUT (0)     ' this will make pin 0 an input
```

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 a more complex way to access or control the pins. When IO (*expression*) is read, the pin corresponding to *expression* is converted to an input and the value on that pin is returned.

When assiging a value to IO(*expression*), 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 bitwise 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, HIGH or LOW.

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, ARMmite PRO, ARMexpress and ARMexpress LITE these pin numbers correspond to the pin numbers shown in the **Hardware Section** . For the ARMweb, DINkit, SuperPRO these pin numbers correspond only to the Port 0 assigned by NXP, for instance IO(3) corresponds to P0.3

For port pins after Port 0, use the **P1 .. P4 commands**, or a #define FIO0DIR.

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

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 compatablity.

For the ARMmite, ARMmite PRO, ARMexpress and ARMexpress LITE these pin numbers correspond to the pin numbers shown in the **Hardware Section** . For the ARMweb, DINkit, SuperPRO these pin numbers correspond only to the Port 0 assigned by NXP, for instance LOW 3 corresponds to P0.3

For port pins after Port 0, use the **P1 .. P4 commands**, or a #define FIO0DIR.

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 hign 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, ARMmite PRO, ARMexpress and ARMexpress LITE these pin numbers correspond to the pin numbers shown in the **Hardware Section** . For the ARMweb, DINkit, SuperPRO these pin numbers correspond only to the Port 0 assigned by NXP, for instance OUT(3) corresponds to P0.3

For port pins after Port 0, use the **P1 .. P4 commands**, or a #define FIO0DIR.

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.

INPUT and OUTPUT were added for PBASIC compatability, 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.

For the ARMmite, ARMmite PRO, ARMexpress and ARMexpress LITE these pin numbers correspond to the pin numbers shown in the **Hardware Section** . For the ARMweb, DINkit, SuperPRO these pin numbers correspond only to the Port 0 assigned by NXP, for instance OUTPUT 3 corresponds to P0.3

For port pins after Port 0, use the **P1 .. P4 commands**, or a #define FIO0DIR.

Example

```
' Set pin 9 as an output
OUTPUT (9)
```

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 bitwise 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.

Example

On the SuperPRO and PROplus:

```
#define FIO1DIR  *&H2009C020          ' or use #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)
```

On the ARMweb or DINkit:

```
#define FIO1DIR       *&H3FFFC020    ' or use #include <LPC21xx.bas>
#define SCB_SCS      *&HE01FC1A0


SCB_SCS  = 3          ' required to enable port1 for firmware before 7.47


' 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**

Miscellaneous

Miscellaneous

Data Abort

Prefetch Abort

Undefined Routine

Description

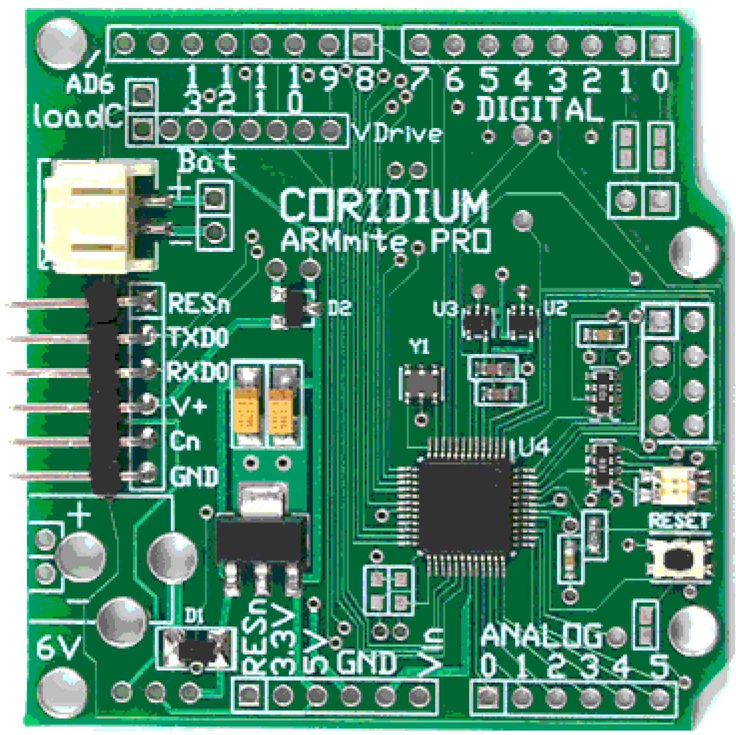
Data Aborts are generated when a user's BASIC program accesses non-existant memory. One way is accessing an array with an index that is larger than available RAM space. Another is using a pointer for hardware access, but with a value that does not correspond to a valid location.

Prefetch aborts indicate an attempt to access an instruction from non-existant 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-existant 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.

Hardware Specs



ARMmite Pin Description

24 pins available to the user, 8 of which can be analog inputs

IO0 IO1 IO2 IO3 IO4 IO5 IO6 IO7 IO8 IO9 IO10 IO11 IO14 IO15	P0.9 P0.8 P0.30 P0.21 P0.20 P0.29 P0.4 P0.5 P0.6 P0.7 P0.13 P0.19 P0.16 P0.15	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	P0.18 P0.17			Input/Outputs -- user controlled Open drain 4mA pulldown when configured as Outputs 5V tolerant
AD0 AD1 AD2 AD3 AD4 AD5 AD6 AD7	P0.22 P0.23 P0.24 P0.10 P0.11 P0.12 P0.25 P0.26	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.

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.

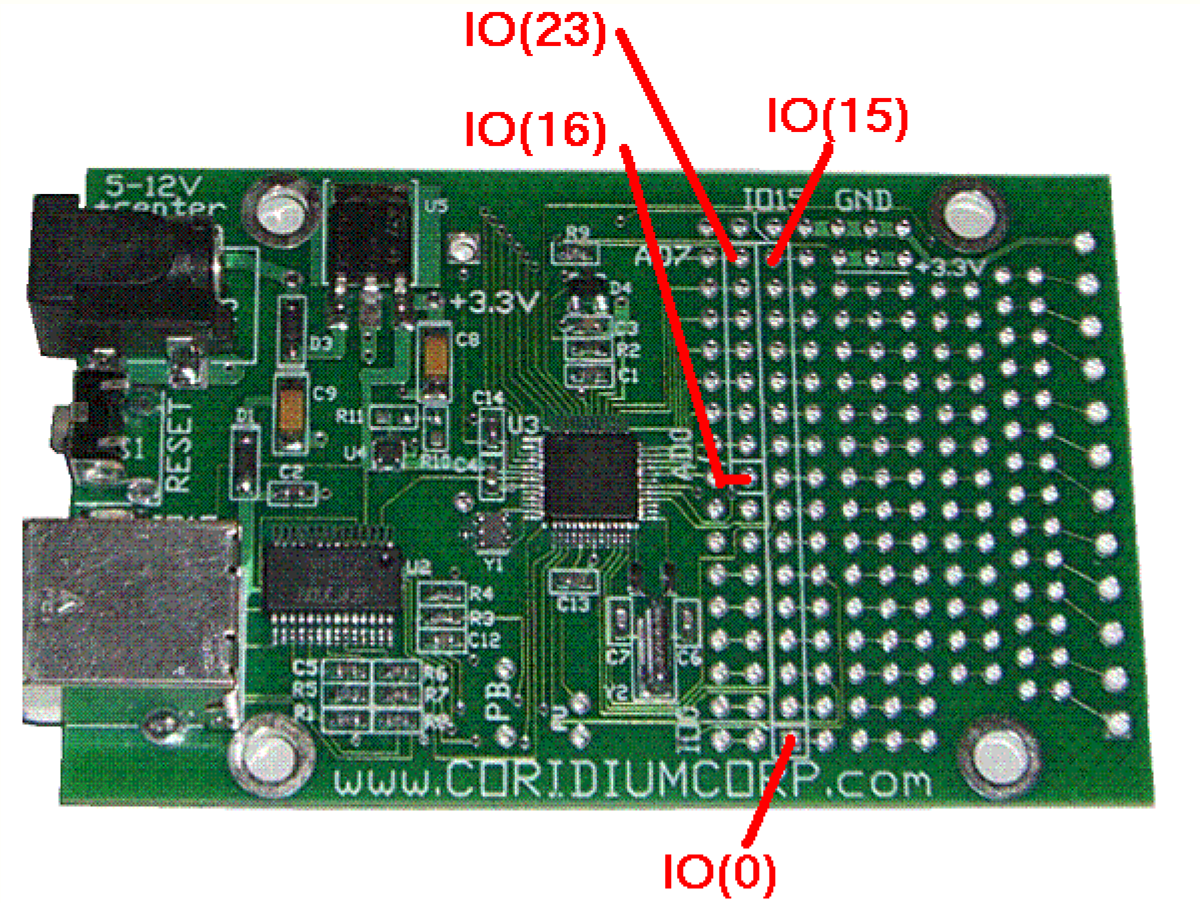
Power connection

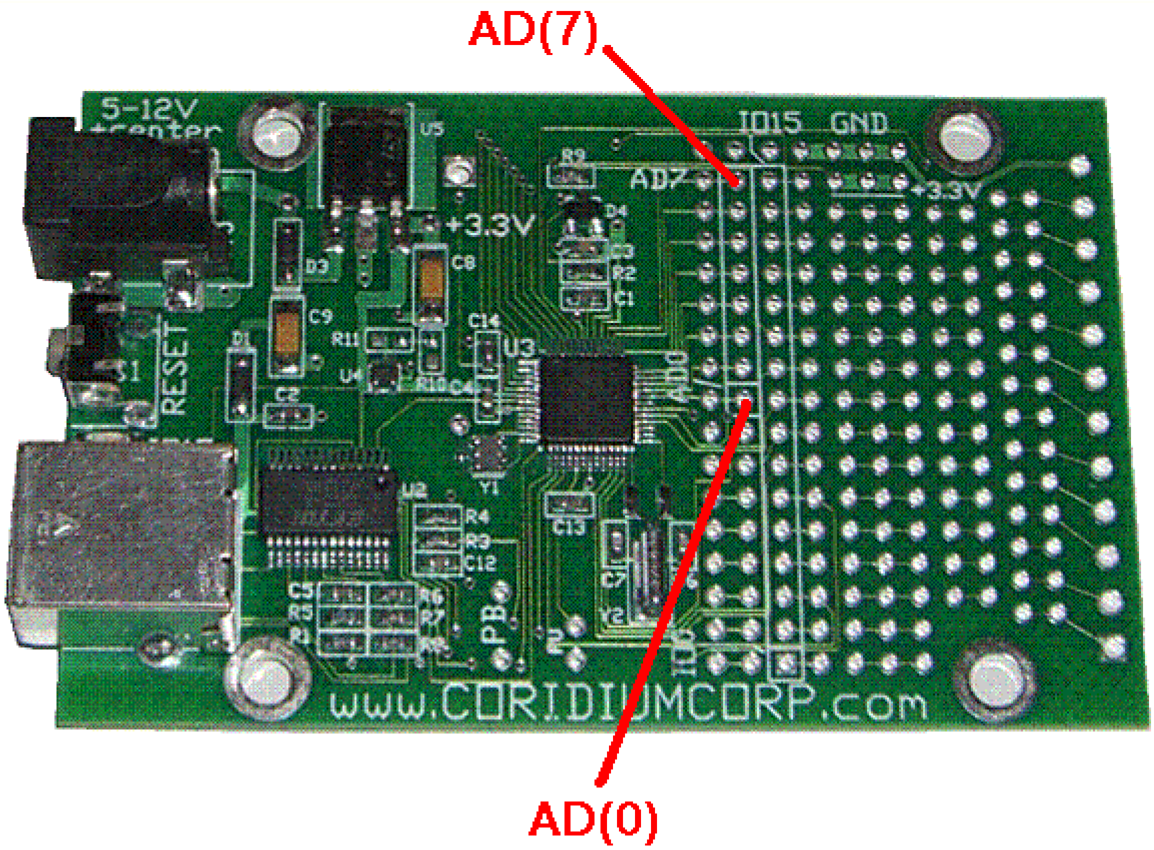
Power when not being supplied by a USB connection uses a 2.1mm barrel connector (Cui PJ-002A). Diodes allow both USB and seperate power to be connected simultaneously. If you are using an unregulated wall transformer, you must check the open circuit voltage and it MUST be less than 12V.

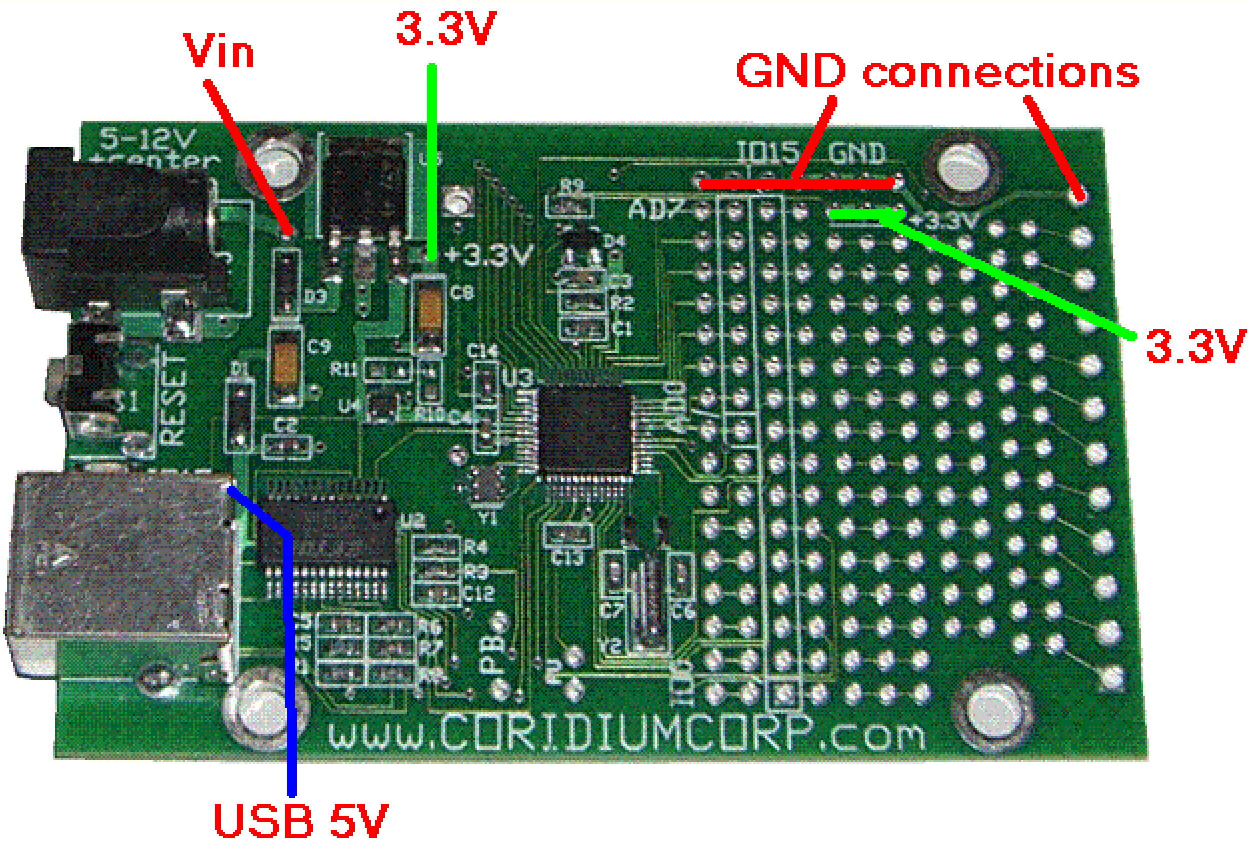
Pin spacing

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

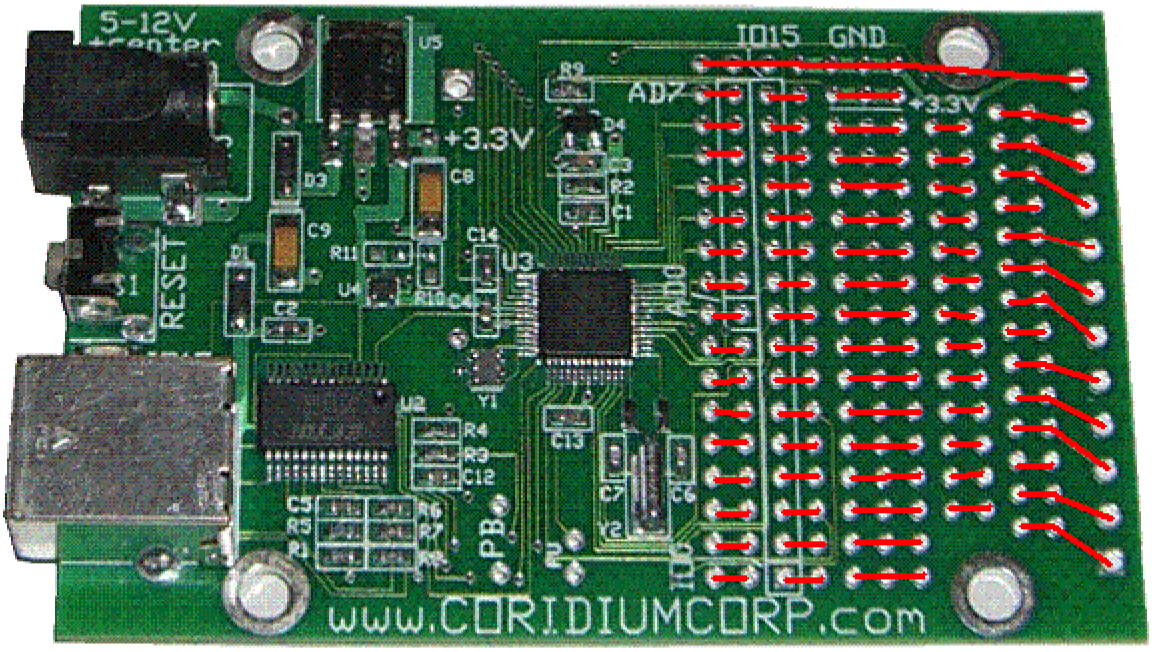


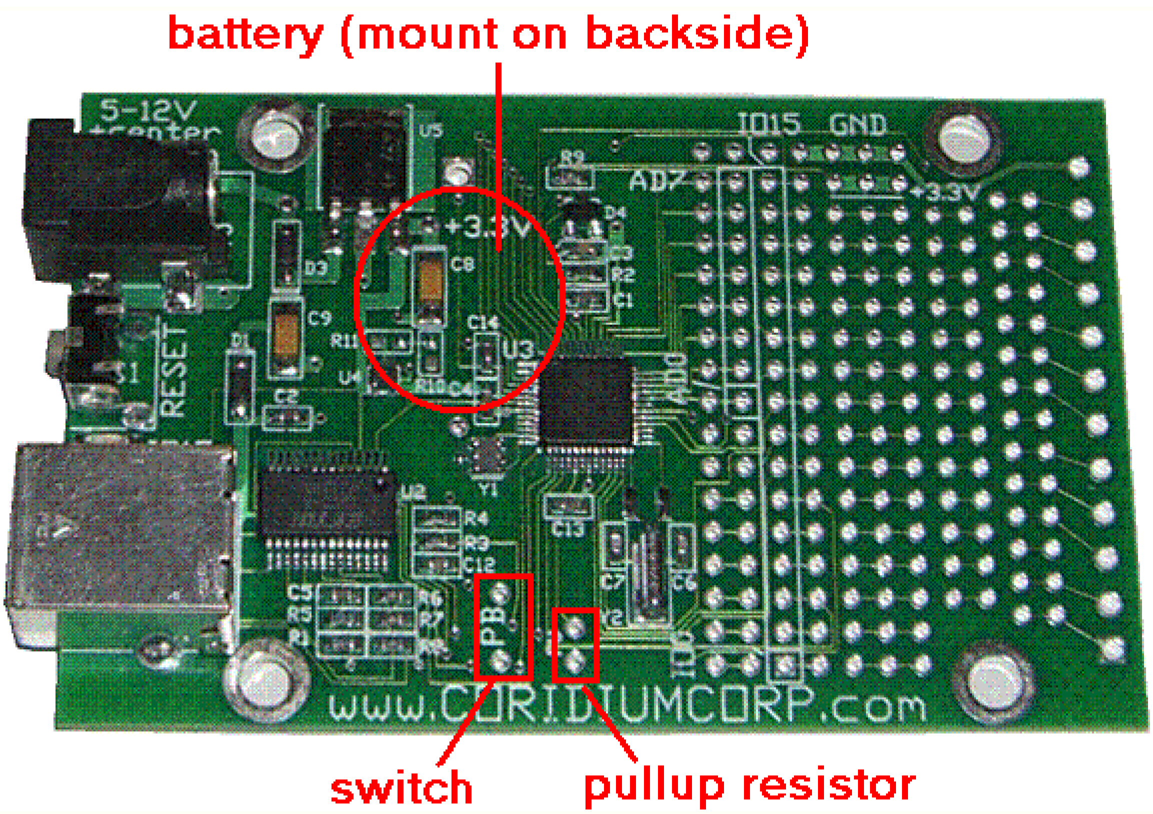




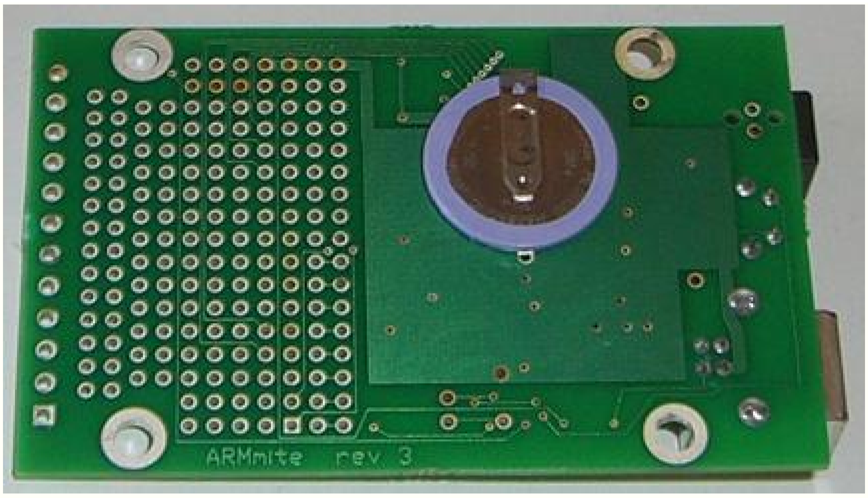
When USB power is not used, a 5-12V supply is required. If 5V is required for some portion of your circuit, it is suggested that a regulated 5V supply be used for input power. These are available from SparkFun.

PROTOTYPE Connections

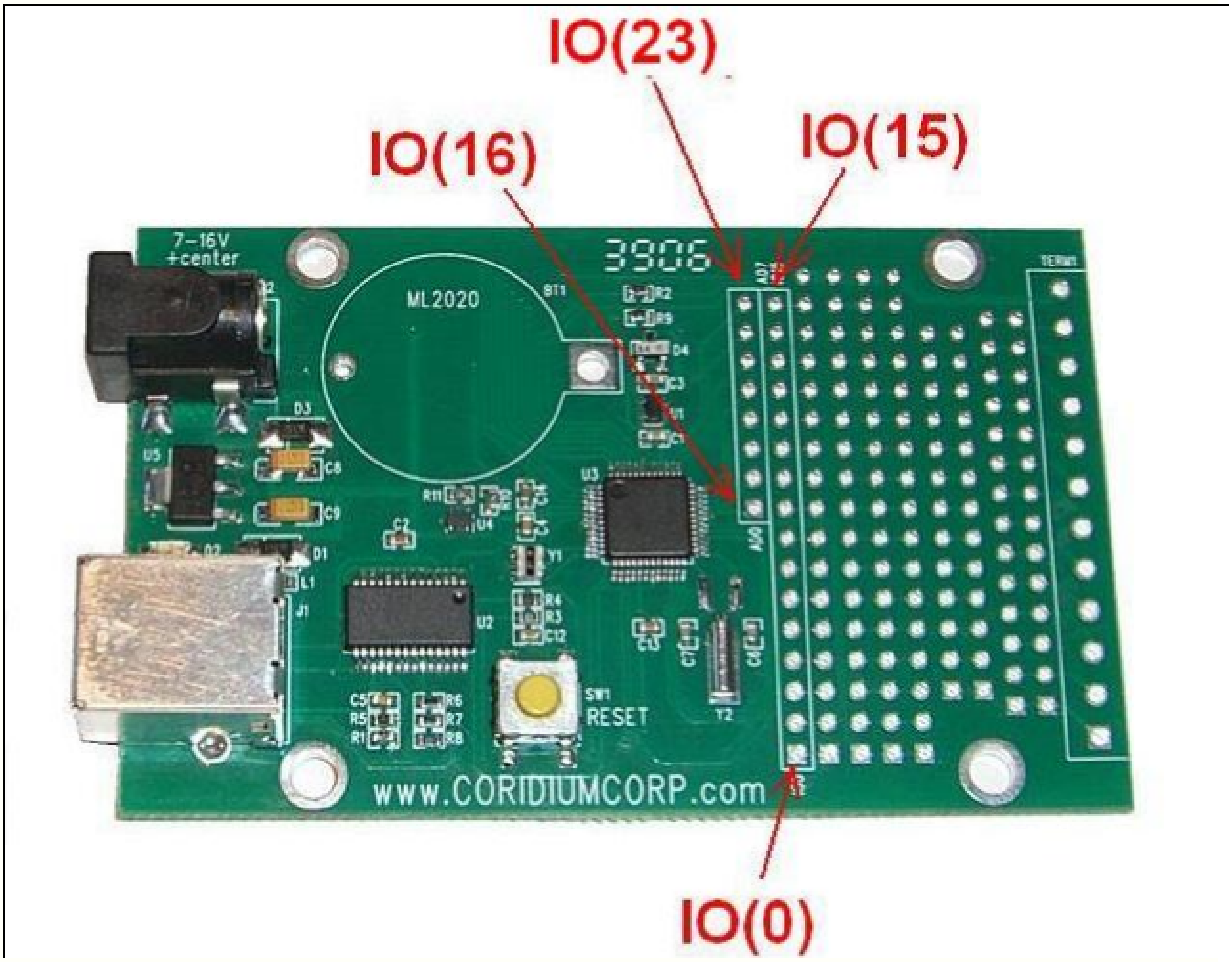


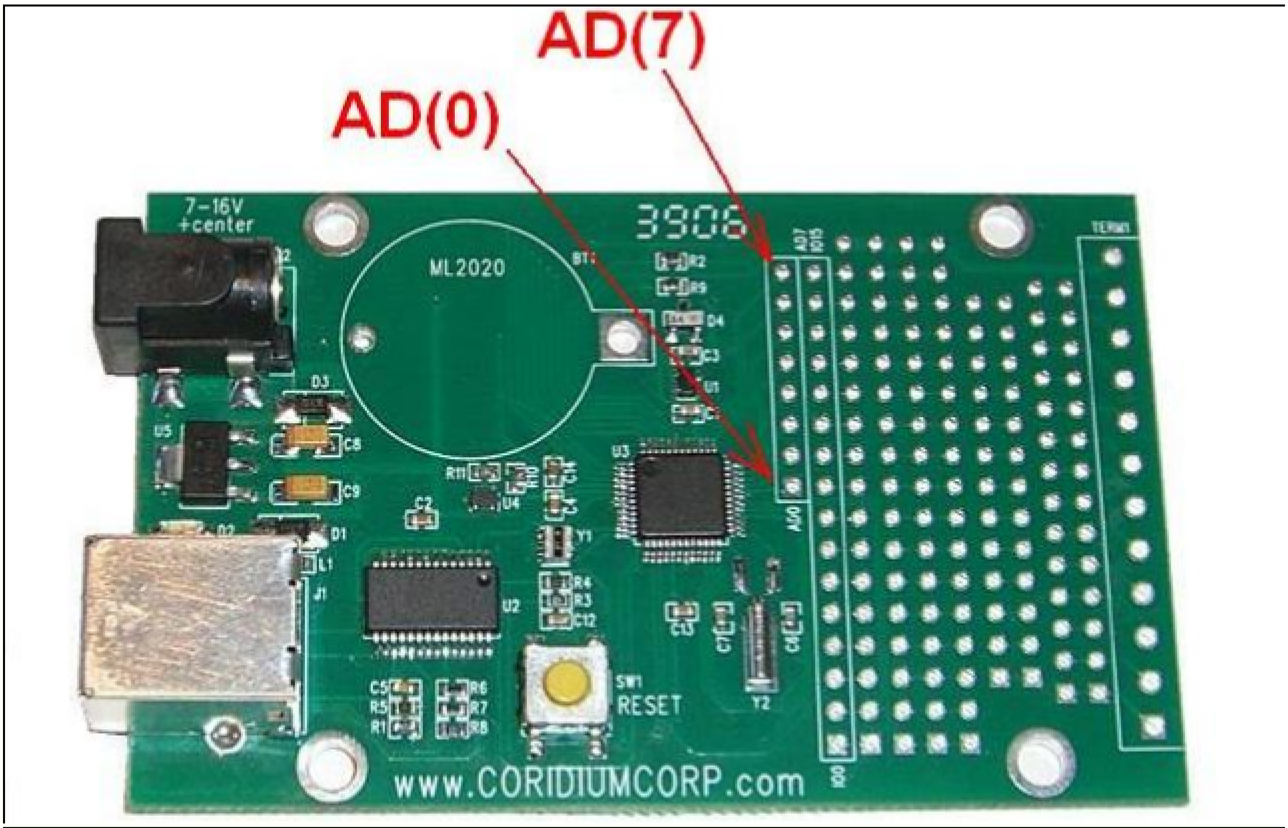


A push button switch and pullup resistor can also be mounted (connected to IO(2)). The optional battery for the real time clock (Panasonic ML2020) can be mounted on the back of the PCB. The VL2020/HFN will also work, though it is more expensive and has less power.

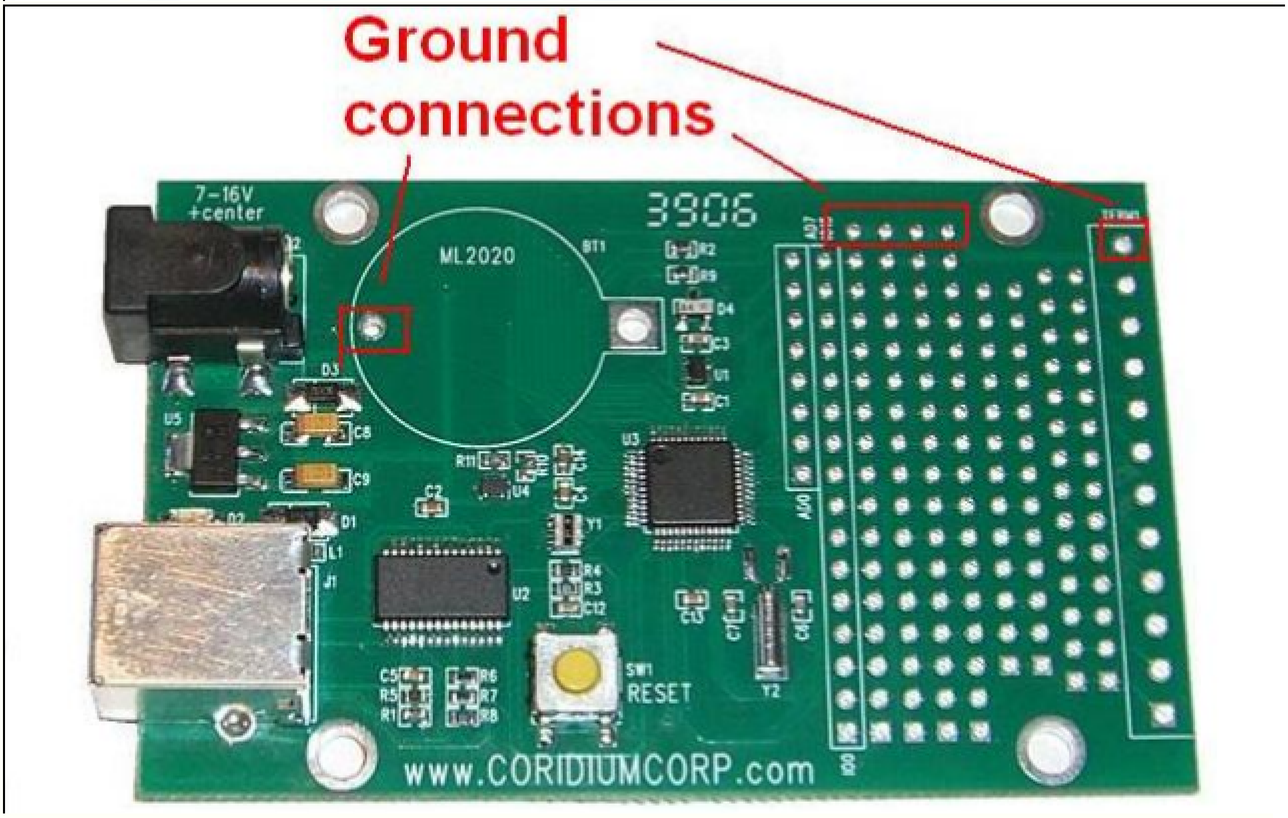

ARMmite rev 3

REV 2

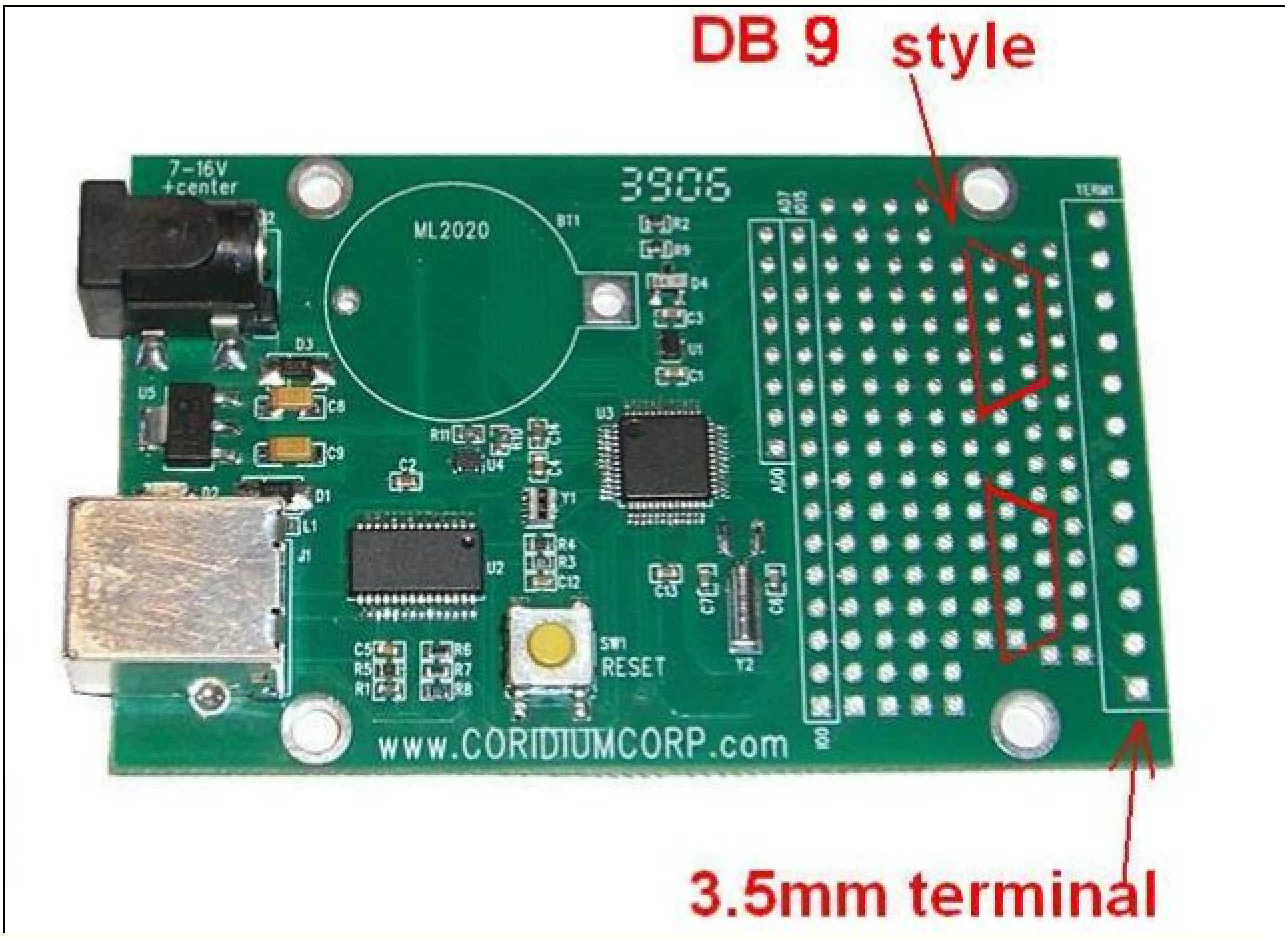

IO(23)
IO(16)
IO(15)
7-16V +center
ML2020
3906
RESET
www.CORIDIUMCORP.com
IO(0)



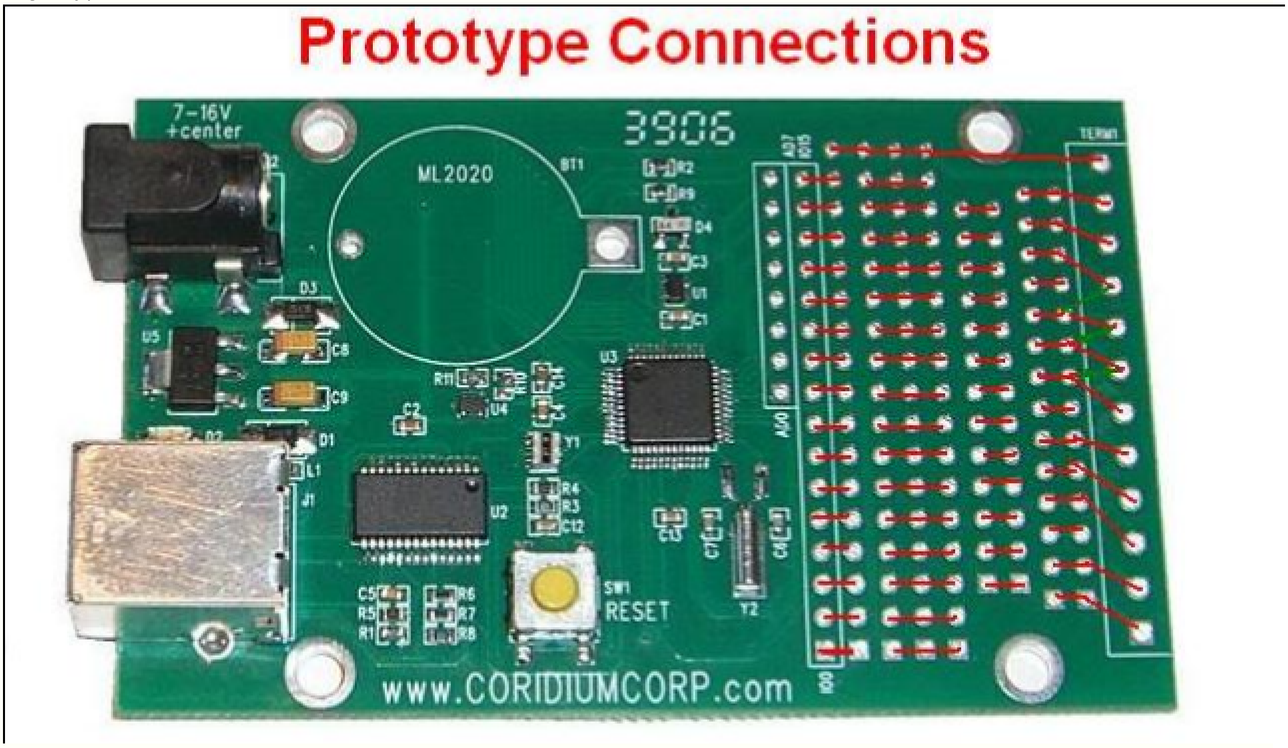
AD(7)
AD(0)
7-16V
+center
ML2020
3906
RESET
www.CORIDIUMCORP.com



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)



ARMmite PRO Pin Description

The ARMmite 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 **MakerShed** . More details on **these connections here**.

Pins available to the user, 7 of which can be analog inputs

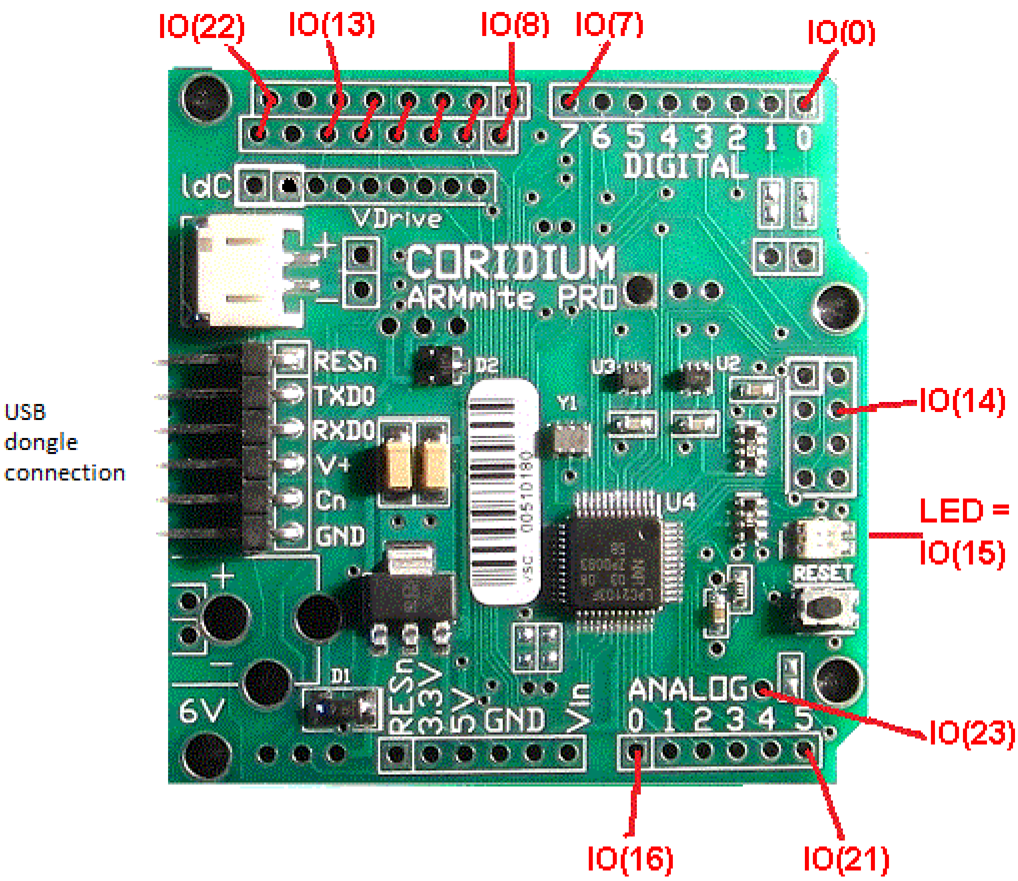
IO0	P0.9	RXD1	PWM1	Input/Outputs -- user controlled
IO1	P0.8	TXD1	PWM2	
IO2	P0.27			0-3.3V level
IO3	P0.19		PWM8	
IO4	P0.28			4mA drive when configured as Outputs
IO5	P0.21		PWM4	
IO6	P0.5			5V tolerant - use limiting resistor when connecting to a 5V supply
IO7	P0.29			
IO8	P0.30		PWM3	
IO9	P0.16	EINT0		
IO10	P0.7		PWM6	
IO11	P0.13		PWM7	
IO12	P0.4			
IO13	P0.6			
IO14	P0.20		PWM5	
IO15	P0.15	EINT2		IO15 connected to LED -- no other connection
AD0	P0.22	IO16		10 bit A/D inputs
AD1	P0.23	IO17		
AD2	P0.24	IO18		may also be used as digital Input/Outputs IO(16-23) -- user controlled
AD3	P0.10	IO19		
AD4	P0.11	IO20		when used as analog lines, voltage levels should not exceed 3.3V
AD5	P0.12	IO21		
AD6	P0.25	IO22		AD6 connected to Arduino AREF pin
AD7*	P0.26	IO23		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

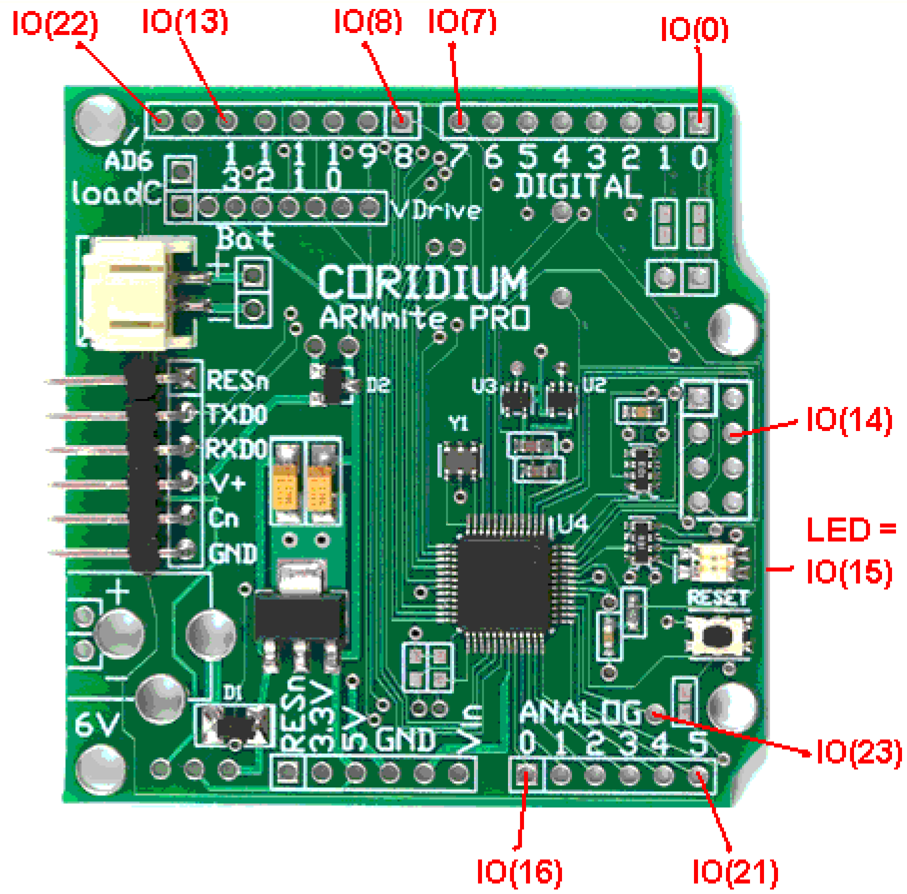
All pins can be used for the software PWM function, and 8 pins can be used for the hardware driven HWPWM function.

Digital IO connections
REV4



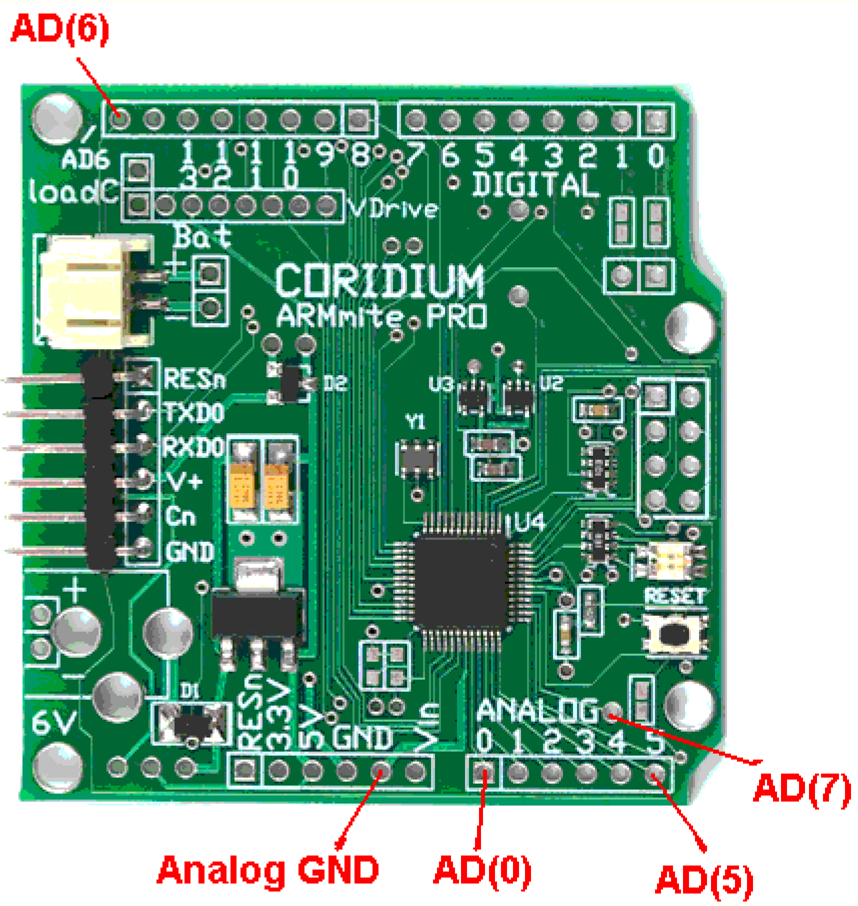
The major change for rev 4 is to add a parallel connection for the 8 IOs IO(8)-IO(13), GND and IO(22) that is on 0.1" centers in relation to the other connections.
In addition the loadC jumper was rotated 90 degrees to make room for this extra connection. And it is also easier to add a battery to the board, by making 1 cut, and adding a diode, resistor and battery (details below).

REV 3



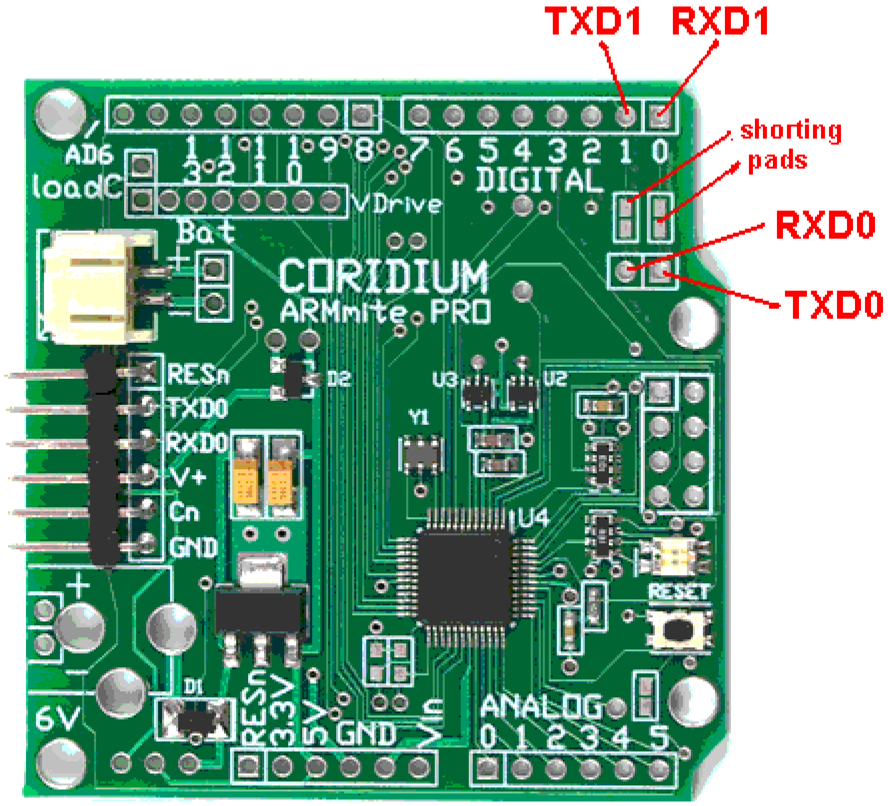
Picture is for the Rev 3 production board. On the Rev 1, IO(23) is available on the via next to AD(5)/IO(21).

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.

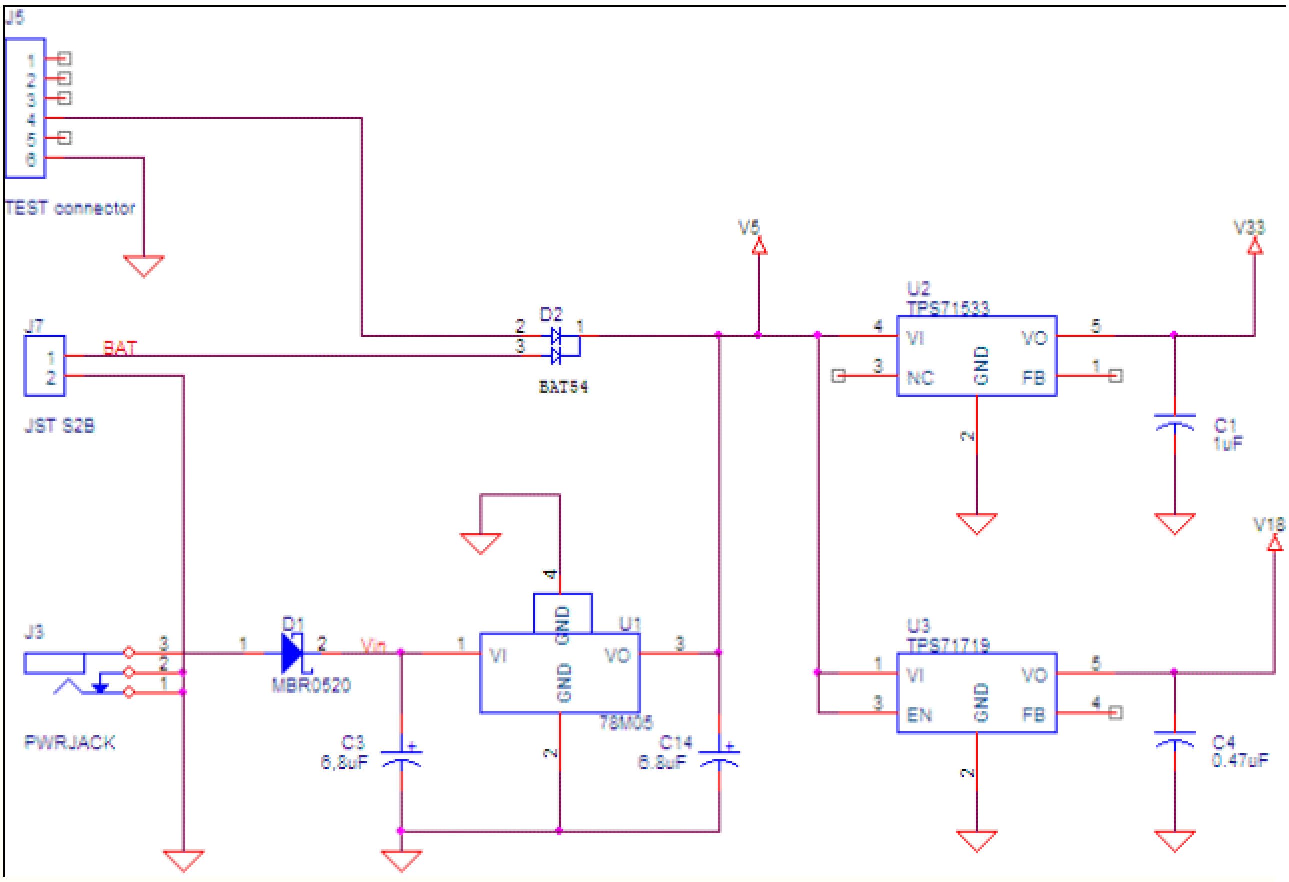


Power connections

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 currrent, 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.

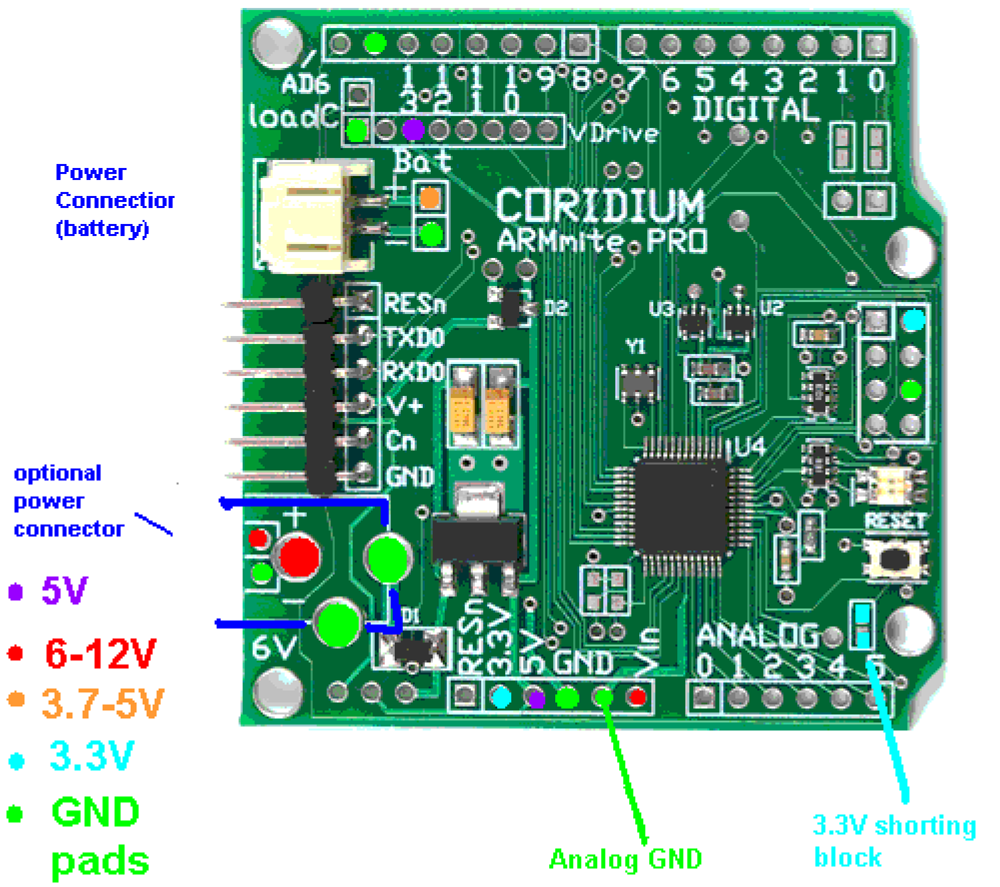
When the 6V source is used, 5V Arduino shields can be powered from the ARMmite PRO.

The schematic describes this circuit



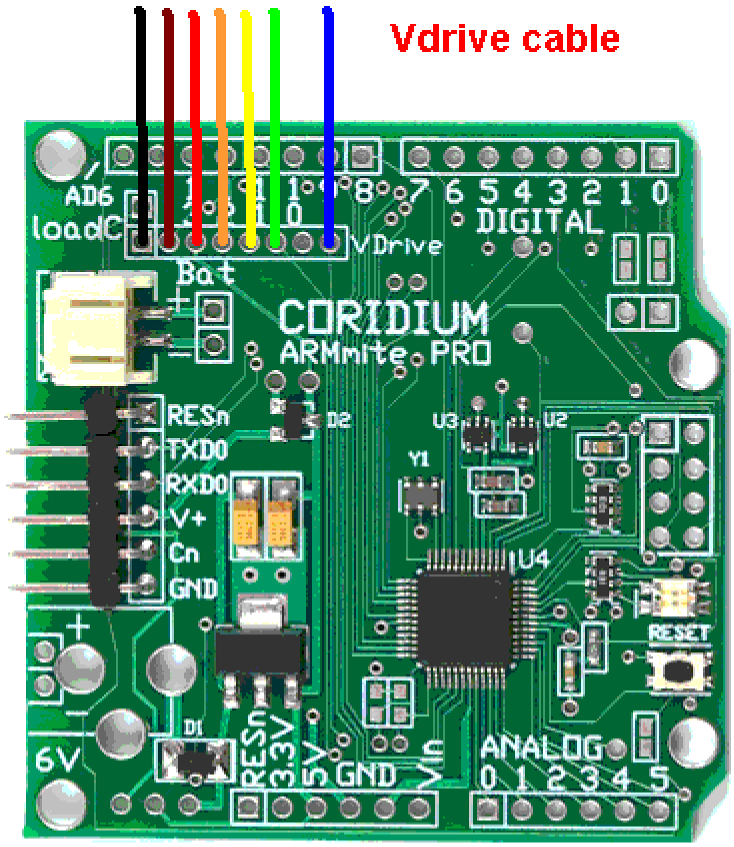
The full schematic can be seen here

Power connections details



The 3.3V regulator can supply 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.
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** .

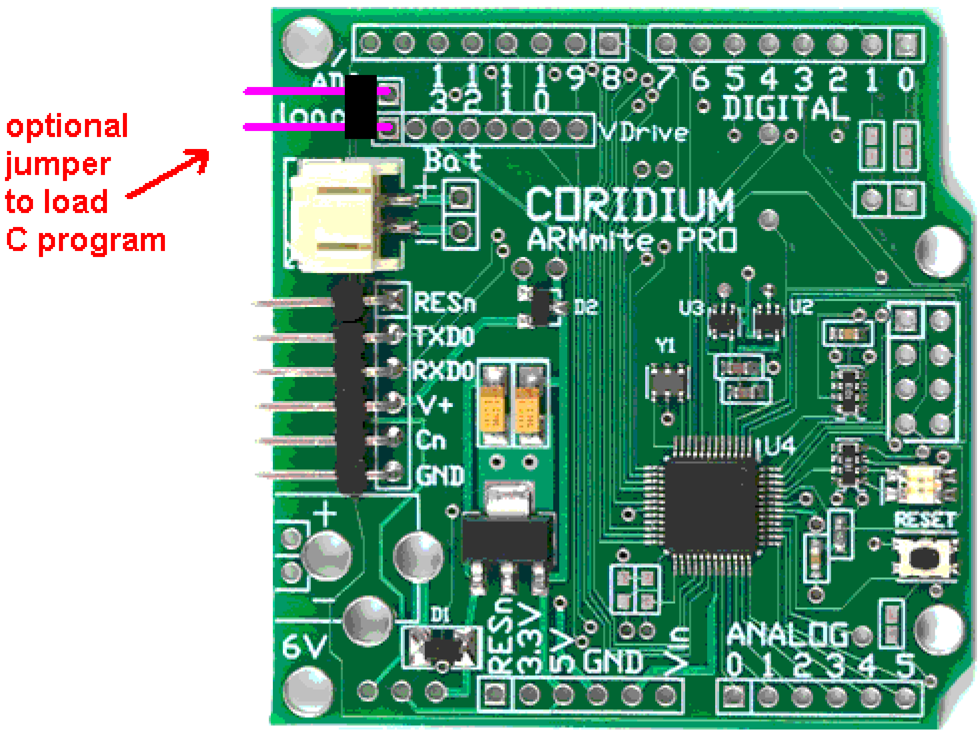


Jumpers and test connector for Program Download

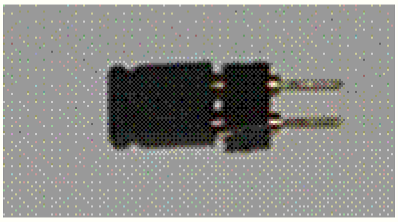
The USB Dongle from Coridium will supply 5V from the USB to power the ARMmite PRO. It also controls the RESET and BOOT signals to automatically load C or BASIC programs using MakeItC or BASICtools.

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.

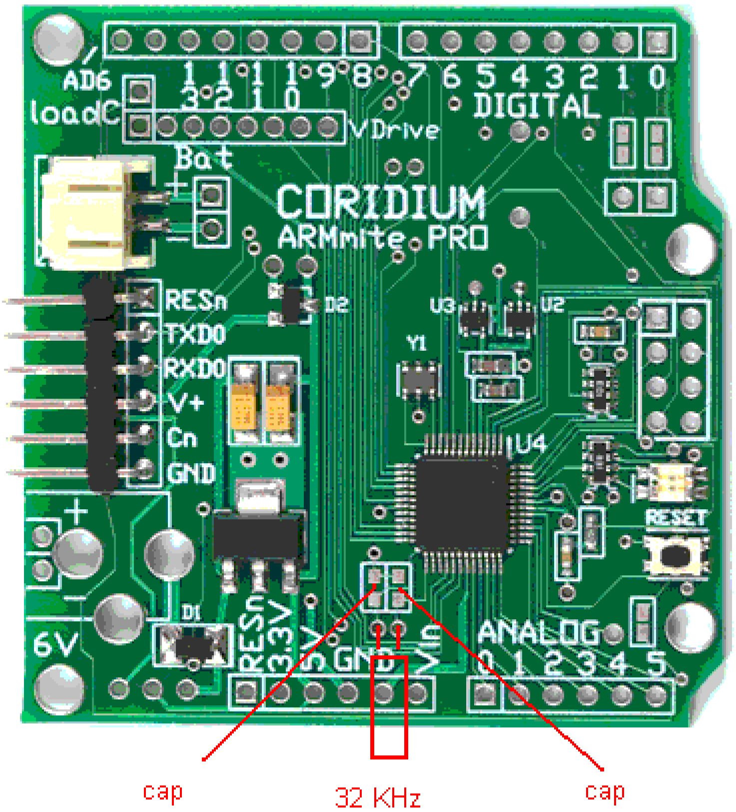


An alternative is to use a 2 pin header with a shorting block (pictured below)



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 cyrstal and 2 cap to use that for the Real Time Clock.

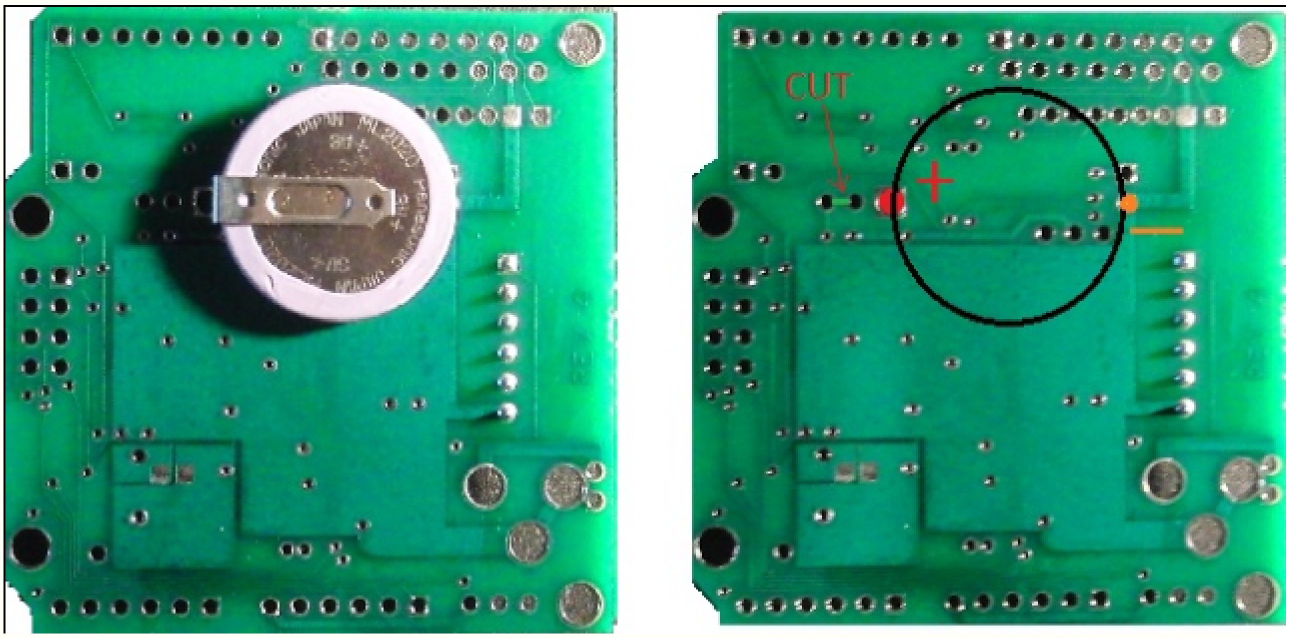


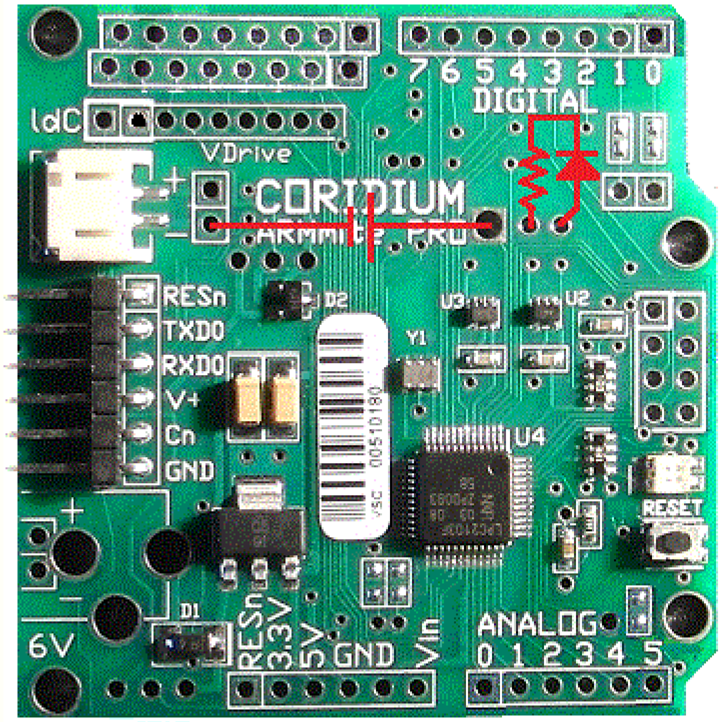
The crystal should be a 32.768 KHz can type, and depending on the rating the capacitors are 0603 size 18-27pF.

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.




7 6 5 4 3 2 1 0
DIGITAL
VDrive
CORIDIUM
RESn
TXD0
RXD0
V+
Cn
GND
D2
Y1
U3
U2
U4
RESET
6V
D1
RESn
3.3V
5V
GND
Vin
ANALOG
0 1 2 3 4 5

Wireless ARMmite Pin Description

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.

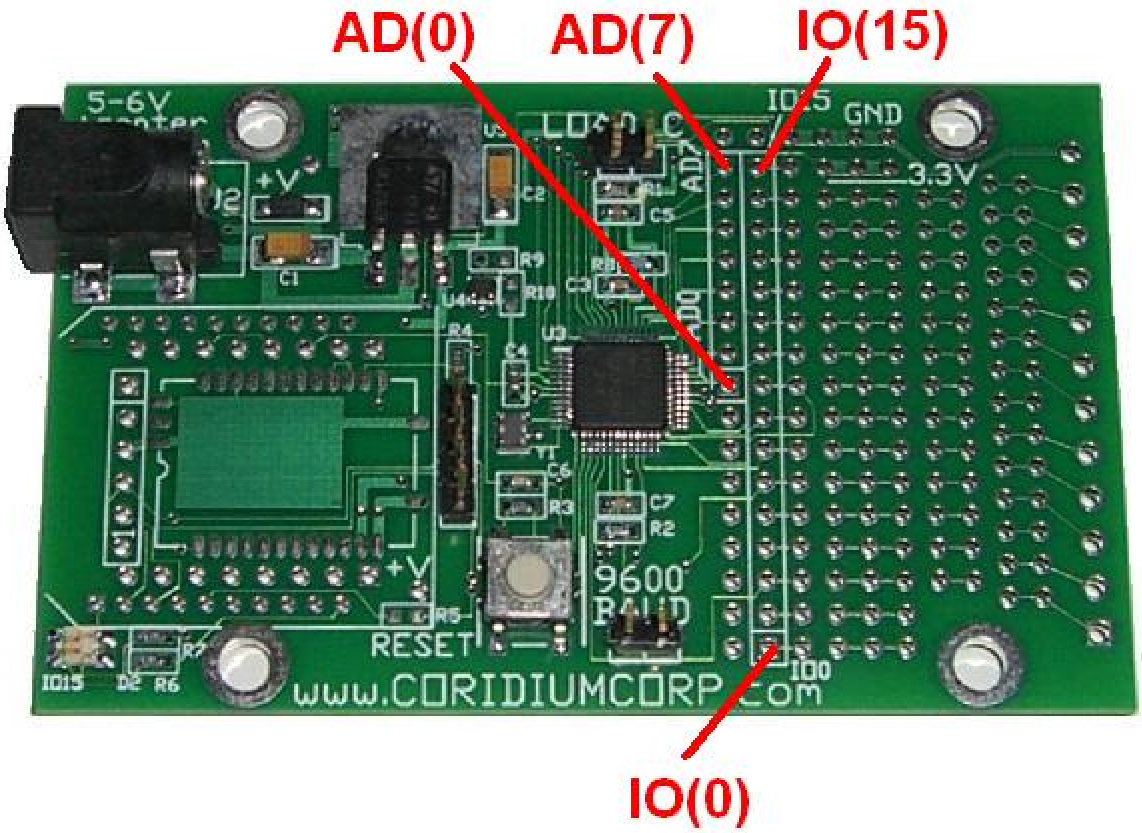
IO0 IO1 IO2 IO3 IO4 IO5 IO6 IO7 IO8 IO9 IO10 IO11 IO14 IO15	RXD1 TXD1 EINT0 EINT2	PWM1 PWM2 PWM3 PWM4 PWM5 PWM6 PWM7 PWM8	Input/Outputs -- user controlled 0-3.3V level 4mA drive when configured as Outputs 5V tolerant - use limiting resistor when connecting to a 5V supply IO15 connected to LED
IO12 IO13			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

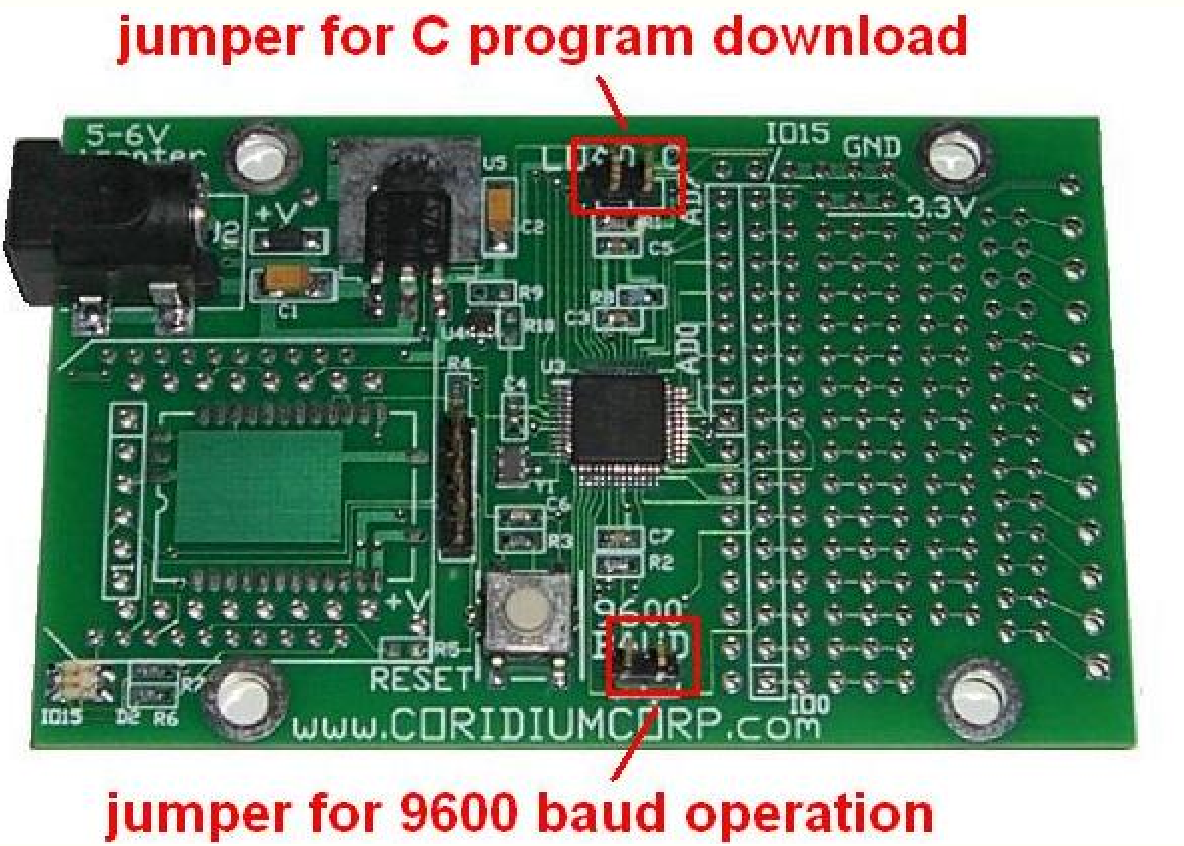
All pins can be used for the software PWM function, and 8 pins can be used for the hardware driven HWPWM function.



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.

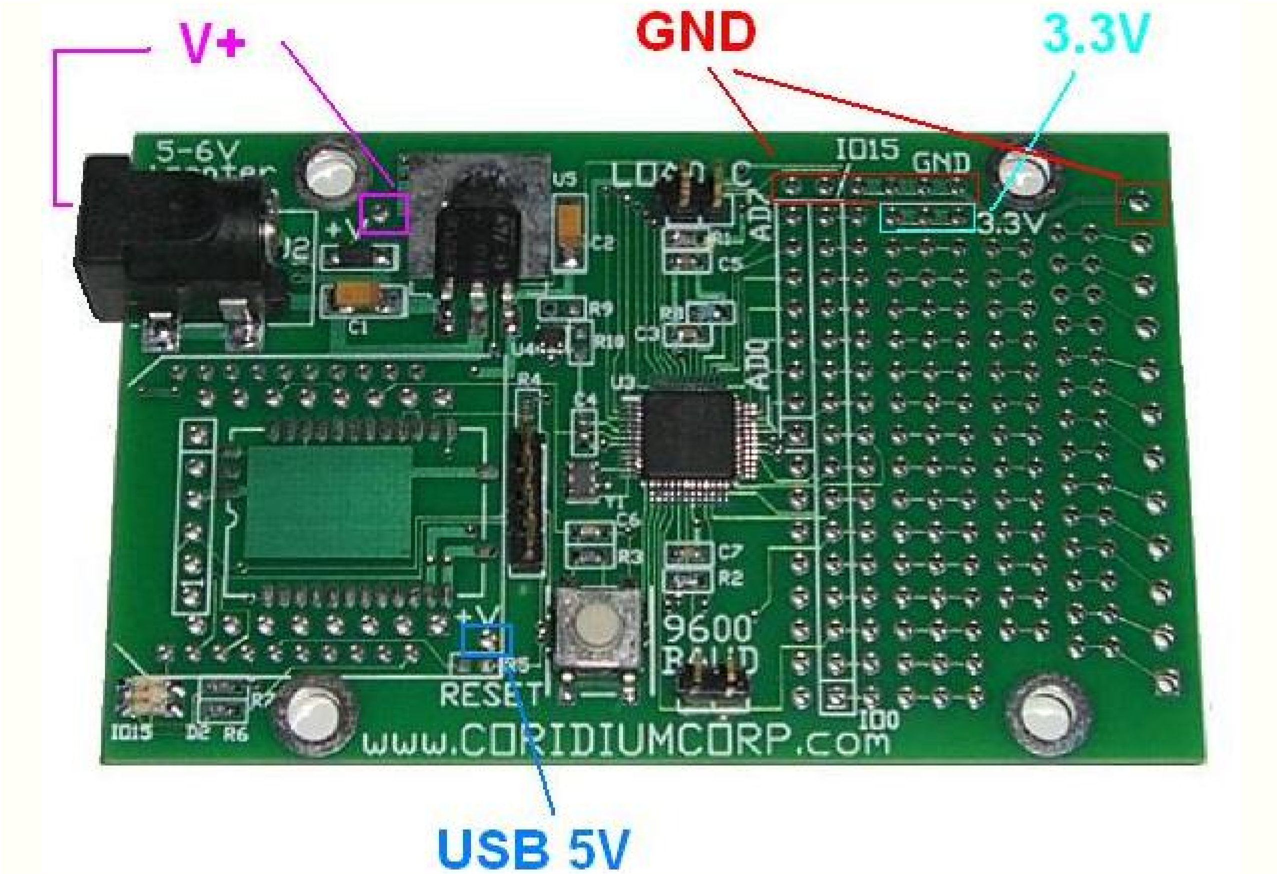
The wireless connections do not have sufficient control lines such that RESET can be controlled from the PC, as well as the RTS line which is used to load C programs. So the BASICtools and MakeItC will prompt you to add a jumper or push the reset button where appropriate.



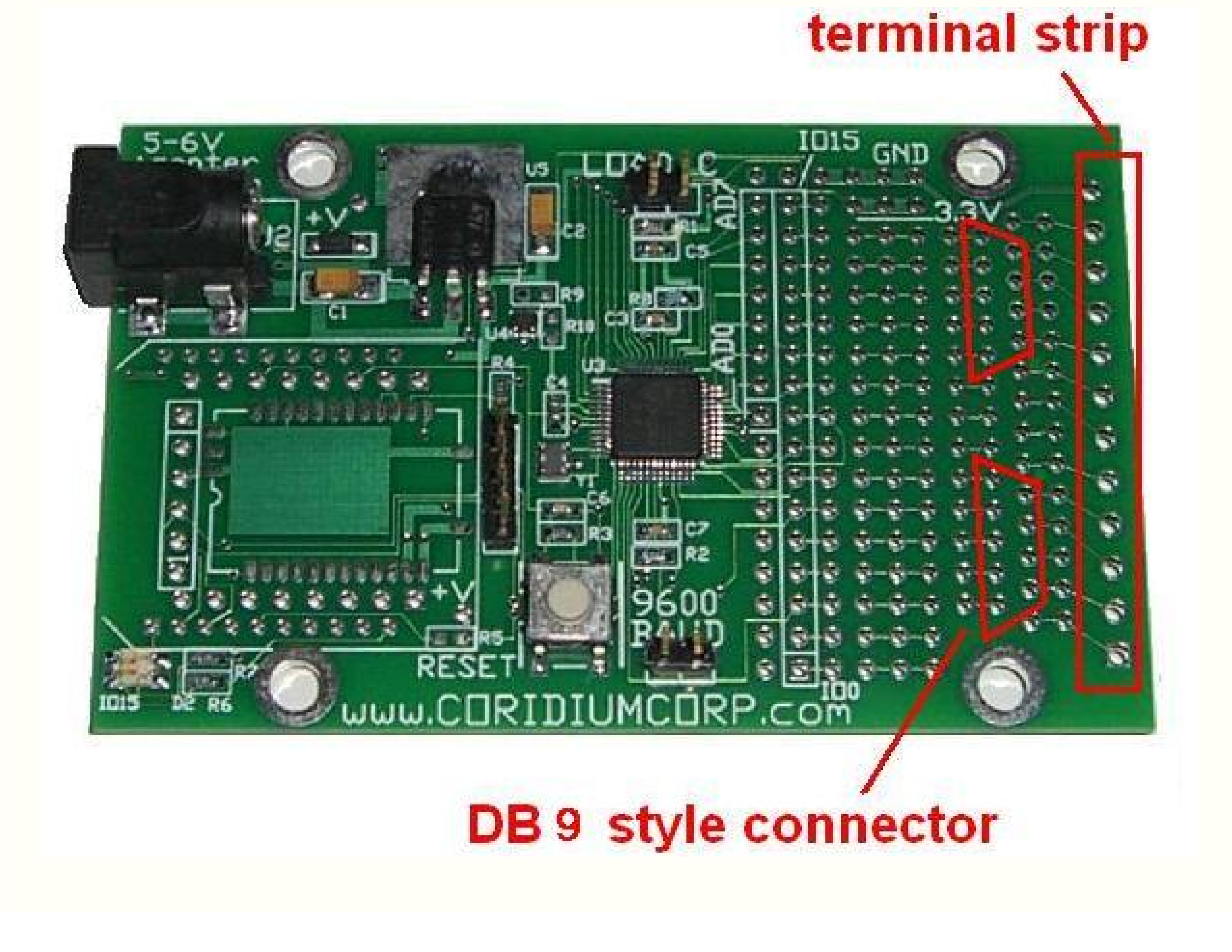
Power

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.



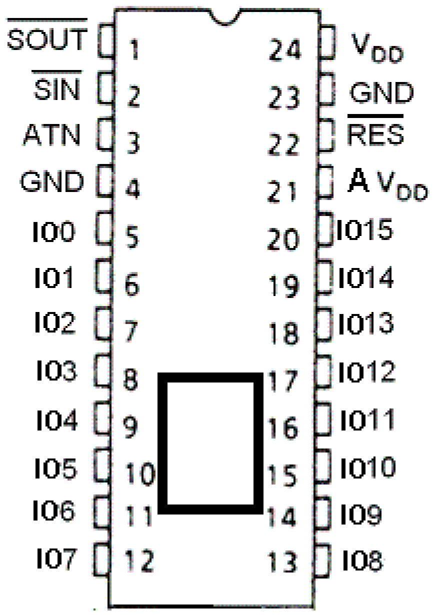
If the all the connections are made to the USB breakout board then 5V can be supplied from the USB. That is also available at the USB 5V pad. When using power from the USB, it should be the only connection for power (do not connect the 5-6V power).

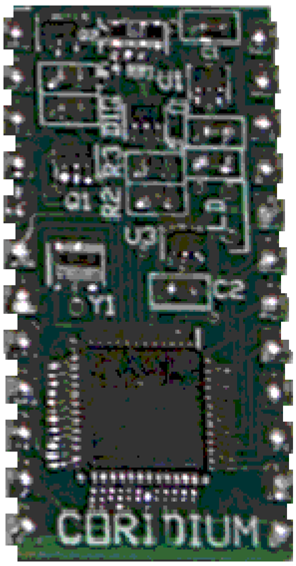


suggested terminal strip On Shore Tech ED550/12DS or equivalent 3.5mm pitch connector (available at Digikey)

prototype connections

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 compatable (active low)
/SIN	2		Serial Input, RS-232 compatable (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 pullup) (active low)
IO0	5		
IO1	6		Input/Outputs -- user controlled
IO2	7		
IO3	8	PWM3	0-3.3V level
IO4	9		
IO5	10	PWM1, RXD1	4mA drive when configured as Outputs
IO6	11	PWM2, TXD1	
IO7	12	AD0	5V tolerant - use limiting resistor when connecting to a 5V supply
IO8	13	AD2	
IO9	14	AD5	
IO10	15	AD1	
IO11	16	AD6	
IO12	17	AD7	
IO13	18	PWM7	EINT2
IO14	19	PWM5	EINT0
IO15	20	PWM8	
GND	4,23		Ground (0V)
VDD	24		Power 5-12V input power

Alt-VDD	21		Alternate 5V input power (for Parallax compatability) DO NOT exceed 5V on this pin connection to pin 24 is preferred this pin is pulled low during download of a C program

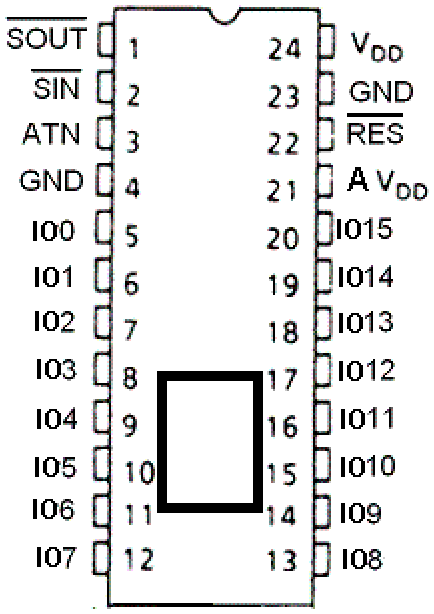
Dual Use AD pins

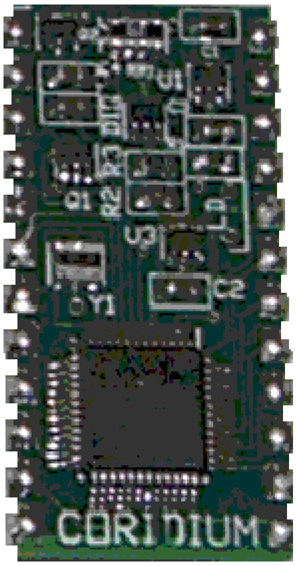
On reset or power up the AD pins are configured as digital IOs on the ARMexpress LITE. When the BASIC accesses these pins they are changed to analog inputs. After that they will remain analog inputs until the next reset or power up.

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 compatable (active low)
/SIN	2		Serial Input, RS-232 compatable (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 pullup) (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 compatability) 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.

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.

IO7			IO7 is connected to LED and PUSHBTTON
IO8	TXD1		as an input the push button is 0 when pressed
IO9	RXD1		
IO10			
IO11**			**IO11 is open drain when an output (i.e. can not pull up)
IO12			
IO13			
IO15			IO15 also controls LED (when low, the LED will be lit)
IO17			
IO18			Input/Outputs -- user controlled
IO19			0-3.3V level, 4mA drive when configured as Outputs
IO20			
IO21			5V tolerant - use limiting resistor when connecting to a 5V supply
IO22			
IO23			
IO25	AD4, DAout		
--	AD5		AD5 is always an analog input, IO26 does not exist
IO27	AD0		
IO28	AD1		10 bit A/D inputs
IO29	AD2		when used as analog lines, voltage levels should not exceed 3.3V
IO30	AD3		
IO31++			++IO31 is always an output
B0			
B1			**BYTEBUS**
B2			
B3			Input/Outputs -- user controlled
B4			
B5			0-3.3V level, 5 volt tolerant, 4ma drive when outputs
B6			
B7			this functions as a byte-wide bus with control of RW and CS
RW			
CS			

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 implimented

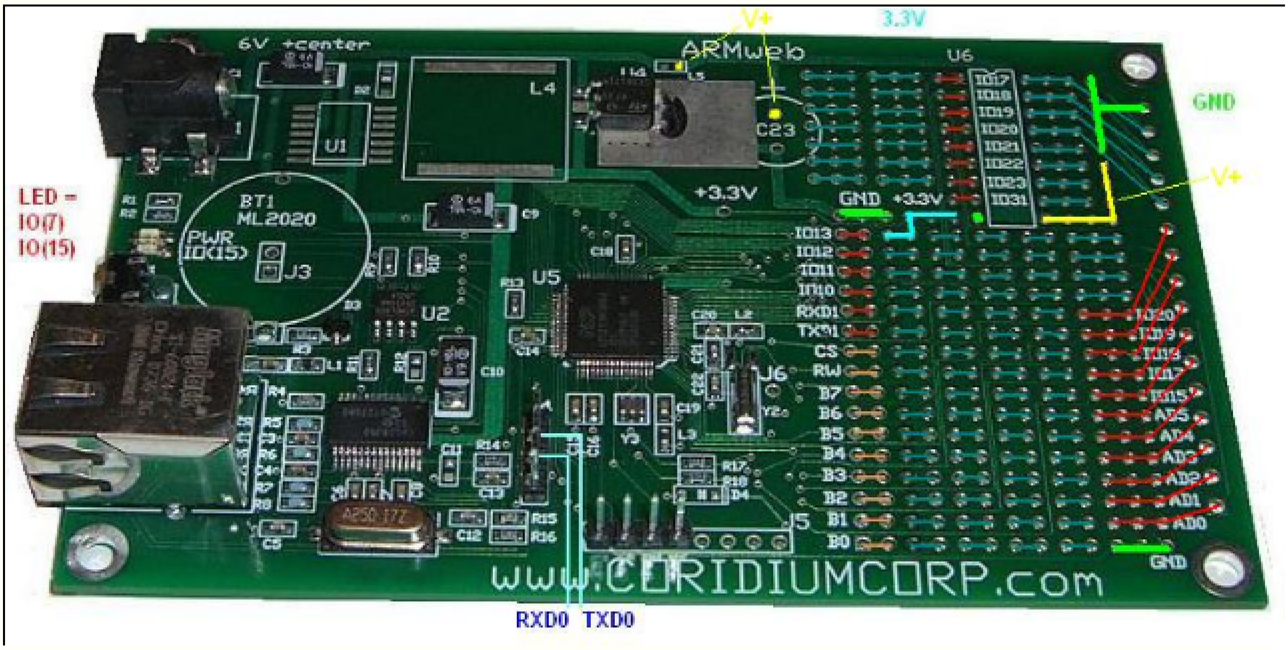
All pins can be used for the software PWM function, and <TBD> pins can be used for the hardware driven HWPWM function.

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.

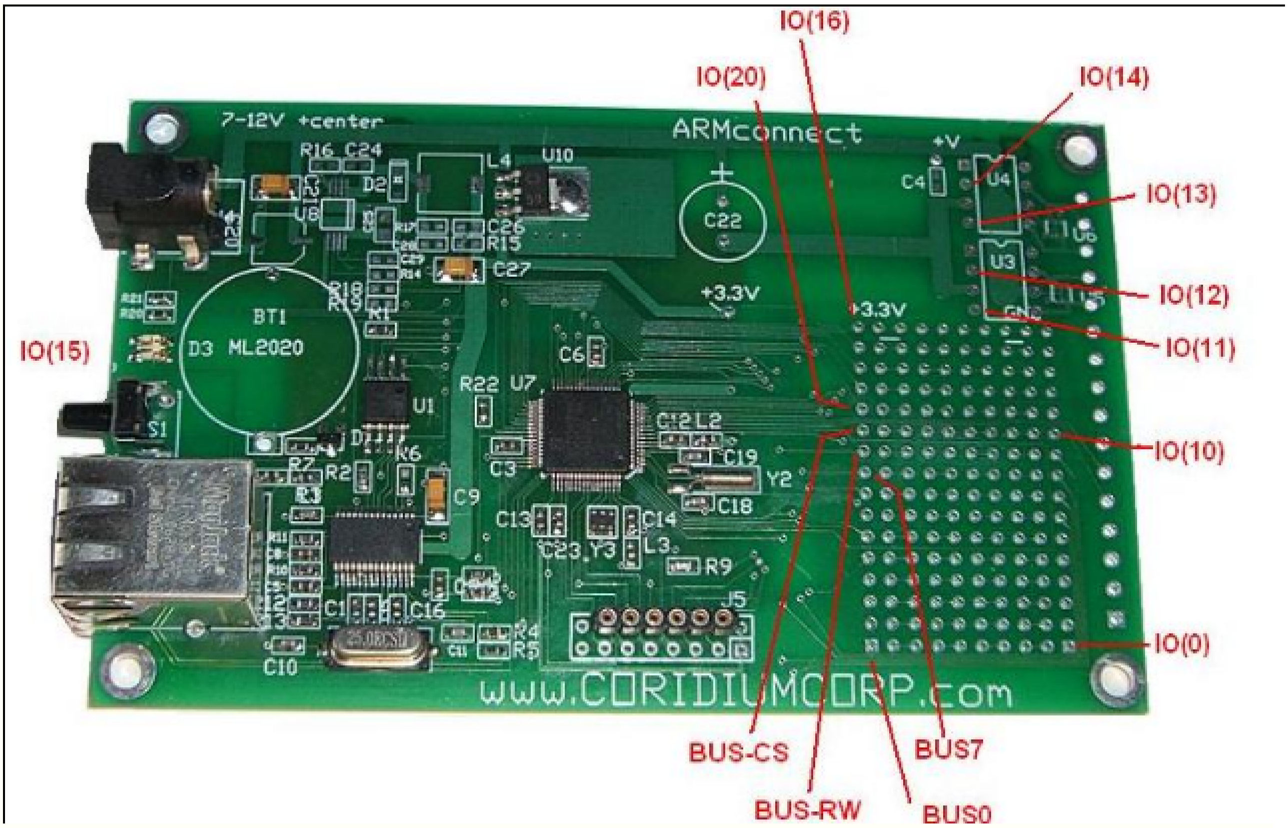
Rev 2,3

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


IO(16)
IO(20)
IO(14)
IO(13)
IO(12)
IO(11)
IO(15)
IO(10)
IO(0)
BUS-CS
BUS7
BUS-RW
BUS0

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.

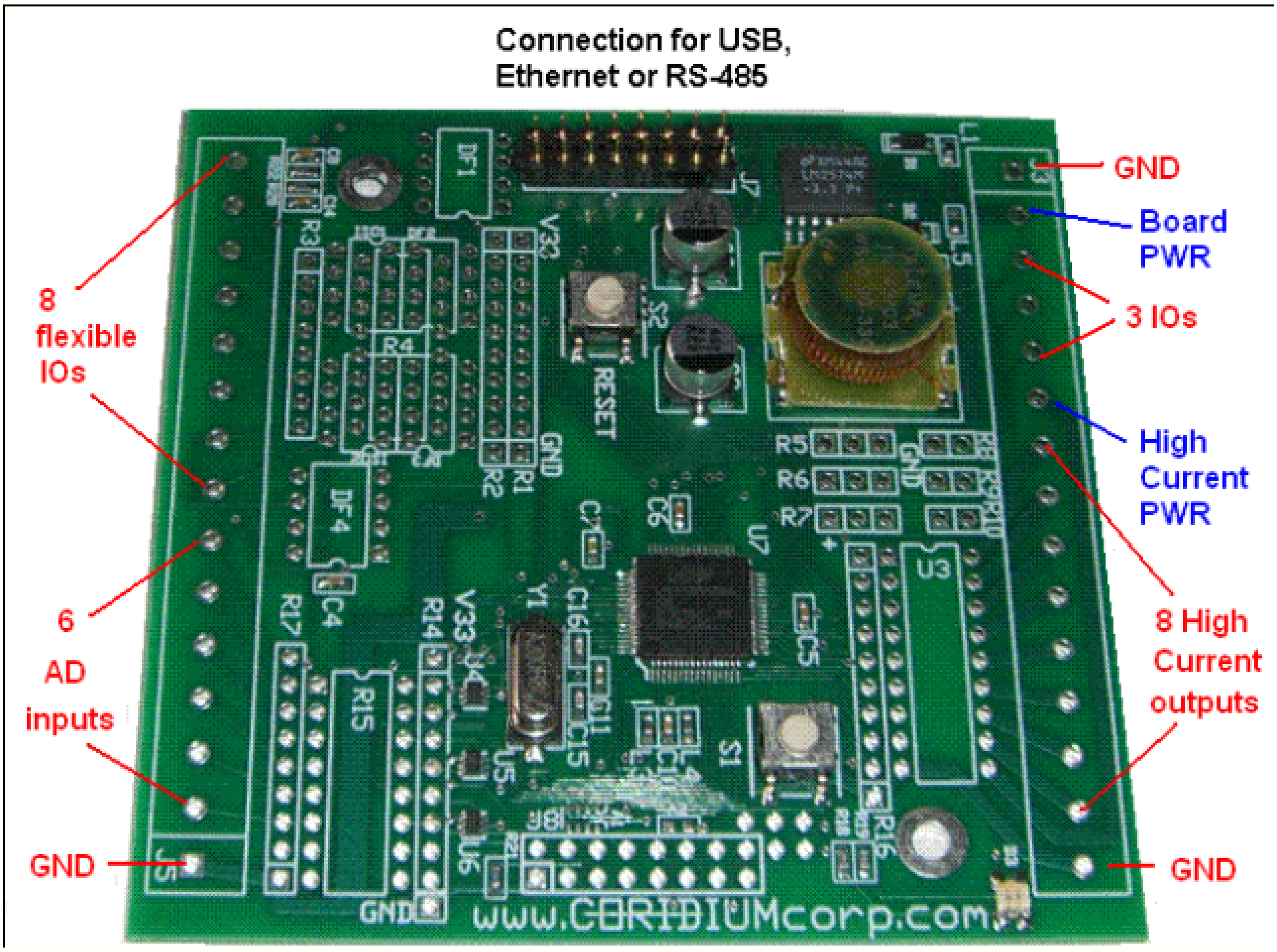
The USB version uses the standalone ARMbasic compiler on the PC.

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 seperate 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

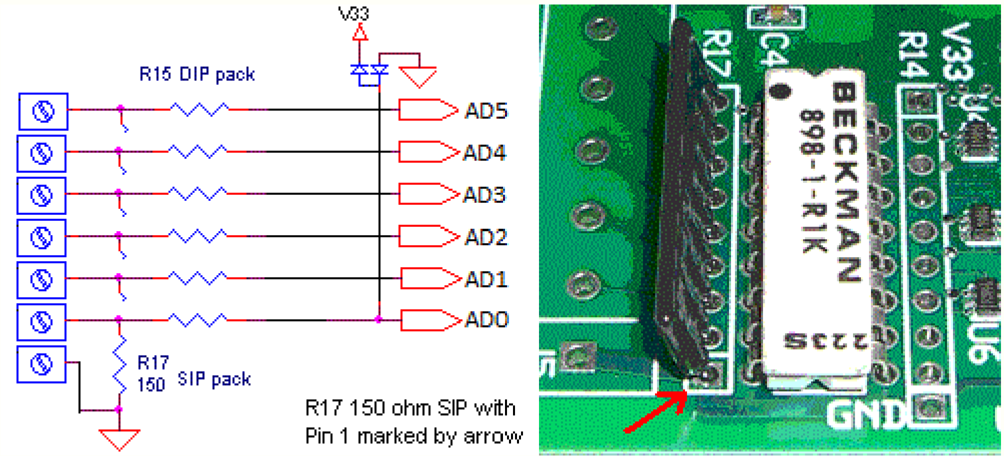
OKW B6704100 The kits include custom cutouts for either Ethernet or USB connections. Mechanical drawing for the **enclosure is here** ,

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.

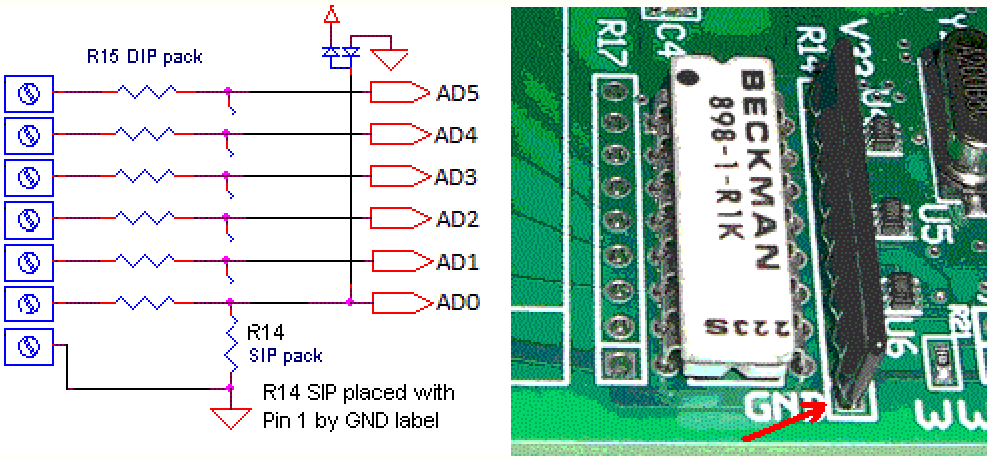
4-20mA sensor --



load 150 ohm SIP into R17

suggested components
  Bourns 4600X Bussed SIP resistor
  Bourns 4100R Isolated DIP resistor

A/D resistor divider --

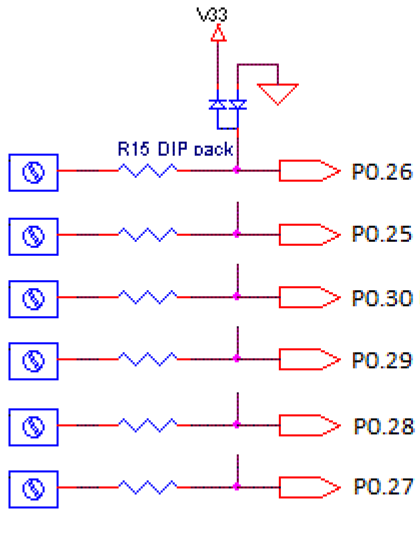


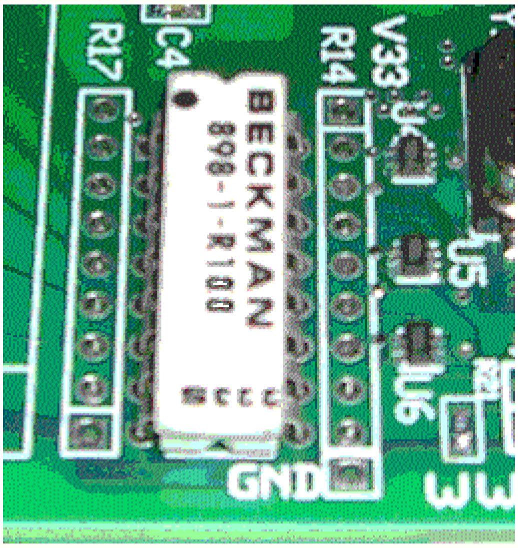
load R15 DIP resistor and R14 SIP with appropriate values

AD = Vin * R14/(R14+R15)

Source impedance to AD should be less than 10K.

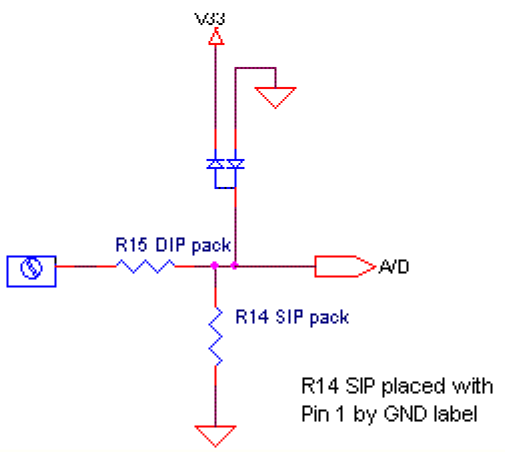
digital IO --

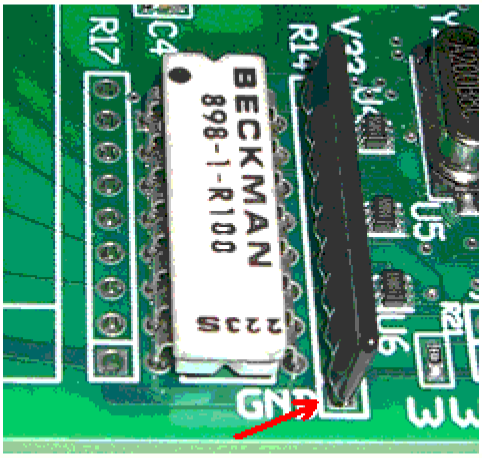




load R15 with 100 or 1K

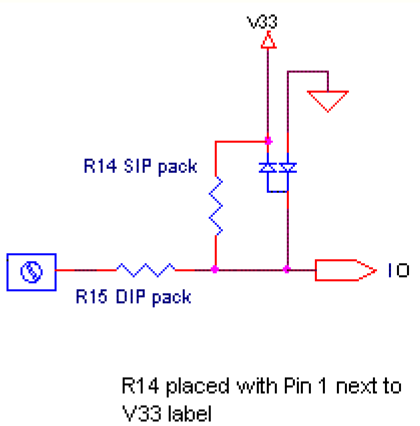
digital IO (pulldown)--





load R15 with 100, R14 with 10K

digital IO (pullup) --

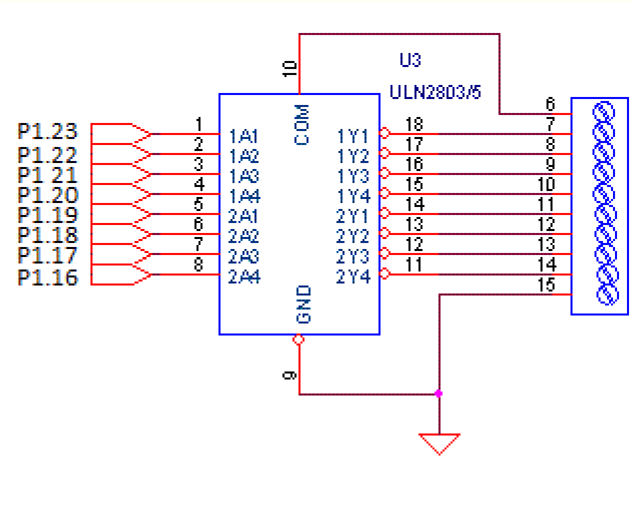


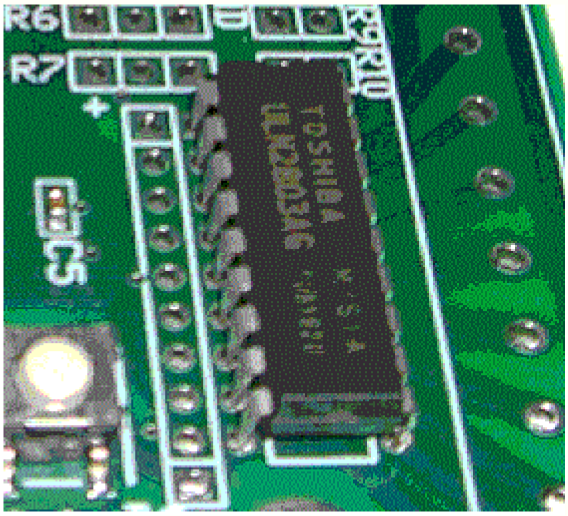
load R15 with 100, R14 with 10K

High Current Drivers

These may use a high sink current driver, or configured as digital IOs with optional pullups or pulldowns

High Current drive --

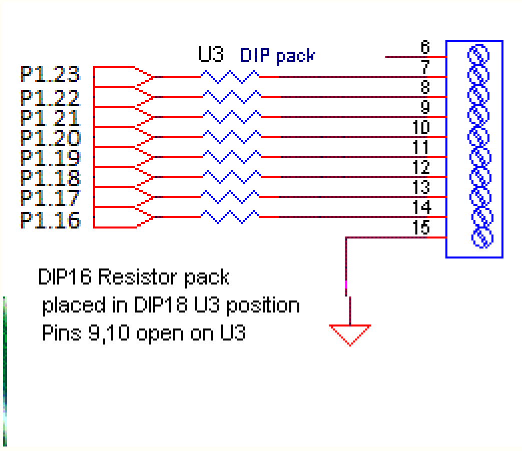


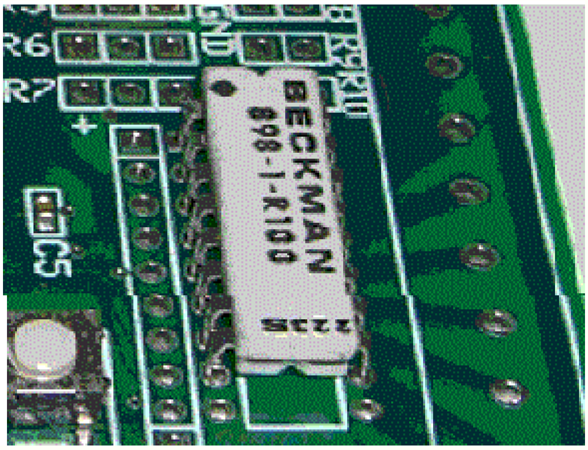


This driver can sink a surge current of 500mA upto 50V, this driver is a **ULN2803** .

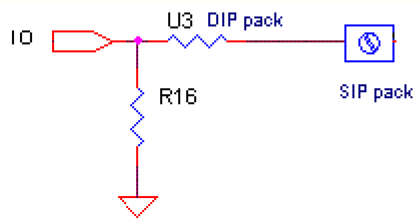
suggested components
TI ULN2803AN
Toshiba ULN2803APG
STmicro ULN2803A

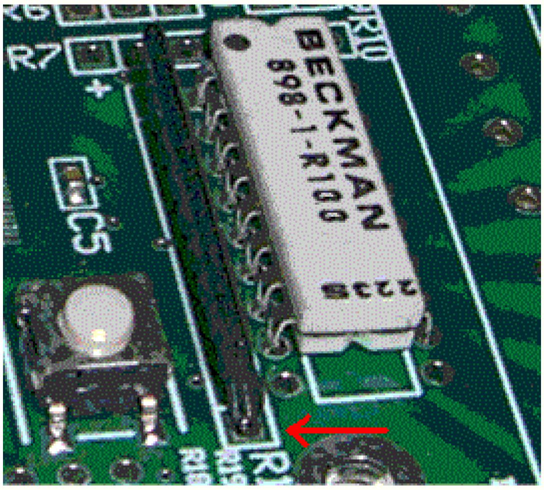
digital IO --





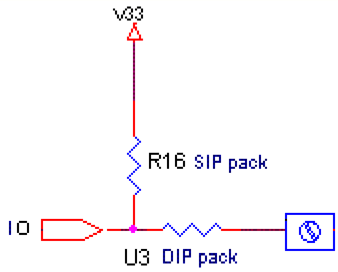
digital IO (pulldown) --

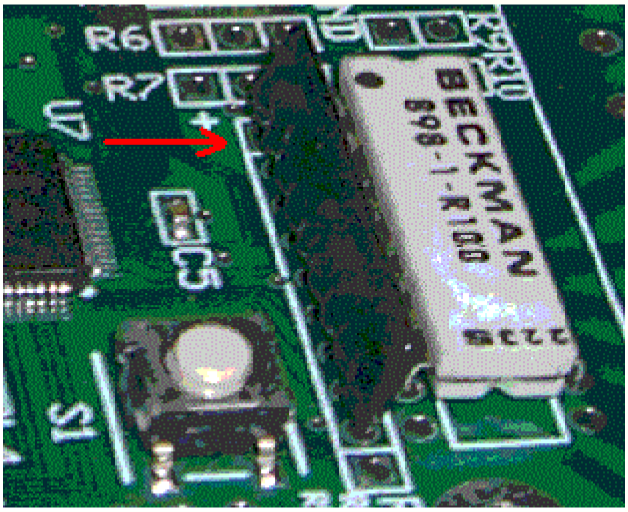




Pin 1 of SIP resistor loaded
near R16 label (arrow)

digital IO (pullup) --



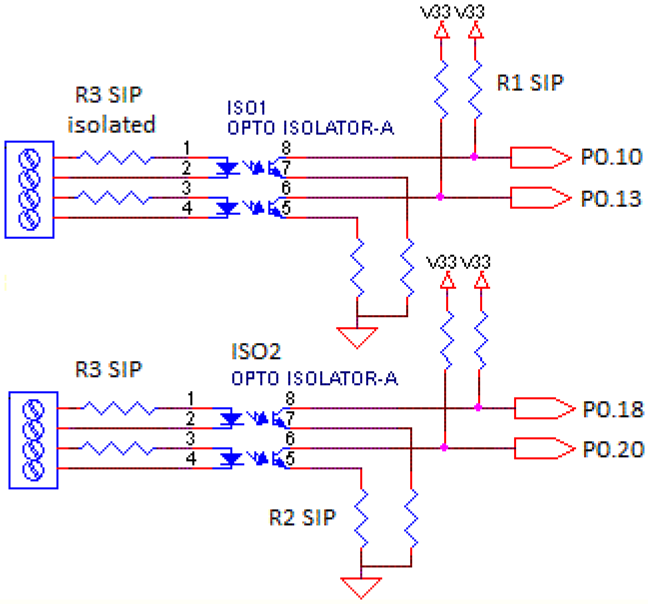


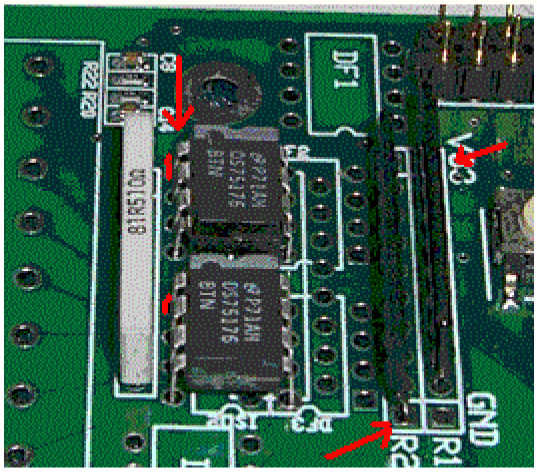
Pin 1 of SIP pullup loaded
near + sign (arrow)

Flexible IOs

These may be configured as 8 digital IOs (with and without pullup/pulldown), 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 --

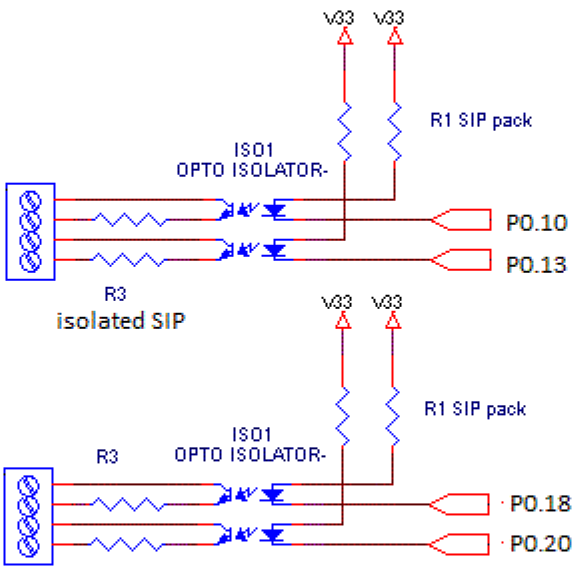


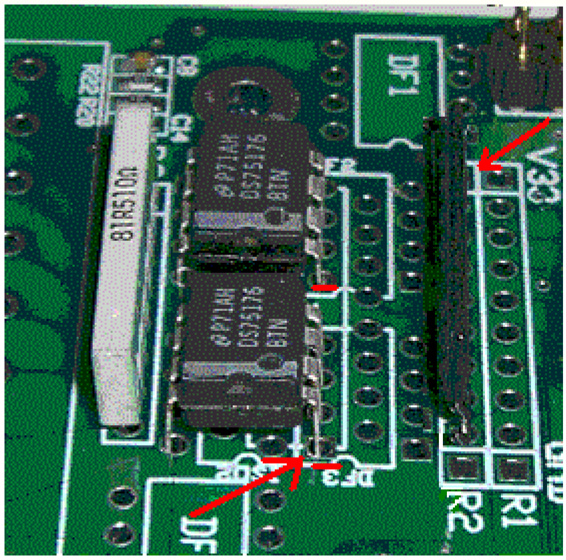


Note Pin 1 orientation of opto-isolators
Pin 1 of R2 near GND,   Pin 1 of R1 near V33

suggested components
  Liteon LTV-827
  Fairchild MCT9001
  Toshiba TLP621-2

opto-isolated output --



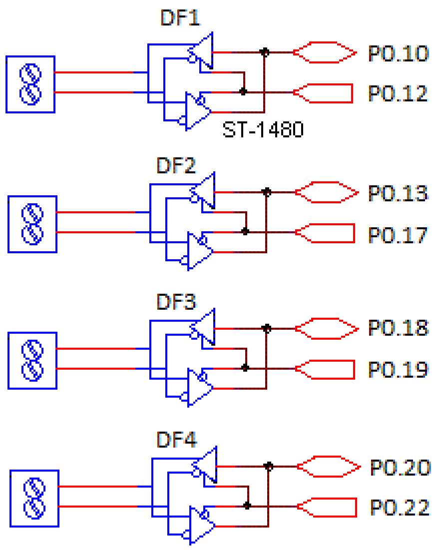


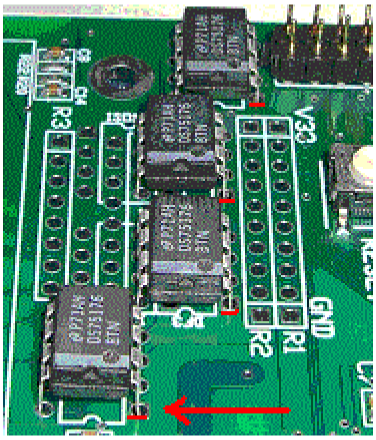
Note Pin 1 orientation of opto-isolators

Pin 1 of R2 near V33 marking

same components as above, rotated 180 degrees

bidirectional RS-422 driver --

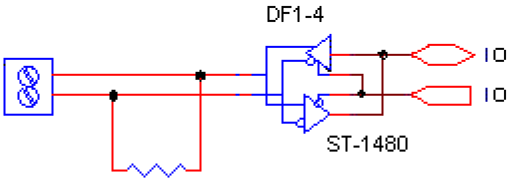




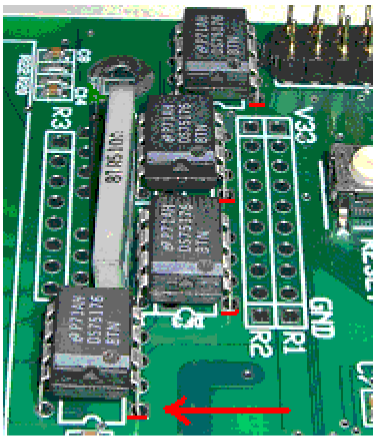
Note Pin 1 orientations

suggested components
  National DS75176BN
  TI SN75176AP

bidirectional RS-422 driver with termination --

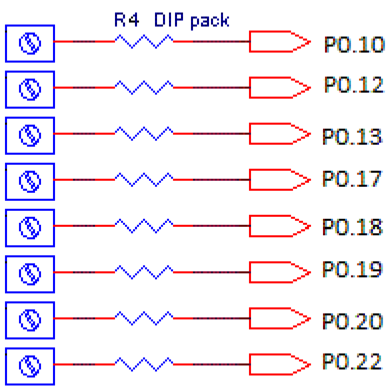


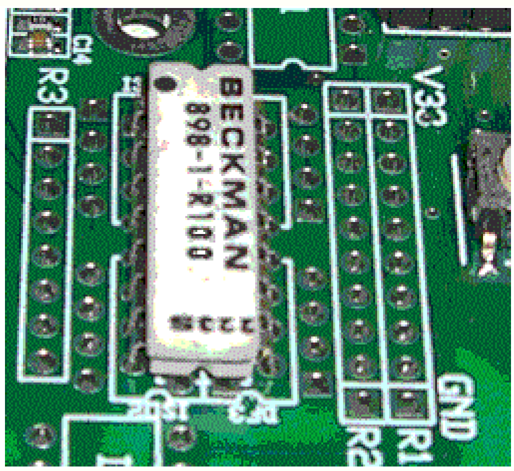
Note Pin 1 orientations



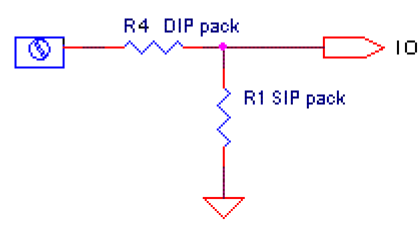
suggested components
  Bourns 4600 Isolated SIP resistor

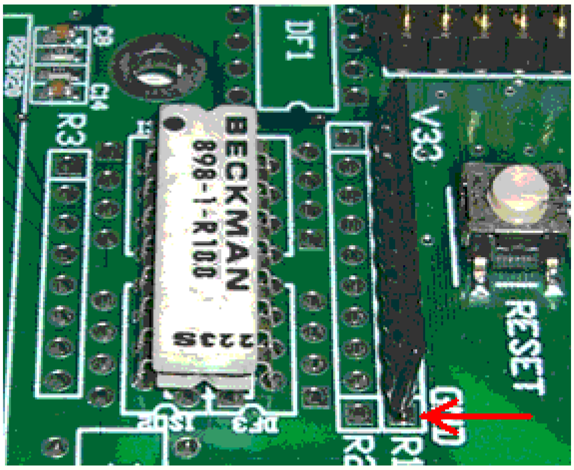
digital IO --





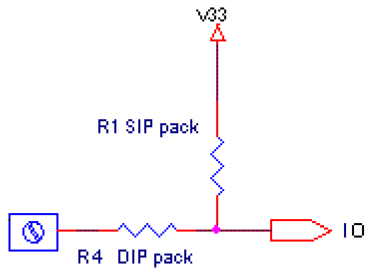
digital IO (pulldown) --

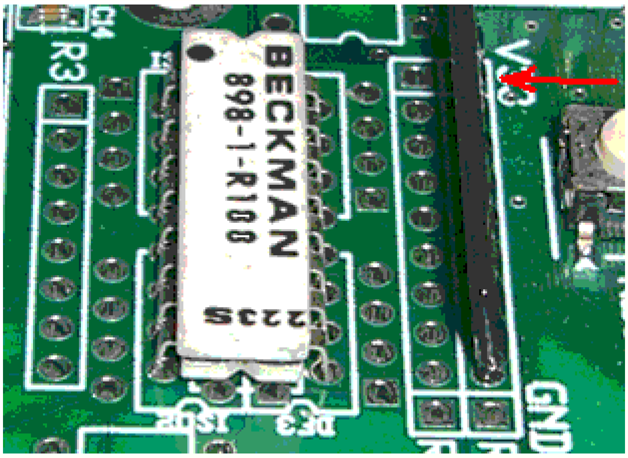




Pin 1 of R1 SIP located
at GND marking

digital IO (pullup) --



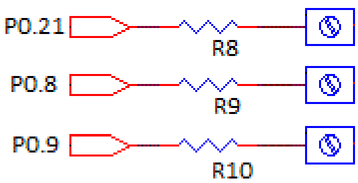


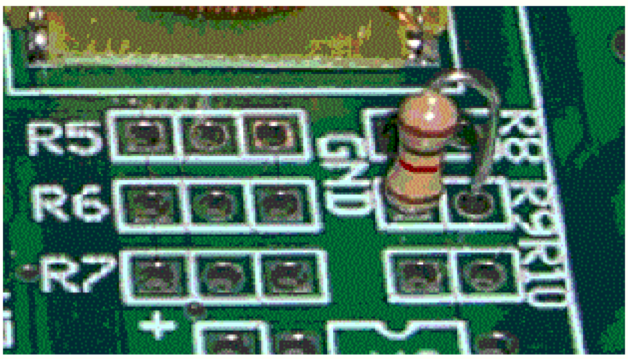
R1 Pin 1 located next
to V33 marking

3 digital IOs

These may be configured as staight thru, or with pullups or pulldowns

digital IO --

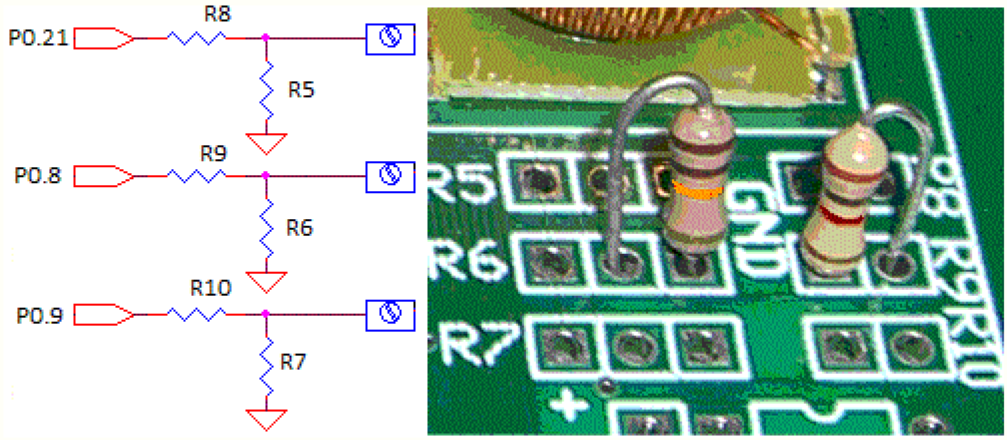




R9 loaded on right

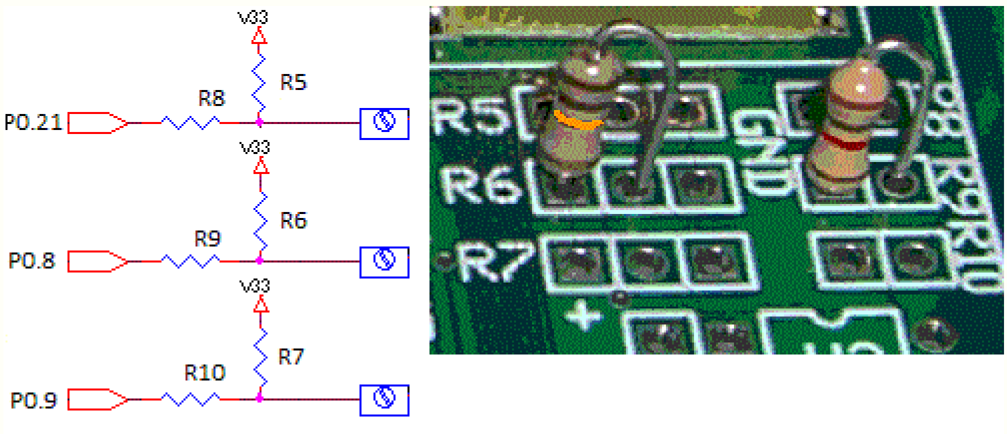
shown with 100 ohm series

digital IO (pulldown) --



shown with 10K pulldown and 100 series

digital IO (pullup) --

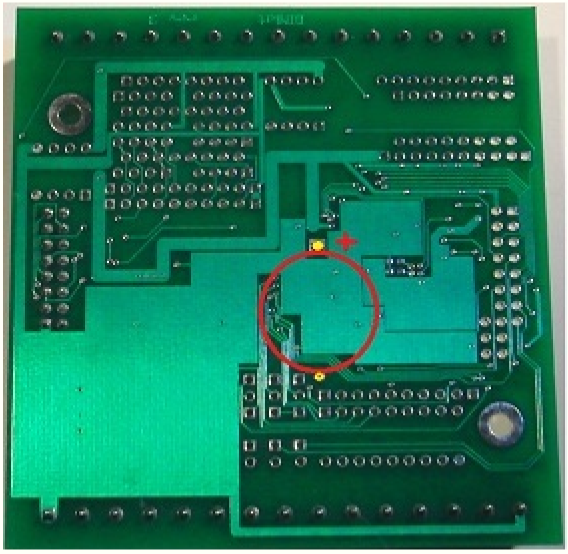


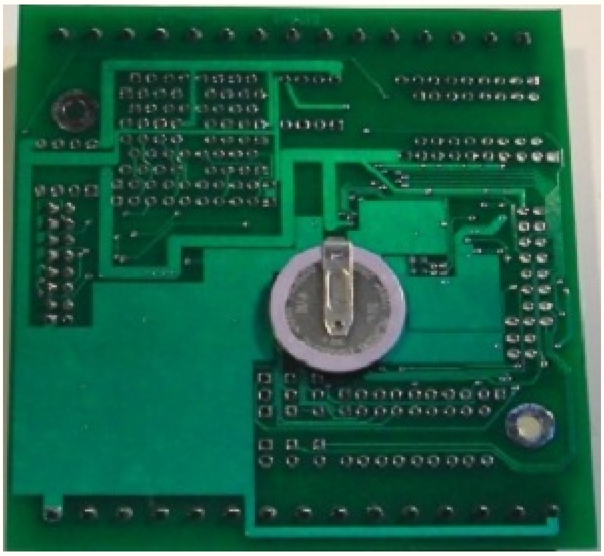
shown with 10K pullup and 100 series

RTC options

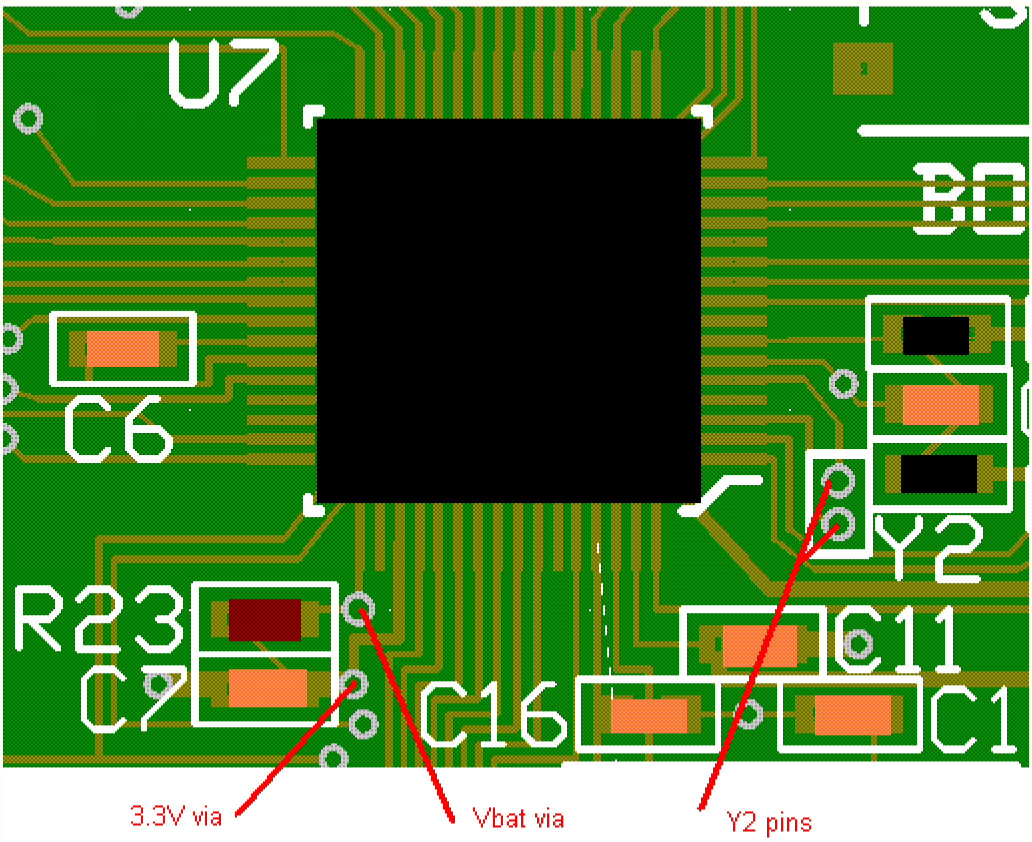
Rev 3

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



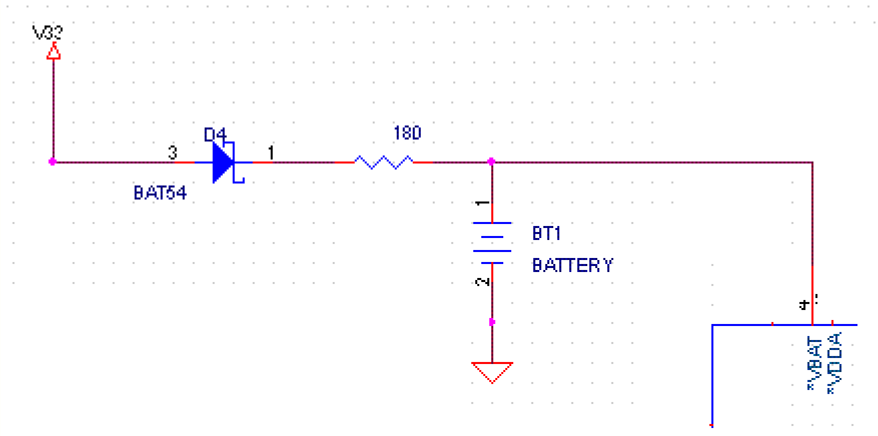


Rev 2



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.

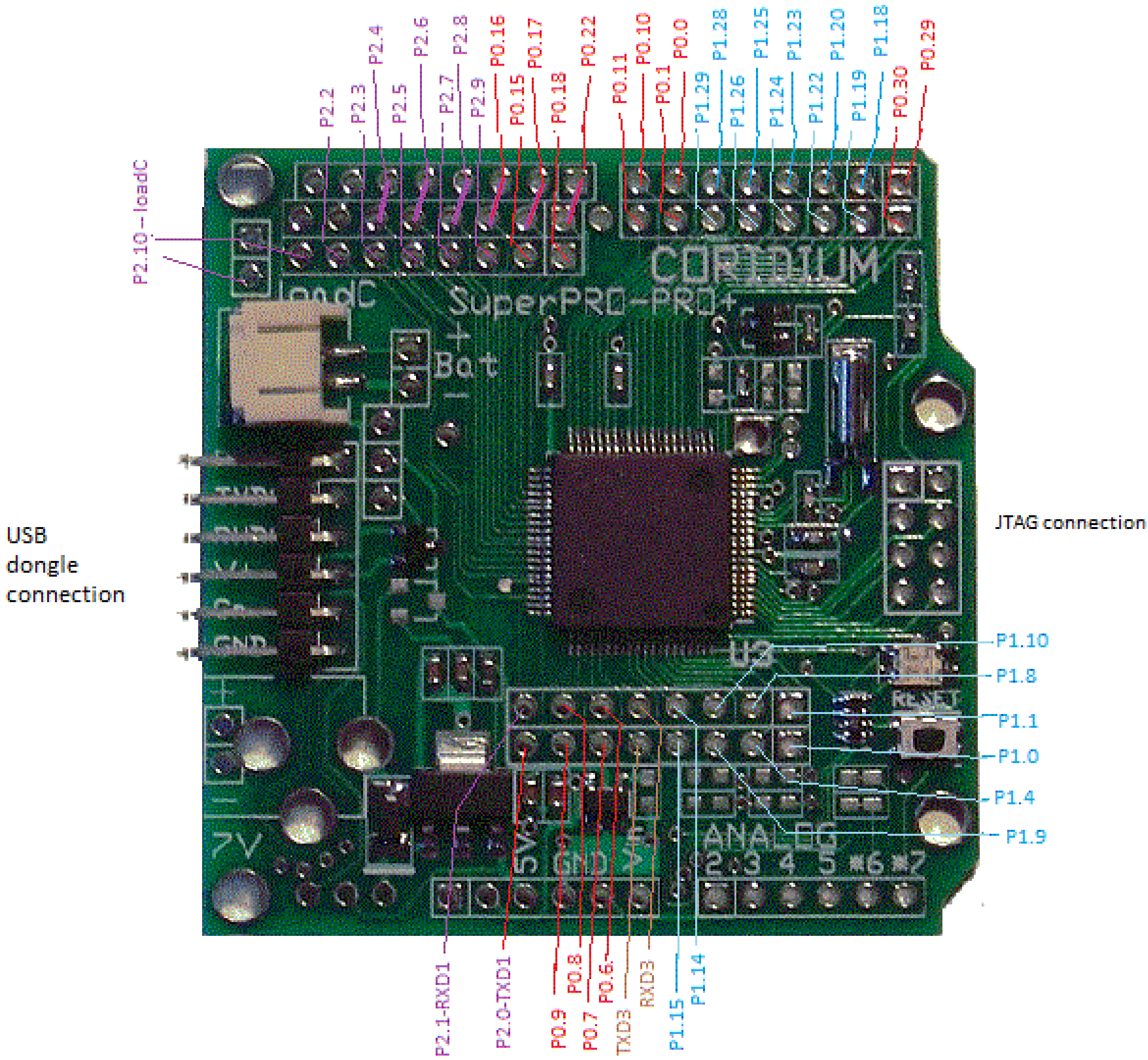
SuperPRO Pin Description

PROplus Pin Description

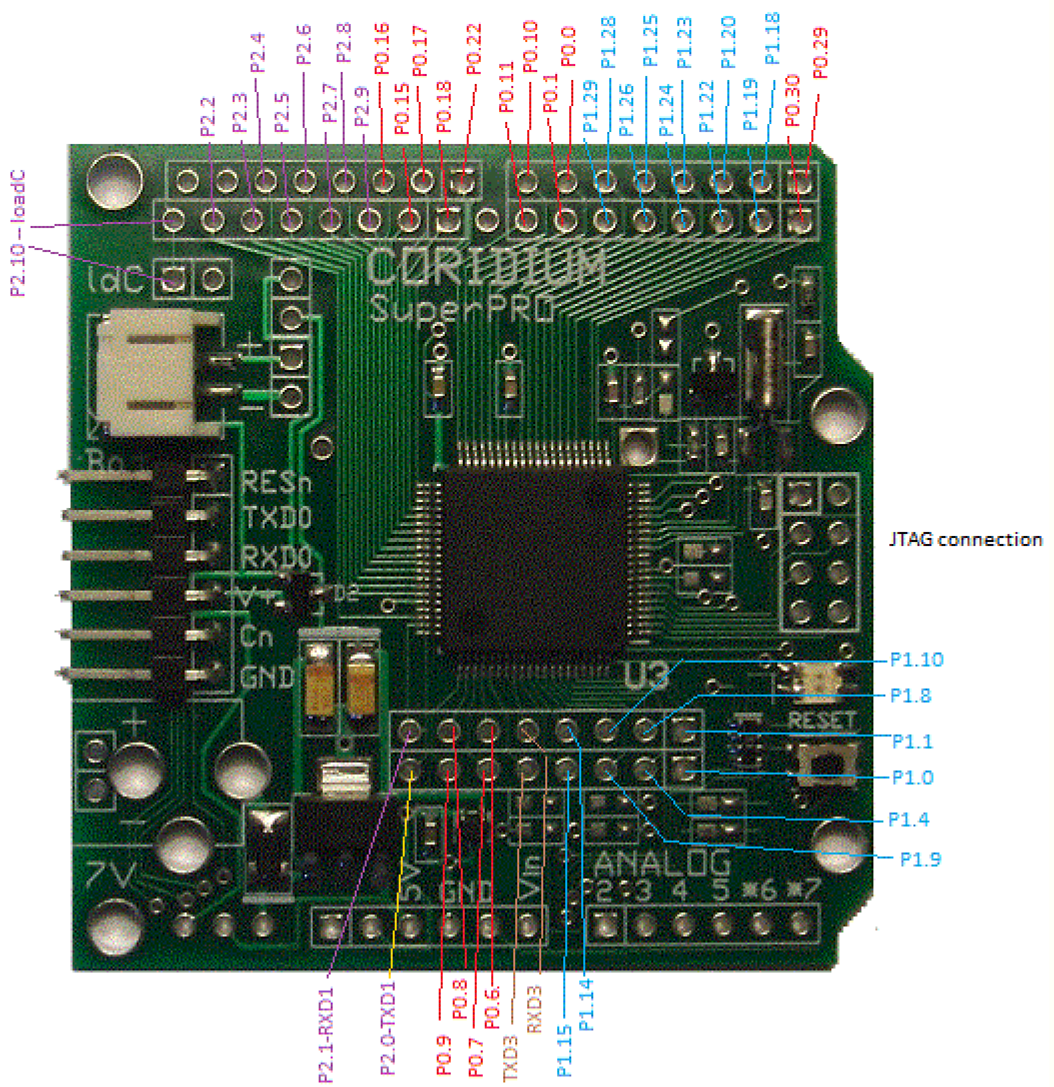
The SuperPRO is footprint and pin compatible with the Arduino PRO. In addition it has an onboard 5V regulator so it is compatible with 5V shield boards.

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 **MakerShed** . More details on **these connections here**.

Digital IO connections -- rev 5
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**. Port 0 pins can be accessed with the original **IN, OUT... keywords** . More details on the GPIOs can be found in the NXP User Manual.



Digital IO connections -- rev 4



Special purpose pins

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 pullups and default to pullups following reset.

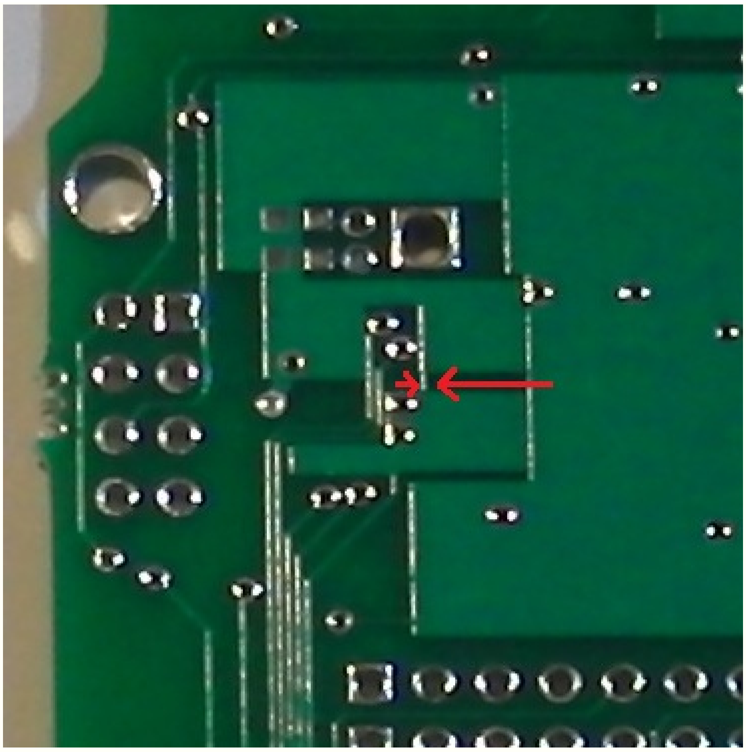
Details can be found in NXP's User manual.

Analog connections

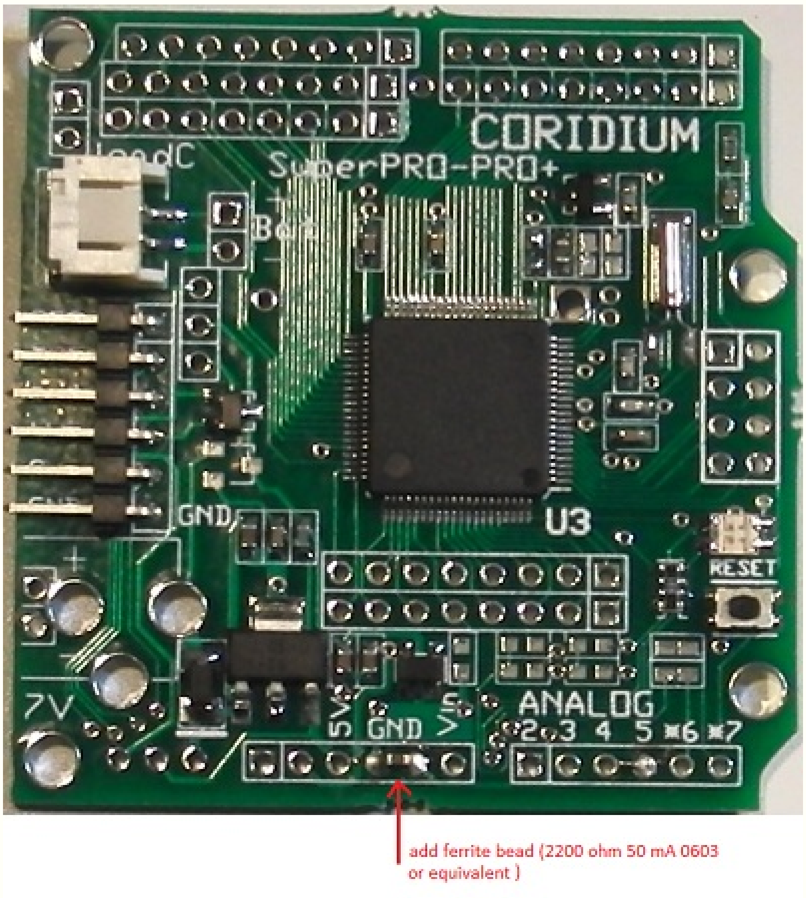
4 A/D converters are readily available. 2 more are available, but share the pins with UART0 -- what was NXP thinking, I have no idea.
1 10 bit DAC is available shared with AD(3) available on the SuperPRO (not on PROplus)
On reset or power up the AD pins are configured by software as AD inputs. To change those to digital IOs, the user must write to the appropriate PINSEL register.
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)

update

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 occurance 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.

Power connections -- SuperPRO

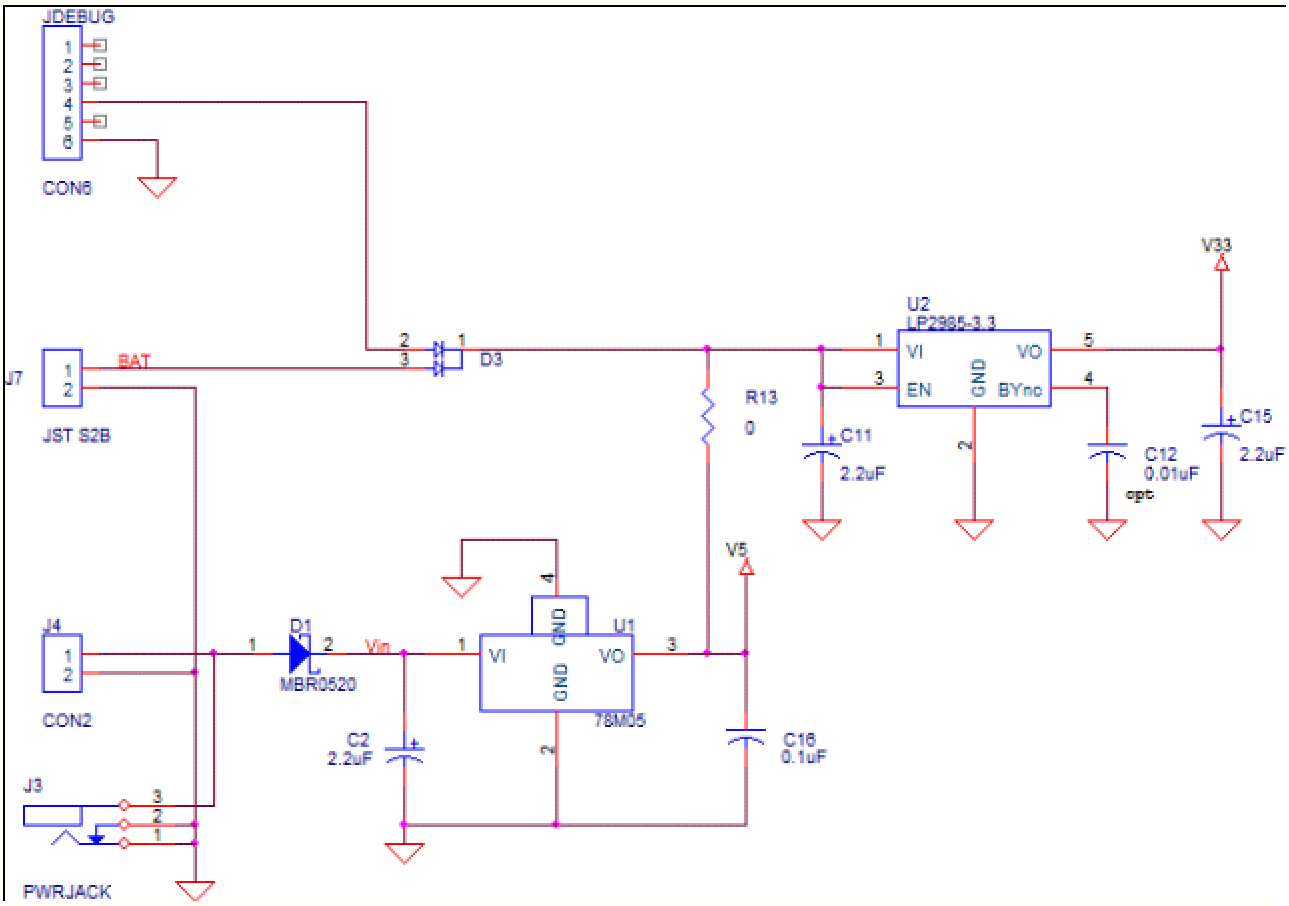
The board is shipped with a **2mm power jack compatible** with a JST PHR/S2B or **SparkFun PRT8671** or various battery packs from SparkFun.

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 currrent, 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.

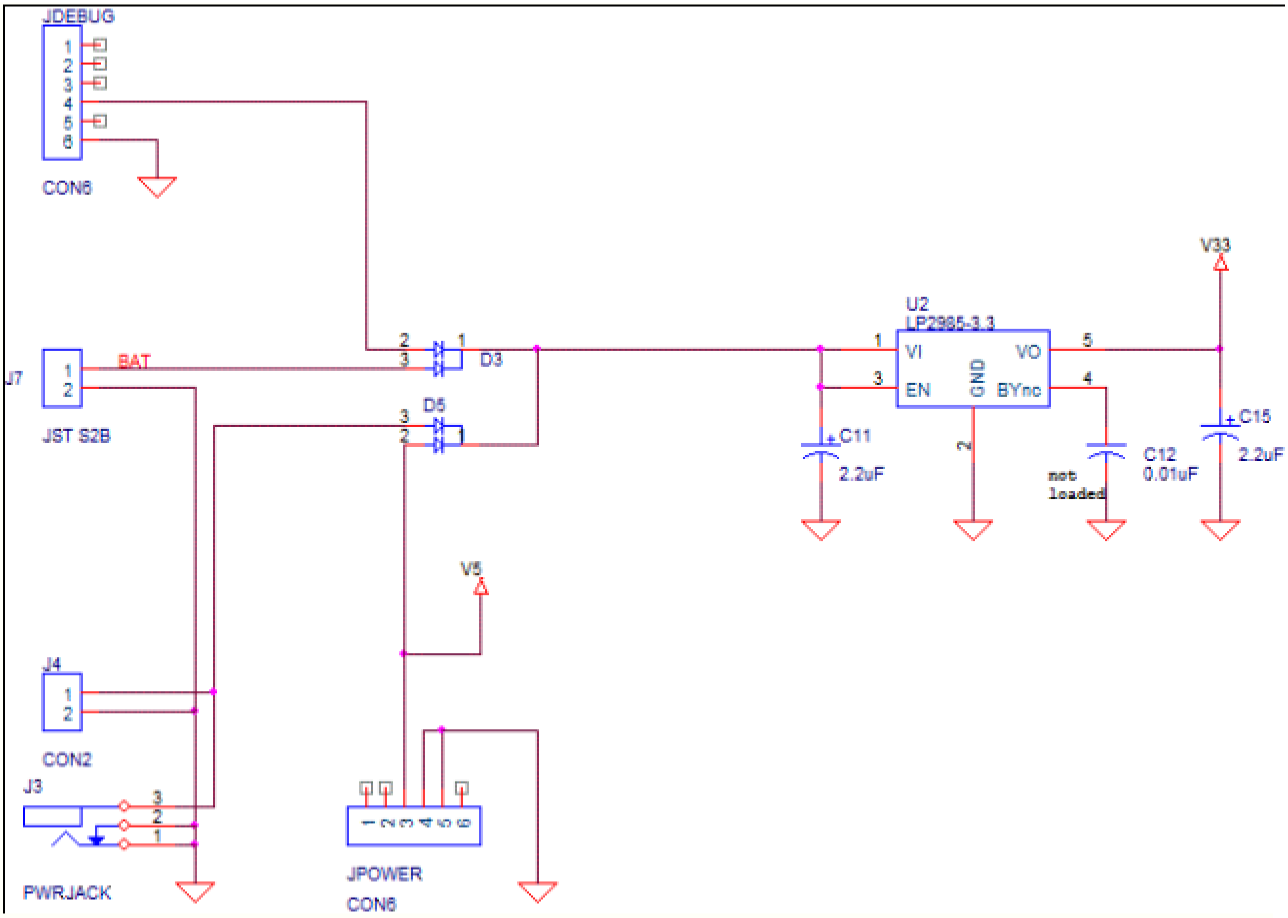
The schematic below describes this circuit on the SuperPRO



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 currrent, 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.

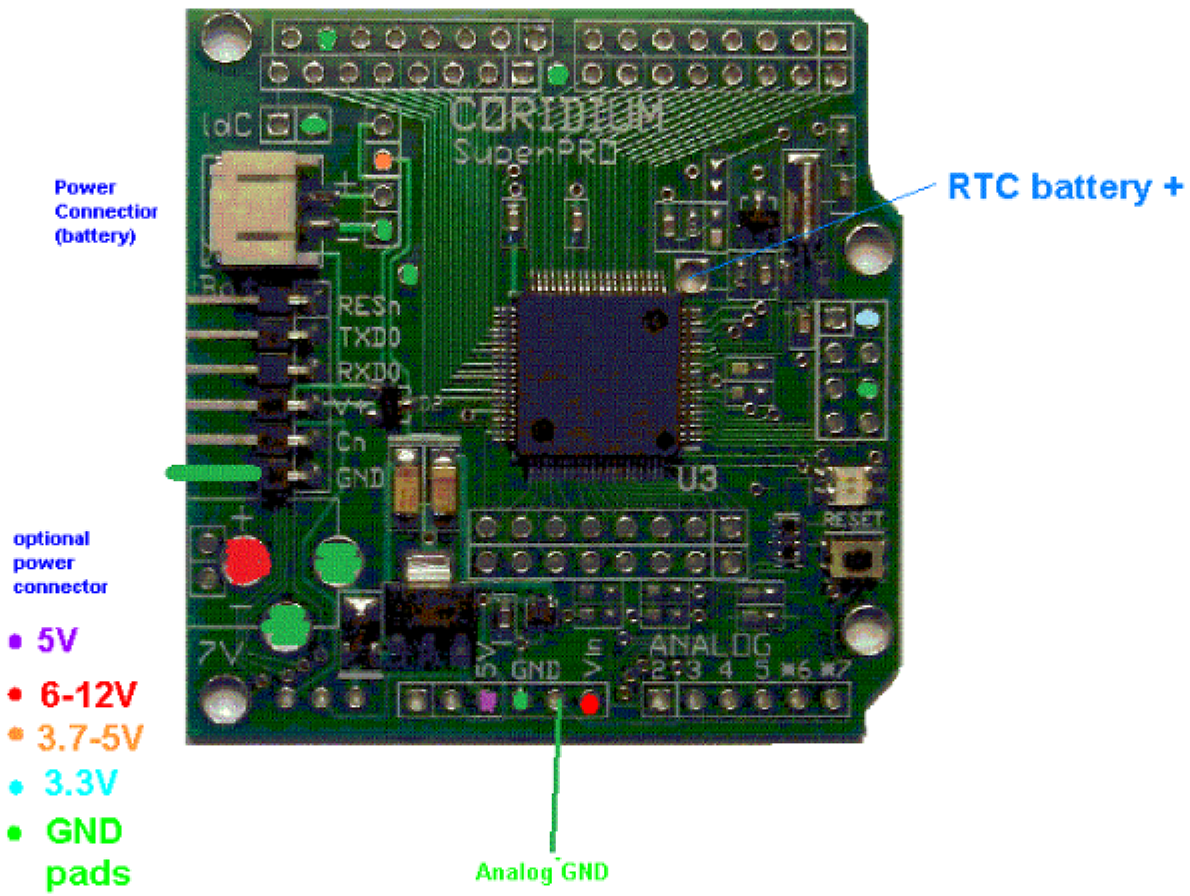
The PROplus only has the 3.3V regulator, so it cannot supply power to a 5V shields.

The schematic below describes this circuit on the PROplus



The **full schematic can be seen here**

Power connections details



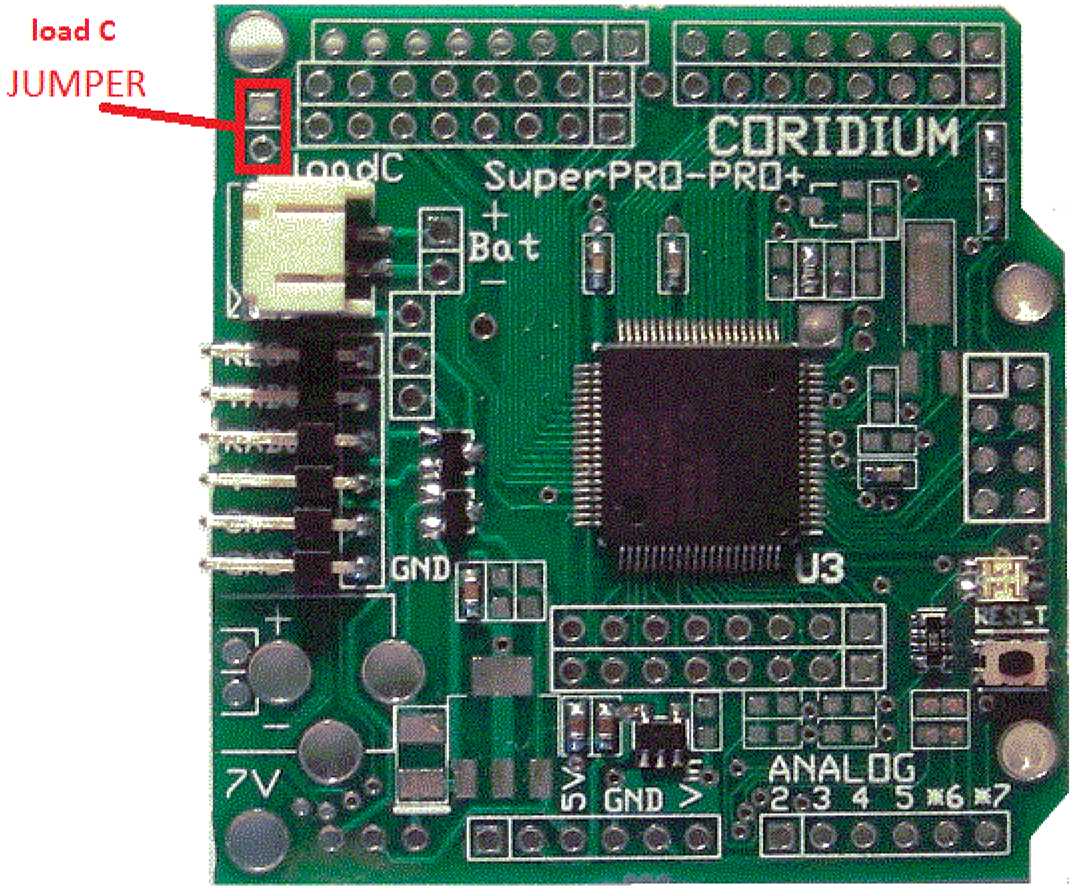
The 3.3V regulator can supply 50 mA, with most being used by the LPC2103. The 3.3V connection next to RESn on the lower power connector is only connected if the shorting pads are shorted (NOT the factory default).
The analog GND should be used to connect to the GND of analog inputs. Digital and Analog 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

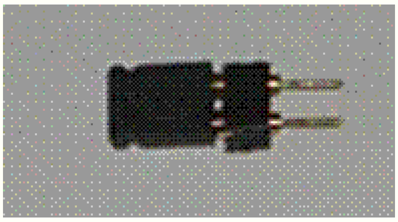
The USB Dongle from Coridium will supply 5V from the USB to power the ARMmite PRO. It also controls the RESET and BOOT signals to automatically load C or BASIC programs using MakeItC or BASICtools. Remember, if you load a C program, it will erase the BASIC firmware and you will not be able to load BASIC 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.



An alternative is to use a 2 pin header with a shorting block (pictured below)

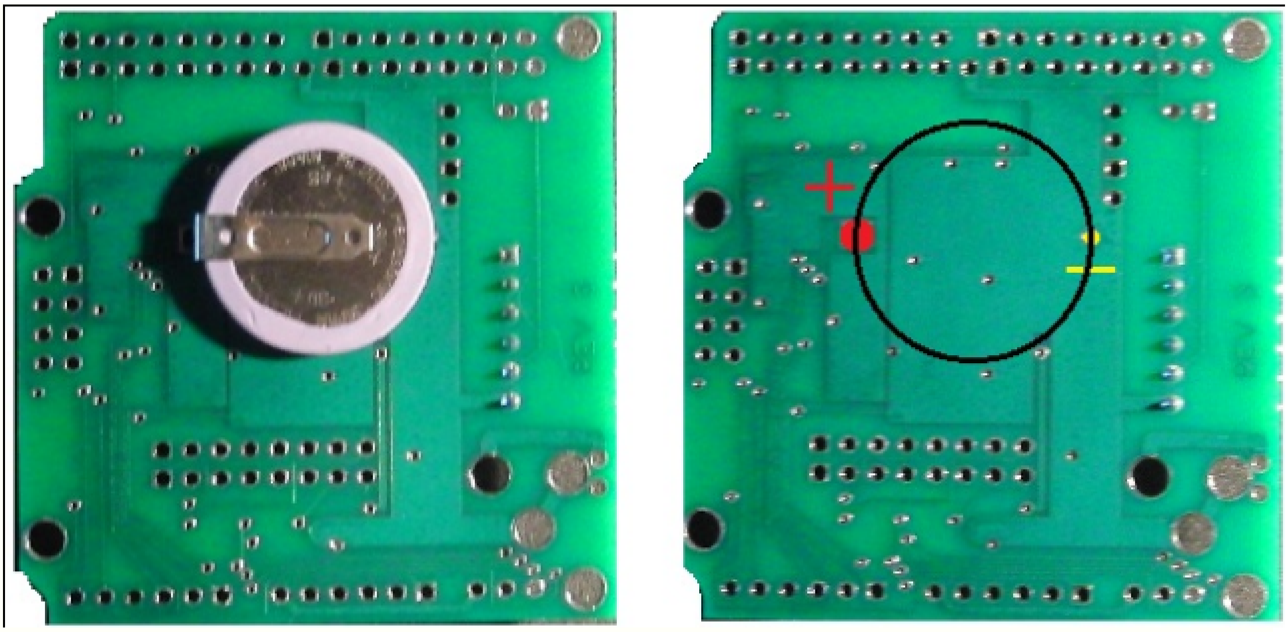


Real Time Clock Oscillator

The RTC oscillator of the LPC17xx parts is not currently reliable see **their errata sheet** . Until that has been resolved, probably with a new revision of the chip, that feature is not available in either the SuperPRO or PROplus.

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.



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.

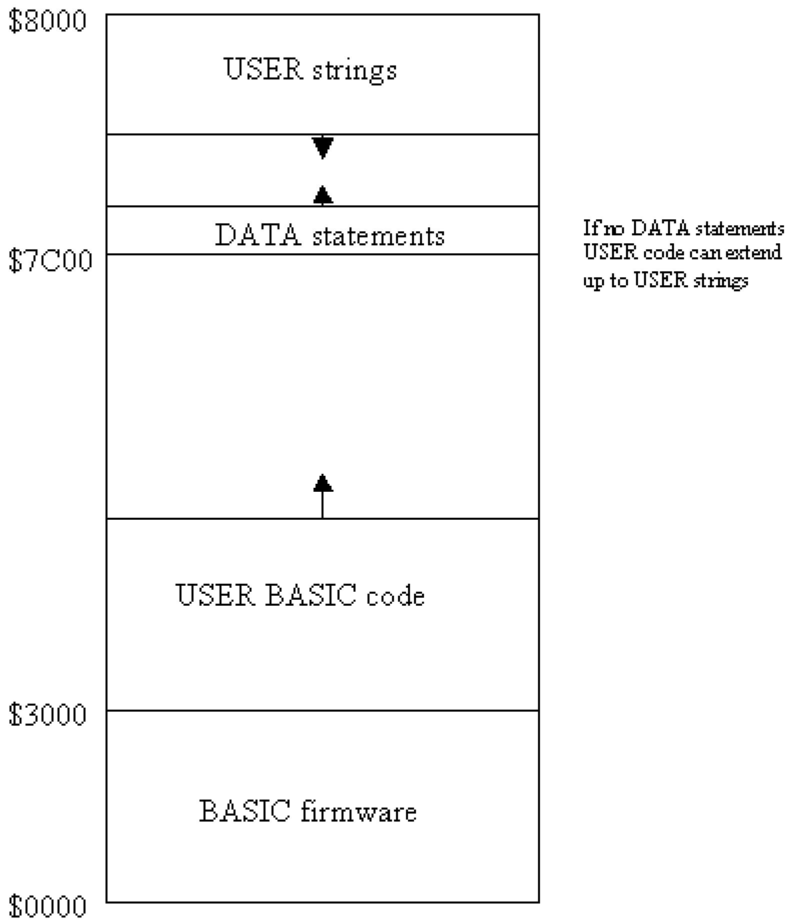
- **ARMmite schematic**
  - **ARMmite rev 2 schematic**
- **ARMmite PRO schematic**
- **PROplus schematic**
- **Super PRO schematic**
  - **USB dongle schematic**
- **ARMexpress LITE schematic**
- **ARMexpress schematic**
  - **ARMexpress Eval PCB**
- **ARMweb schematic**
  - **ARMweb rev 3 schematic**
- **DINkit schematic**
  - **DINkit USB board**
  - **DINkit Ethernet board**

DXF files are mechanical drawings of the boards, they are also available from these links or in the Schematics directory.

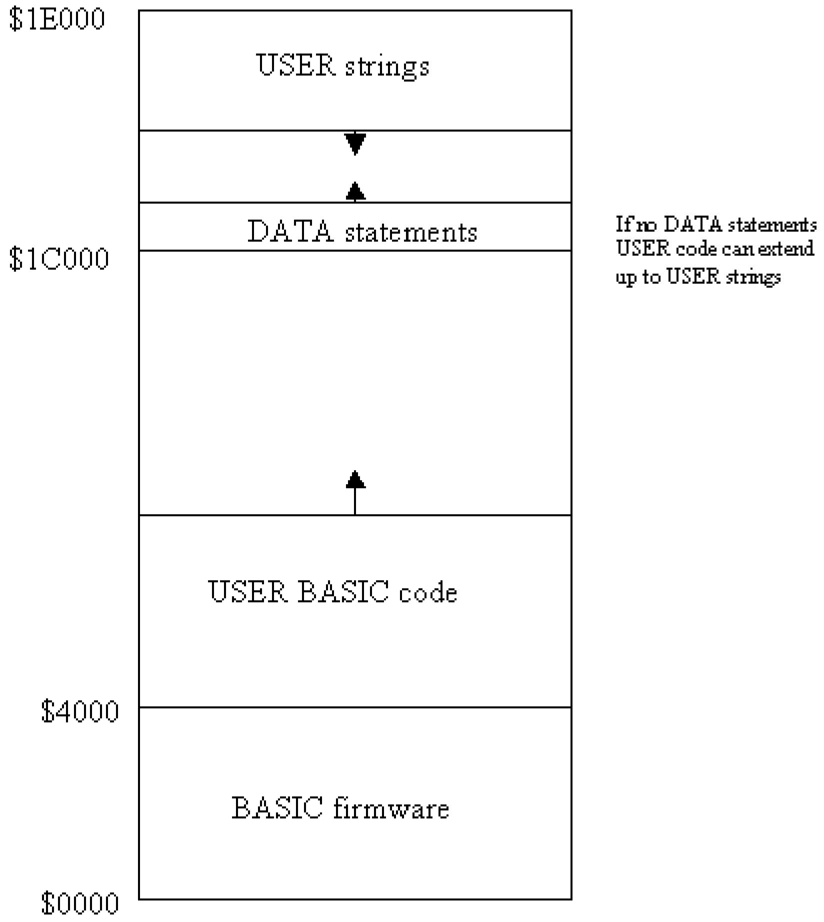
- **ARMmite mechanical**
- **ARMmite PRO mechanical**
- **ARMweb mechanical**
- **SuperPRO PROplus mechanicals**

Memory Maps

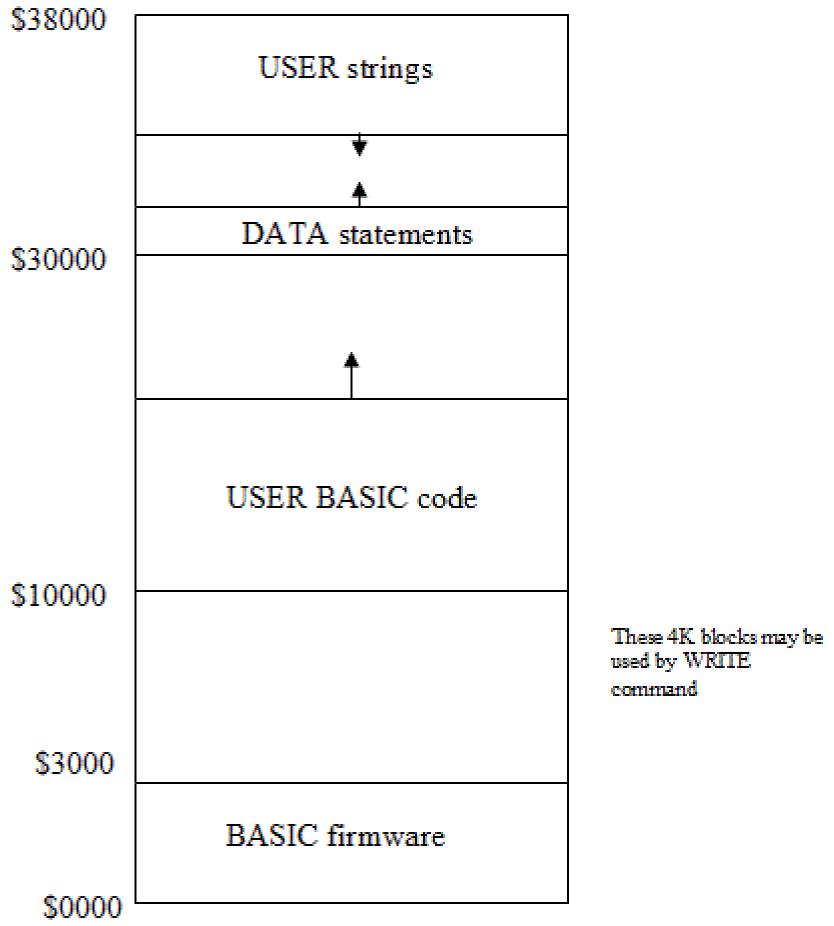
ARMmite ARMexpress LITE, ARMmite PRO, PROplus



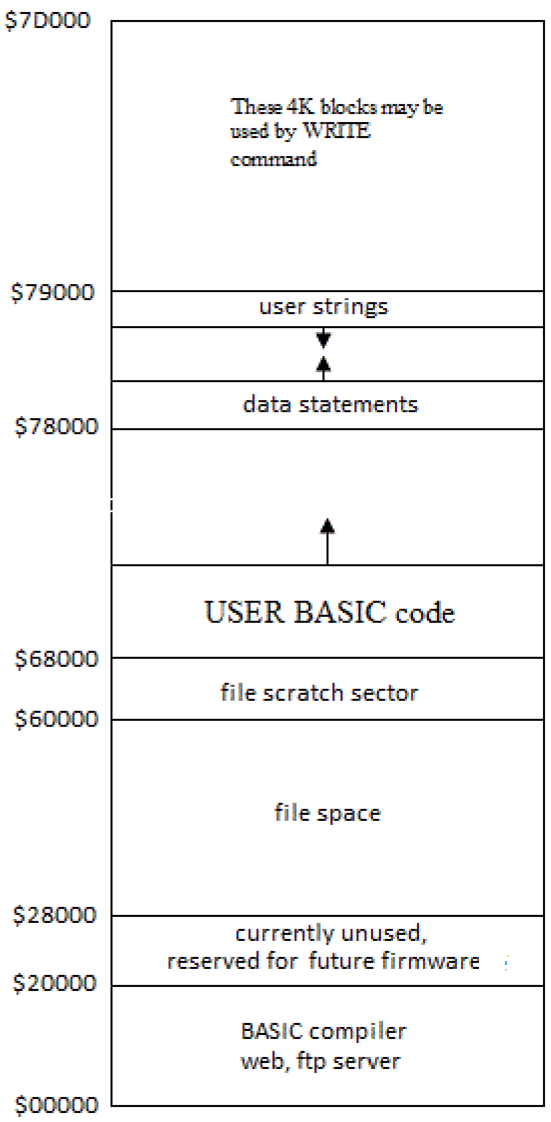
ARMexpress


$1E000
USER strings
DATA statements
$1C000
If no DATA statements USER code can extend up to USER strings
USER BASIC code
$4000
BASIC firmware
$0000

SuperPRO



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.

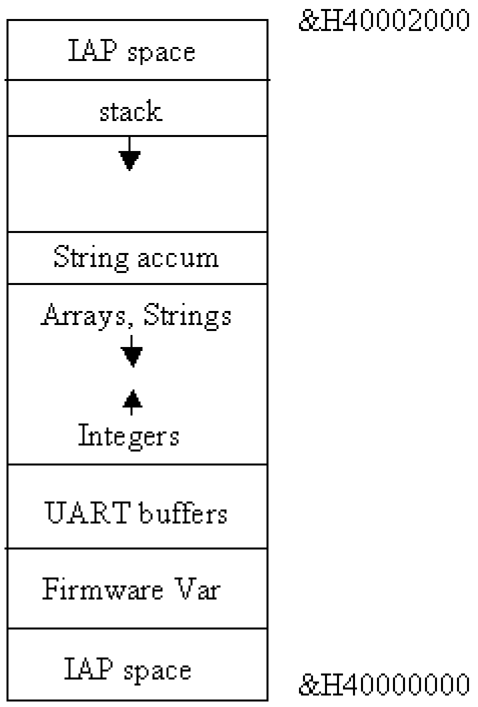
62.5K is available for data (15K words)

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.

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 pullup resistor.

Following reset, the board waits 0.5 seconds for an ESC character, which if received stops the user program from running. If no ESC is received the user program starts.

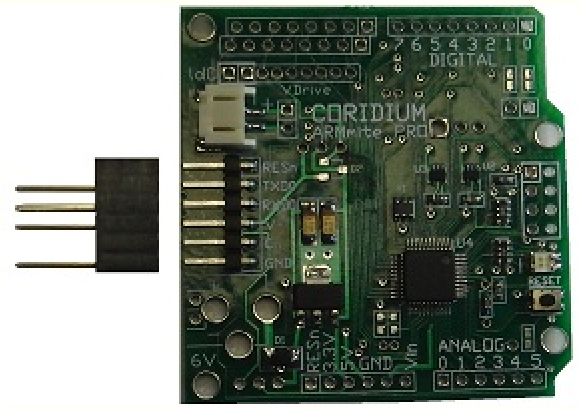
Restarting the program

If the user has entered a BASIC program into the ARMexpress/ ARMmite, 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).

If the user program ends by getting to the last statement of the program or executing an END instruction, the ARMexpress/ARMmite will await either input on the debug serial port, or a RESET.

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 ARMexpress/ARMmite 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/ARMmite 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 ARMmite, ARMexpress or 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

At a STOP the ARMexpress/mite will respond to **^** run or **@** dump-memory commands which are used in the BASICtools variables page.

CPU details

These are links to detailed documentation for the CPUs used in the ARMexpress and ARMmite products. These files are at the NXP website. The links may move so if they are broken here, search their site www.nxp.com

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

LPC1756 used in the Super PRO and LPC1751 used in the PROplus

LPC1756 data sheet

LPC1756 user manual

Serial Configuration

Though we recommend using BASICtools to talk to the ARMexpress, here are settings for other terminal programs.

Baudrate

19.2 kbaud, 8 bit, No Parity, 1 stop bit

End of Line

expects a LF (line feed),

CR is currently ignored.

Voltage Levels

/SOUT, /SIN and ATN (pins 1,2,3) will accept either TTL or RS-232 levels. ATN when high resets the ARMexpress, 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.

Handshaking

XON/XOFF (software handshaking) is used only during programming of the Flash. When downloading a large program, a pause is required when the current amount of code in the buffer exceeds 8k (about 5-600 lines). That buffer will be written to Flash which takes between 0.5 and 1 second (2103 writes 4K blocks and the 2106 writes 8K blocks).

This XON/XOFF is still sent, but a + character is also sent back at the completion of the write Flash Block. And BASICtools now pauses waiting for the +, before sending more data, not relying on the XON/XOFF control in the lower level driver. It was found that especially on the 2106 when not using USB, that the serial driver would drop characters and end up corrupting downloaded code. This is also why you see ...*+*+ during the programming process. The ... indicates the start, the * when BASICtools determines a Flash block will be written, and the + when the ARMexpress/mite responds with the block being completed.

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 (200 msec, 500 msec on Wireless) where the ARMexpress/mite 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/mite is ready to be reprogrammed. Or the user can restart the program by typing RUN or using the RUN button in BASICtools.

Program Running Signaling

When the user code starts running, an SOH (\1) character is sent, and when the user code stops an EOTX (\4) is sent. This was added for the ARMmite, as BASICtools needs to know when the user code is running. ARMexpress versions starting with 6.11 also support this.

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, and in some cases reboot the system.

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.

TclTerm.tcl when used as a stand-alone terminal emulator will also maintain its own initialization file TclTerm.ini.

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 the ARMexpress family. 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 at the ARMexpress Yahoo Groups Files Section where you will also find a sample C program (writen for MinGW) that will also communicate with the ARMexpress family.

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, and for version 7 TXD0 or for version 6 SEROUT 16,... , or TXD(16)=

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 practive 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

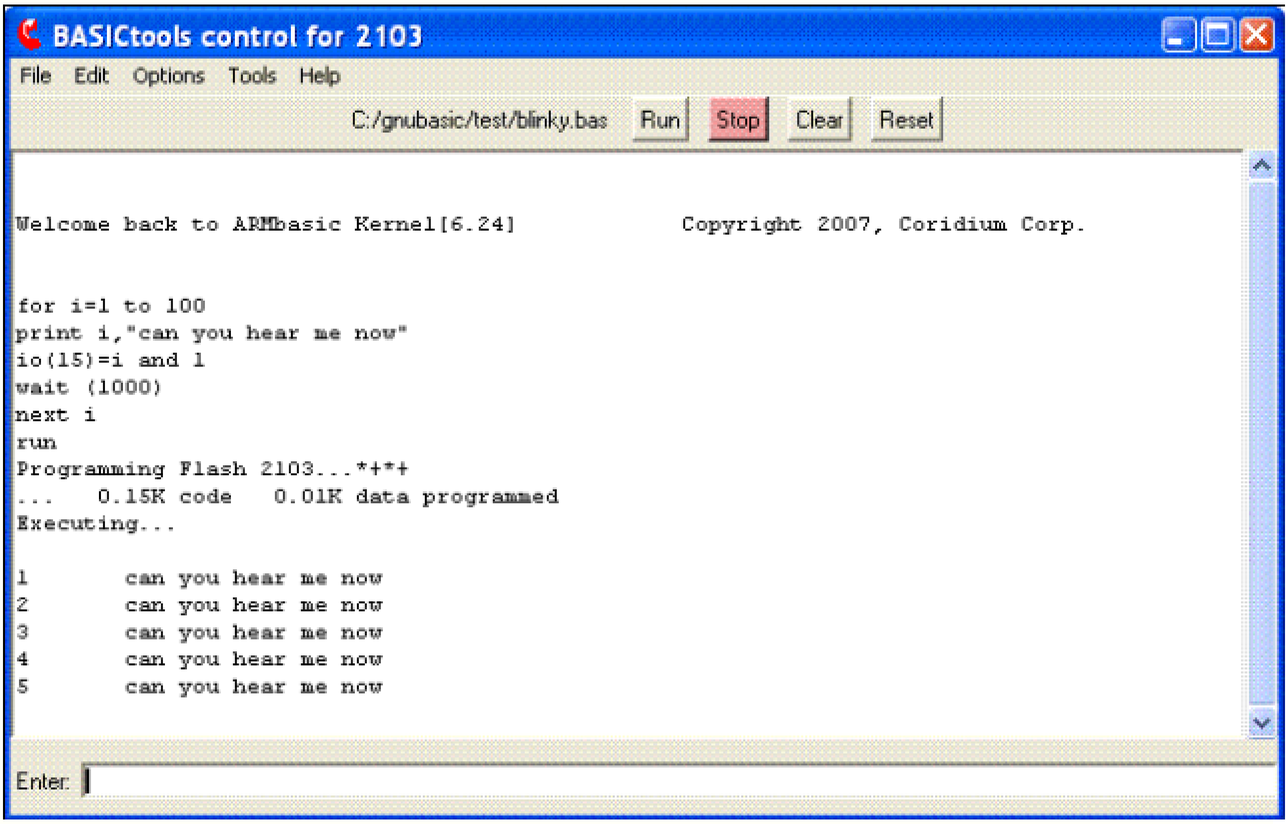
The ARMmite and ARMexpress use the DTR and RTS serial control lines to control programming and reset for the device. The state chosen allows the ARMmite/express to run and be reset by the push button while idle (ie. no serial program running).

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 Tclterm

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.

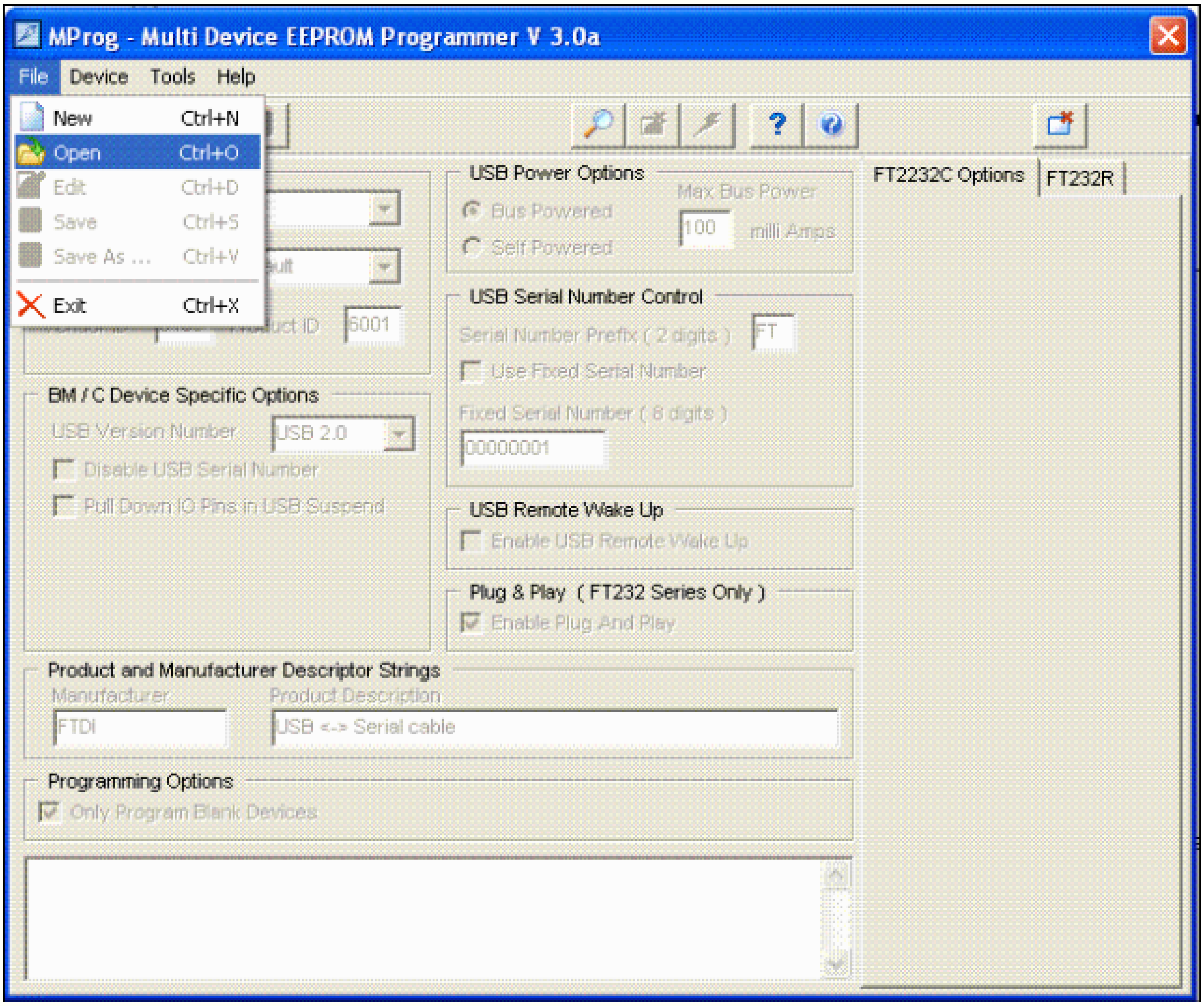
http://www.ftdichip.com/Resources/Utilities/MProg3.0_Setup.exe

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.**

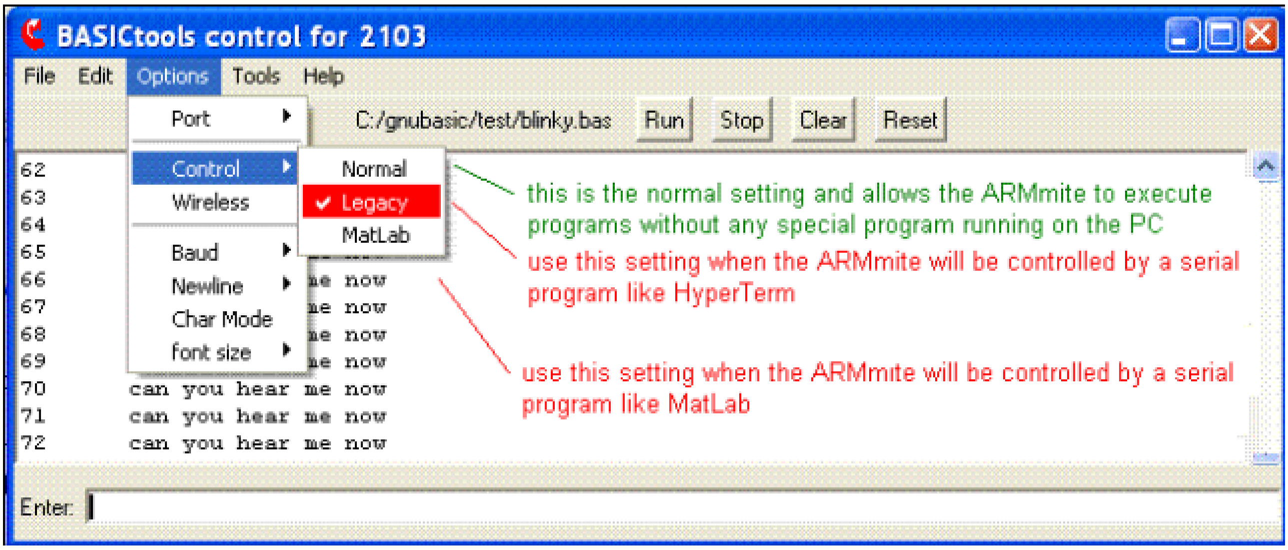


Exit this program and close any serial programs such as BASICtools. For this change to take effect, the ARMmite/express must be disconnected from the PC and reconnected.

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

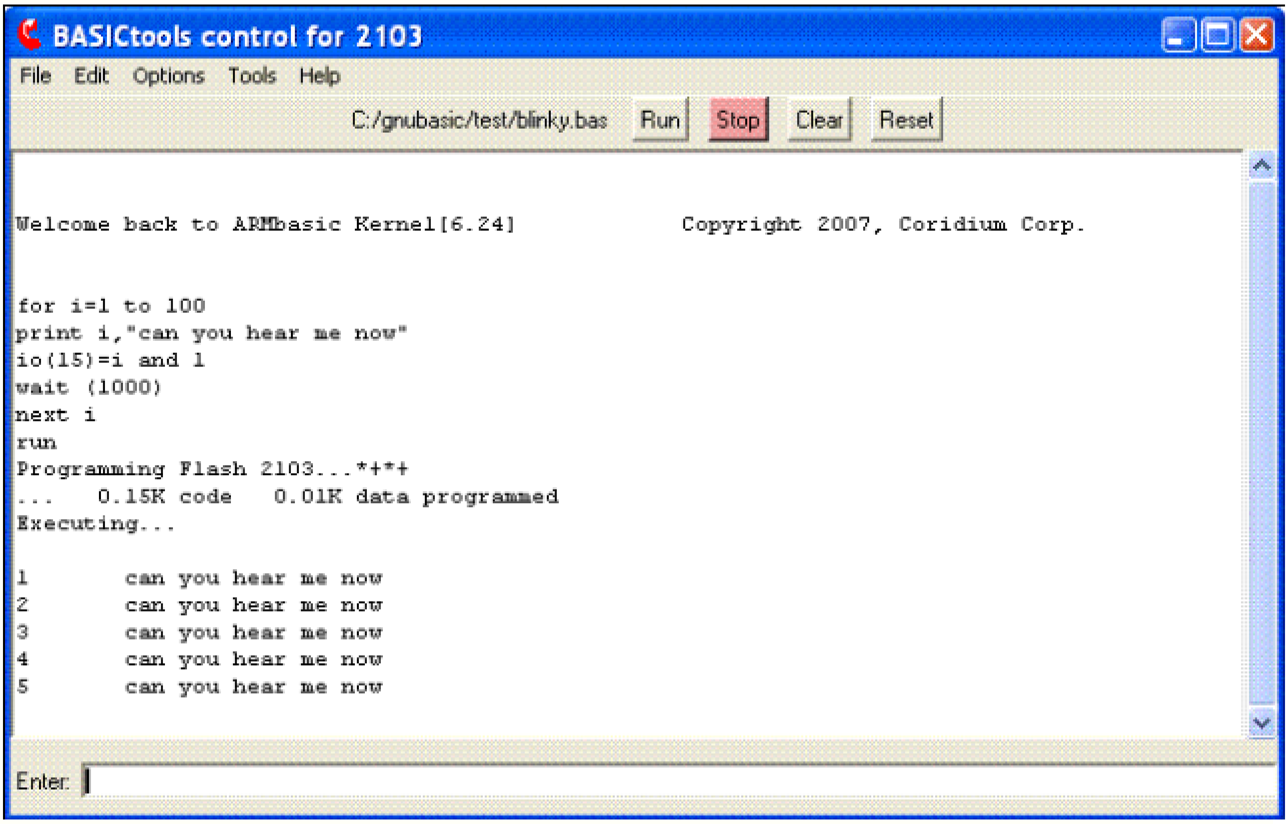
General Info

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 (ie. 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 Tclterm

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.

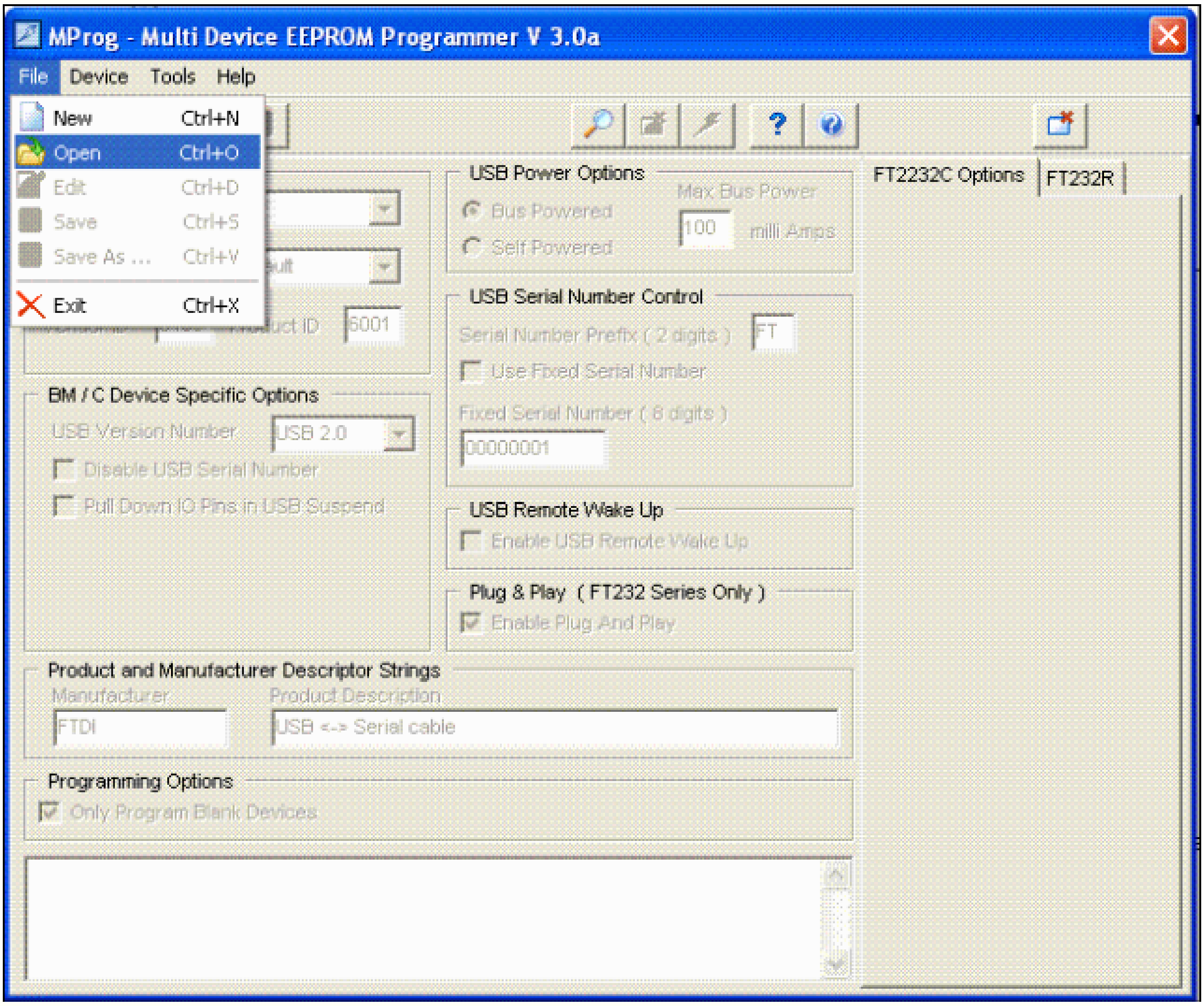
http://www.ftdichip.com/Resources/Utilities/MProg3.0_Setup.exe

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.**

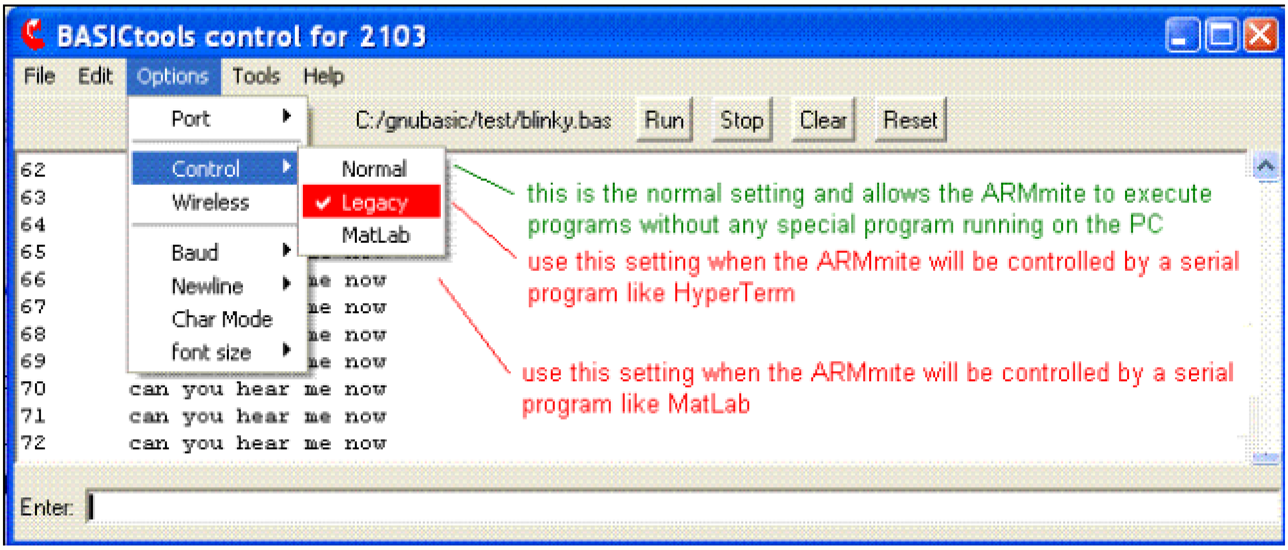


Exit this program and close any serial programs such as BASICtools. For this change to take effect, the ARMmite/express must be disconnected from the PC and reconnected.

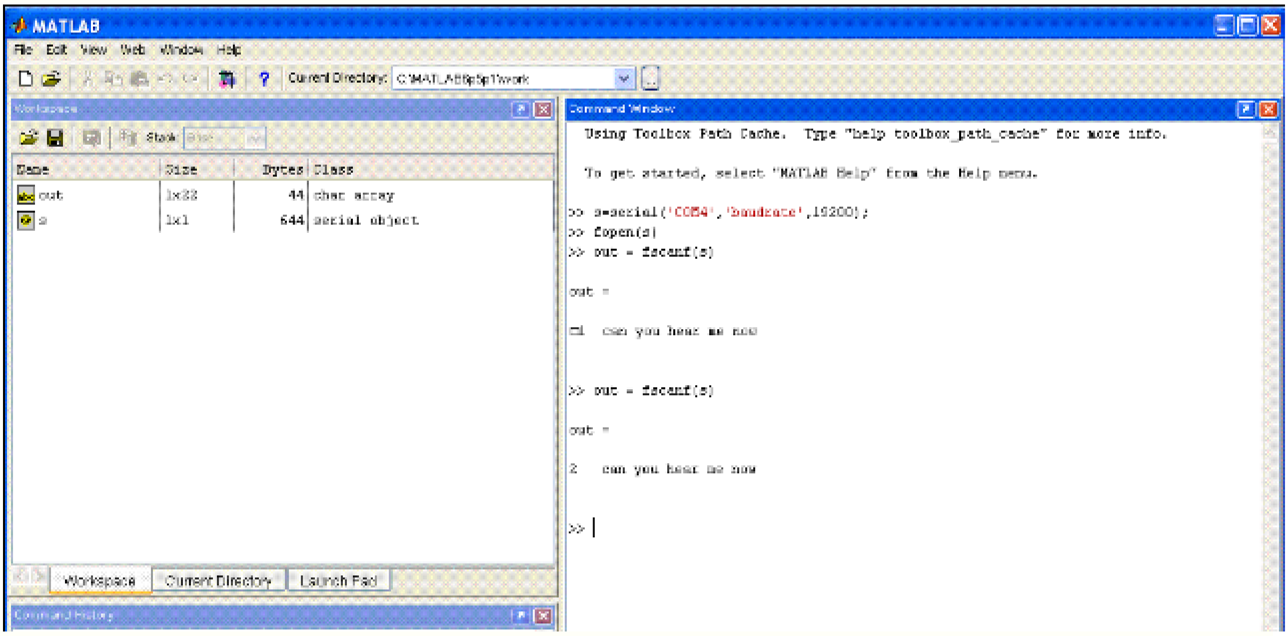
Now the ARMmite/express will be idle until the MatLab serial port is open. Then after those programs are run, to start your BASIC or C program press the RESET pushbutton on the ARMmite/express.

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).

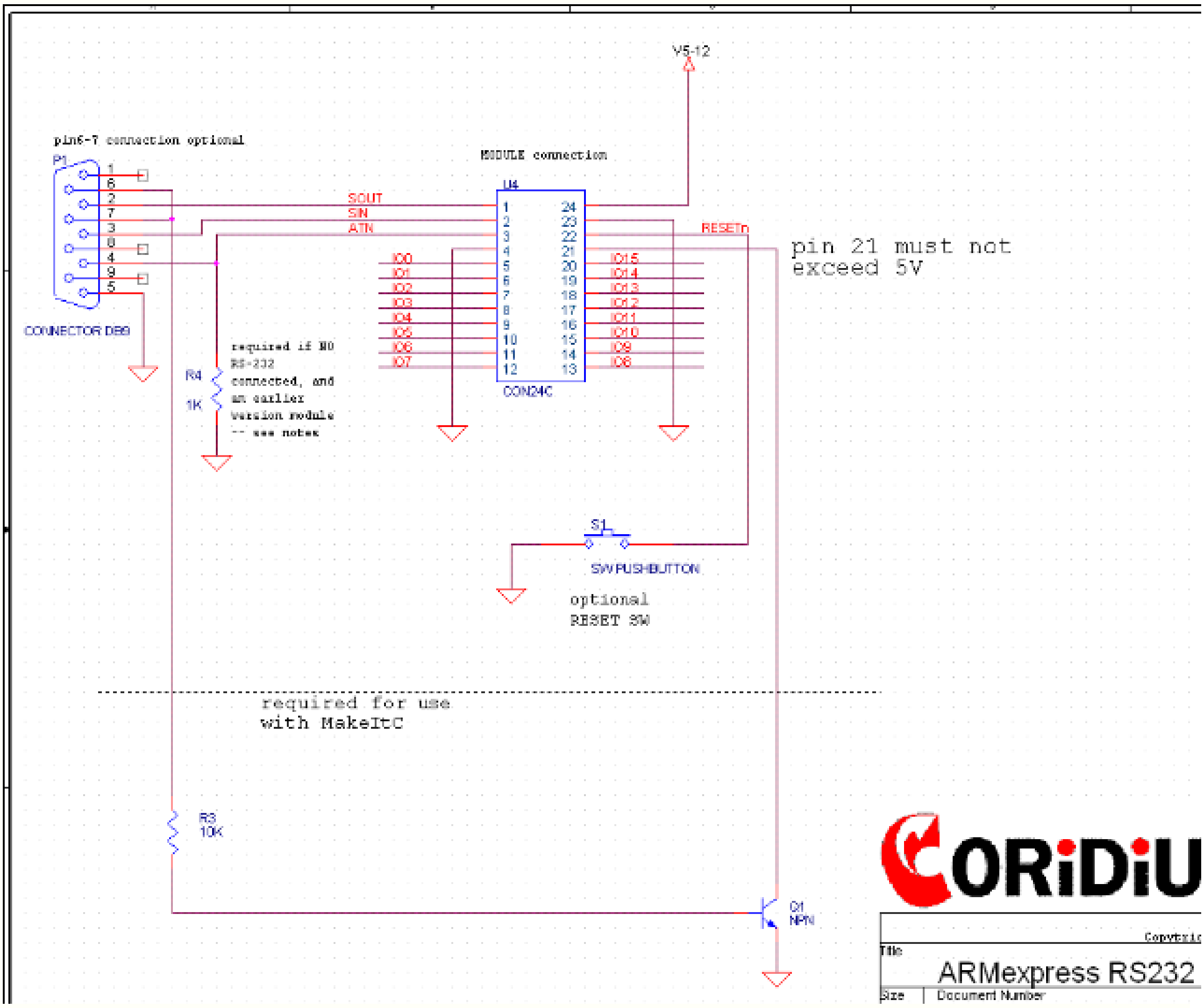


Check operation with MatLab



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.

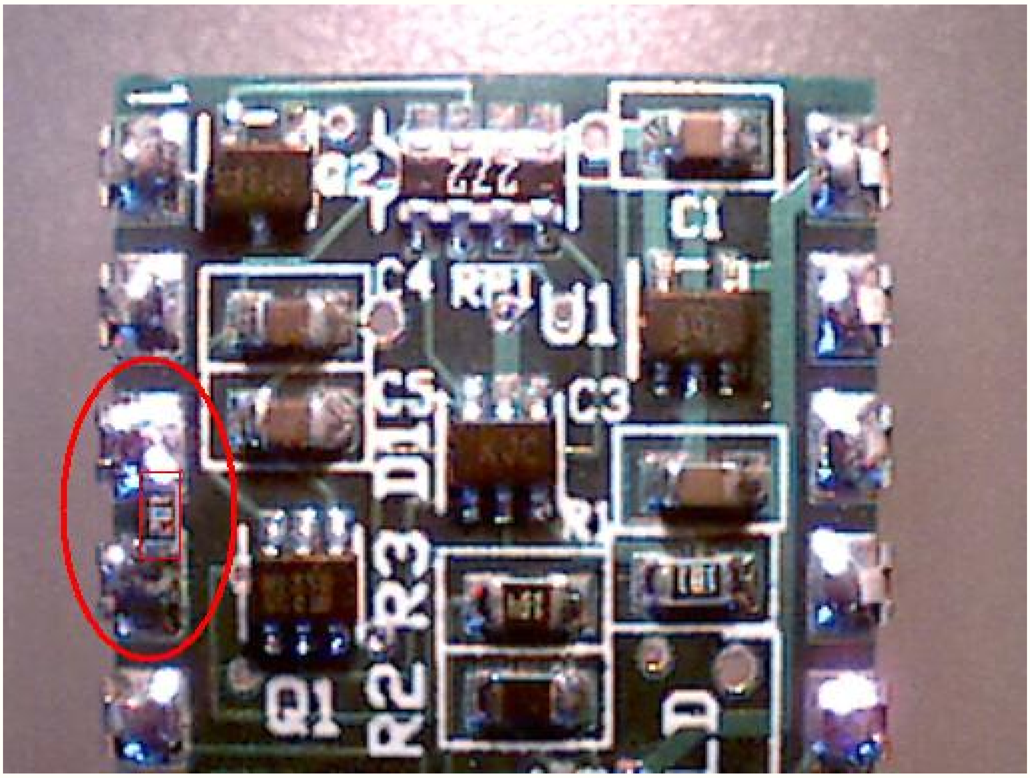
Do not connect a power source greater than 5V directly to pin 21.

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.

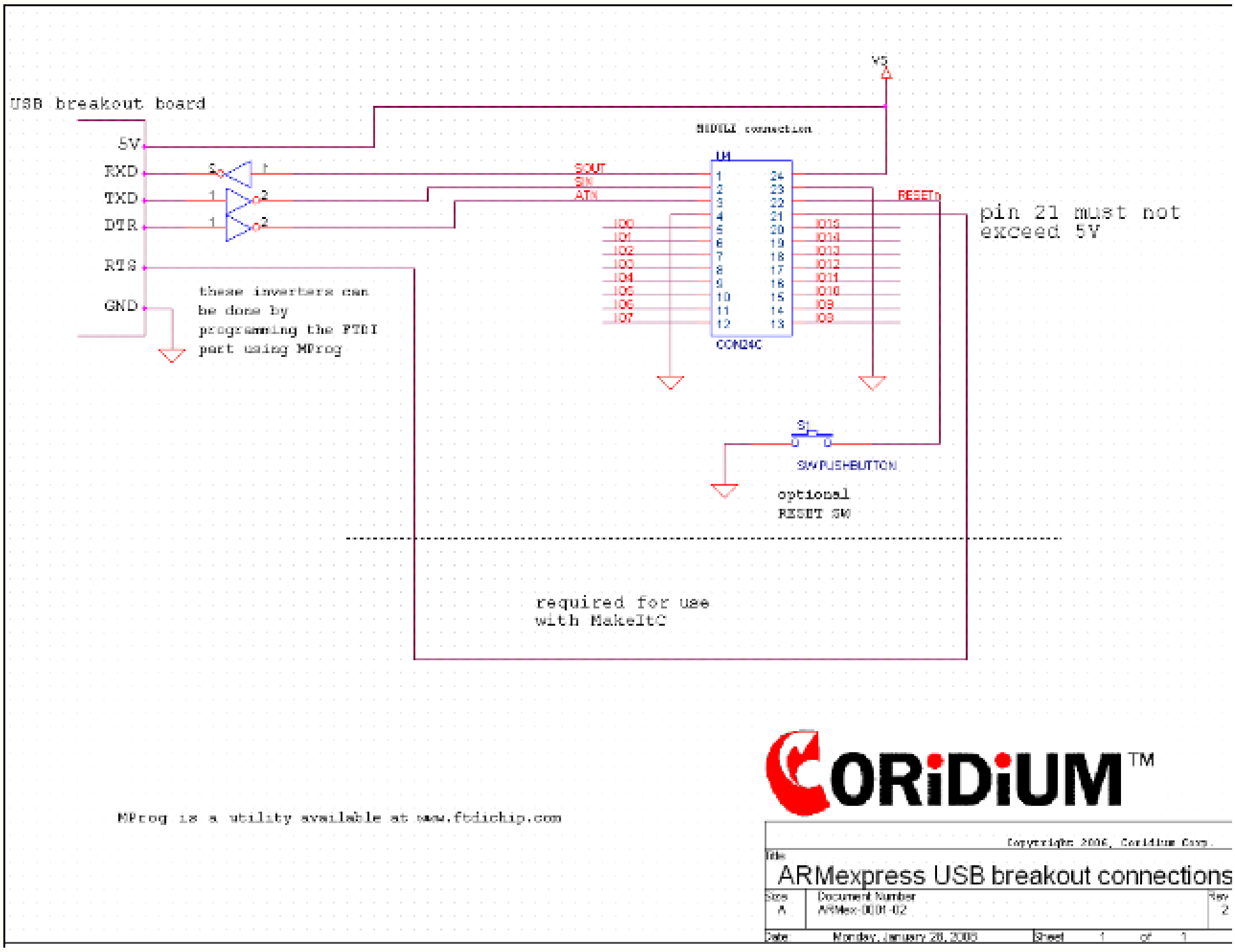
Pin 3

On later revision ARMexpress and ARMexpress LITE a 1K pulldown has been added on the module between pins 3 and 4 (as pictured below. If your unit does not have this, then a 1K pulldown 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

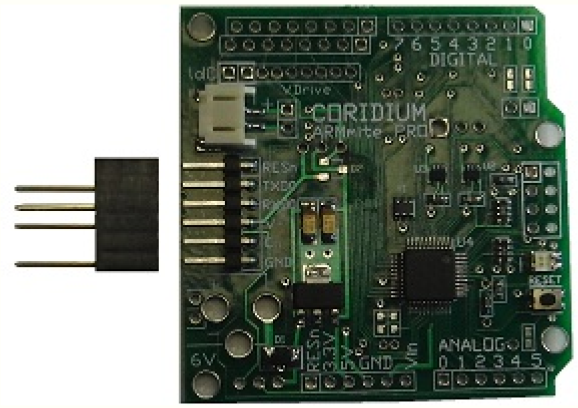
PIN 22 high

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**).



CORIDIUM
ARMmite PRO
DIGITAL
ANALOG
0 1 2 3 4 5
7 6 5 4 3 2 1 0
RESET
RESn
TXD0
GND
6V
3.3V
5V
GND
Vin

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 upto 4mA of current. Most TTL devices will recognize these as valid logic levels (normally defined to be 0.8 and 2.0V)

Inputs

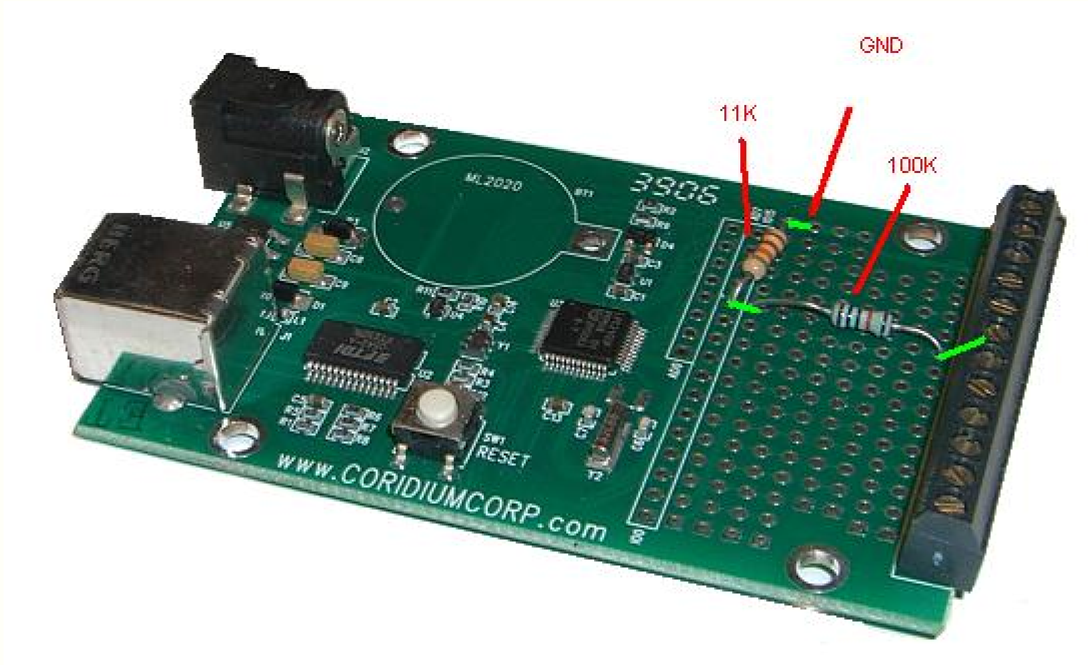
The ARMexpress and ARMmite may also be directly connected to 5V TTL outputs. If they are TTL compatable the voltage levels of the TTL output would normally be (0.4 and 3.4V), though they may go higher. The inputs for these ARM devices are 5V compatable.

Tieing to Supply lines

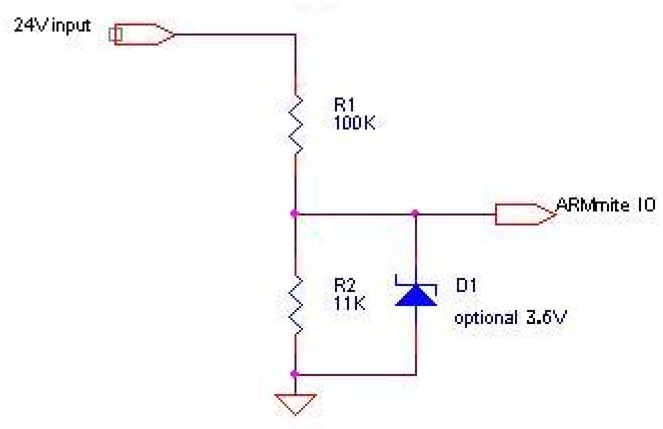
The ARMexpress and ARMmite inputs may be connected directly to a GND pin, but if connecting to a fixed voltage supply, then a 1K or greater resistor in series is recommended. This is the same recommendation for any TTL compatable device. The reason being is that the 5V supply may exceed the 5V at times, 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.



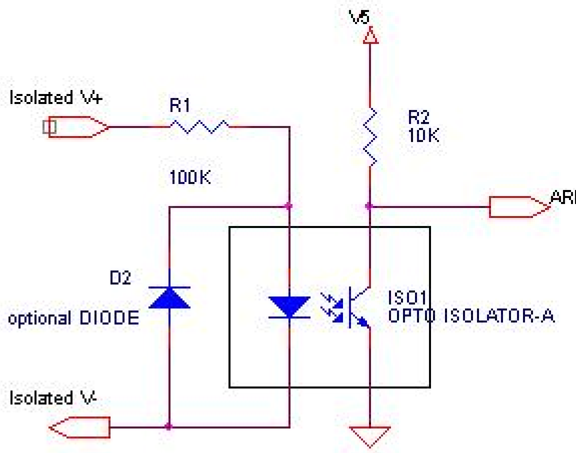
This resistor divider divides the 24V by 10 and also limits the current if that 24V goes higher. The circuit below shows schematically the connection that was made.



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, thats what the catch diode provides is a path for that current to flow back into the supply of the relay.

Power

Common to all boards

Initial Power on conditions
On power up all pins are tri-stated on the ARMexpress/ARMmite.

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 compatable 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 compatability

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 compatable 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 excercised, 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.

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, PWM and FREQOUT 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 interupted for TBD microseconds. The ARMconnect also uses a 10 msec timer interupt. 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 interupts during that critical period. These operations include OWIN, OWOUT, SERIN and SEROUT. Other operations that would be negatively impacted by an interupt also disable the interup for a period of time. Those include PULSIN, PULSOUT, PWM, RCTIME and FREQOUT.

Interupts 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 noticedable, 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 FREQOUT or 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.

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.

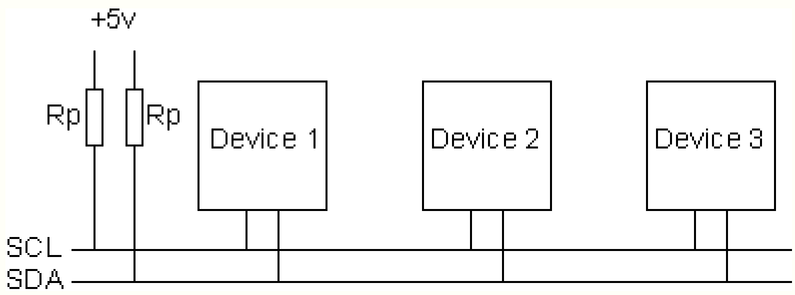
In practice, many devices have exceptions. Some read data as the clock goes up (**leading edge**), others read as it goes down (**falling edge**). Writing is almost always on clock movement that goes the opposite direction of reading. Some devices have two clocks, one to "capture" or "display" data, 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.

Using the I2C Bus

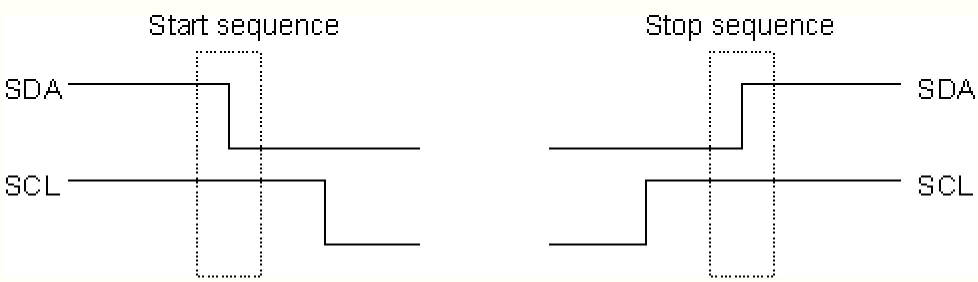
The physical I2C bus
This is just two wires, called SCL and SDA. SCL is the clock line. It is used to synchronize all data transfers over the I2C bus. SDA is the data line. The SCL & SDA lines are connected to all devices on the I2C bus. There needs to be a third wire which is just the ground or 0 volts. 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:



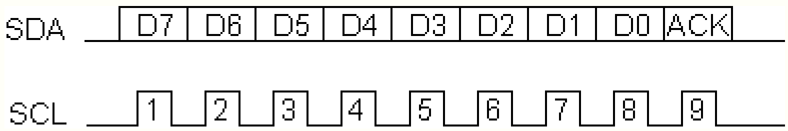
The value of the resistors should be from 1.8K (1800 ohms) to 4.7k (4700 ohms). It depends on the length of the I2C bus, the longer the bus, the smaller value should be used. If the value is too large, the rise time of the signals will be too slow and the bus may not work properly. If the resistors are 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.

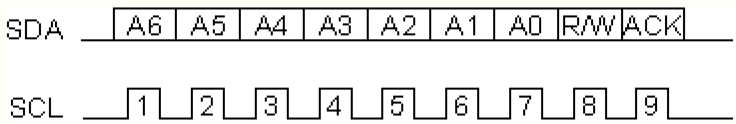


Data is transferred in sequences of 8 bits. The bits are placed on the SDA line starting with the MSB (Most Significant Bit). The SCL line is then pulsed high, then low. Remember that the chip cannot really drive the line high, it simply "lets go" of it and the resistor actually pulls it high. For every 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).



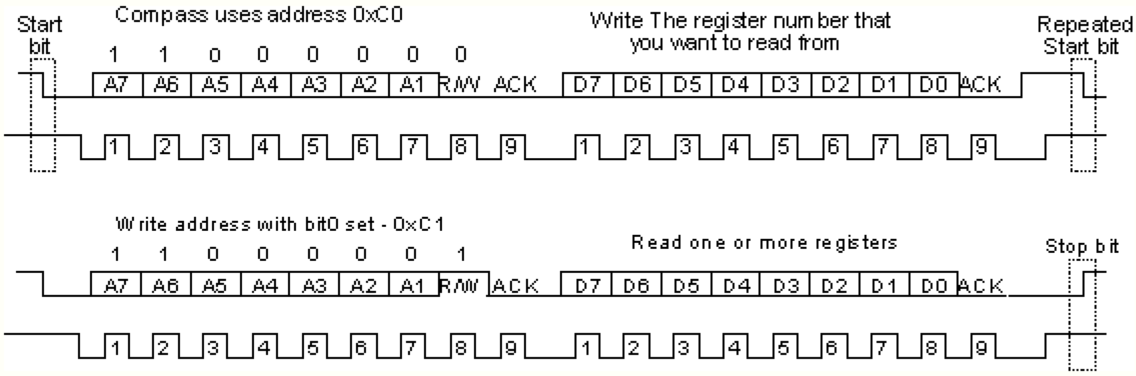
The placement of the 7 bit address in the upper 7 bits of the byte is a source of confusion for the newcomer. It means that to write to address 21, you must actually send out 42 which is 21 moved over by 1 bit. It is probably easier to think of the I2C bus addresses as 8 bit addresses, with even addresses 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 seperate 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".

ARM Peripheral Use

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   Not Used unless requested by user with BAUD1
PWM   used when HWPWM is engaged on PROplus, SuperPRO
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

```
ISR21          UART0
ISR22          UART1
```

```
If a function is not included in the BASIC code the interrupt is available,
for instance ON TIMER uses TIMER0 interrupt and RXD1 uses the UART1 interrupt.
In Idle just the CPU clock stops and any interrupt will wake it.
```

Background Tasks

Except for the ARMweb, the only background tasks are interrupt handlers for UART0 and UART1. UART1 is not active until the BAUD1 function is called.

ARMweb Ethernet Services

ARMweb Ethernet Services 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

ARMweb Getting Started

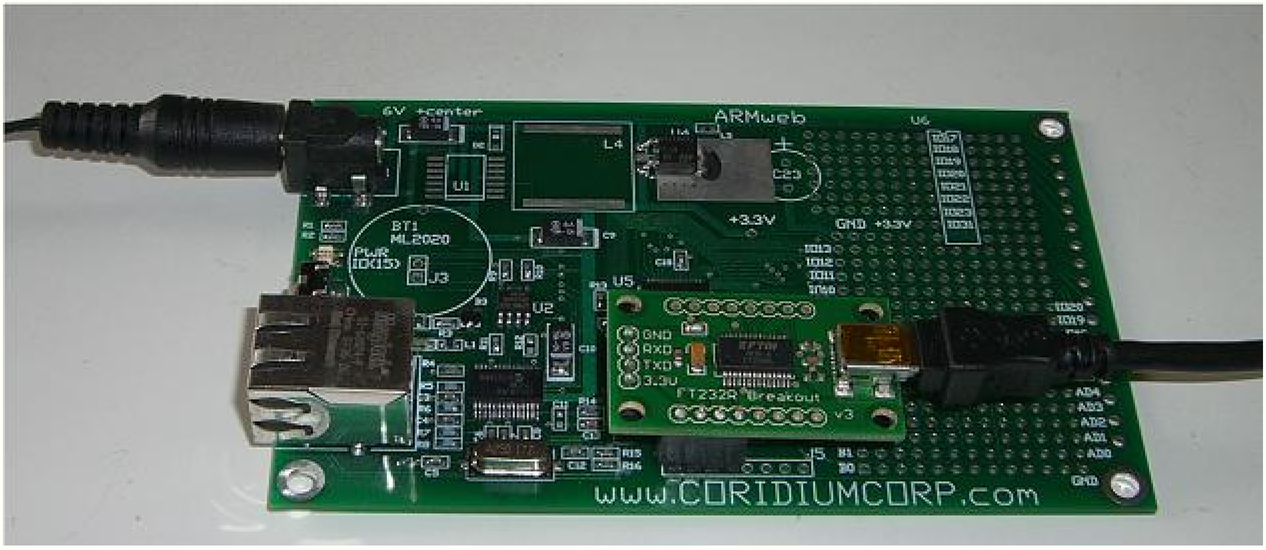
Getting Started

- Install Software
- Connect Ethernet
- USB connection for ARMweb
- Writing simple programs via the web
- Writing programs with BASICtools

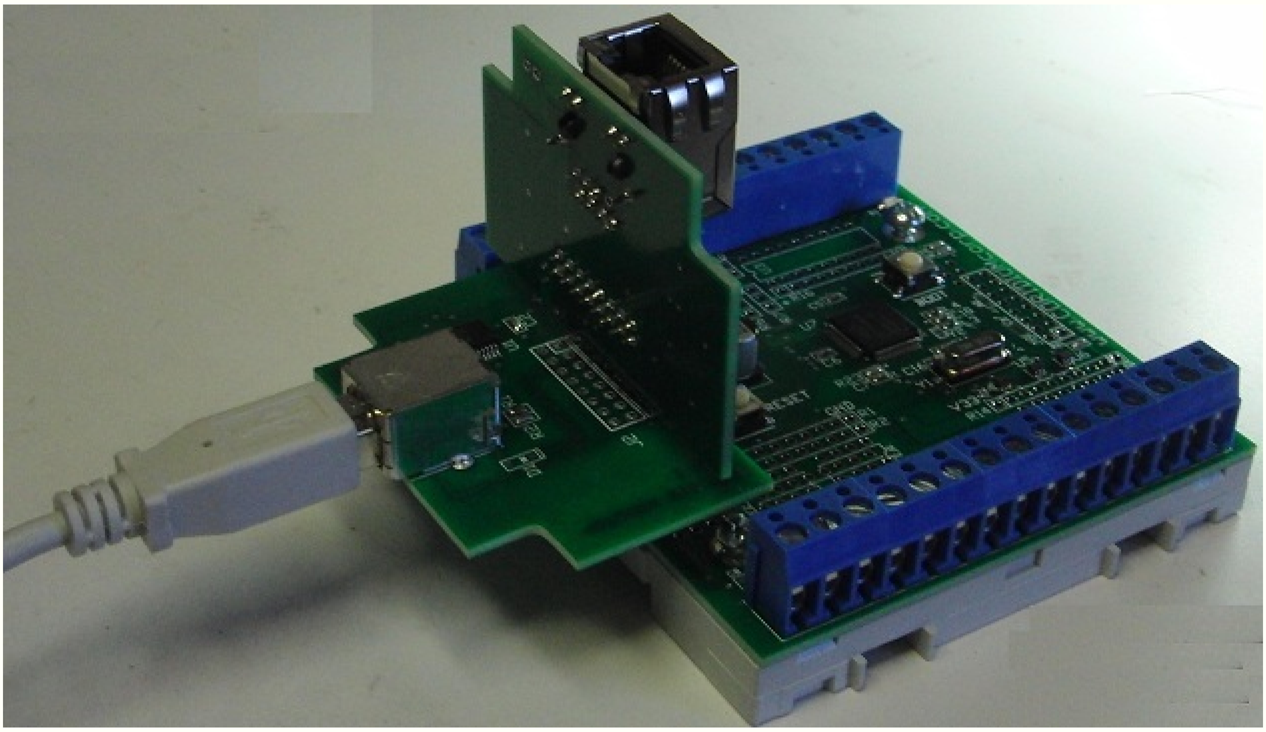
Optional: USB connection for BASICtools

While the ARMweb can be programmed through the webpage, during the development cycle BASICtools can be used via a USB connection. BASICtools has a much faster response than a browser.

The attachment of the USB and power supply is shown below. 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).



ARMweb


DINkit (ethernet)

Why use BASICtools?

Browsers are very slow when refreshing a webpage, so the interaction with the programmer is better with BASICtools.

#include can not be used from a webpage, as the ARMweb does not have direct access to the #include'd file

The BASIC compiler on the PC has more memory for the symbol table and can handle larger programs than when compiling on the builtin ARMweb compiler.

The variable dump tool is available in BASICtools. Debug messages are sent to the USB port, as well as <?BASIC ... ?> source and output when processing web requests. When your program is debugged and AutoRun is turned on the USB port is turned off. You can improve the performance of the web server BASIC compiler by increasing the speed of UART0, by changing baud settings in BASICtools and executing BAUD0(937500) in your main program.

For an introduction to BASICtools refer to the **ARMmite sections** .

BASIC and Webpage interaction

BASIC can be embedded in the webpage served by the ARMweb. That BASIC code can access global variables of the user program running on the ARMweb. At present, BASIC embedded in the webpage can not call a FUNCTION or SUB (this will be a future enhancement).

The user (client) can also interact with an ARMweb BASIC 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 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. 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.

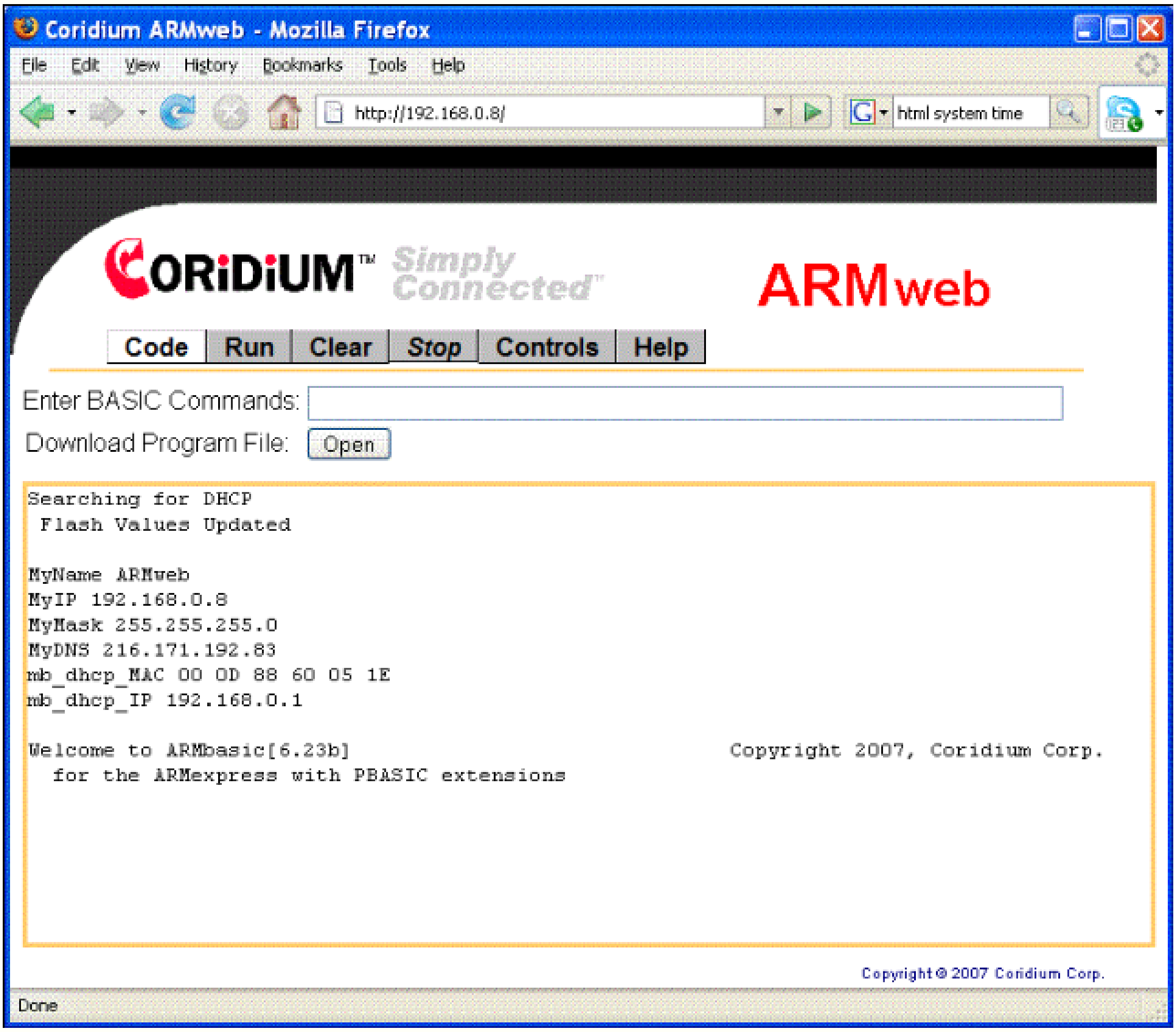
or

More details on ARMweb and BASIC...

armweb.htm PAGE

Description

This page is the main control page for the ARMweb. It is always available even if the main page served is a user generated page. It can be accessed at the armweb.htm page.



Code :
The default is to come to this page. A user BASIC program can be typed in line by line, or downloaded using the OPEN button.

Values: (potentially obsolete)
Variables in the user BASIC program can be accessed from this page. Those variables have to be declared as either WEB or WEB READONLY.

With the new user webpage features, this function may go away in a future release.

Run/Stop :
This button will either run or stop the previously loaded user BASIC program. This function is disabled when when security is set on the **Controls Page** .

Clear :

This will erase any user program. This function is disabled when when security is set on the **Controls Page**.

Controls :
This accesses the **Controls Page**. It will be disabled when security is set.

Help:
Currently has no function, and may be eliminated in a future release, or linked to the Coridium Web Site help files.

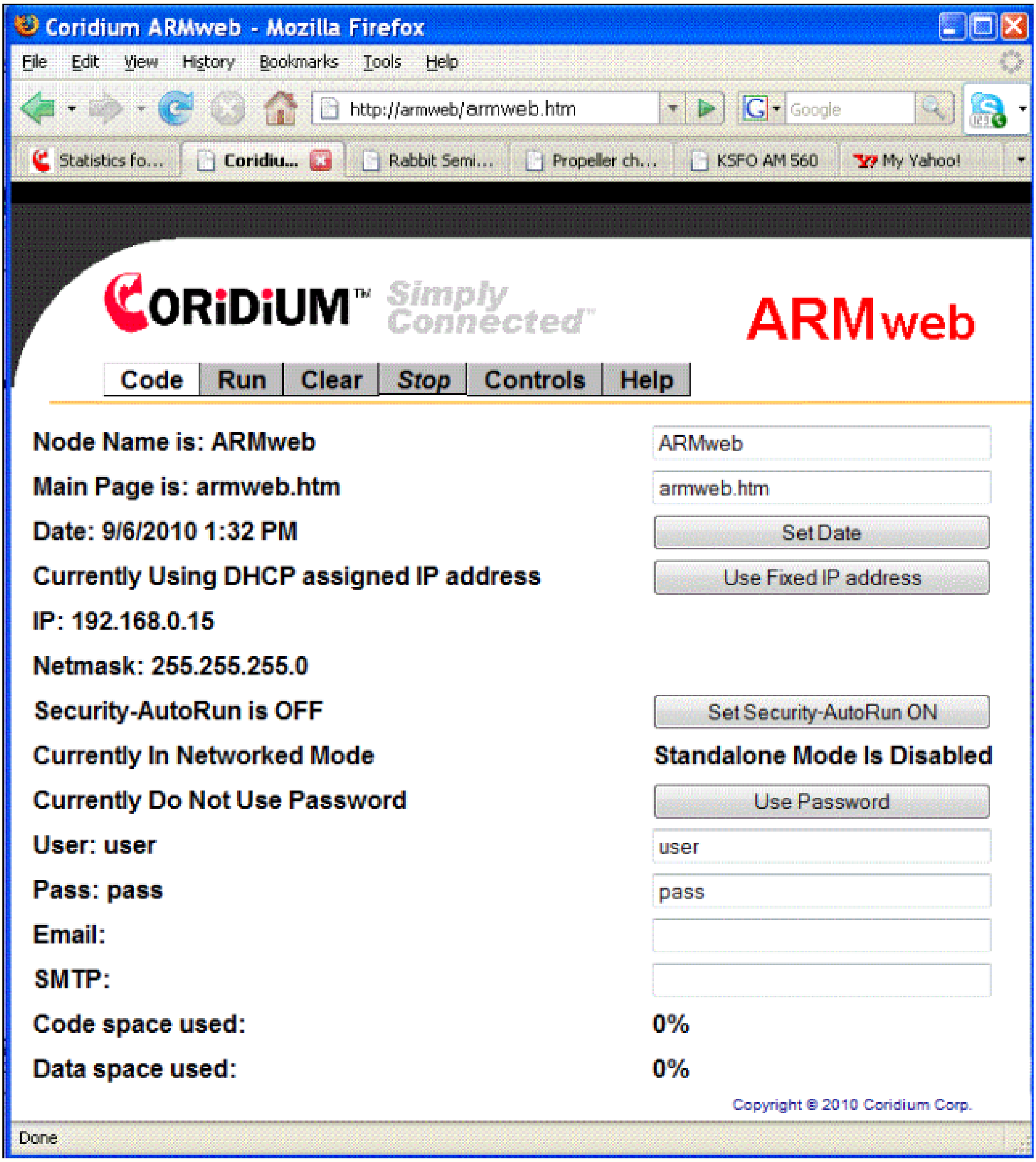
See also

- **UDP Services**
- **FTP Services**

Controls PAGE

Description

This page controls the ARMweb. It is always available even if the main page served is a user page. It can be accessed at the armweb.htm page.



Main Page :
The default is to come to this page. When the user loads their own pages via ftp, then this can be changed to make the main page the user generated page

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.

Security is OFF :
When Security is OFF the user's BASIC program will not start on reset. Also you have access to the Controls Page, ftp server and run/stop clear buttons.

When Security is ON, the users BASIC program is run on reset. This will also lock out ftp, control, stop, clear and code page access when the user program is running. When enabled updates can not be made from the web as long as the user BASIC program is running. When enabled and the user program is running, the only way to make changes is to physically hold the push-button on the ARMweb during power up, which returns to the factory defaults (including erasing the program and any files in the ftp space).   You must also change the default passwords for this to work.

When Security is ON, debug messages to UART0 (and via USB dongle to BASICtools) are disabled. This improves the performance of web server.

Standalone/Networked Mode :
This is part of the initial configuration. The default is Standalone mode, but it will switch to Network mode if the ARMweb ever gets a response from a DHCP server. While in Standalone mode, the IP address will be normally 192.168.0.50, and the ARMweb will act as a mini-DHCP server for a PC connected directly to it. This allows a very minimal system to configure the ARMweb (see the **Getting Started** section)

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 .

Program Statistics :
The compiler keeps track of the amount of code and variable space that has been used, and is represented by a percentage of the whole space (64KB code, and 4KB data).

Accepting Changes:
Any changes you make will not be permanent until the next power cycle (power off and on). If you do not want to make changes there is an **Undo All Changes** button, that will revert to the last saved configuration.

See also

- **UDP Services**
- **FTP Services**

CGI Services

Syntax

FUNCTION CGIIN AS STRING

Description

CGIIN functions like a serial channel to the webpage. When someone accesses the webpage 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 webpage, if the webpage 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 for TBD seconds or unitl it is read by a CGIIN.

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

 ...

    CGIinput = ""    ' erase the input line
loop
```

See also

- CGI example
- Web Basic
- FTP Services

Webpage Programming

Building a webpage on the ARMweb is much like any other web server. An HTML webpage is ftp'ed 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 webpage 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 webpage, here the webpage includes an image of a logo. In this example user actions on the webpage 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

<u>Upload to ARMweb using FTP</u>

No special tools to compile your page, just upload it to the ARMweb. Here the 2 files used for the webpage, the main HTML and the banner image.

```
C:\WINDOWS\system32\cmd.exe
C:\gnubasic\armweb>ftp 192.168.0.6
Connected to 192.168.0.6.
220 ARMweb Coridium Corp FTP Service (Version.0.0).
User (192.168.0.6:(none)):
331 Password required for .
Password:
230 user logged in.
ftp> put simple.htm
200 PORT command successful.
150 Opening BINARY mode data connection for simple.htm
226 Transfer complete.
ftp: 365 bytes sent in 0.00Seconds 365000.00Kbytes/sec.
ftp> put banner.gif
200 PORT command successful.
150 Opening BINARY mode data connection for banner1.gif
226 Transfer complete.
ftp: 72731 bytes sent in 13.41Seconds 5.43Kbytes/sec.
ftp> dir
200 PORT command successful.
150 Opening BINARY mode data connection for LIST
01-16-60 11:44AM 365 simple.htm
01-16-60 11:44AM 72731 banner1.gif
226 Transfer complete.
ftp: 103 bytes received in 0.00Seconds 103000.00Kbytes/sec.
ftp> quit
221 Goodbye.

C:\gnubasic\armweb>
```

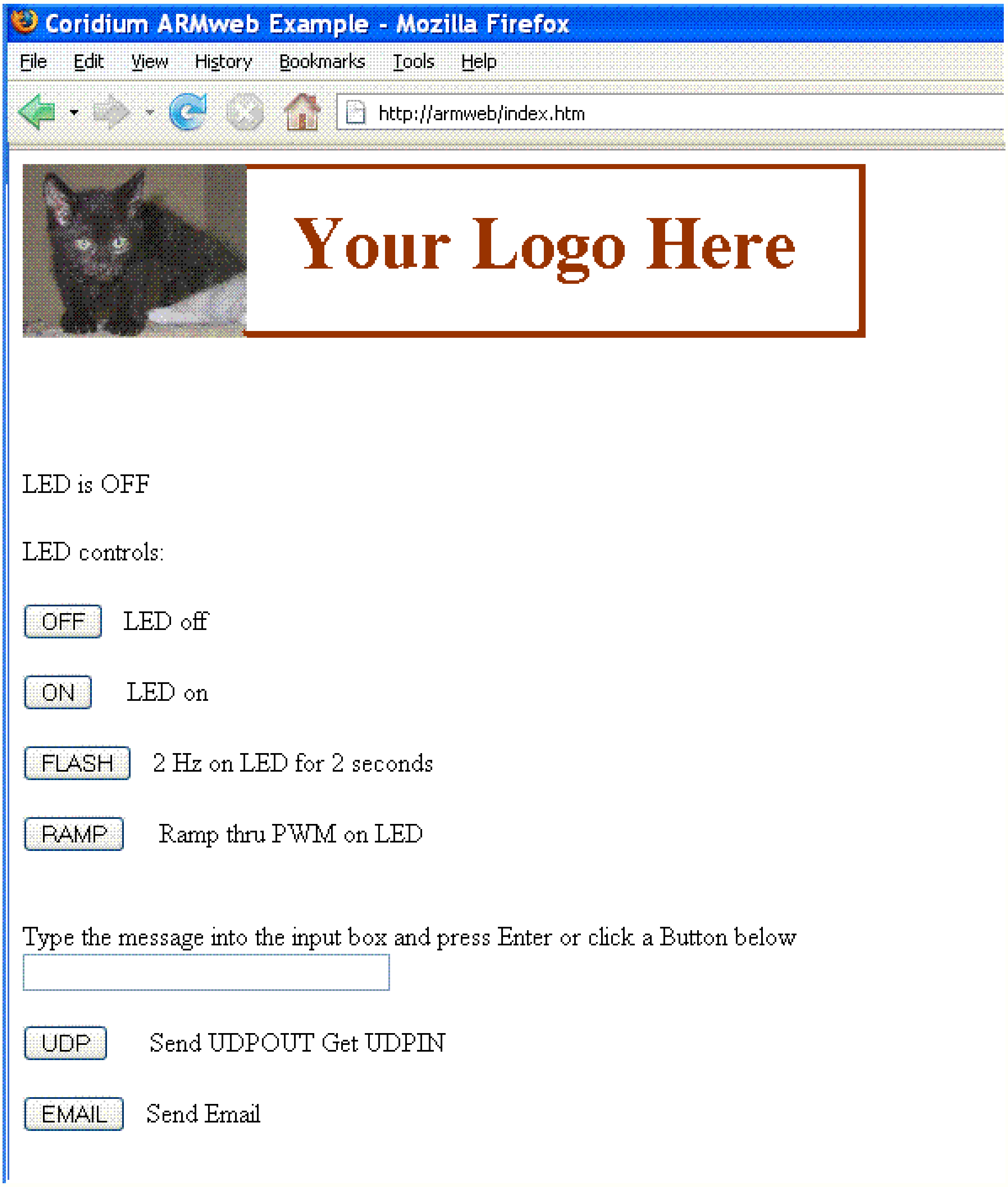
Interact with a BASIC program running on the ARMweb

The webpage 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

#include <FREQOUT.bas>
#include <PULSE.bas>

while 1                                  get input from client
    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 "3"
        FREQOUT(16,4000,2,0)
    case "4"
        for i=0 to 256
            PWM (16,255-i,20)
        next i
    case "5"
        Message = "Your ARMweb says "+Message
        MAIL (Message)                                   email 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

ARMweb 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 stopped, 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".

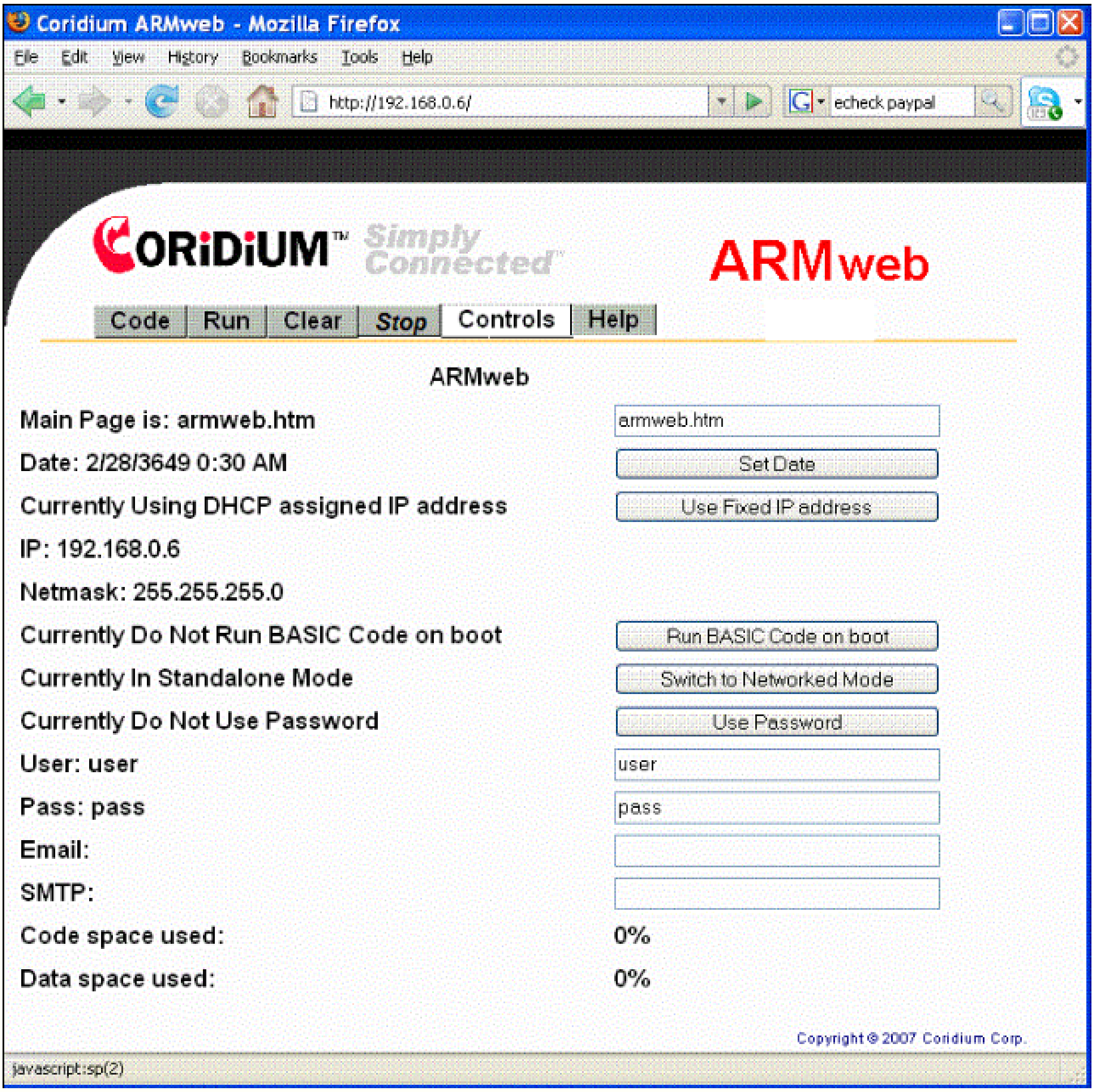
Change these as desired and reset the node to apply changes.
Also on the Controls page, a file from the system may be chosen as the Main Page.
This then becomes the default page when browsing to **http://ARMweb** or to **http://Nodes-IP-address** .

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 webpage to the ARMweb (192.168.0.6 was assigned by the DHCP, and the default username/pass were used (either enter user and pass, or just hit enter both times).

```
C:\WINDOWS\system32\cmd.exe

C:\gnubasic\armweb>ftp 192.168.0.6
Connected to 192.168.0.6.
220 ARMweb Coridium Corp FTP Service (Version.0.0).
User (192.168.0.6:(none)):
331 Password required for .
Password:
230 user logged in.
ftp> put simple.htm
200 PORT command successful.
150 Opening BINARY mode data connection for simple.htm
226 Transfer complete.
ftp: 365 bytes sent in 0.00Seconds 365000.00Kbytes/sec.
ftp> put banner.gif
200 PORT command successful.
150 Opening BINARY mode data connection for banner1.gif
226 Transfer complete.
ftp: 72731 bytes sent in 13.41Seconds 5.43Kbytes/sec.
ftp> dir
200 PORT command successful.
150 Opening BINARY mode data connection for LIST
01-16-60 11:44AM 365 simple.htm
01-16-60 11:44AM 72731 banner1.gif
226 Transfer complete.
ftp: 103 bytes received in 0.00Seconds 103000.00Kbytes/sec.
ftp> quit
221 Goodbye.

C:\gnubasic\armweb>
```

After the above ftp session a webpage has been setup on the ARMweb. It can be viewed at **http://192.168.0.6/simple.htm**.

To make that the main page served by the ARMweb go to the controls page.



At this point the change will not take affect until the ARMweb is reset, the easiest way is to cycle power on and off.

From then on, when you navigate to **http://192.168.0.6** the simple.html page will be displayed. If you want to go to the ARMweb BASIC, Controls or Values page, go to **http://192.168.0.6/armweb.htm**.

See also

- **Web Basic**
- **Web Services**

MAIL

Syntax

MAIL (*string*) ' does not use authenication,

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.

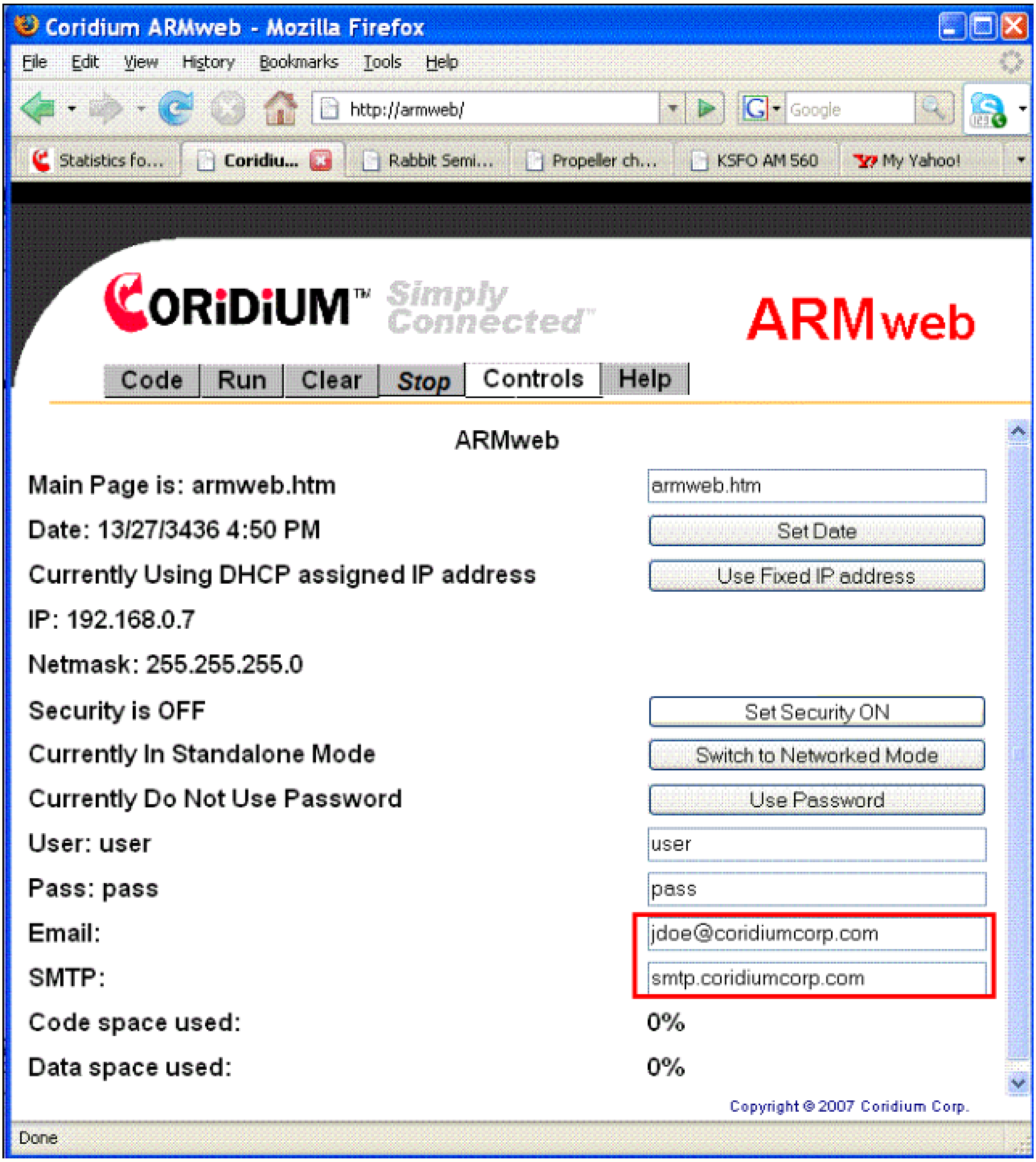
So you can send an email to yourself.

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 A$(10)
A$= "the current temperature is "+STR(temperature)
MAIL (A$)
...

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

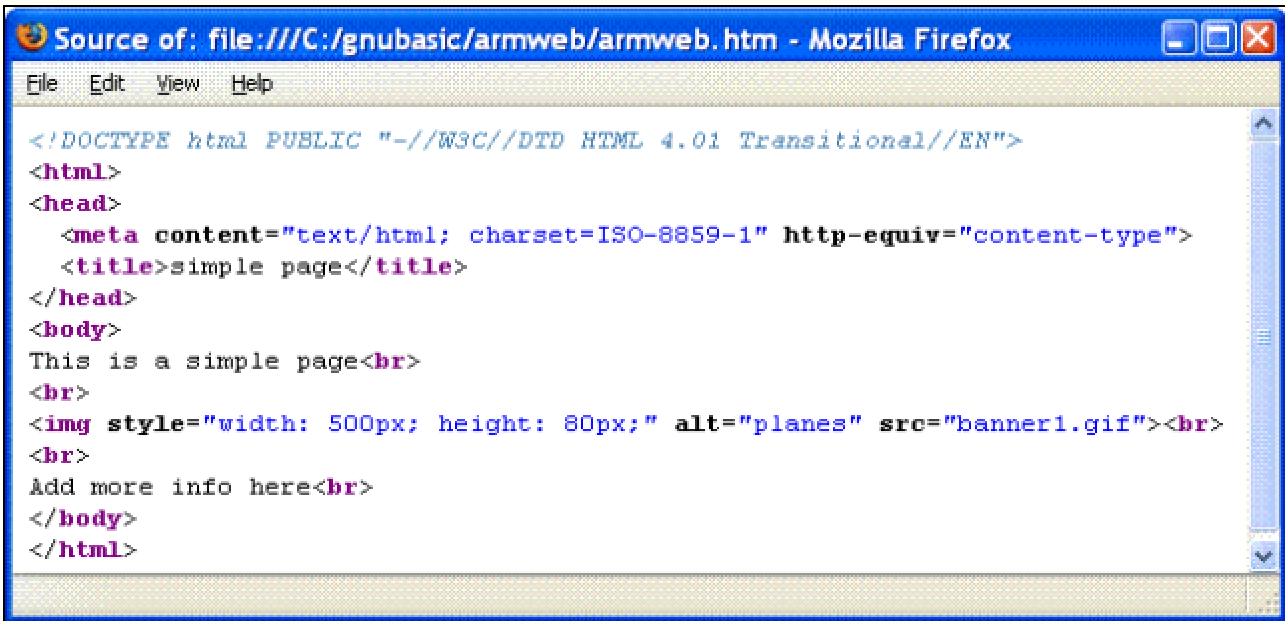
- **UDP Services**
- **FTP Services**

Web Services

ARMweb 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.

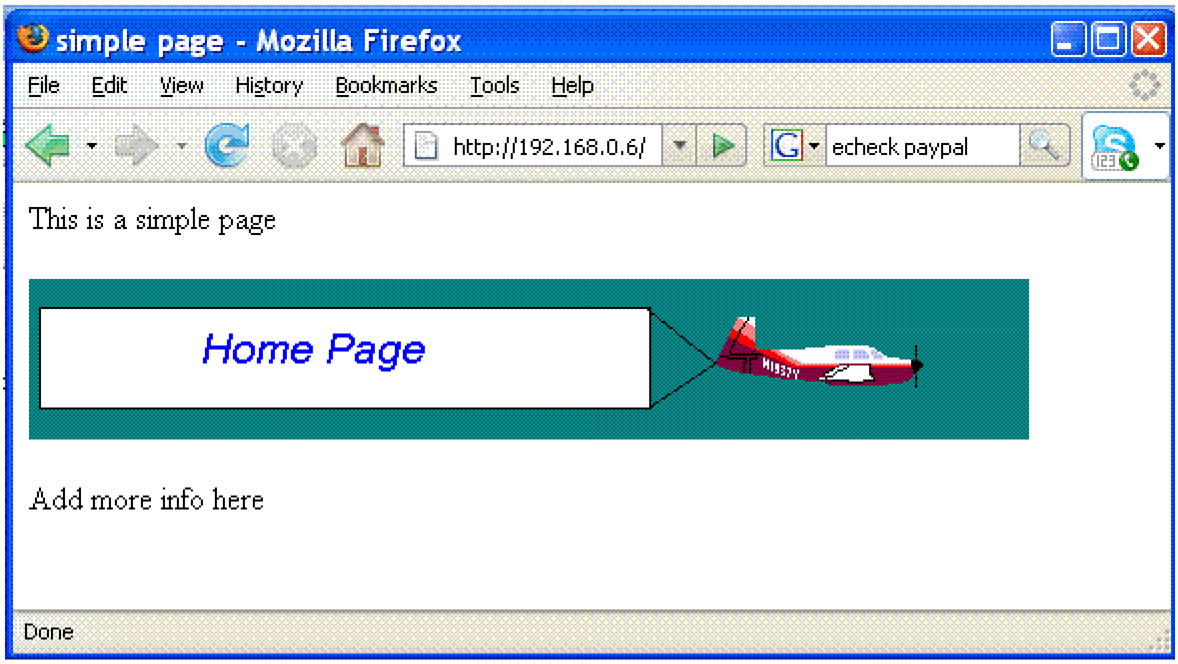
Building a web page

This is not the venue to teach webpage design, but a simple example will be presented here. Various ways can be used to build a webpage 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.




```
<!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>
This is a simple page<br>
<br>
<img style="width: 500px; height: 80px;" alt="planes" src="banner1.gif"><br>
<br>
Add more info here<br>
</body>
</html>
```

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-



See also

- Web Basic
- FTP Services

Web BASIC

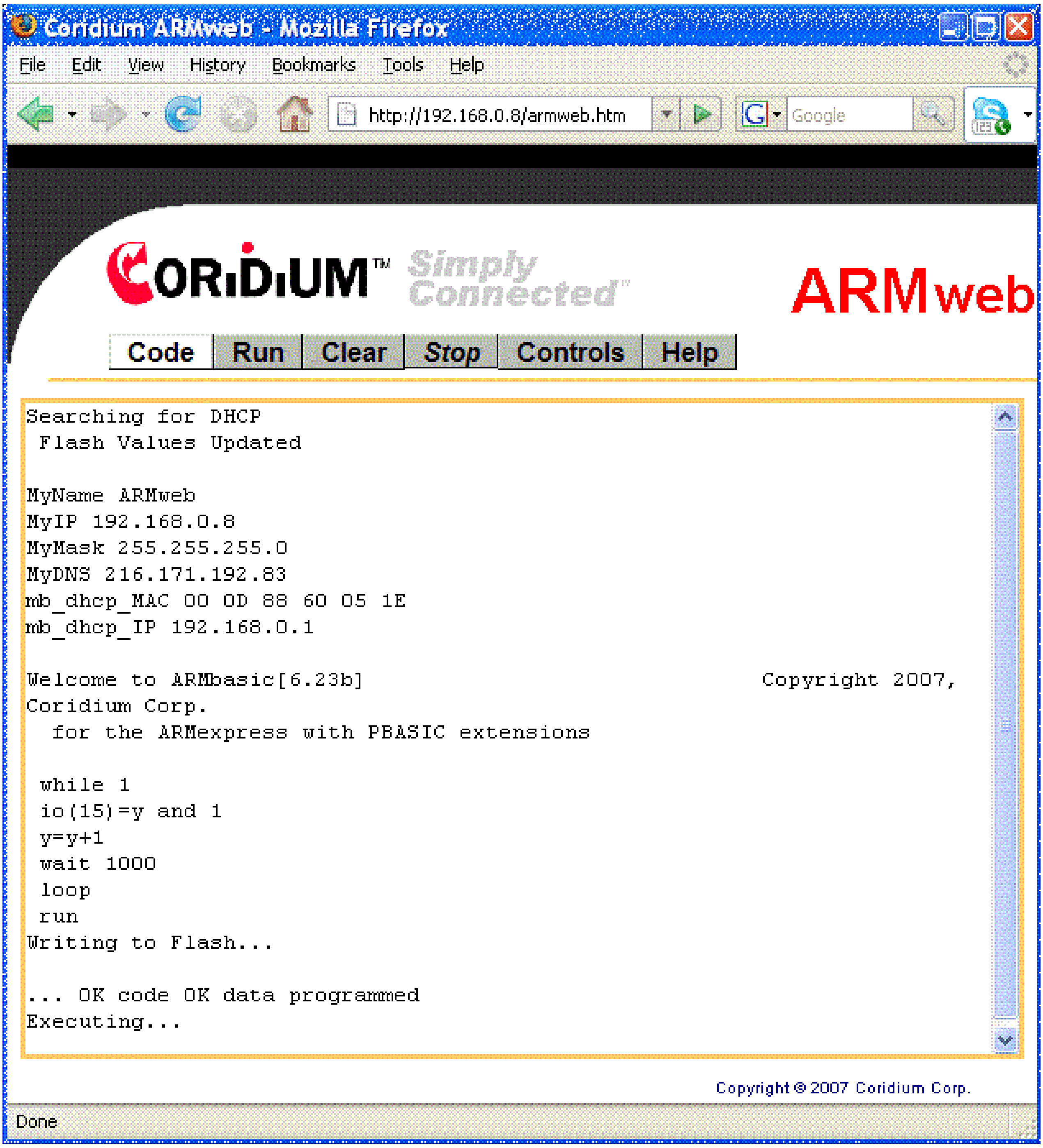
ARMweb allows for basic code to be embedded in the web pages much like PHP or JavaScript Variables may be accessed from the User program. The intention is not to place your BASIC code in this program, but to interact with your program from a webpage. For example if you put an endless loop in the BASIC embedded in the webpage, the webpage will hang.

Example: Add reading a User variable through the webpage.

Here is a modified version of the webpage loaded from **Web Services** .

```
<!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 y is ";y
?>
<br>
<br>
<img style="width: 500px; height: 80px;" alt="planes" src="banner1.gif"><br>
</body>
</html>
```

Now from Code page of ARMweb enter the following program (its can be accessed at armweb.htm)




```
WHILE 1
  IO(15) = Y AND 1   ' Flash the LED
  Y = Y + 1
  WAIT 1000
LOOP
RUN
```

The program is running and the value of Y is incremented every half second.
Browse to **http://ARMweb/simple.htm**

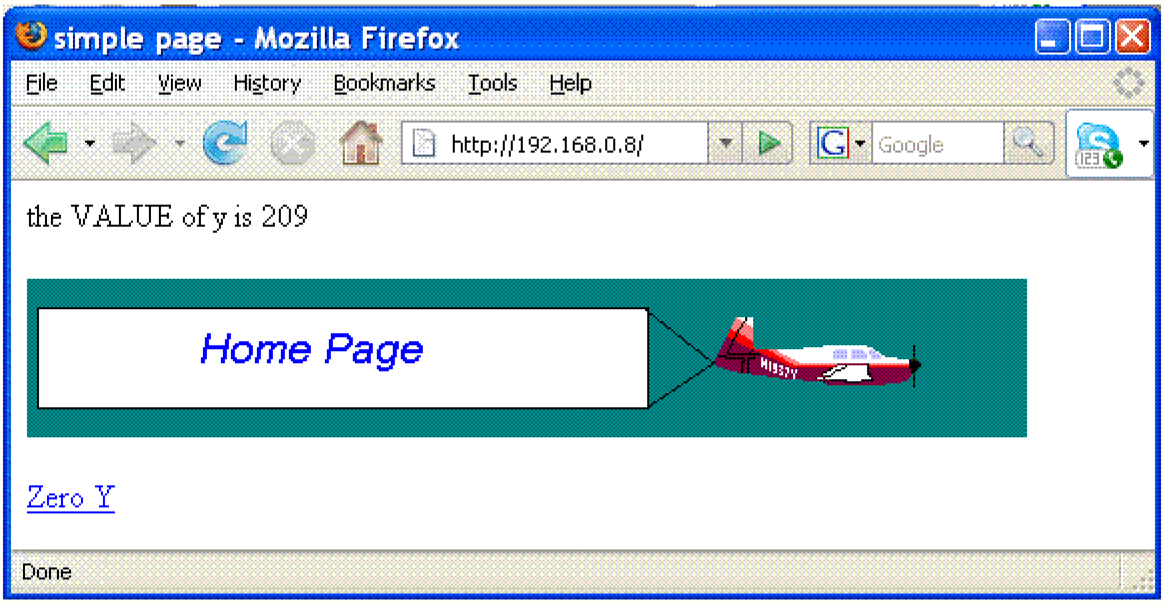
Refreshing the browser will show the updated values of Y.

Example: Executing a BASIC command from a webpage.

To the above example we will add a method to set the variable y to 0, by accessing another webpage that runs a BASIC program. This may also be accomplished with CGI, see the **CGI examples** .

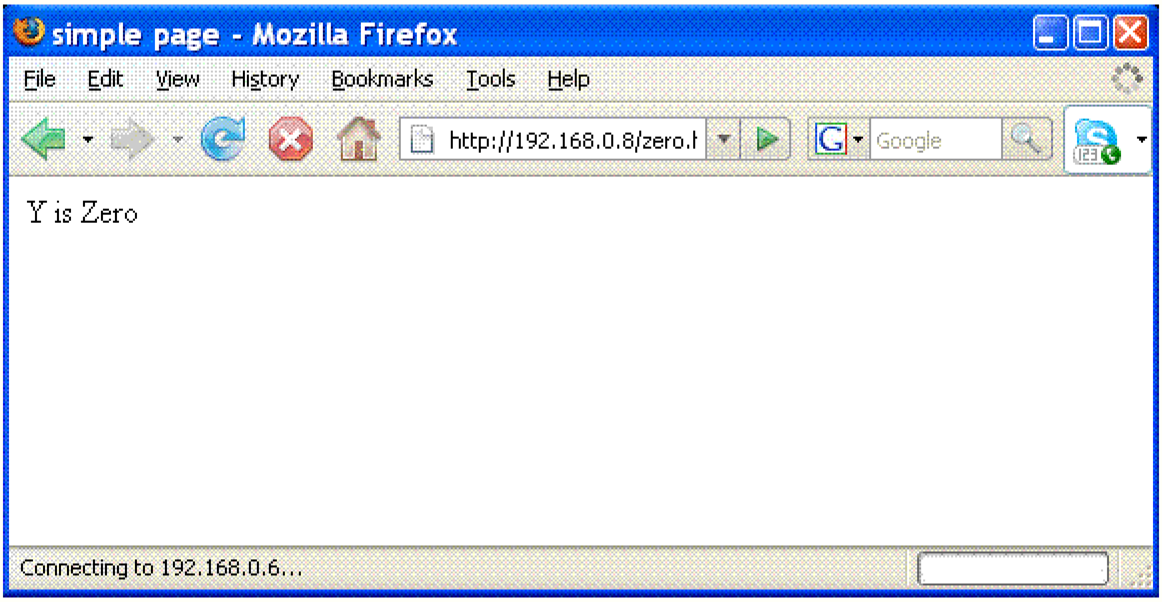
First add an anchor to another webpage that will be served by the ARMweb

```
<!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 y is ";y
?>
<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.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
  <meta content="text/html; charset=ISO-8859-1" http-equiv="content-type">
  <META HTTP-EQUIV=Refresh CONTENT="1; URL=http://192.168.0.8">
  <title>simple page</title>
</head>
<body>
<?BASIC
  y=0
?>
Y is Zero
</body>
</html>
```



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 to BASICtools). This slows down the web server, that can be improved by increasing the baud rate of UART0, by executing BAUD0(937500) in your main program. And to continue to view those debug messages reset the baud rate in 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 reponses 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 A$(100)

'sit and listen for any incoming UDP on port 50000

A$ = ""
WHILE A$(0) = 0
  A$ = UDPIN  (50000)
  x = LASTIP
LOOP
PRINT A$; " 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

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 configuered 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.

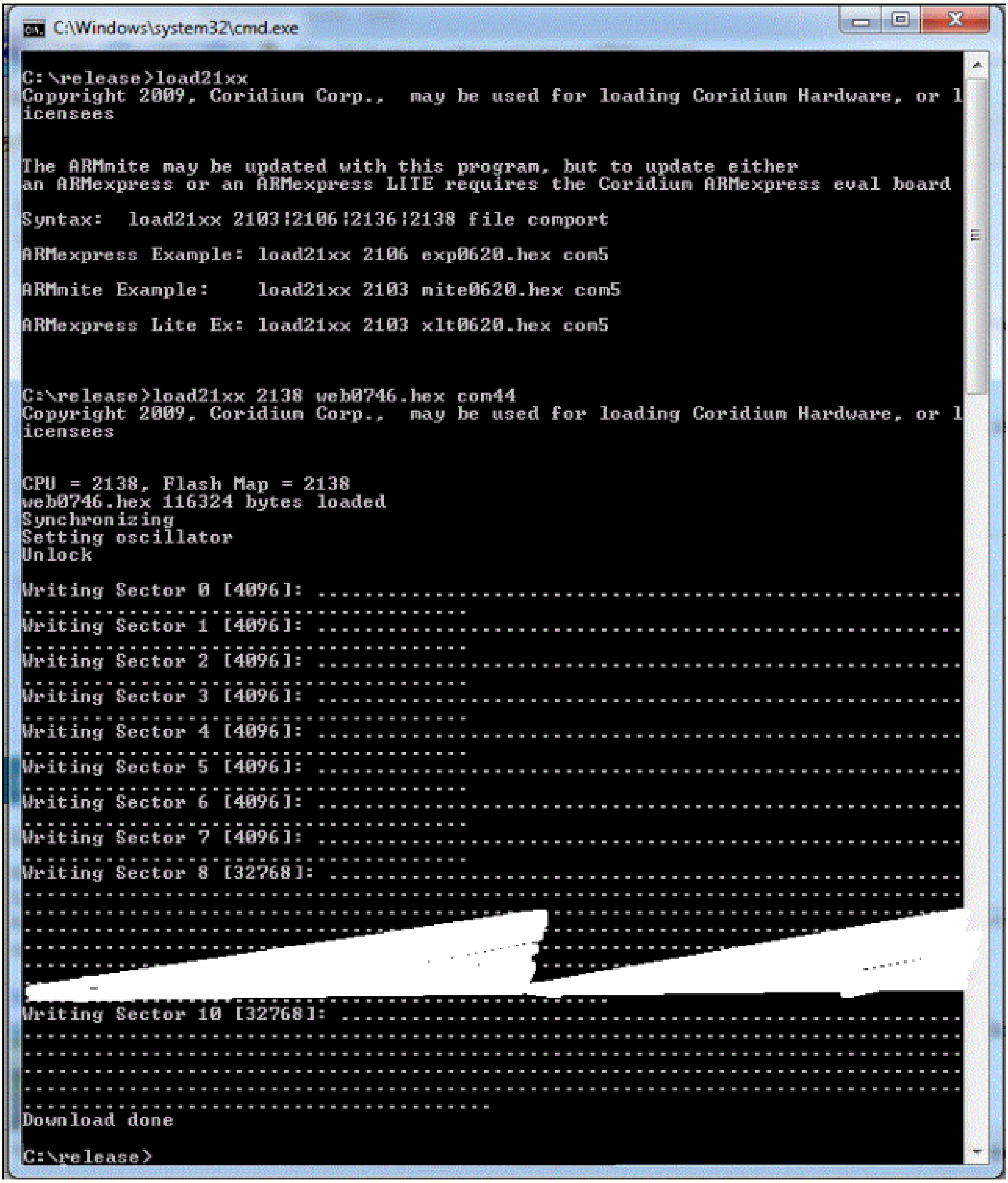
From a command line run load21xx.exe.

It will prompt you for the proper format of the command to update, the CPU is a 2138 -- see below for an example session.

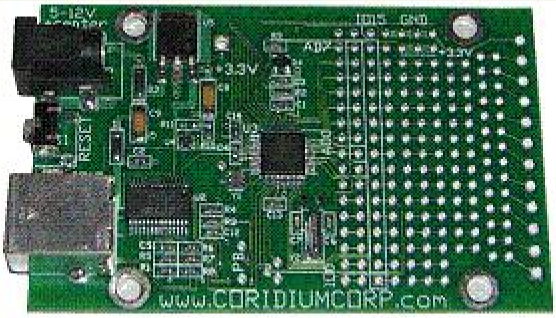
Restoring Factory Defaults

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 licensees
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 com44
Copyright 2009, Coridium Corp., may be used for loading Coridium Hardware, or licensees
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 4 [4096]: ....
Writing Sector 5 [4096]: ....
Writing Sector 6 [4096]: ....
Writing Sector 7 [4096]: ....
Writing Sector 8 [32768]: ....
Writing Sector 10 [32768]: ....
Download done
C:\release>

Tables



Tables

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 ARMexpress. 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.

Two codes XON and XOFF are used for flow control. When a large **ARMbasic** program file is sent to the ARMexpress module, the module may require a delay when writing code into Flash memory. During these writes of code to Flash, an XOFF character will be sent to the PC that indicates that the PC should pause sending data. After the block is written (about 0.4 second) an XON will be sent to resume communication.

However when using SERIN or SEROUT, there is no special interpretation of characters, so all codes 0 to 255 may be sent without any change.

The ARMmite 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	Logical
- (Negation)	= <> < > <= >=	AND
*, / (Multiplication and division)		OR
MOD (Modulus Operator)		XOR
+, - (Addition and subtraction)		NOT
<<, >> (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
ARRAY	fixed length	signed integer	-2147483648	+2147483647
STRING	variable/max length 256 bytes	zero terminated	0	+255
STRING	used as byte array no max length		0	+255

Support

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	Installing Firmware on other vendors boards
Install Software Unlock Firmware installer	Install Software Install Demo Firmware Installing purchased full feature Firmware

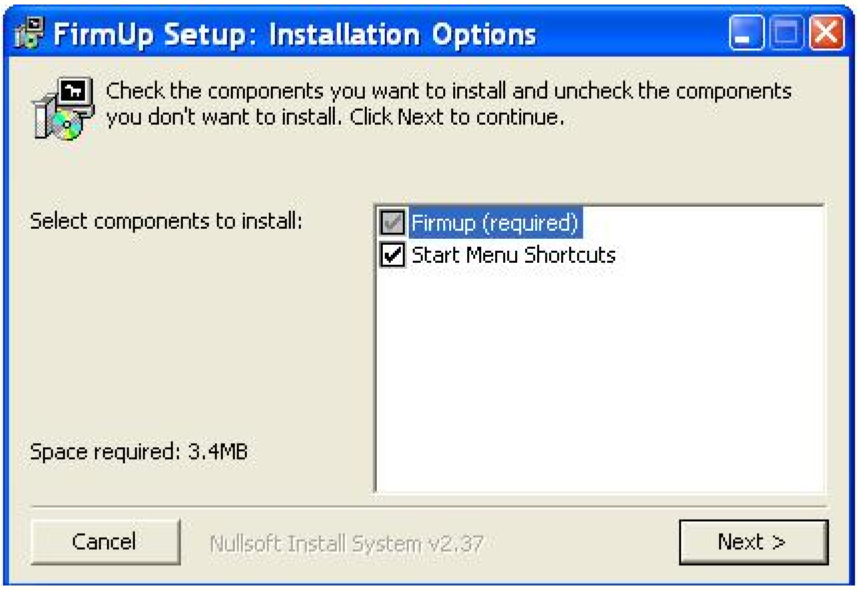
Step 1: Install Software

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.

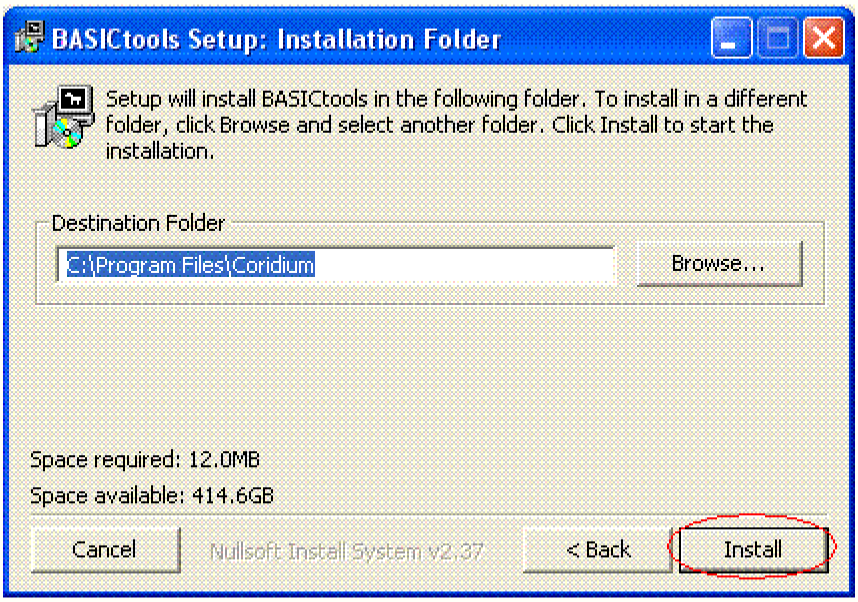
Purchase page from Coridium Web store

This installer is meant for 32 bit WIndows either NT, XP or XPx64 and Vista.

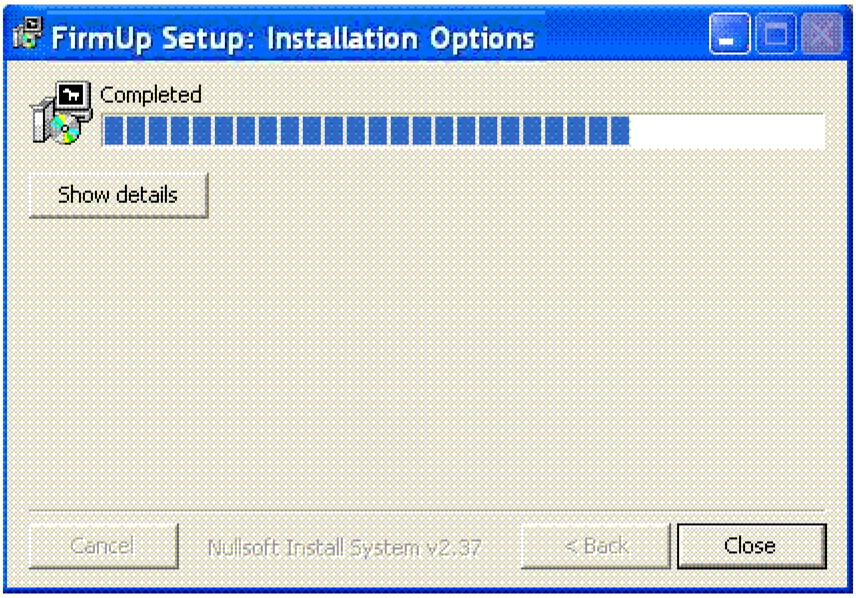
The software is downloaded from the web, and run as an installer SETUP program.



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** .

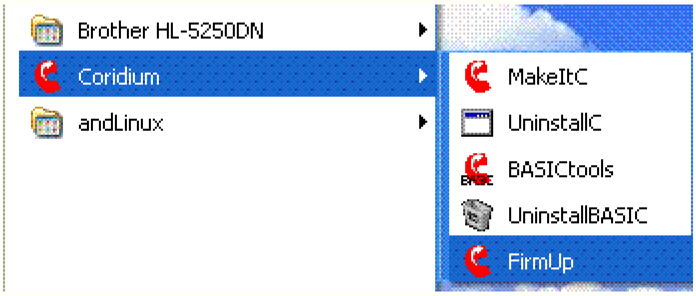
And its as easy as that.

On to Step 2

Step 2: Writing the Firmware.

The **ARMbasic** compiler is freely downloaded, but the utility to install BASIC support libraries is locked. To unlock that you need to receive a special version of this program from Coridium after purchase. There is a demo version available for the stand-alone **ARMbasic** compiler.

The software installed in the previous step would either be FirmUp for firmware upgrades, or NewFirm for the standalone **ARMbasic** compiler.



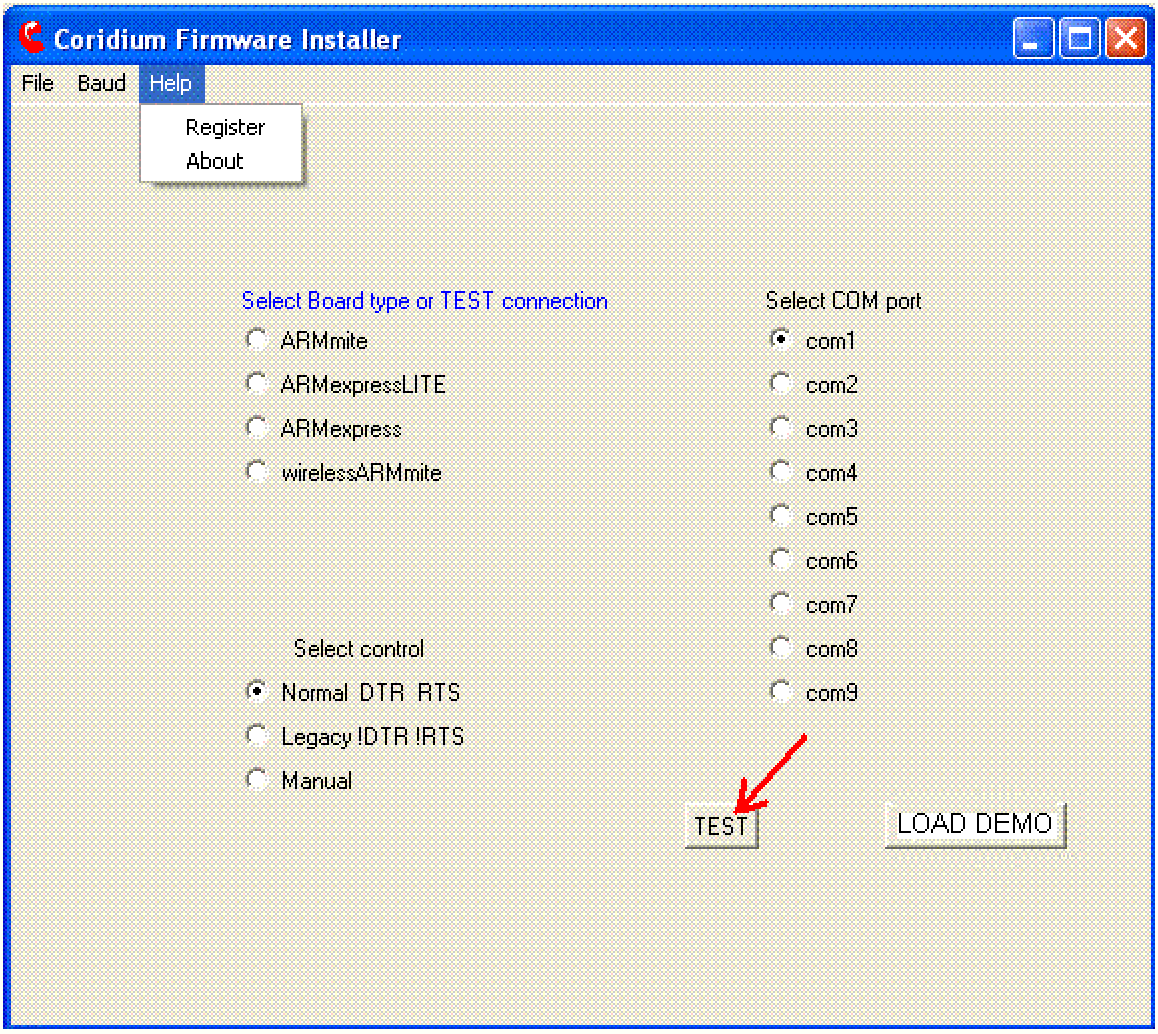
To run FirmUp/NewFirm you must have network access, as information is downloaded from the Coridium website.

Step 2: Establish communication

Before you can run **ARMbasic** 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 NewFirm/FirmUp program. The installation of Step 1 has installed a Start Menu shortcut.

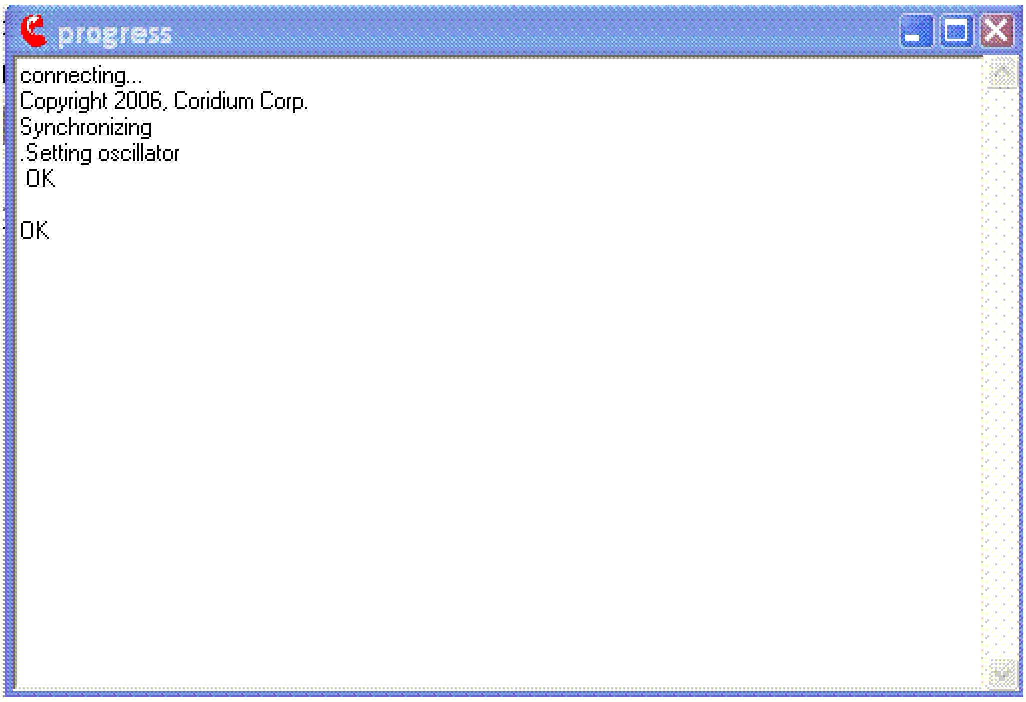
FirmUp allows you to choose the serial port on the PC from a list of known ports. Ports in that list that are capitalized were determined to be using FTDI USB serial devices. You must also set the control type, which for most will be Normal mode. Legacy mode is for those users who have inverted the control signals, for instance to run Hyperterm or Linux, details **here** . For wireless boards, Manual mode should be chosen.

LOAD DEMO code will erase any other programs on the board, do NOT do this unless you know the BASIC firmware was already erased.



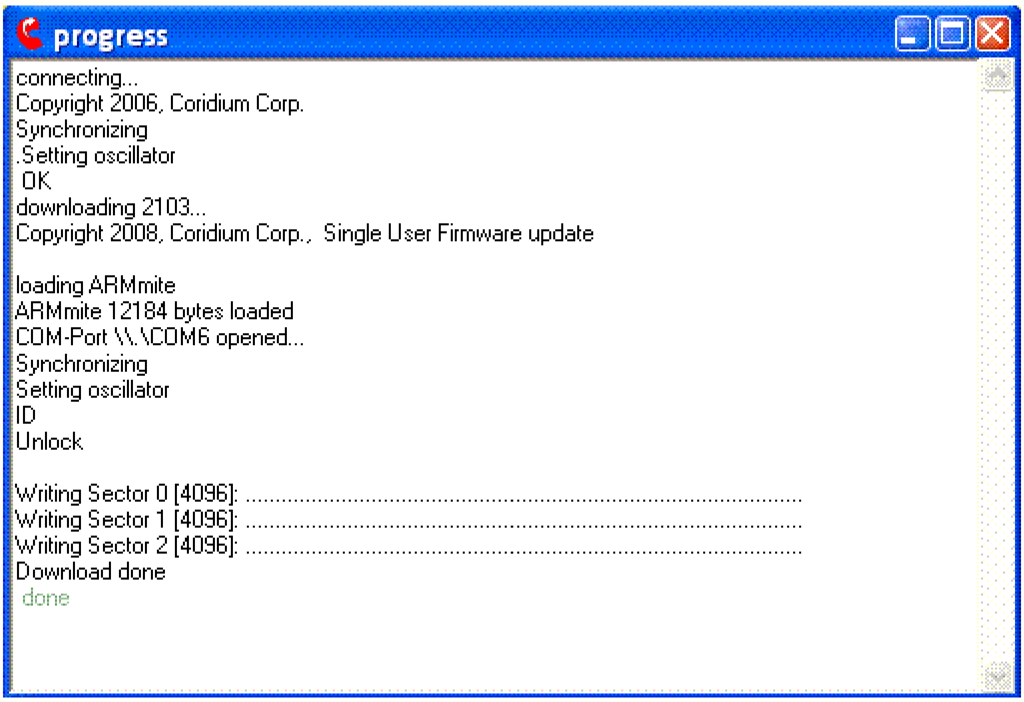
So select your comport and choose the control method. To test that push the soft button TEST on the FirmUp program. It will prompt you for any action required (like pushing buttons on the target board), and then test the communication with the PC. If this does not pass, then you cannot go on to the next step.

Loading this DEMO code will erase any other programs on the board, do NOT do this unless you know the BASIC firmware was already erased.

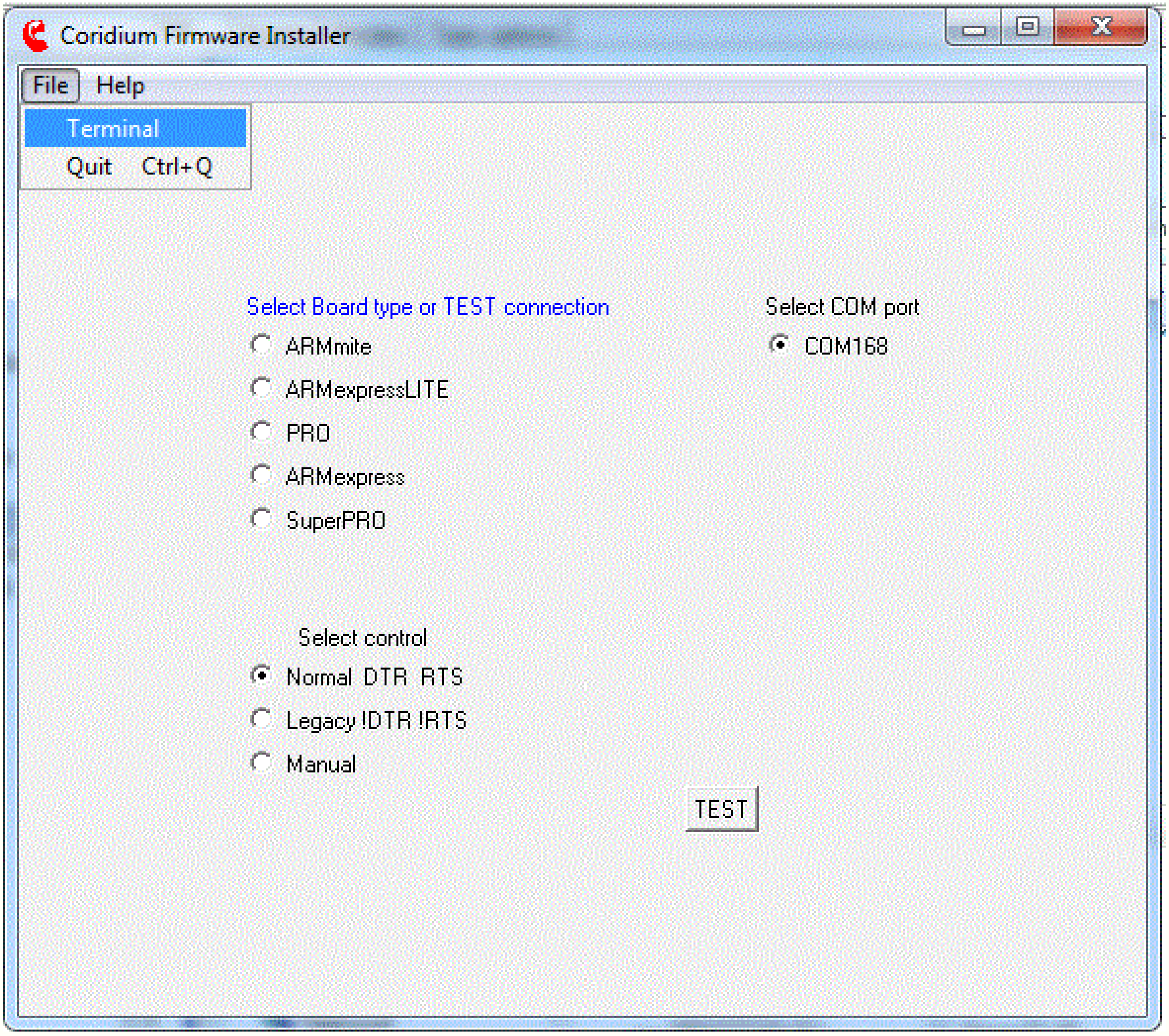


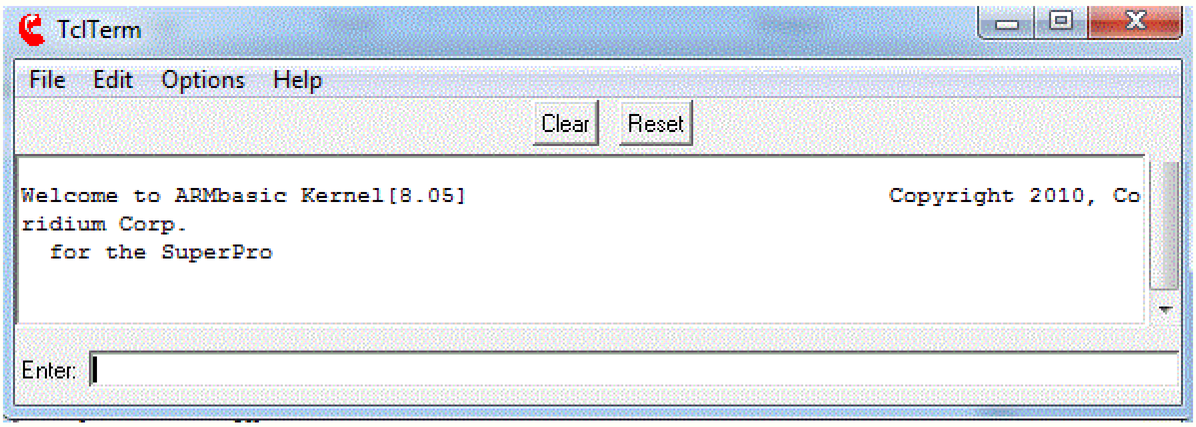
Step 3: Install Firmware on ARM

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.



Firmware has been succssfully 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, Coridium Corp.
for the SuperPro
Enter:

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 FREQOUT runtime library function fails when it is called with a value of 1234" is much better than "It crashes".

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.

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.

A PBASIC translator (in development) will use GNU sed v3.02.80 and MinGW cpp.

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.
```

Index