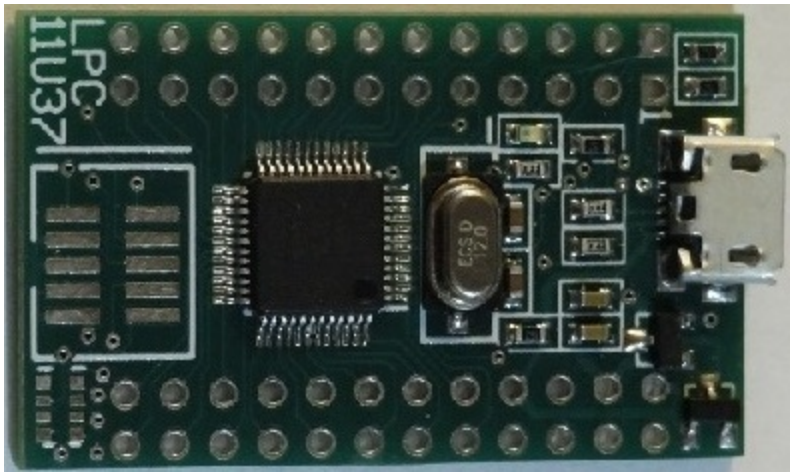
ARMbasic Documentation

Getting Started

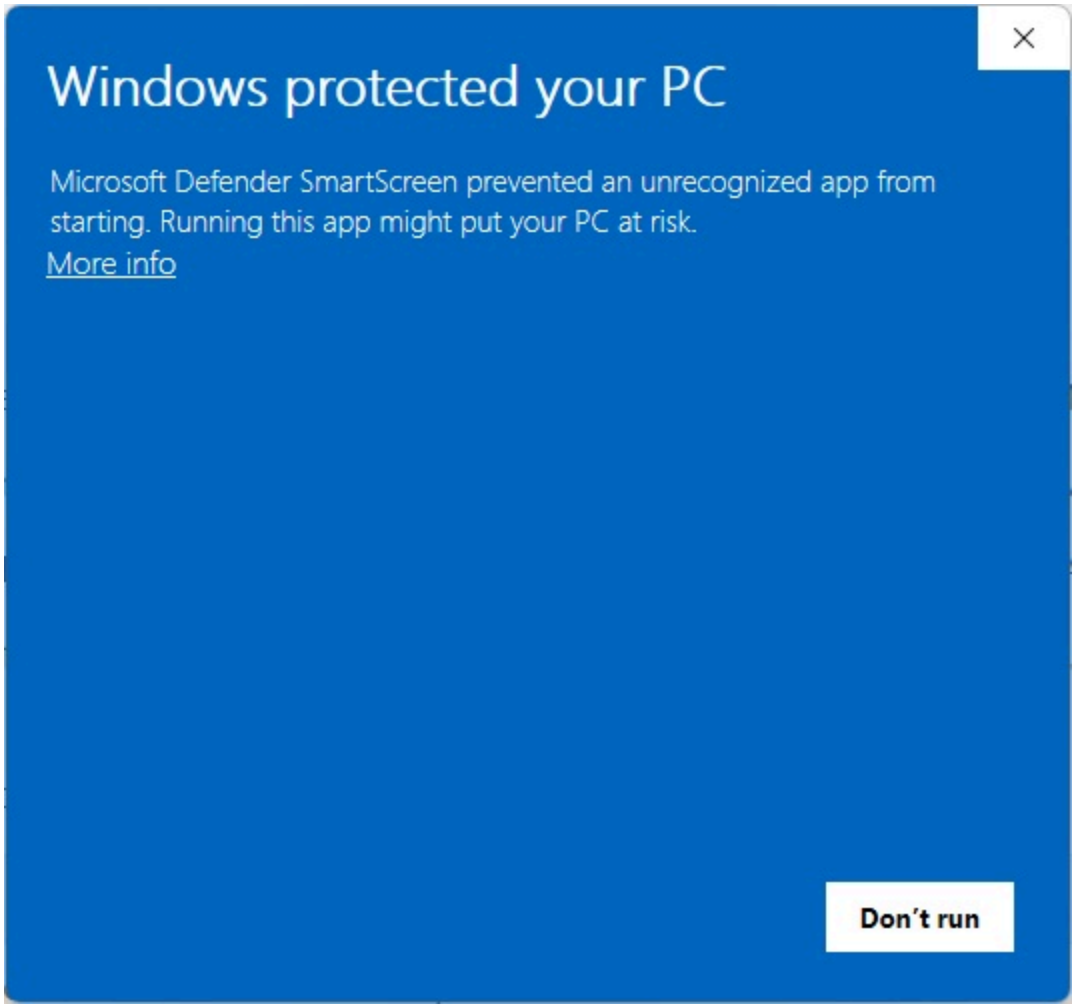


Install Software

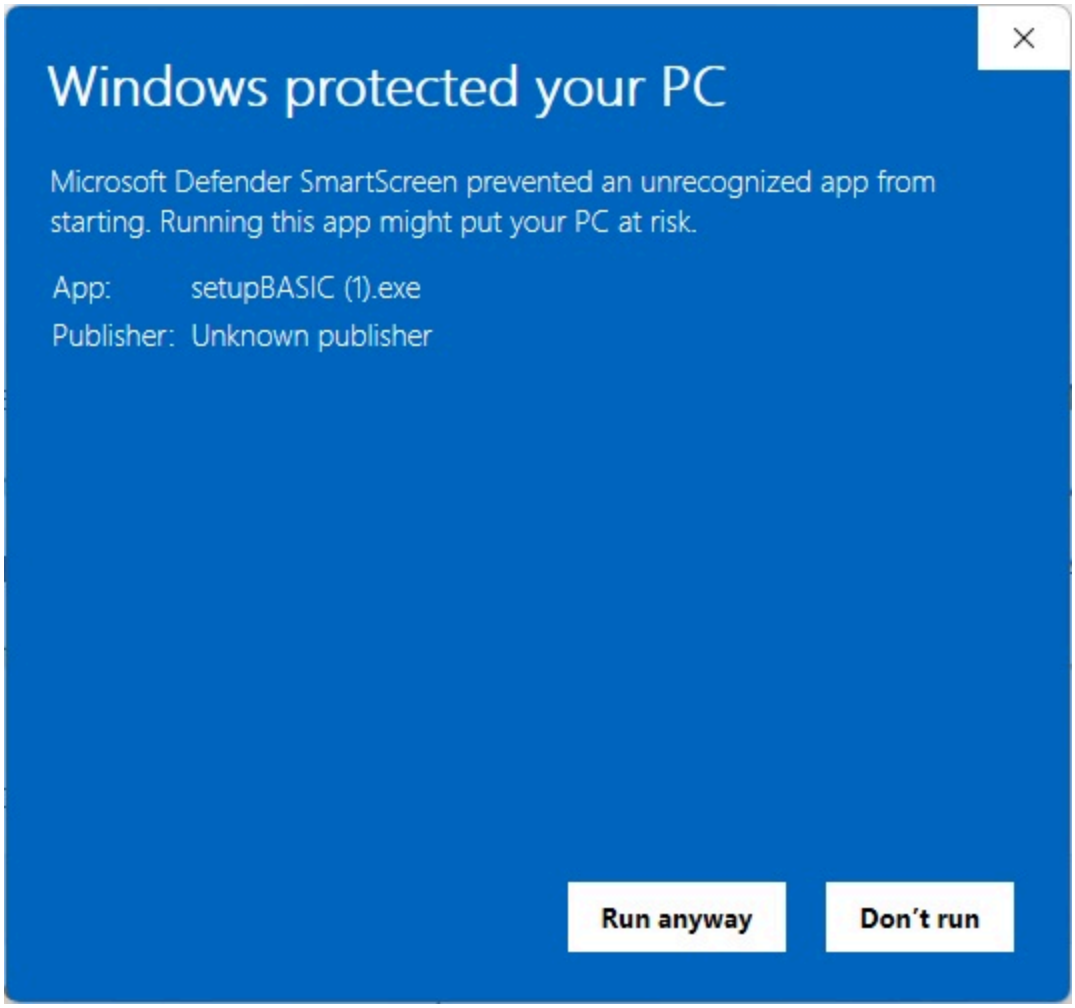
Download **setupBASIC.exe** from the coridium.us website and run it.

Windows SmartScreen

Windows may show a "Windows protected your PC" dialog. Click **More info** to expand the options.



Then click **Run anyway** to start the installer.



Installer prompts

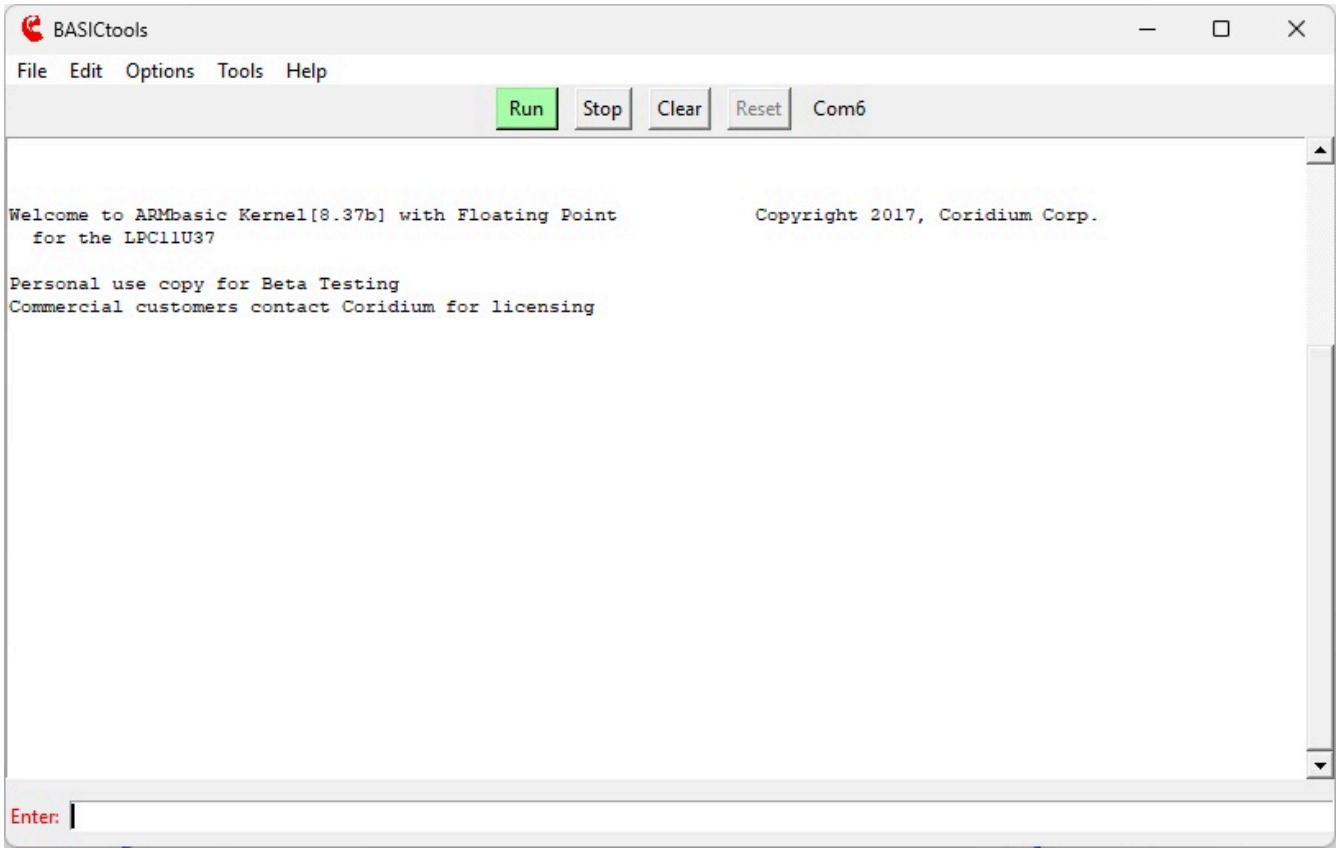
Accept the prompts to install each of the following components:

- **ARMbasic compiler**
- **BASICtools IDE**
- **Notepad++ editor**
- **Examples and libraries**

Connect USB

Writing your first program

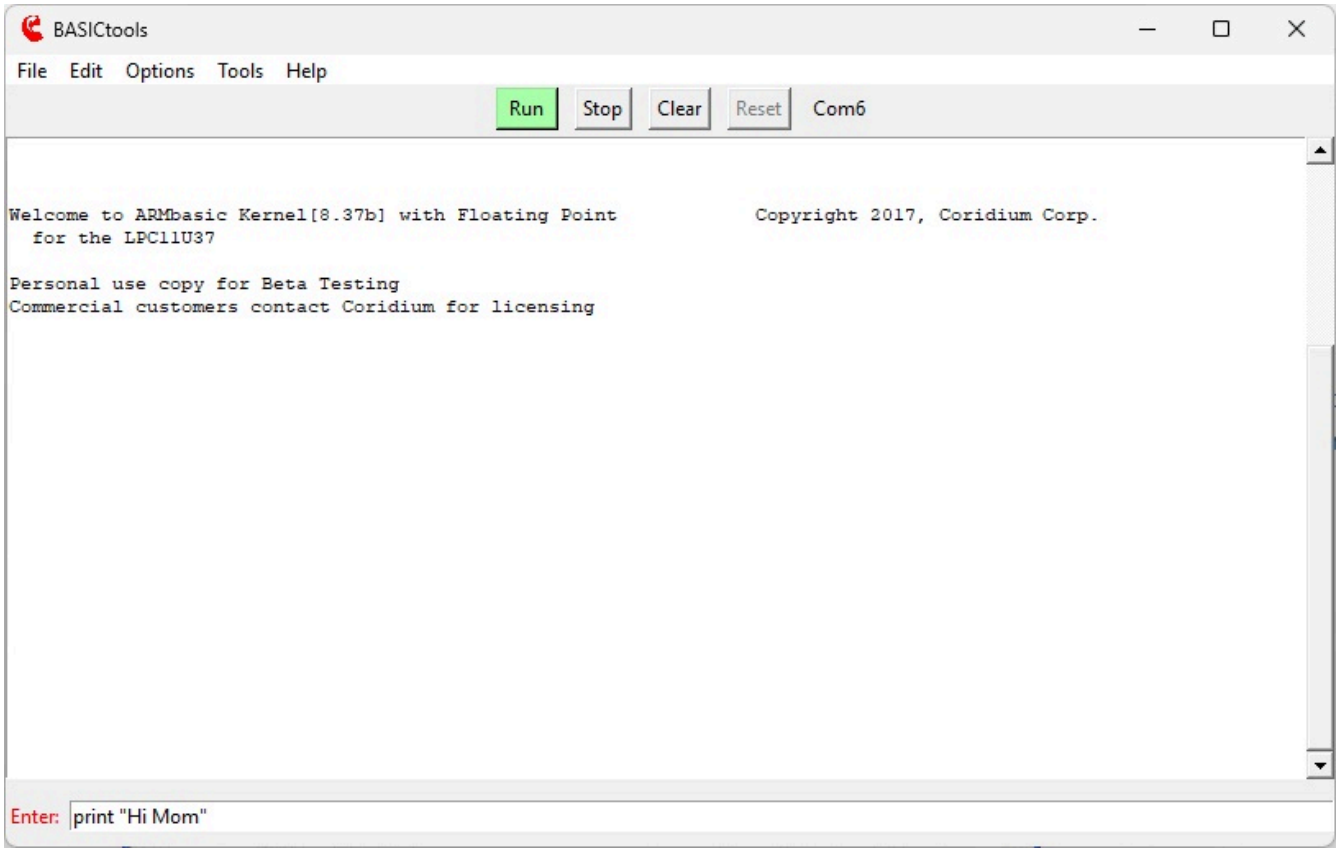
Reset ARMstamp



Once you launch BASICtools you should be able to communicate with the ARMstamp. To check this, Reset the ARMstamp with the option button.

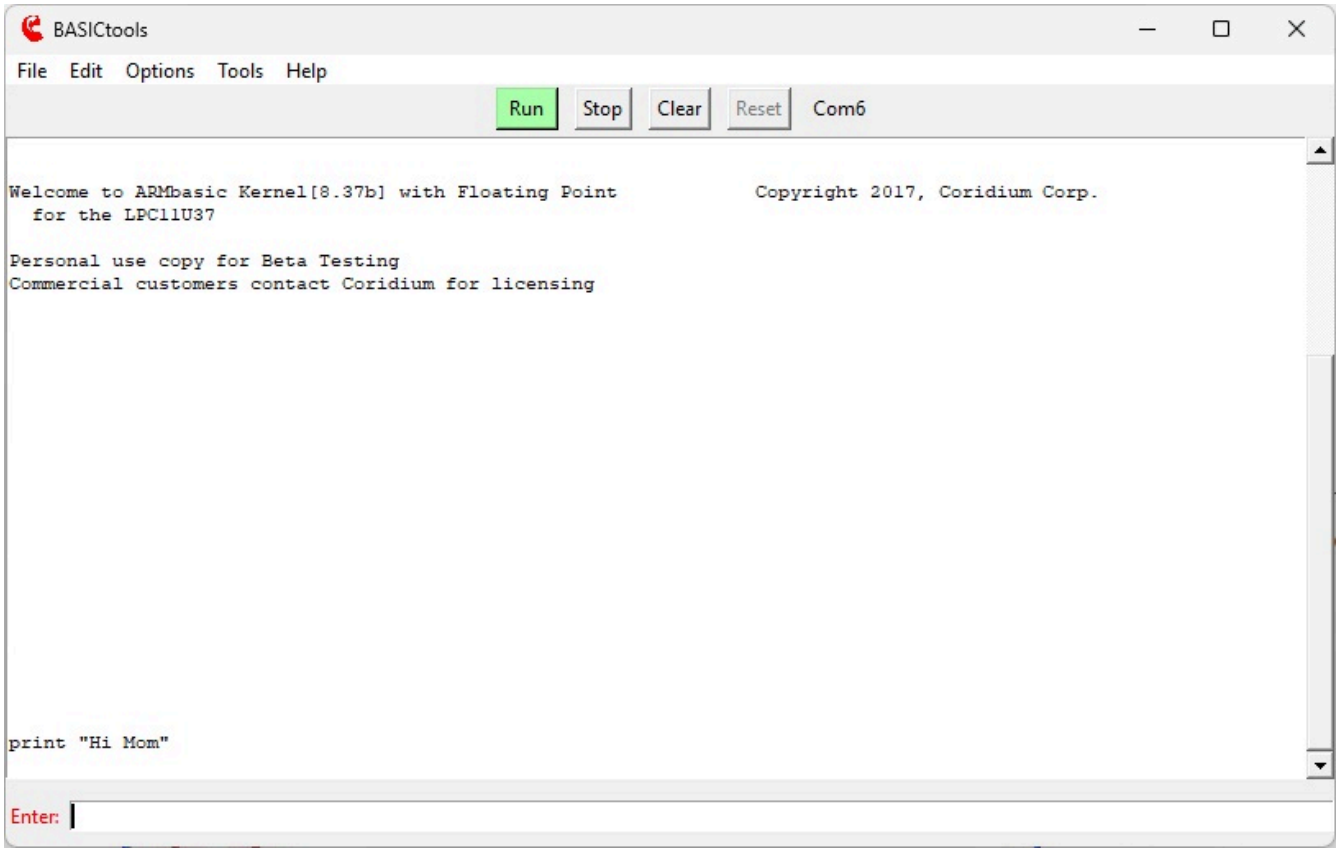
If all is connected correctly, the ARMstamp will display the welcome message, and the ARMstamp is ready to receive **ARMbasic** commands, which can be entered in the input window.

The traditional "Hi Mom" program



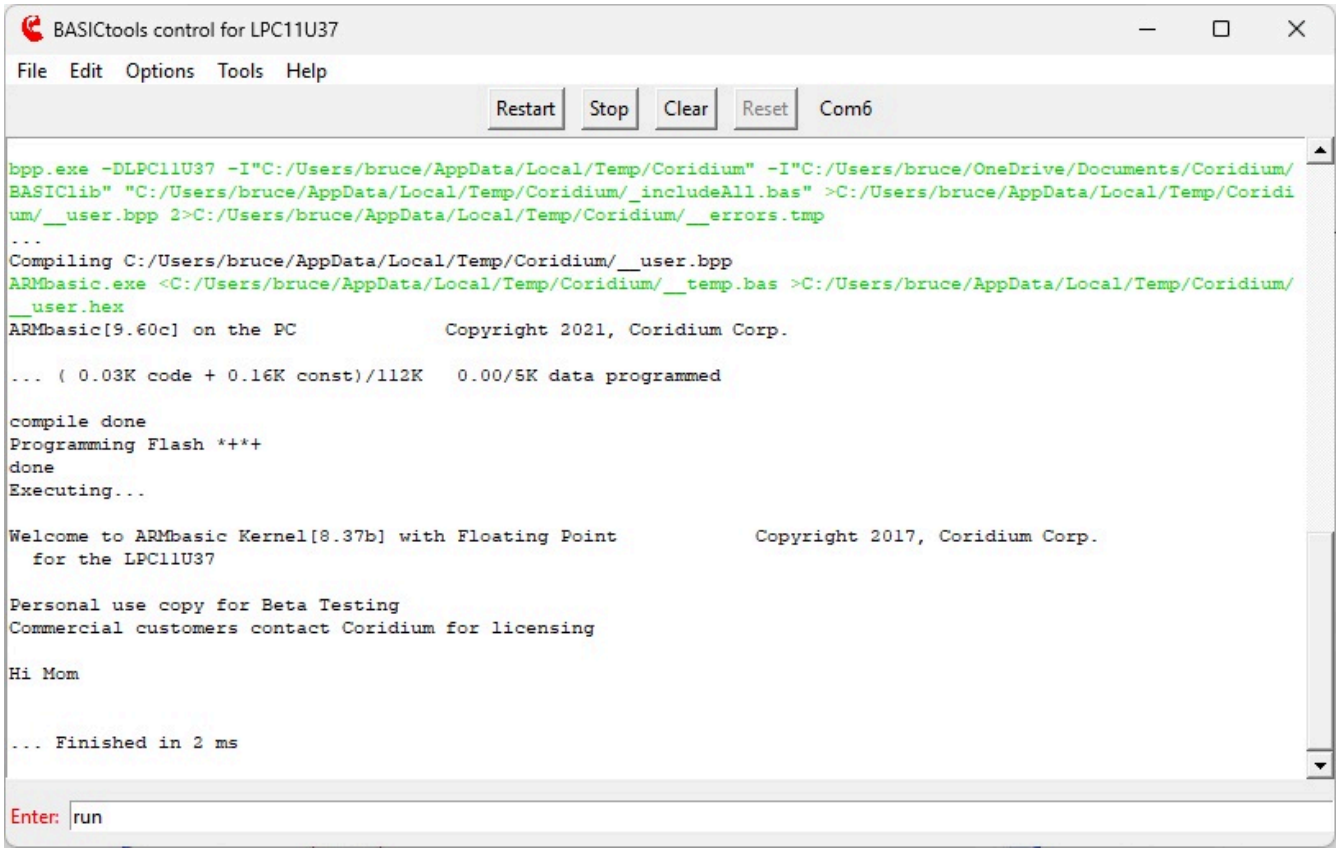
So type something like the traditional `PRINT "Hi Mom"`. When you hit the ENTER key it will be sent to the ARMstamp and be echoed back in the console window.

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, which normally takes less than 1400 msec. Next the program will be executed, as evidenced by the output of "Hi Mom" to the console. ARMstamp also reports back how long the program executed, in this case 19 msec.

On to Step 3

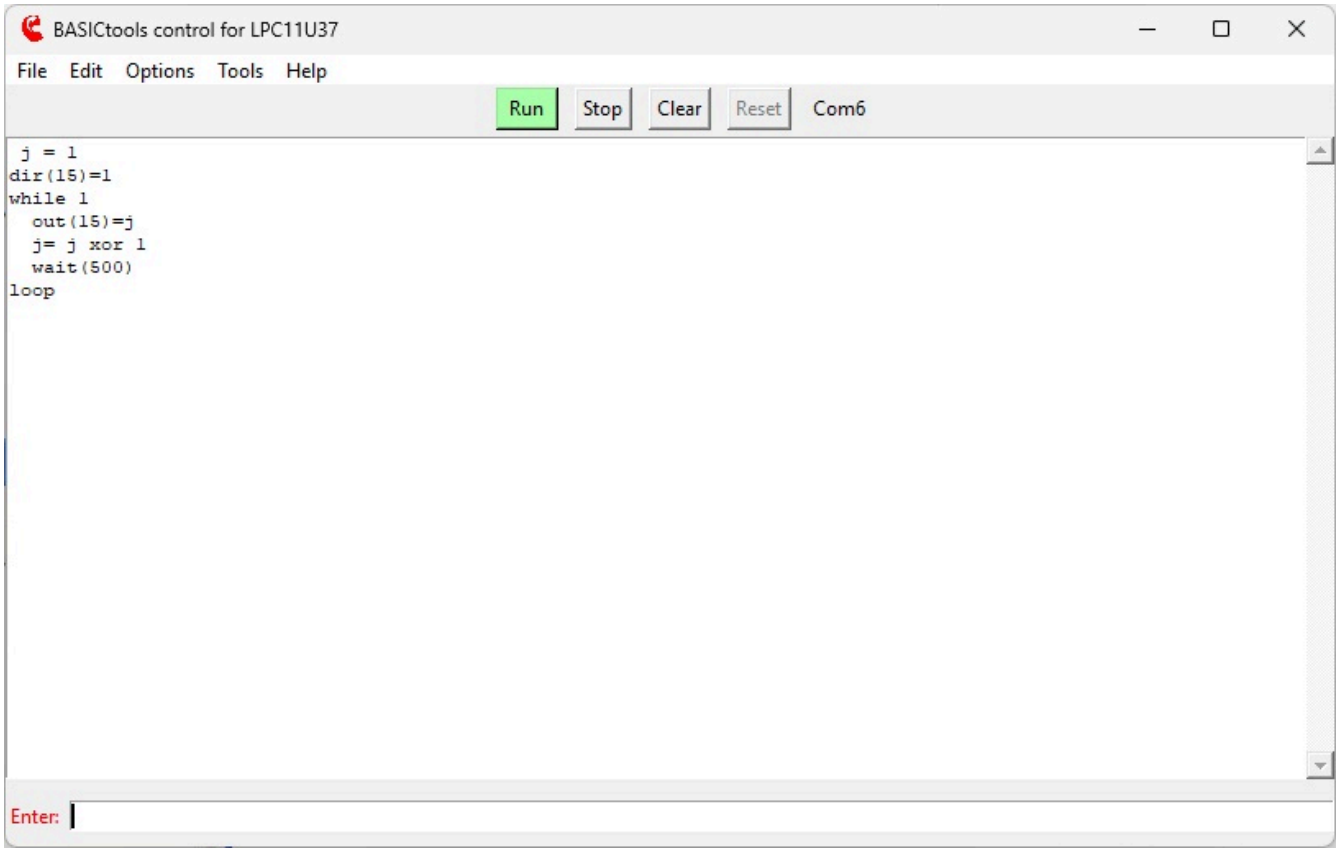
Programming the IO

A program that uses IO

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

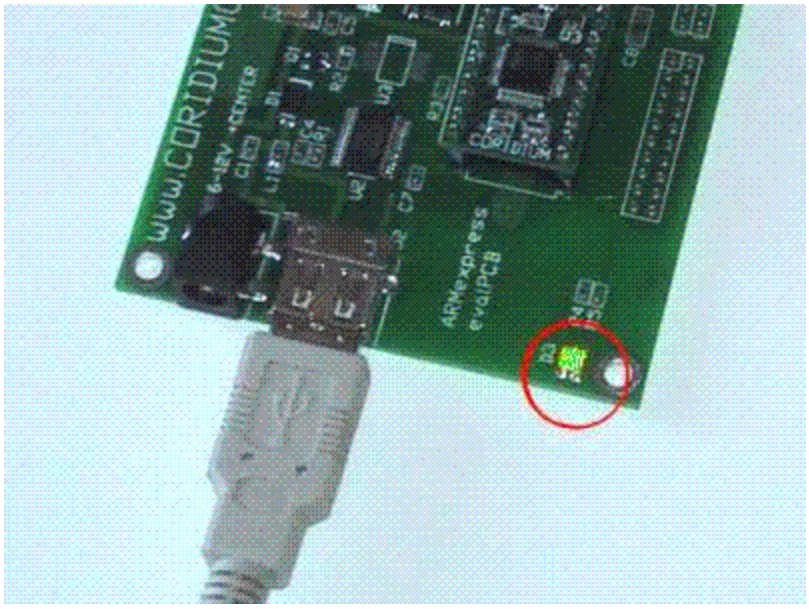
```
DIR(15)= 1
' enable pin 15 as an output WHILE X<30 OUT(15) = X AND 1 ' drive pin 15 high when x is odd, low when x is even X=X+1 WAIT(500) LOOP
```

Now RUN the program

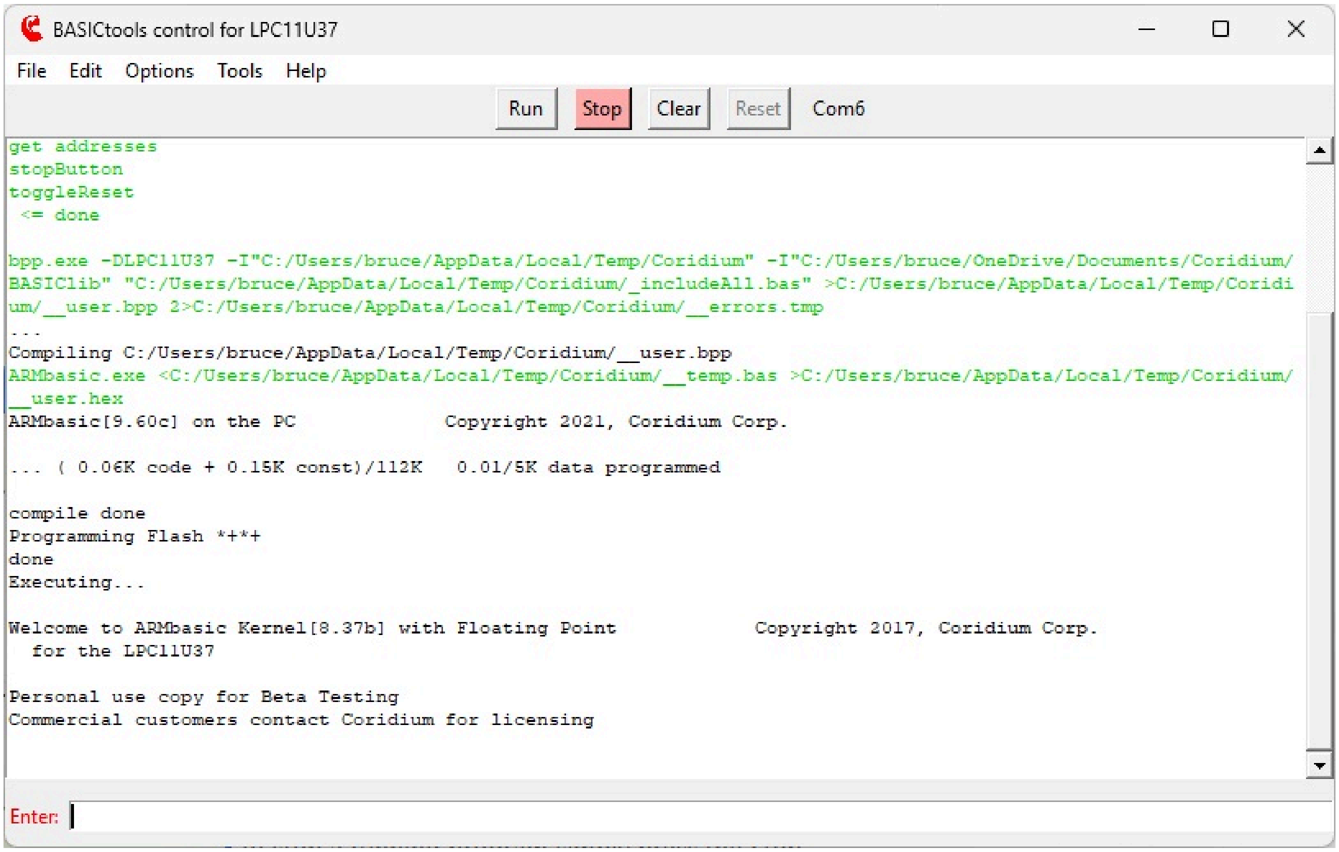


The LED on the PCB should Flash 15 times.

And see the results



Stop the program



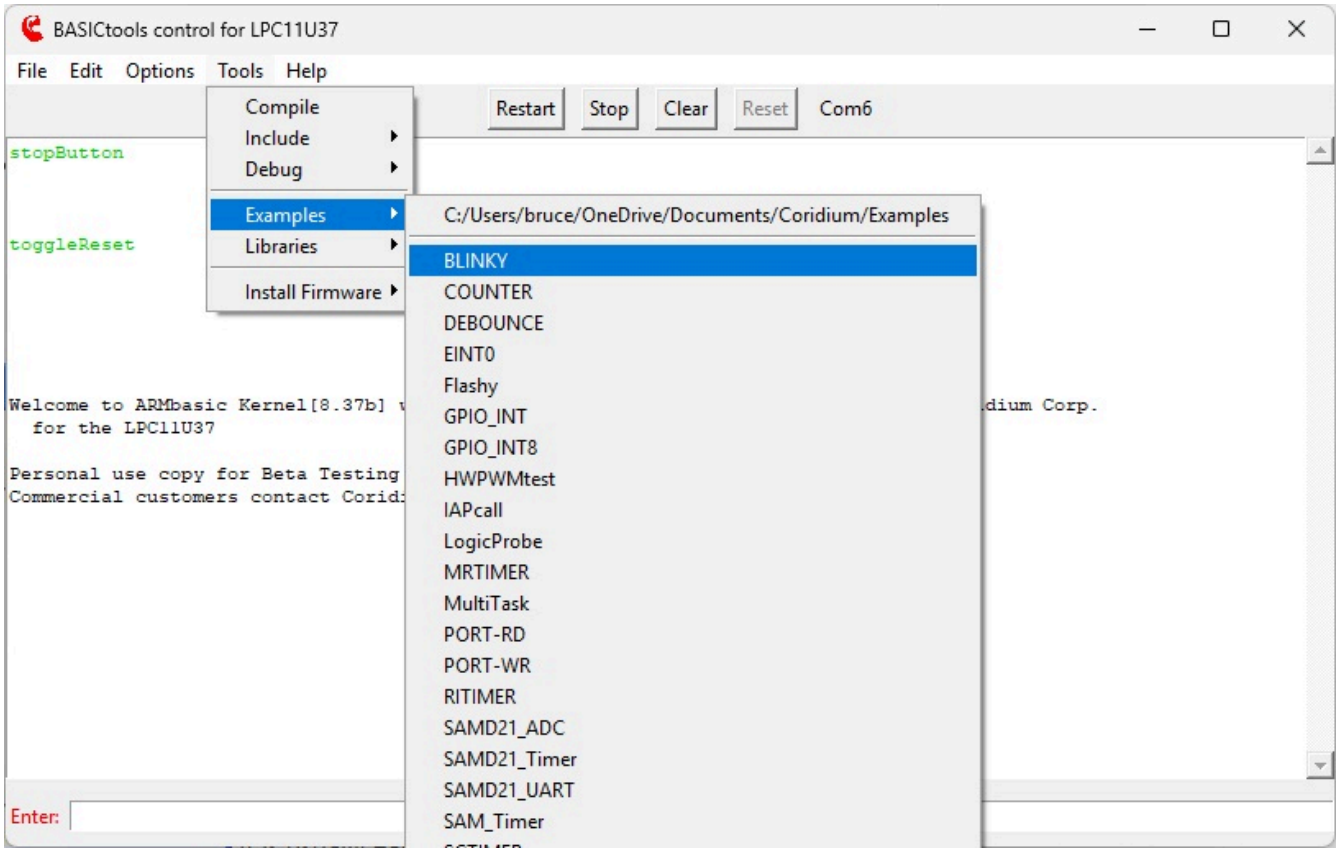
To stop a running program simply press the Stop button.

On to Step 4

More complex programs

Choose a File

It is usually easier to download a program with Send or ReSend File. You can choose a file from the file system or one of our examples.



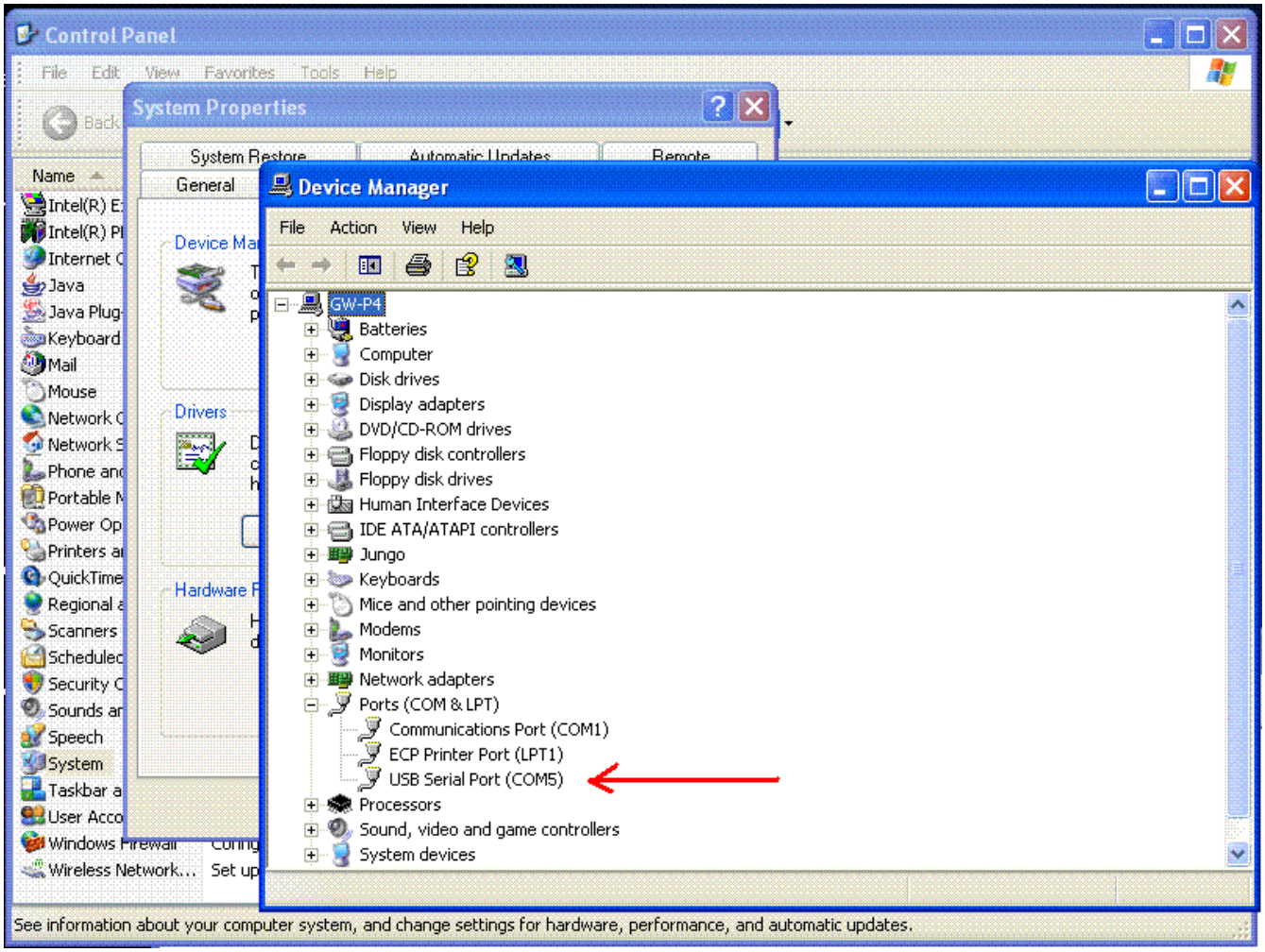
On to Step 5

Trouble Shooting

Step 5: Trouble Shooting

Determining which COM port should be used

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



COM port naming

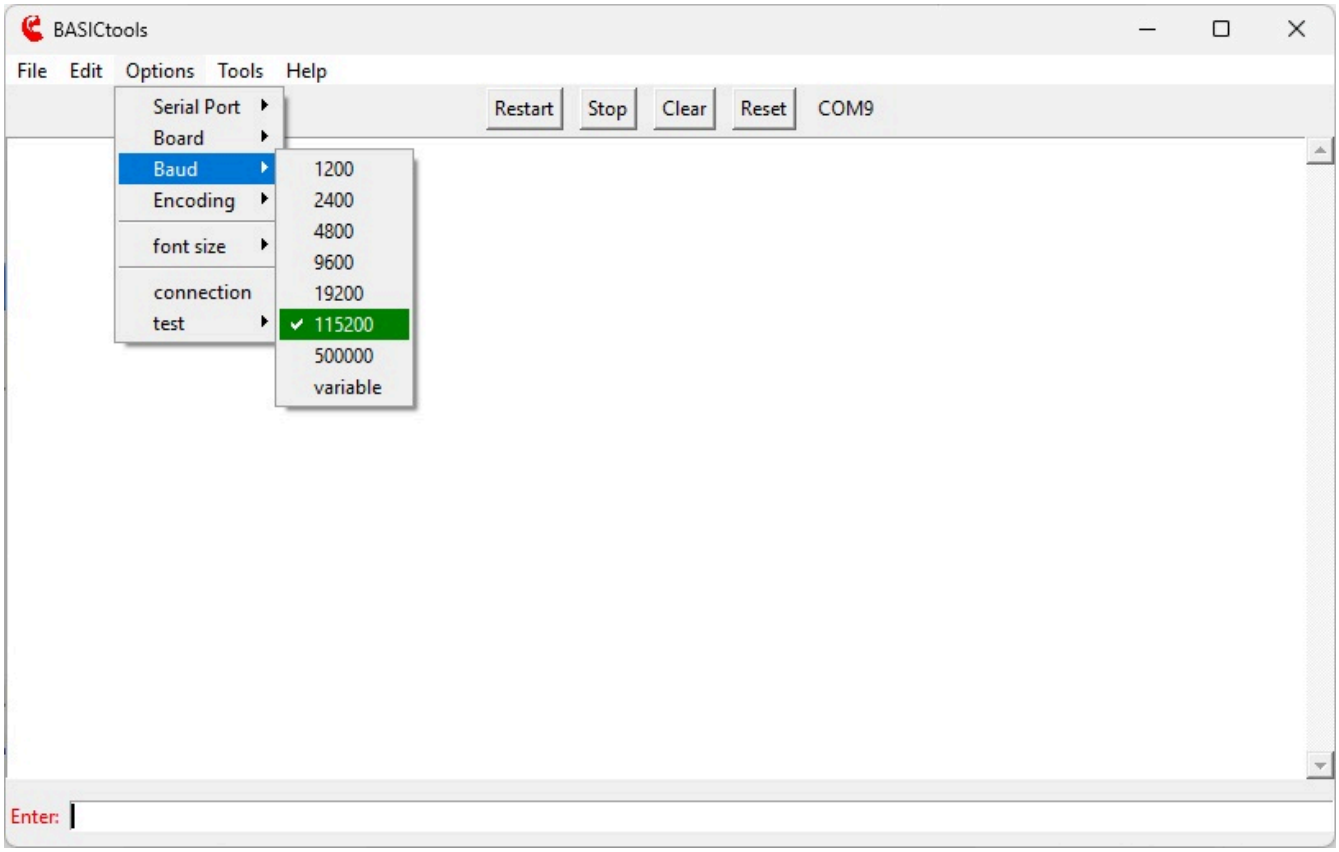
Ports named in all capital letters (for example **COM4**) are FTDI ports.

Ports with just the first letter capitalized (for example **Com5**) are recognized ports on ARM USB equipped boards.

FTDI ports show 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

Check Baud Rate



Or you might not have the correct baud rate selected.

Check your cables, check the LED

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 ARMstamp PCB, which is pin compatable with other DIP24 modules such as the Parallax BASICstamp.

ARMbasic is simple to use, and runs totally on the ARM CPU and can be programmed from any device with 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. It's quick and easy to debug programs. Why learn a new development system — you can either enter programs directly from the console or use any text editor you are already familiar with.

BASIC Compatibility

- **ARMbasic** from Coridium is not a "new" BASIC language. You don't need 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; a MAIN function is not required. Syntax follows much of that of Microsoft Visual BASIC.

PBASIC IO functions

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 3 pins
- PWM on any pin with 256 levels
- PULSIN will measure a pulse
- SHIFTIN, SHIFTOUT can be used for SPI or MicroWire devices
- OWIN and OWOUT support one-wire devices
- SERIN, SEROUT can be used for low duty cycle asynchronous serial ports on any pin up to 9600 baud

Support for 32-bit variables and Strings

- Integer: (32-bit math)
- Floating point (32-bit)
- String support
- Automatic type conversions

Arrays

- Static arrays supported, up to 32 KB in size.

Direct Hardware Access

- Uses the same syntax as C-pointers.

Debugging support

- The ease and speed of an interpreter.
- Quick access to variables.

Optimized code generation

- While **ARMbasic** is not an optimizing compiler, it does many kinds of general optimizations to generate fast code on ARM CPUs. It runs more than 10 million BASIC instructions per second.

Requirements

Installing

Windows

Follow the installer instructions, which are also outlined in the Install Software section. The compiler resides on the ARMstamp, so the installer is just setting up documentation and the BASICtools IDE. Connection to a PC is done over USB.

Linux

Linux is supported. Search the coridium.us website for instructions; details and discussion are on the forum.

Mac

Mac is supported. Search the coridium.us website for instructions; details and discussion are on the forum.

Raspberry Pi

Raspberry Pi is supported. Search the coridium.us website for instructions; details and discussion are on the forum.

Others

To communicate with the ARMstamp, 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.

Running

Getting Started, Using TclTerm

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 (INTEGER), and also supports arrays of 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.
- The archaic `$` suffix for strings is supported but not necessary
- Only single dimension arrays are supported.

Differences from PBASIC

Although **ARMbasic** has an extremely similar syntax to PBASIC, there are subtle differences, so some legacy code may not compile right away. 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 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 is 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 & 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 elimentary 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

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 and only requires a host to have a serial port or USB serial port. Aside from having a syntax compatible with Visual BASIC, **ARMbasic** introduces several features of the popular PBASIC, including I2C, SERIAL, PWM, IN, and OUT. **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.coridium.us . 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.

Floating Point Math

- ARMbasic uses 32 bit math for all numeric operations. There is no plan to add floating point at this point.

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.

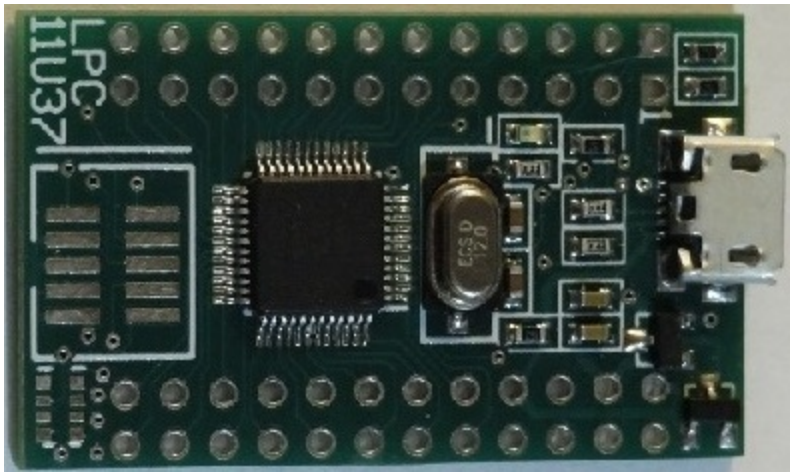
-

Revision History

Notices

NO WARRANTY

The Language



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

```
A$ =     "this is a string"

A$(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

END

Syntax

`END`

Description

`END` terminates the program. When **ARMbasic** is used in a control application, `END` is rarely reached, since most control programs run an indefinite loop — once a program ends, the only way to resume is a restart or reboot.

Example

```
' Halt the program if a hardware self-test fails.
IF NOT SelfTestPassed THEN
    PRINT "Self-test failed – halting."
    END
ENDIF

' ...rest of program runs only when SelfTestPassed is true...
```

Differences from other BASICs

- None.

See also

- STOP
- PAUSE
- 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

Syntax

GOSUB label

Description

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

Example

```
GOSUB message

END

message:

PRINT "Welcome!

return
```

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

Syntax

DEBUGIN *variable*

Description

Normally the programs running on an ARMstamp 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.

Another way to change values is to use the > execute statement immediately operation.

INPUT is used to control the direction of one of the 16 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.

Example

```
DIM yn$(10)

DO
    PRINT "Please enter a number: ";
    DEBUGIN a
    PRINT "and another ";
    DEBUGIN b

    PRINT "Thank you"
    SLEEP (500)
    PRINT
    PRINT "The total is "; a + b
    PRINT
    PRINT

    DO
        PRINT "Would you like to enter some more numbers";
        DEBUGIN yn$
    UNTIL (yn$(0) = "y") OR (yn$(0) = "n")
UNTIL (yn$(0) <> "y")
```

Differences from other BASICs

- ARMstamp 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
- Visual BASIC 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.

Example

```
DIM A(10)                      ' A$ is acceptable for backward compatibility
A = "World"

PRINT "Hello World"             ' new line printed
PRINT "Hello"; "World"; "!";    ' all on one line, no new line follows
PRINT                           ' prints just a newline

' 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

READ

RESTORE

RETURN

Syntax

RETURN

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

Example

```
PRINT "Let's Gosub!"

GOSUB MyGosub

PRINT "Back from Gosub!"

END

MyGosub:

PRINT "In Gosub!"

RETURN
```

Differences from other BASICs

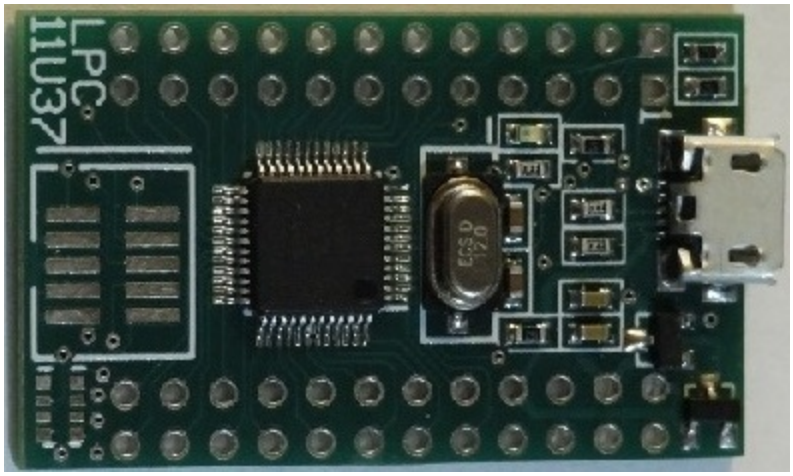
- a subset of the RETURN of Visual BASIC
- none from PBASIC

See also

- GOSUB.

WRITE

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)* [ELSE] *statement(s)* [ELSEIF *expression* THEN] *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.

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
        DEBUGIN "guess"; 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" 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

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.

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

CLEAR

Syntax

CLEAR

Description

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

Example

Example

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

Differences from other BASICs

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

See alsoRUN

CONST

Syntax

`CONST symbolname = value`

`CONST symbolname = { value, value, ... }`

Description

Declares compile-time constant symbols.

CONST supports multiple types — integer, byte, string, or single. You **cannot** mix multiple types inside a single `CONST { ... }`. The compiler looks at the first value listed and makes that the type for the whole statement.

Example

```
CONST Count = 5

' Array initializer syntax (replaces DATA / READ)
CONST ROWS AS BYTE = { ROW1, ROW2, ROW3, ROW4, ROW5 }   ' ROW1... done by #define elsewhere
CONST COLS = { 1, 2, 3, 4 }

' String constant
CONST failmsg = "FAIL**FAIL**FAIL**FAIL"

' Floating-point array constant
CONST factorials = { 1., 1., 2., 6., 24., 120., 720., 5040., 40320., 362880. }
```

Differences from other BASICs

- Visual BASIC allows more complex CONST declarations.
- PBASIC syntax is `symbolname CON value` (also accepted by ARMbasic).

See also

- Preprocessor

DATA

DIM

Syntax

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

Declaring Strings: DIM *symbolname$* (*size*)

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 number of elements in the array plus 1. This allows indexing from 0 to *size* .

All strings must 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.

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*)]
- this syntax is accepted by the compiler, though not recommended

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. 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.

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

RESTORE

RUN

Syntax

RUN

Description

Run or execute the program which has been loaded.

Example

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

Differences from other BASICs

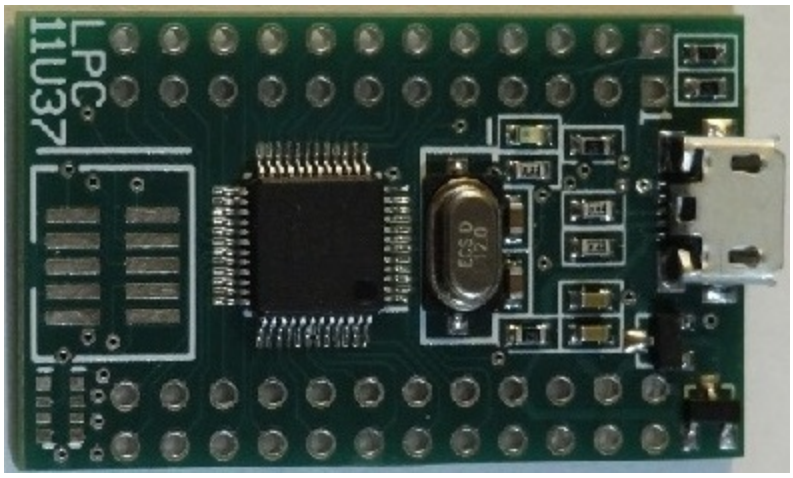
- same as many BASICs
- no equivalent in Visual BASIC

- no equivalent in PBASIC, done with the editor

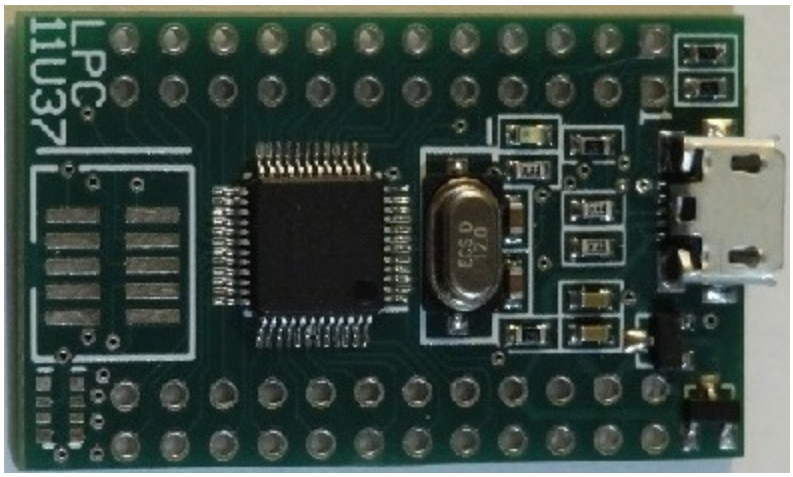
See also

- CLEAR

Operators



Operators List



& (String concatenation with conversion)

*** (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*

string1 `&` *string2*

Description

The concatenation operator takes two string variables and returns a string made of sticking both strings together. As of version 6.08 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 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

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. ABS returns its value as the same data type as the argument *number*.

Example

```
PRINT ABS ( -1 )

PRINT ABS ( 42 )

PRINT ABS ( N )

N = -69

PRINT ABS ( N )
```

The output would look like:

```
1

42

0

69
```

AND (Conjunction)

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

COS

MOD (Integer modulo)

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

- Arithmetic Operators

NOT (Bit-wise complement)

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 is
false   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. The same logic is also used when working with conditions. Not, when used bitwise, is mostly performed with positive integers, such as UInteger, UByte, or ULongInt. It can, however, be used with any numerical type.

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 (Disjunction: Inclusive 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

<< (Shift-left)

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

- >>
- Arithmetic Operators

>> (Shift-right)

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

- <<
- Arithmetic Operators

REV

SIN

XOR (Exclusive Or)

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

```
' Identical operands cancel
x = 5 XOR 5                ' x = 0
```

```
' Using the XOR operator on two numeric values
numeric_value1 = 15        ' 00001111
numeric_value2 = 30        ' 00011110
PRINT numeric_value1 XOR numeric_value2     ' 17 = 00010001
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_
END

' This outputs "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

When several operations occur in a single expression, they are evaluated in a predetermined order — the **operator precedence**. ARMbasic groups operators into three main categories: arithmetic, comparison, and logical. If an expression contains operators from more than one category, **arithmetic** is evaluated first, then **comparison**, and finally **logical**.

If operators have equal precedence, they are evaluated left to right. All comparison operators have equal precedence among themselves.

Parentheses override the default precedence. Operations inside parentheses are performed first; within a set of parentheses, normal precedence applies.

Precedence (highest to lowest within each column)

Arithmetic	Comparison	Logical
`-` (negation)	`=`, `<>`, `<`, `>`, `<=`, `>=`	`AND`
`*`, `/` (multiplication, division)		`OR`
`MOD` (modulus)		`XOR`
`<<`, `>>` (shift left, shift right)		`NOT`
`+`, `-` (addition, subtraction)		

See also

- Operator List

Data Types

Constants

Variables

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 or allowed.

Example

```
FirstNumber = 1

SecondNumber = 2

PRINT FirstNumber, SecondNumber, ThirdNumber 'This will print 1      2      0
```

Differences from other BASICs

- similar to Visual BASIC
- different syntac in PBASIC

See also

- Standard Identifier Rules
- 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 (1 to 10), and was filled with random numbers, you could look at it like this:

```
Index        Data
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)*

Description

A STRING is an array of characters, and is limited to 256 characters. 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. STRINGs are not checked for length at run time, so care must be taken to avoid filling it beyond the declared DIM. String variable names must end in a dollar sign.

Individual characters within a string can be accessed like an array, such as a$(12) returns the character in position 12, starting from 0.

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. String$ names allowed in Visual BASIC, but $ not enforced. Also 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.

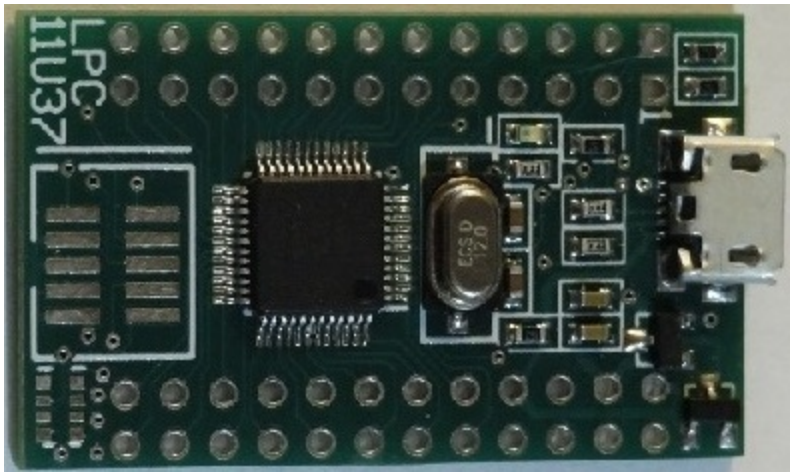
Example

```
DayOfWeek = *
($E0024034)    ' read the real time clock day of week
register
```

See also

-

Converting Data Types



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".

Example

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

Differences from other BASICs

- does not exist in PBASIC
- same function exists in Visual BASIC

See also

- STR
- HEX
- VAL

HEX

Syntax

HEX (*expression*)

Description

This returns the hexadecimal string representation of the integer *expression*. Hexadecimal values contain 0-9, and A-F. The size of the result string depends on the integer type passed, it's not fixed.

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.

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

- ABS
- AND
- BAUD
- CASE
- CHR
- CLEAR
- CONST
- DAY
- DIM
- DIR
- DIRS
- DO...LOOP
- DOWNTO
- ELSE
- ELSEIF
- END
- ENDIF
- ENDSELECT
- EXIT
- FOR...NEXT
- GOSUB
- GOTO
- HEX
- HIGH
- HOUR
- IF...THEN
- IN
- INS
- IO
- LEFT
- LOOP
- LOW
- MAIN
- MAX
- MIN
- MINUTE
- MOD
- MONTH
- NEXT
- NOT
- OR
- OUT
- OUTS

- PAUSE
- PRINT
- RD_BYTE
- RD_HALF
- RETURN
- REV
- RIGHT
- RUN
- RXD
- SECOND
- SELECT [CASE]
- SLEEP
- STEP
- STOP
- STR
- STRCOMP
- THEN
- TO
- TXD
- UNTIL
- WAIT
- WAITMICRO
- WEEKDAY
- WHILE...LOOP
- WR_BYTE
- WR_HALF
- XOR
- YEAR

ABS

AND

BAUD

Description

Sets the baud rate for *pin*, used by subsequent TXD(pin) and RXD(pin) calls on that pin.

Implemented as the `bbBAUD()` array declared inside `SERIAL.bas`, so that file must be included.

Syntax

```
#include <SERIAL.bas>
BAUD(pin) = baudrate
```

Parameters

Parameter	Type	Range	Notes
pin	Integer	Board-specific	e.g., 0–15
baudrate	Integer	up to 115.2 Kbaud	Receive is limited to 57.6 Kbaud

Example

```
#include <SERIAL.bas>

BAUD(2) = 19200          ' set baud rate for serial I/O on pin 2
BAUD(1) = BAUD(2)        ' set pin 1 to the same baud rate as pin 2
```

Differences from other BASICs

- No equivalent in Visual BASIC.
- No equivalent in PBASIC.

See also

- TXD
- RXD

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

CLEAR

CONST

COS

DATA

DAY

Syntax

DAY

Description

Function setting or returning the day of the month. Range 1 to 28, 29, 30, or 31 (depending on the month and whether it is a leap year).

Example

```
 SECOND = 30

 MINUTE = 15

 HOUR = 13

 DAY = 14

 MONTH = 4

 YEAR = 2006

 SELECT WEEKDAY

 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$
```

```
  PRINT MONTH; "/"; DAY; "/"; YEAR

  PRINT "The time is "; HOUR; ":"; MINUTE; ":"; SECOND
```

The output would look like:

```
This is Friday

  4/14/2006

  The time is 13:15:30
```

Differences from other BASICs

- no equivalent in PBASIC

See also

- SECOND
- MINUTE
- HOUR
- WEEKDAY
- MONTH
- YEAR

DIM

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.

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

- DIRS

DIRS

Syntax

DIRS

Description

DIRS reads or writes the 16 bits controlling the directions of the 16 IO pins. A 1 in one of the 16 bit positions corresponds to that IO pin being driven as an output.

The ARMmite allows control of 24 pins (0..23), with pins 16..23 shared with the AD pins.

Example

```
'set direction of pins 0 to 7 output, 8 to 15 input

DIRS = $00FF
```

Differences from other BASICs

- no equivalent in Visual BASIC
- none from PBASIC

See also

- DIR

DO...LOOP

DOWNTO

Syntax

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

Description

DESCRIPTION

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!"

END IF
```

Differences from QB

-

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

ENDIF

Syntax

if [statement] then [action] ENDIF

Description

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

Example

```
IF a = 1 THEN

 PRINT "A is equal to one!"
 ENDIF
```

See also

- IF...THEN

ENDSELECT

Syntax

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

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 both **ARMbasic** and PBASIC
- END SELECT (seperate words) are used in Visual BASIC

See also

- IF...THEN
- SELECT CASE

EXIT

FOR

GOSUB

GOTO

HEX

HIGH

Syntax

HIGH (*expression*)

Description

Drive the output of the pin corresponding to *expression * high.

See also

- LOW
- IO
- OUT

HOUR

Syntax

HOUR

Description

Function setting or returning the hour. Range 0 to 23.

Example

```
 SECOND = 30

 MINUTE = 15

 HOUR = 13

 DAY = 14

 MONTH = 4

 YEAR = 2006

 SELECT WEEKDAY

 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$
```

```
  PRINT MONTH; "/"; DAY; "/"; YEAR

  PRINT "The time is "; HOUR; ":"; MINUTE; ":"; SECOND
```

The output would look like:

```
This is Friday

  4/14/2006

  The time is 13:15:30
```

Differences from other BASICs

- no equivalent in PBASIC

See also

- SECOND
- MINUTE
- WEEKDAY
- DAY
- MONTH
- YEAR

IF...THEN

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*.

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.

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

- OUTS
- OUT
- INS
- IO

INS

Syntax

INS

Description

INS represents all 16 pin inputs. When read, it will return all corresponding values on the pins.

Writing to INS is not allowed.

The ARMmite allows control of 24 pins (0..23), with pins 16..23 shared with the AD pins.

Example

```
'set direction of pins 0 to 7 input, 8 to 15 output

DIRS = $FF00

'assume an external device is driving pins 0 to 7 high

PRINT "The current value of Input pins is "; INS

The current value of Input pins is 15
```

Differences from other BASICs

- no equivalent in Visual BASIC
- none from PBASIC

See also

- IN
- OUTS
- OUT
- IO

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.

Using IO simplifies pins that are being used as both inputs and outputs.

The ARMmite allows control of 24 pins (0..23), with pins 16..23 shared with the AD pins.

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
- OUTS
- IN
- INS

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

LOOP

Description

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

LOW

Syntax

LOW (*pin number*)

Description

Drive the output of the selected pin low.

See also

- HIGH

MAIN

MAX

Syntax

expression MAX *limit*

Description

Set the maximum value that expression may be.

Example

```
FOR k = 1 TO 5

PRINT k MAX 3

NEXT k
```

The output would look like:

```
1

2

3

3

3
```

See also

- MIN

MIN

Syntax

expression MIN *limit*

Description

Set the minimum value that expression may be.

Example

```
FOR k = 1 TO 5

PRINT k MIN 3

NEXT k
```

The output would look like:

```
3

3

3

4

5
```

See also

- MAX

MINUTE

Syntax

MINUTE

Description

Function setting or returning the minute. Range 0 to 59.

Example

```
 SECOND = 30

 MINUTE = 15

 HOUR = 13

 DAY = 14

 MONTH = 4

 YEAR = 2006

 SELECT WEEKDAY

 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$
```

```
  PRINT MONTH; "/"; DAY; "/"; YEAR

  PRINT "The time is "; HOUR; ":"; MINUTE; ":"; SECOND
```

The output would look like:

```
This is Friday

  4/14/2006

  The time is 13:15:30
```

Differences from other BASICs

- no equivalent in PBASIC

See also

- SECOND
- HOUR
- WEEKDAY
- DAY
- MONTH
- YEAR

MOD

MONTH

Syntax

MONTH

Description

Function setting or returning the month. Range 1 to 12.

Example

```
 SECOND = 30

 MINUTE = 15

 HOUR = 13

 DAY = 14

 MONTH = 4

 YEAR = 2006

 SELECT WEEKDAY

 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$
```

```
  PRINT MONTH; "/"; DAY; "/"; YEAR

  PRINT "The time is "; HOUR; ":"; MINUTE; ":"; SECOND
```

The output would look like:

```
This is Friday

  4/14/2006

  The time is 13:15:30
```

Differences from other BASICs

- no equivalent in PBASIC

See also

- SECOND
- MINUTE
- HOUR
- WEEKDAY
- DAY
- YEAR

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

OR

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, -1 or 0.

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.

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

- OUTS
- IN
- INS
- IO

OUTS

Syntax

OUTS

Description

OUTS represents all 16 pin outputs. When written to, it will transfer all corresponding values to the output pins that are currently configured as outputs.

Reading OUTS is the same as reading INS, in that it reads the current state of the 16 pins.

Writing or reading OUTS does not change the direction (input/output) of the pins.

The ARMmite allows control of 24 pins (0..23), with pins 16..23 shared with the AD pins.

Example

```
'set direction of pins 0 to 7 output, 8 to 15 input

DIRS = $00FF

'set pins 0 to 3 high, 4 to 7 low

OUTS = $000F

PRINT "The current value of Output pins is "; OUTS

The current value of Output pins is 15
```

Differences from other BASICs

- no equivalent in Visual BASIC
- none from PBASIC

See also

- OUT
- IN
- INS
- IO

PAUSE

Syntax

PAUSE (*milliseconds*)

Description

Delay program execution a number of milliseconds. 1000 milliseconds is one second WAIT and PAUSE are equivalent.

Example

```
DIRS = $00FF
'  set pins 0 to 7 to output
OUTS
= 255       '  and
then set them hign or to 3.3 V

FOR I=0 TO 7

  PAUSE (1000)

  LOW I
     ' set each pin LOW one after the other every second

NEXT I
```

Differences from other BASICs

- none from Visual BASIC
- none from PBASIC

See also

- STOP
- TIMER
- SLEEP
- END

PRINT

READ

RESTORE

RETURN

REV

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

```
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

RUN

RXD

Description

Receives a single byte from an asynchronous serial stream on *pin*, using the baudrate previously set via BAUD(pin). Similar to SERIN but a more efficient, lower-overhead implementation.

Returns 0–255 if data was received. If no serial stream is detected within 0.5 seconds, RXD times out and returns -1 (`$FFFFFFFF`).

Syntax

```
value = RXD(pin)
```

Parameters

Parameter	Type	Range	Notes
pin	Integer	Board-specific (e.g., 0–15)	On ARMstamp, pin 16 = SIN

Returns

Value	Meaning
0–255	Byte received
-1	Timeout (no data within 0.5 s)

Example

```
BAUD(1) = 9600          ' set 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 PBASIC's SERIN.

Legacy hardware-UART form (obsolete)

> Older Coridium firmware exposed the hardware UARTs through a channel-indexed form: `BAUD(channel)`, `RXD(channel)`, `TXD(channel)`. The channel selected which UART rather than which IO pin. Documented here for users still on older firmware.

`BAUD(channel) = rate` configures the UART for *channel* and sets its baud rate. On reset the pins are general-purpose IOs.

`value = RXD(channel)` returns 0–255 if a byte is available from UARTx, or -1 (`&HFFFFFFFF`) if not. The hardware UART is used, so the CPU is not tied up; up to 256 bytes are buffered by an interrupt routine.

`TXD(channel) = character` queues a byte for transmission on UARTx. Transmit uses interrupts and a FIFO (128 bytes on most boards, 64 on PROplus). If the buffer fills, the program stalls until there's room.

Per-board notes:

- **SuperPRO / PROplus** — 4 UARTs supported; pin assignments are in the Hardware Info. UART0 and UART1 work on all firmware versions; access to all 4 UARTs requires firmware 8.14 or later.
- **BASICchip / ARMmite** — UART0 only (RXD0/TXD0 on the schematic, LPC2103 UART0). Data is positive-true.
- Baudrates up to 115.2 Kbaud.

Legacy example:

```
SUB PrintUART1(Astr(100) AS STRING)
    DIM I AS INTEGER
    I = 0
    WHILE Astr(I)
        TXD(1) = Astr(I)
        I = I + 1
    LOOP
END SUB

main:
BAUD(1) = 19200          ' enable UART1
PrintUART1("Hello World") ' send a string out UART1

BAUD(2) = 9600           ' enable UART2 (not all boards have UART2)

' Wait for serial input on UART2
DO
    MyByte = RXD(2)
UNTIL MyByte >= 0
```

See also

- BAUD
- TXD
- SERIN

SECOND

Syntax

SECOND

Description

Function setting or returning the second. Range 0 to 59.

Example

```
 SECOND = 30

 MINUTE = 15

 HOUR = 13

 DAY = 14

 MONTH = 4

 YEAR = 2006

 SELECT WEEKDAY

 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$
```

```
  PRINT MONTH; "/"; DAY; "/"; YEAR

  PRINT "The time is "; HOUR; ":"; MINUTE; ":"; SECOND
```

The output would look like:

```
This is Friday

  4/14/2006

  The time is 13:15:30
```

Differences from other BASICs

- no equivalent in PBASIC

See also

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

SELECT CASE

SIN

SLEEP

Syntax

`SLEEP`

Description

Puts the CPU into low-power sleep mode. The processor halts until an interrupt wakes it. Unlike WAIT or PAUSE, SLEEP does not block on a fixed time — execution resumes when an enabled interrupt fires.

SLEEP no longer accepts a parameter. For a timed delay, use WAIT or PAUSE.

Example

```
' Wake on EINT0 (set up elsewhere with INTERRUPT SUB)

DO
    SLEEP                ' park the CPU; an interrupt resumes execution
    ' ... handle whatever the ISR signaled ...
LOOP
```

Differences from other BASICs

- None from Visual BASIC.
- PBASIC's `SLEEP seconds` form is no longer supported — use WAIT or PAUSE.

See also

- INTERRUPT
- INTERRUPT SUB
- WAIT
- PAUSE

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. The ARM module will not respond until it has been reset.

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

See also

- EXIT

STR

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

THEN

Description

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

TO

Syntax

FOR *iterator intial_value* TO *ending_value* ... NEXT [*iterator*] SELECT *case_comparison_value* CASE *lower_bound* TO *upper_bound* ... END SELECT

Description

The TO keyword is used to define a certain numerical range. This keyword is valid only if used with FOR ... NEXT and SELECT / CASE . In the first syntax, the TO keyword defines the initial value of the iterator in a FOR

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

Example

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

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 ) ; " degr

          CASE min_medium_danger TO max_medium_danger

             COLOR 14

             PRINT "Temperature" ; it ; " is in the medium danger zone at" ; array( it ) ; " d

          CASE min_high_danger TO max_high_danger

             COLOR 12

             PRINT "Temperature" ; it ; " is in the high danger zone at" ; array( it ) ; " degr

          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

TXD

Description

The byte (0–255) is transmitted with a START bit and trailing STOP bit. The CPU bit-bangs the output, so the program blocks at this instruction until the shift completes.

Syntax

`TXD(pin) = value`

Parameters

Parameter	Type	Range	Notes
pin	Integer	Board-specific (e.g., 0–15)	On ARMstamp, pin 16 = SOUT
value	Integer	0–255	Single byte to transmit

Example

```
DIM A$(10)
BAUD(2) = 19200          ' set baud rate for serial I/O on pin 2
A$ = "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.
- Preferred alternate to PBASIC's SEROUT.

See also

- BAUD
- RXD

- SEROUT

UNTIL

Syntax

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

WAIT

Syntax

WAIT (*milliseconds*)

Description

Delay program execution a number of milliseconds. 1000 milliseconds is one second

WAIT and PAUSE are equivalent.

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 that uses a CPU dependent "tick" value

See also

- PAUSE
- SLEEP

WEEKDAY

Syntax

WEEKDAY

Description

Function returning the number of the day of the week. 0 corresponding to Sunday through 6 corresponding to Saturday

Example

```
 SECOND = 30

 MINUTE = 15

 HOUR = 13

 DAY = 14

 MONTH = 4

 YEAR = 2006

 SELECT WEEKDAY

 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$
```

```
  PRINT MONTH; "/"; DAY; "/"; YEAR

  PRINT "The time is "; HOUR; ":"; MINUTE; ":"; SECOND
```

The output would look like:

```
This is Friday

  4/14/2006

  The time is 13:15:30
```

Differences from other BASICs

- no equivalent in PBASIC

See also

- SECOND
- MINUTE
- HOUR
- DAY
- MONTH
- YEAR

WHILE

WRITE

XOR

YEAR

Syntax

YEAR = 2006

PRINT YEAR

Description

Function setting or returning the year.

Example

```
 SECOND = 30

 MINUTE = 15

 HOUR = 13

 DAY = 14

 MONTH = 4

 YEAR = 2006

 SELECT WEEKDAY

 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$
```

```
  PRINT MONTH; "/"; DAY; "/"; YEAR

  PRINT "The time is "; HOUR; ":"; MINUTE; ":"; SECOND
```

The output would look like:

```
This is Friday

  4/14/2006

  The time is 13:15:30
```

Differences from other BASICs

- no equivalent in PBASIC

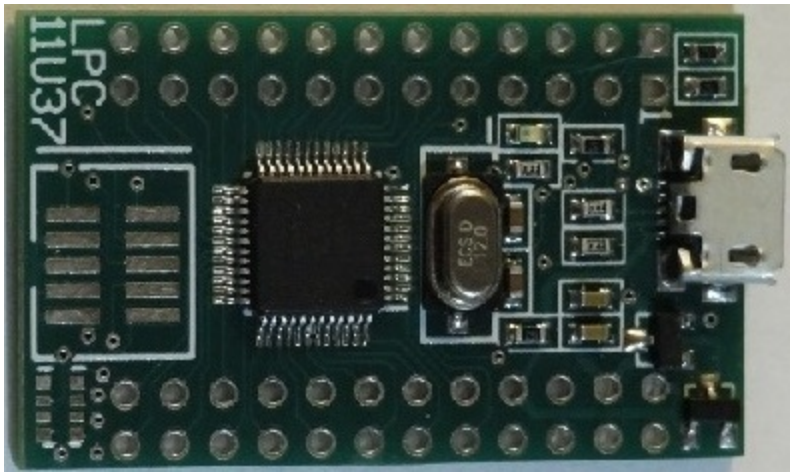
See also

- SECOND
- MINUTE
- HOUR
- WEEKDAY
- DAY
- MONTH

Additional Reserved Words

Runtime Library

Date and Time Functions



DAY

HOUR

MINUTE

MONTH

PAUSE

SECOND

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

```
TIMER = 0

WHILE (wait for something to happen)

PRINT "That took "; TIMER; " microseconds"
```

Differences from other BASICs

- no equivalent in PBASIC
- no equivalent in Visual BASIC

See also

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

WAIT

WEEKDAY

YEAR

Mathematical Functions

ABS

COS

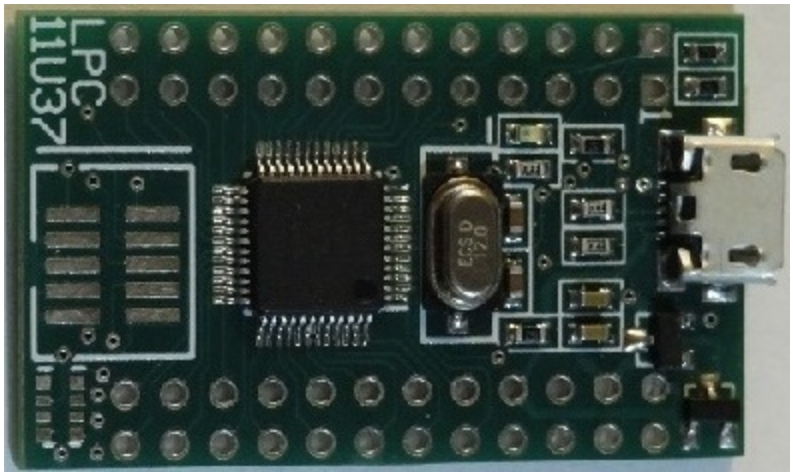
MOD

<< (Shift-left)

>> (Shift-right)

SIN

String Functions



CHR

HEX

LEFT

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

RIGHT

STR

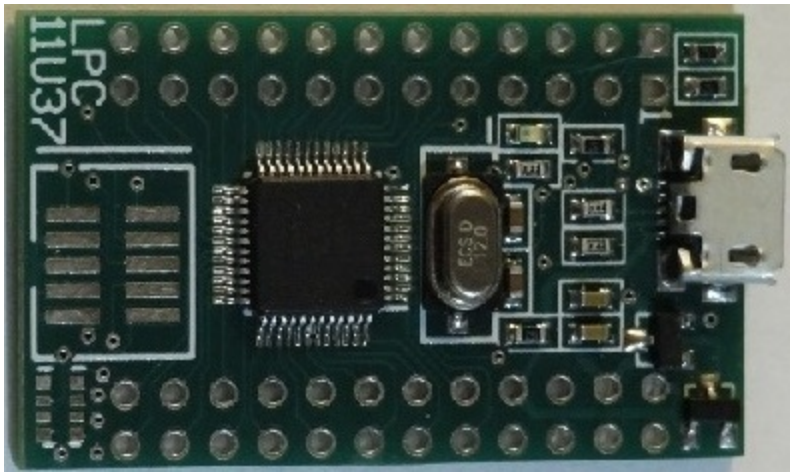
STRCOMP

VAL

User Input Functions

DEBUGIN

Hardware Library



Pin Controls

AD

Syntax

AD (expression)

Description --- Not available on ARMstamp

AD will return 0.. that corresponds to the voltage on the pin corresponding to *expression*. The value returned will be between 0 and ... 65472, with the top 10 bits being significant. (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. 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.

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

DIR

DIRS

IN

INS

IO

OUT

OUTS

Function List

* (ARM periph access)

Syntax

* *variable*

* *constant*

Description

The Cpointer syntax is used to give direct access to the ARM peripheral registers. Caution must be taken, for instance in interupts are altered, this will interfere with the operation of **ARMbasic** and your program.

Example

```
X=
$E0014018

*X =
1234
' set the ARM PWM0 match register
T = * $E001401C
' read the ARM PWM1 match register

*
$E0014020 = 4321   ' set the ARM PWM2 match
register
```

Differences from other BASICs

- No equivalent in Visual BASIC
- no direct equivalent in PBASIC, CONFIGPIN is a similar function

See also

- Hardware Library Functions

BAUD

COUNT

Syntax

COUNT *pin, milliseconds, variable*

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

```
'Report the number of transition cycles on pin 7
during a 10 second interval

COUNT 7, 10000, val

PRINT "Pin 7 transitioned "; val; " times"

Pin 7 transitioned 3 times
```

```
'test the operation of the COUNT routine  --
attach a 100 KHz signal generator to pin 15

' this uses the > run immediate operator and the . print immediately
operator

>COUNT 15, 10000, x

.x

99630
```

Differences from other BASICs

- no equivalent in Visual BASIC
- none from PBASIC

See also

- RCTIME

FREQOUT

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, OUT(x)=1 will be faster.

Example

```
DIRS = $00FF      '  set
pins 0 to 7 to output

OUTS   =

  0
  '  and then set them low or to 0 V

FOR I=0 TO 7
  PAUSE (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

I2CIN

Syntax

I2CIN pin, slaveADDR, [opt1, [... opt5,]] [InputList] InputList = variable | stringname$ \count | *arrayname* *count* [, InputList]

Description

I2CIN will read a series of bytes from an I2C slave device. pin is any expression defining the SDA pin to use. pin+1 will be designated the SCL pin. slaveADDR will select a device on the I2C bus. Up to 5 optional byte values may be sent out prior to reading the InputList. After that a series of bytes will be read from the slave to fill the InputList. The InputList enclosed in [] can contain variables or strings. When using a string a count must be specified defines the number of bytes to be copied into the string. I2C is a byte

oriented bus, so each transaction will either send a byte value (0 to 255) or receive a byte for each element of the InputList. Data is shifted in at 380 Kbits/sec.

Example

```
DIM A$(10)

 I2CIN 1,$30,$10, [A$ \10]      ' read 10 bytes
from
slave $30 register $10 connected on pins 1,2

I2CIN 5,$40,$20, [X]
  ' read a single byte from slave $40 register $20 on pins 5,6
```

Differences from other BASICs

- PBASIC output formatting not supported
- PBASIC secondADDR specification is seperated by , rather than
- no equivalent in Visual BASIC

See also

- I2COUT

I2COUT

Syntax

I2COUT *Data pin, slaveADDR, {Present,} * [OutputList] *OutputList* = *expr | arrayname count | stringname$ {\count} | "string"   { , OutputList} *

Description

I2COUT will send a series of bytes from an I2C slave device. Datapin is any expression defining the SDA pin to use. Datapin+1 will be designated the SCL pin. slaveADDR will select a device on the I2C bus.

If a one-wire device responds the optional variable *Present* will be set to 1, else 0. (added in version 6.08)

After that a series of bytes will be written to the slave from the OutputList. The OutputList enclosed in [] can contain expressions array-variables or strings. When using an array-variable without an (element) a count must be specified that defines the number of bytes to be copied from the string.

[OutputList] can also contain a "constant-string" or *stringame*Thelatterwillsendoutbytesstartingfrom * stringname(0)* until a 0 byte is read. I2C is a byte oriented bus, so each transaction will send a byte values (0

to 255) to an I2C slave. If the value from an *expr* ession in the *OutputList* is larger than 8 bits, the MSBs will be truncated. Data is shifted out at 380 Kbits/sec.

Example

```
DIM A$(10)

FOR I=0 TO 9

A$(I) = $30 + I

NEXT I

 I2COUT 1,$30, [$10,A$ \10]  ' send 10 bytes
toslave $30 register $10 connected on pins 1,2

X=$55

I2COUT
5,$40, [$20,X]
' send a single byte to slave $40 register $20 on pins 5,6

  I2COUT 5,$50, [$20,$AA]       '
send $AA to slave $50 register $20 on pins 5,6
```

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

INPUT

Syntax

INPUT expression

Description

INPUT will set the pin corresponding to expression to an input.

INPUT and OUTPUT were added for PBASIC compatability, DIR(x)=0 will be faster.

Example

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

Differences from other BASICs

- INPUT gets a value from the user in Visual BASIC
- none from PBASIC

See also

- IO
- DIRS
- OUTPUT

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, OUT(x)=0 will be faster.

Example

```
DIRS = $00FF
'  set pins 0 to 7 to output
OUTS
= 255       '  and
then set them hign or to 3.3 V

FOR I=0 TO 7
  PAUSE
(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

OUTPUT

Syntax

OUTPUT *expression*

Description

OUTPUT will set the pin corresponding to *expression* to an output. INPUT and OUTPUT were added for PBASIC compatability, DIR(x)=1 will be faster.

Example

```
' Set pin 9 as an output

OUTPUT 9
```

Differences from other BASICs

- no equivalent in Visual BASIC
- none from PBASIC

See also

- DIR
- DIRS
- INPUT

OWIN

Syntax

OWIN *Pin, {Output1, Output2, ... Output6}, ***[** *InputList* **]**

InputList =

Variable | Array(expr) | String(expr) | Array \ count| String \ count{,InputList}

Description

OWIN begins with a RESET/Presence sequence on the designated *Pin*.

Then upto 6 *Output* bytes will be transfered to the device to select the command.

Following that the *InputList* will be read back from the device. The *InputList* consists of a list of variable names, string or array elements, or a string or array followed by a count. In the last case, a series of bytes will be read from the device and placed into individual elements of the string or array. The input list may consist of multiple parts with upto 32 different targets.

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

```
' write to the scratch pad of a DS2430
x = $be
y = $41
owout
7,present,[$cc,$f,$6,x,y]
print present
owin 7,$cc,$aa,$6,[a,b]
print
hex(a),hex(b)
```

Differences from other BASICs

- no equivalent in Visual BASIC
- simplified from PBASIC

See also

- OWOUT

OWOUT

Syntax

OWOUT *Pin, {Present,} [OutputList] OutputList = expr | arrayname \ count | stringname$ {\count} | "string" { , OutputList}*

Description

OWOUT begins with a RESET/Presence sequence on the designated *Pin*.

If a one-wire device responds the optional variable *Present* will be set to 1, else 0.

Following that the *OutputList* will be sent to the device. The *OutputList* consists of a list of numeric *expr* ressions, or a string or array followed by a length. In the last case, a individual elements of the string or array will be written to the device as a series of bytes. Elements will be truncated to byte length. The output list may consist of multiple parts with upto 32 different sources.

[OutputList] can also contain a "constant-string" or *stringame*Thelatterwillsendoutbytesstartingfrom * stringname(0)* until a 0 byte is read.

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.

Other than string$ names, string expressions are not allowed in the *OutputList*.

Example

```
' write to the scratch pad of a DS2430
x = $be
y = $41
owout
7,present,[$cc,$f,$6,x,y]
print present
owin 7,$cc,$aa,$6,[a,b]
print
hex(a),hex(b)
```

Differences from other BASICs

- no equivalent in Visual BASIC
- simplified than PBASIC

See also

- OWIN

PULSIN

Syntax

PULSIN *pin, level, variable*

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

```
'Wait for pin 7 to go high then low.

'Print the number of microseconds pin 7 was high.

PULSIN 7, 1, val

PRINT "Pin 7 pulse high for "; val; " us"
```

```
'this example will only work on the ARMstamp
```

```
' PULSIN routine  -- attach a 10 KHz signal
generator to pin 15

' this uses the > run immediate operator and the . print immediately
operator

>PULSIN 15, 0, x

.x

50
```

Differences from other BASICs

- no equivalent in Visual BASIC
- Times are measured in microseconds rather than CPU dependent ticks in PBASIC

See also

- COUNT

PULSOUT

PWM

Description

Generates a PWM (pulse-width modulated) output on a pin for a fixed duration. 8-bit resolution (256 levels). The pin's direction is set to output for the duration, then returned to tristate (input); if the pin drives an RC filter, the voltage will hold on the capacitor for a period determined by the load.

Syntax

`PWM pin, duty, duration`

Parameters

Parameter	Type	Range	Notes
pin	Integer	Board-specific	e.g., 0–31
duty	Integer	0–255	128 ≈ 50%, 255 = 100%
duration	Integer	ms	how long to drive the pin

Example

```
' Generate a 1.65 V signal (half of 3.3 V) on pin 4 for 6 seconds.
PWM 4, 127, 6000
```

Differences from other BASICs

- None from Visual BASIC.
- In PBASIC, *duration* is CPU-dependent and measured in ticks rather than milliseconds.

RCTIME

RXD

SERIN

Description

Receives bytes as asynchronous serial data on *pin* at *baudrate* and stores them into *InputList*. *PosTrue* is optional; if set to 0 the data is inverted.

InputList can contain a list of variables or `stringname$ \ len` (which receives *len* bytes starting from `stringname$(0)`). SERIN times out after 0.5 seconds and returns -1 (`$FFFFFFFF`) in the next item if no data was received.

Syntax

`SERIN pin, baudrate, { posTrue, } [InputList]`

Parameters

Parameter	Type	Range	Notes
pin	Integer	Board-specific (e.g., 0–15)	On ARMstamp, pin 16 = SIN (negative true, max 19.2 K). On ARMmite, pin 16 = SIN (positive true).
baudrate	Integer	up to 115.2 Kbaud	Pin 16 capped at 19.2 K on ARMstamp
posTrue	Integer	0 or non-zero (optional)	0 = inverted, otherwise positive-true
InputList	List	variables or string	see Description; -1 returned on timeout

Example

```
' Read serial stream from pin 1, save to MyByte
SERIN 1, 19200, [MyByte]
PRINT HEX(MyByte)
```

Differences from other BASICs

- No equivalent in Visual BASIC.
- Simplified from PBASIC.

See also

- SEROUT
- RXD
- BAUD

SEROUT

Description

Sends bytes from *OutputList* as asynchronous serial data on *pin* at *baudrate*. *PosTrue* is optional; if set to 0 the data is inverted.

OutputList can contain a list of constants, variables, `"constant-string"`, `arrayname \ count`, or `stringname$ {\count}` (which sends bytes starting from `stringname$(0)` until a 0 byte is read).

Syntax

`SEROUT pin, baudrate, { posTrue, } [OutputList]`

Parameters

Parameter	Type	Range	Notes
pin	Integer	Board-specific (e.g., 0–15)	On ARMstamp, pin 16 = SOUT (negative true, max 19.2 K). On ARMmite, pin 16 = SOUT (positive true).
baudrate	Integer	up to 115.2 Kbaud	Pin 16 capped at 19.2 K on ARMstamp
posTrue	Integer	0 or non-zero (optional)	0 = inverted, otherwise positive-true
OutputList	List	constants, variables, strings	see Description

Example

```
SEROUT 1, 19200, [$31, $32, $33]   ' sends "123" at 19.2 Kbaud
```

Differences from other BASICs

- No equivalent in Visual BASIC.
- Simplified from PBASIC.

See also

- SERIN
- TXD
- BAUD

SHIFTIN

Syntax

SHIFTIN *Dpin, Cpin, Mode, [Variable{\ Bits} {, Variable{\ Bits}...}]*

Description

SHIFTIN has been kept as a compatable function with PBASIC. It can be used for devices that are not covered by SPI, I2C or 1-Wire.

While most other hardware functions use bytes, SHIFTIN is oriented for bit control. The length of each variable defines the number of bits that will be shifted out (2 - 32). If a Bit value is not specified it is assumed to be 8.

- Mode = 0 data is shifted in MSB first, and sampling starts before the first clock pulse
- Mode = 1 data is shifted in LSB first, and sampling starts before the first clock pulse
- Mode = 2 data is shifted in MSB first, and sampling starts before the second clock pulse
- Mode = 3 data is shifted in LSB first, and sampling starts before the second clock pulse

SHIFTIN does not allow arrays or strings to be part of the InputList. Values shifted in will be positive values, unless a full 32 bit value is shifted in, and then if the MSB is 1 it will be negative. Data is shifted in at 1.1 Mbits/sec.

Example

```
' use SHIFTIN/OUT to control an SPI EN28J60

rw=2

reg = $1b

io(6)=0

shiftout 3,4,1,[rw\3, reg\5, y]      ' set reg $1B

io(6)=1

io(6)=0

shiftout 3,4,1,[reg]
         'select the register

shiftin 5,4,0,[x]
               'and
read it back

io(6)=1
```

Differences from other BASICs

- no equivalent in Visual BASIC
- similar to PBASIC

See also

- SHIFTOUT
- SPIIN

SHIFTOUT

Syntax

SHIFTOUT *Dpin, Cpin, Mode, [OutputData{\ Bits} {, OutputData{ Bits}...}]*

Description

SHIFTOUT has been kept as a compatable function with PBASIC. It can be used for devices that are not covered by SPI, I2C or 1-Wire.

While most other hardware functions use bytes, SHIFTOUT is oriented for bit control. The length of each variable defines the number of bits that will be shifted out (2 - 32). If Bit is not defined it is assumed to be 8.

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

SHIFTOUT does not allow arrays or strings to be part of the OutputList.

Data is shifted out of the device at 1.1 Mbits/sec.

Example

```
' use SHIFTIN/OUT to control an SPI EN28J60

rw=2

reg = $1b

io(6)=0

shiftout 3,4,1,[rw\3, reg\5, y]      ' set reg $1B

io(6)=1

io(6)=0

shiftout 3,4,1,[reg]
         'select the register

shiftin 5,4,0,[x]
               'and
read it back

io(6)=1
```

Differences from other BASICs

- none from Visual BASIC
- simplified from PBASIC

See also

- SHIFTIN
- SPI,Microwire

SPIIN

Syntax

SPIIN *CSpin*, *inpin, clockpin, {outpin}, {out1, {out2,...}}, [InputList]*

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 *clockpin*, to fill the *InputList*.

In bi-directional cases, *out1..out3* will be output on *outpin* before reading the *InputList* .

Data is shifted in LSB first and each element of the *InputList* is filled with a byte of data. The REV operator can be used to change the bit order. Data is shifted in at 1.1 Mbits/sec. (It was 800 Kbits/sec in version 6.07 and before)

Example

```
'debugging the SPI interface to an ENC28J60
-- done in immediate mode

hex

>spiout 6,3,4,[$5b rev 8, $55 rev 8]      '
setting the EIE register

>spiin 6,5,4,3,$1b rev 8,[x]

.x rev 8

55
```

Differences from other BASICs

- no equivalent in Visual BASIC
- no equivalent in PBASIC

See also

- SPIOUT

SPIOUT

Syntax

SPIOUT *CSpin*, *outpin, clockpin, [OutputList] OutputList = expr | arrayname \ count | stringname$ {\count} | "string" { , OutputList}*

Description

SPIOUT 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, *outpin* is used to output data clocked by *clockpin*, from the *OutputList*.

[OutputList] can contain a list of constants, variables, "constant-string"
or *stringame*Thelatterwillsendoutbytesstartingfrom * stringname(0)* until a 0 byte is read.

Data is shifted out LSB first and each element of the *OutputList* is treated as a byte. The REV operator can be used to change the bit order. Data is shifted out at 1.4 Mbits/sec. (version 6.07 and before this was 800 Kbits/sec)

Example

```
SPIOUT
4,1,2,[$04,GAIN]
' set the GAIN register of an ADS7870

SPIIN 4,3,2,1,$44,[GAIN}
       ' read the GAIN register back
```

Differences from other BASICs

- no equivalent in Visual BASIC
- no equivalent in PBASIC

See also

- SPIIN

TXD

Alphabetical Keyword List

BAUD

COUNT

DEC

Syntax

DEC (*number*)

Description

DESCRIPTION

Example

```
EXAMPLE
```

Differences from other BASICs

- none from Visual BASIC
- none from PBASIC

See also

- HEX

DIR

DIRS

FREQOUT

HEX

HIGH

I2CIN

I2COUT

IN

INPUT

INS

IO

LOW

OUT

OUTPUT

OUTS

OWIN

OWOUT

PAUSE

PULSIN

PULSOUT

PWM

RCTIME

RXD

SERIN

SEROUT

SHIFTIN

SHIFTOUT

SLEEP

SPIIN

SPIOUT

STOP

STR

TXD

Hardware Specs

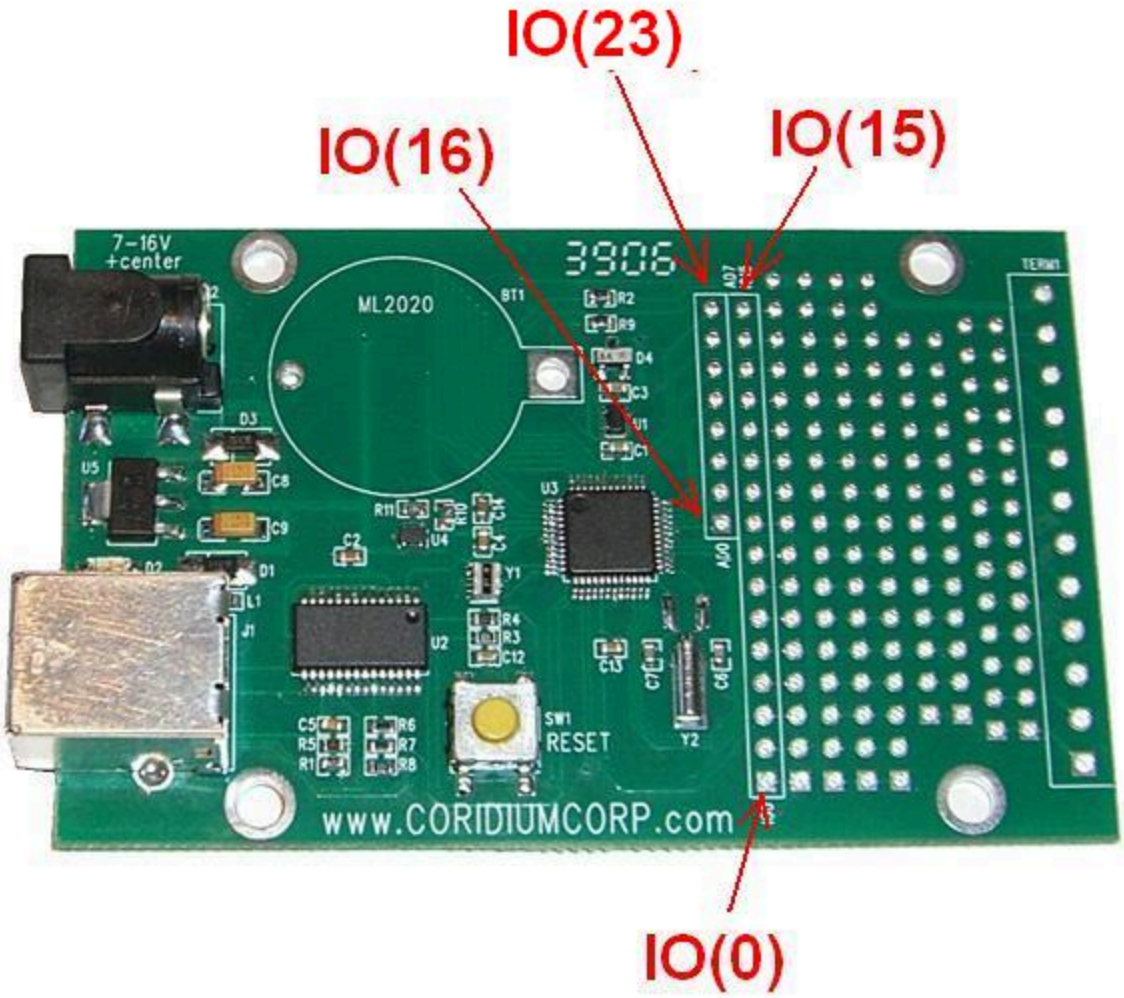
ARMmite Pins

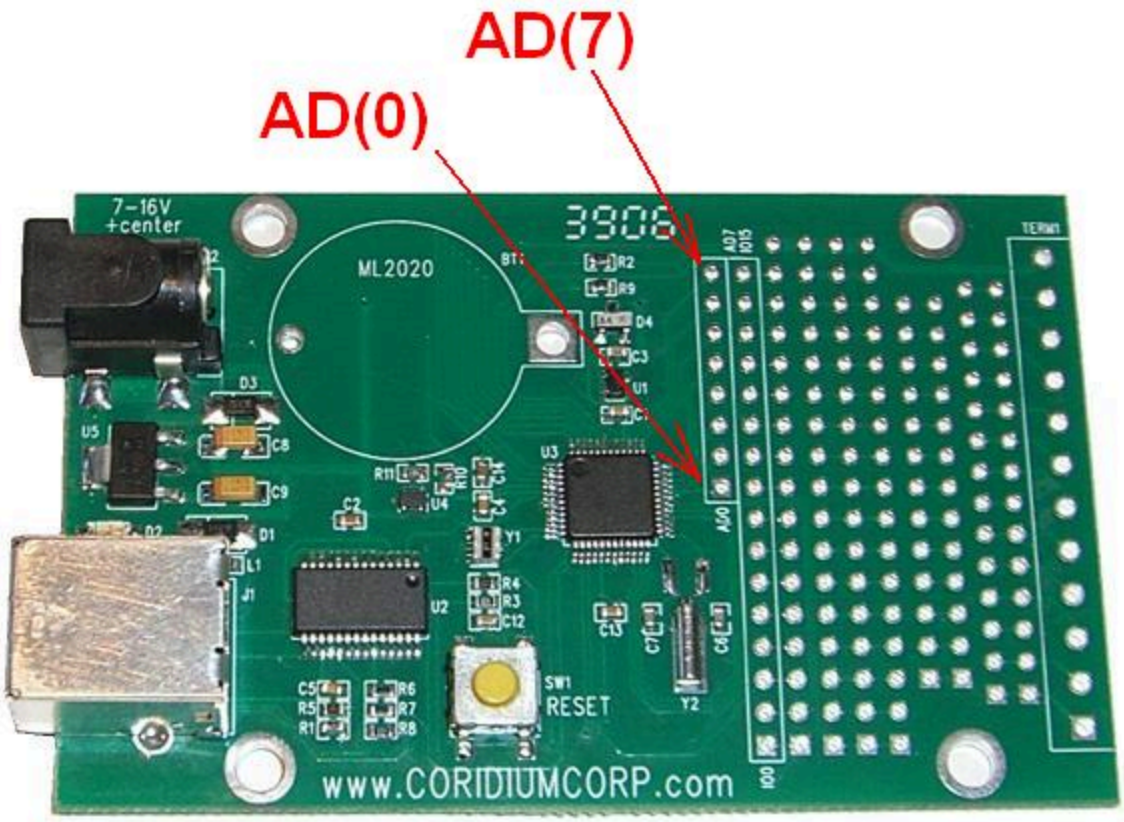
24 pins available to the user, 8 of which can be analog inputs **Dual Use AD pins**

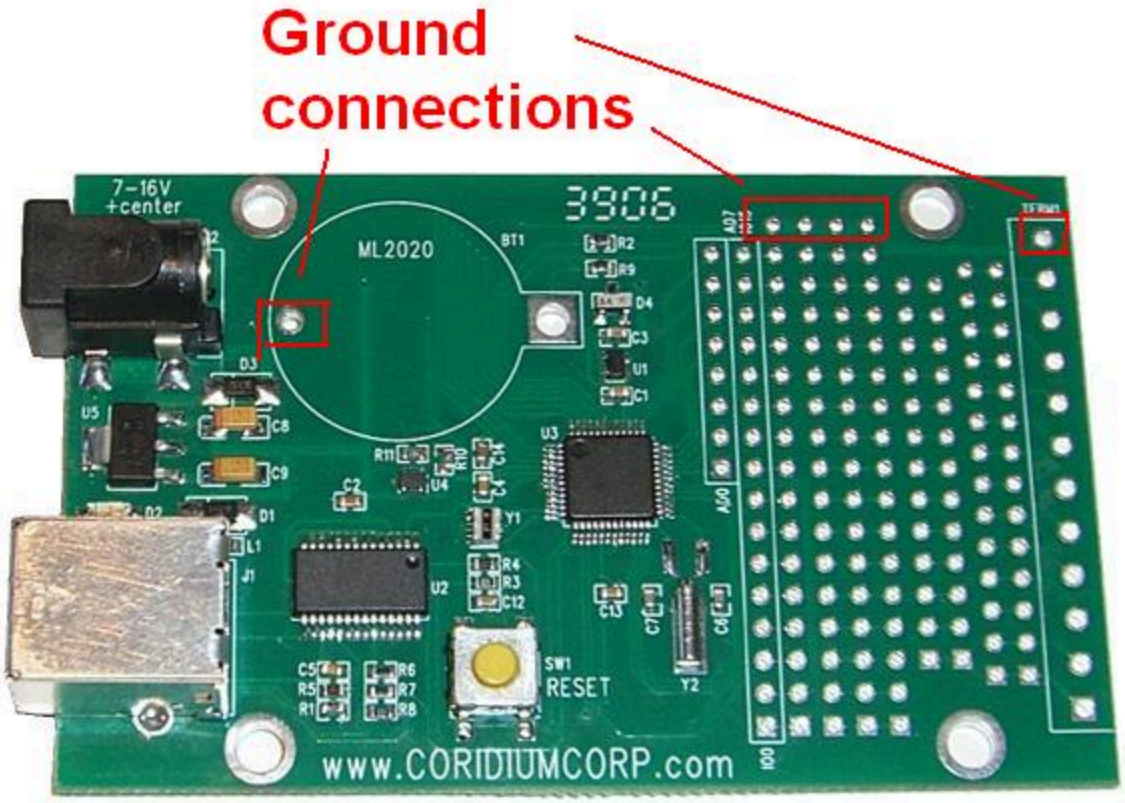
On reset or power up the AD pins are configured as AD inputs. To change those to digital IOs, the user must individually specify a control direction using INPUT x, OUTPUT x, or IO(x) commands. After that they will remain digital IOs until the next reset or power up.

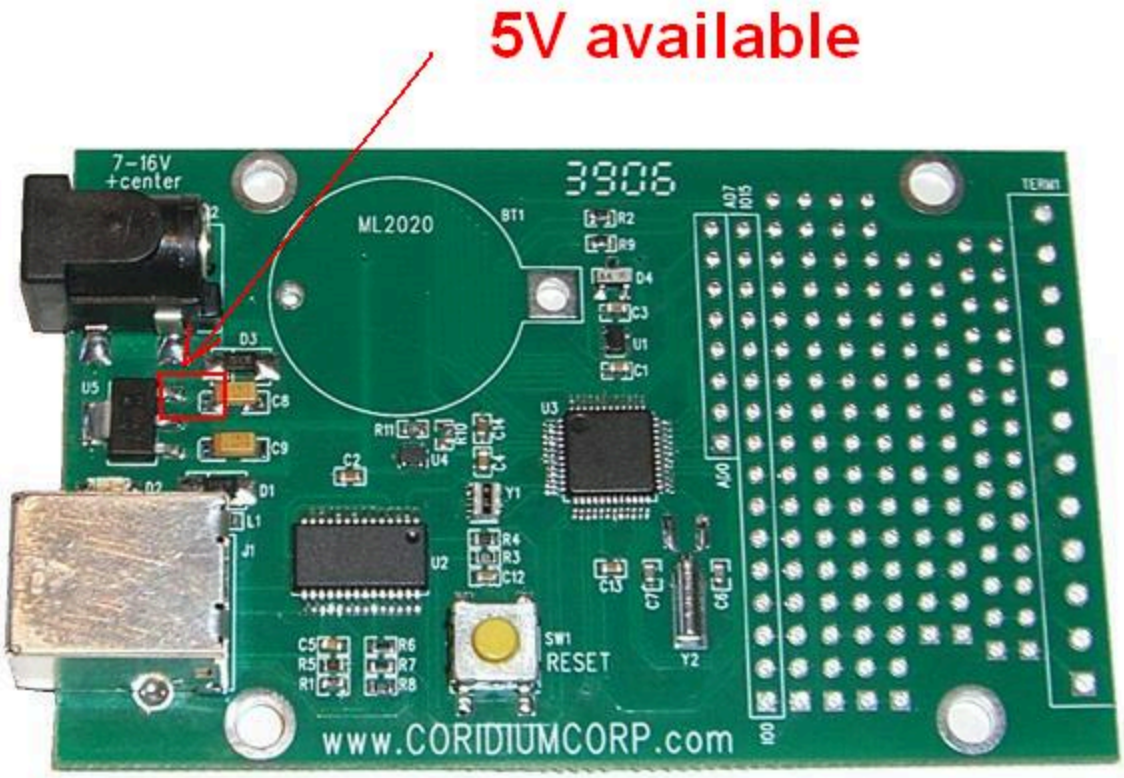
Battery Real Time Clock

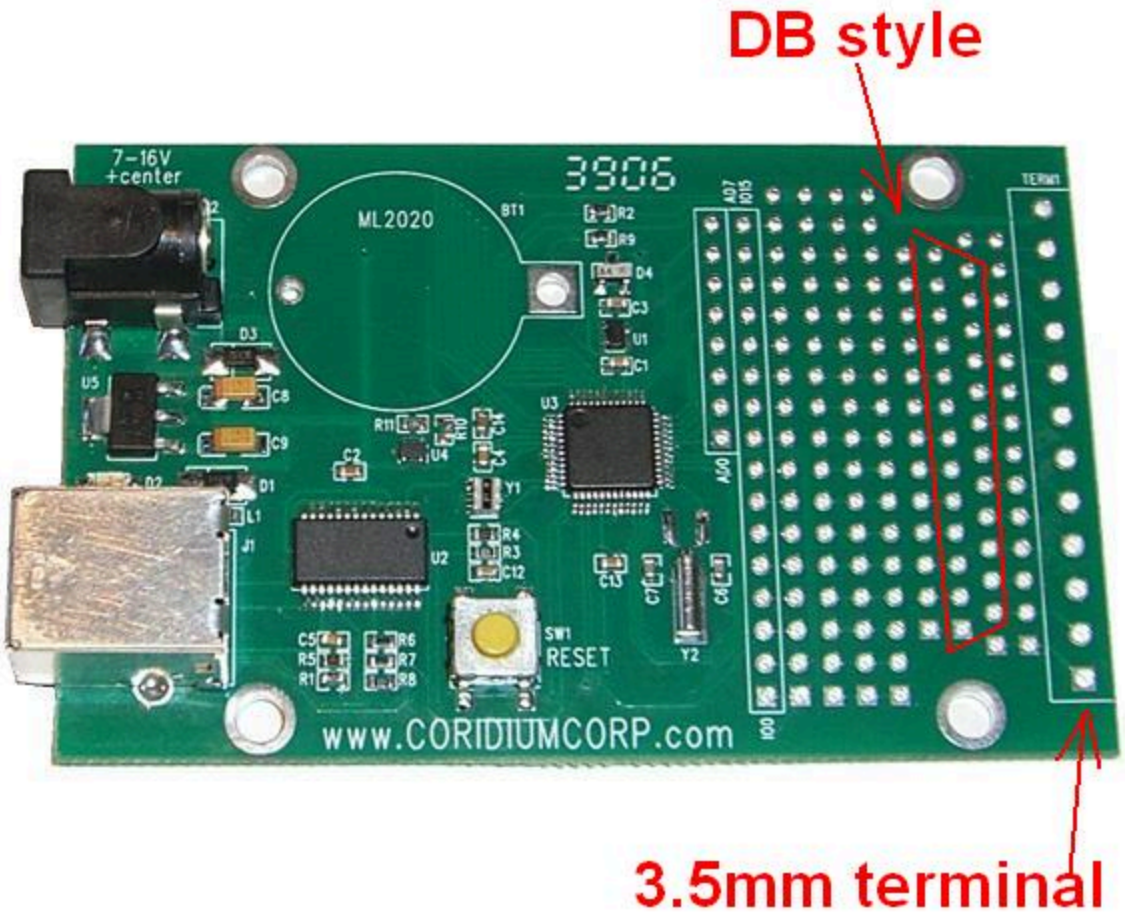
The ARMmite board is designed to accept a Panasonic ML2020/H1C rechargeable Lithium battery at position BT1. This battery powers the real time clock of the LPC2103. The contents of RAM is not kept alive while running on battery, and the CPU restarts the user program in Flash when power is restored. This battery is designed to maintain power for a few days without power, and will recharge fully in about 1 day.


IO(23)
IO(16)
IO(15)
7-16V
+center
ML2020
3906
RESET
www.CORIDIUMCORP.com
IO(0)

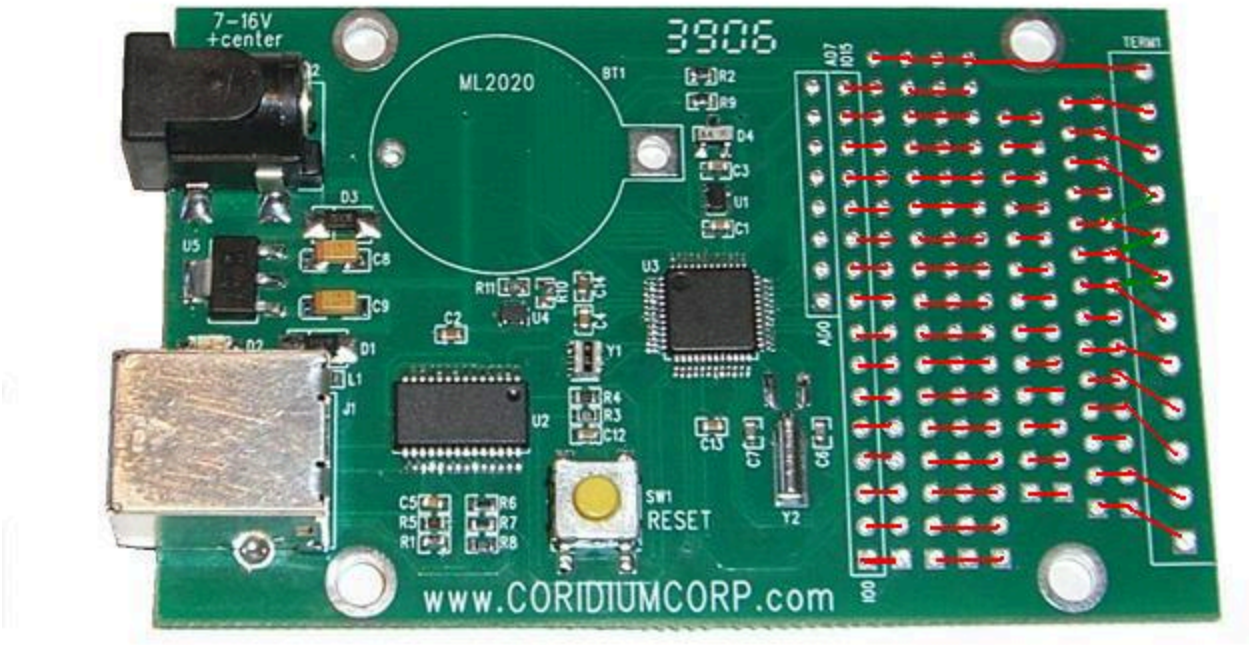

AD(7)
AD(0)
7-16V
+center
ML2020
3906
RESET
www.CORIDIUMCORP.com


Ground
connections
7-16V
+center
ML2020
BT1
3906
SW1
RESET
Y2
www.CORIDIUMCORP.com


5V available
7-16V
+center
ML2020
BT1
3906
RESET
SW1
www.CORIDIUMCORP.com

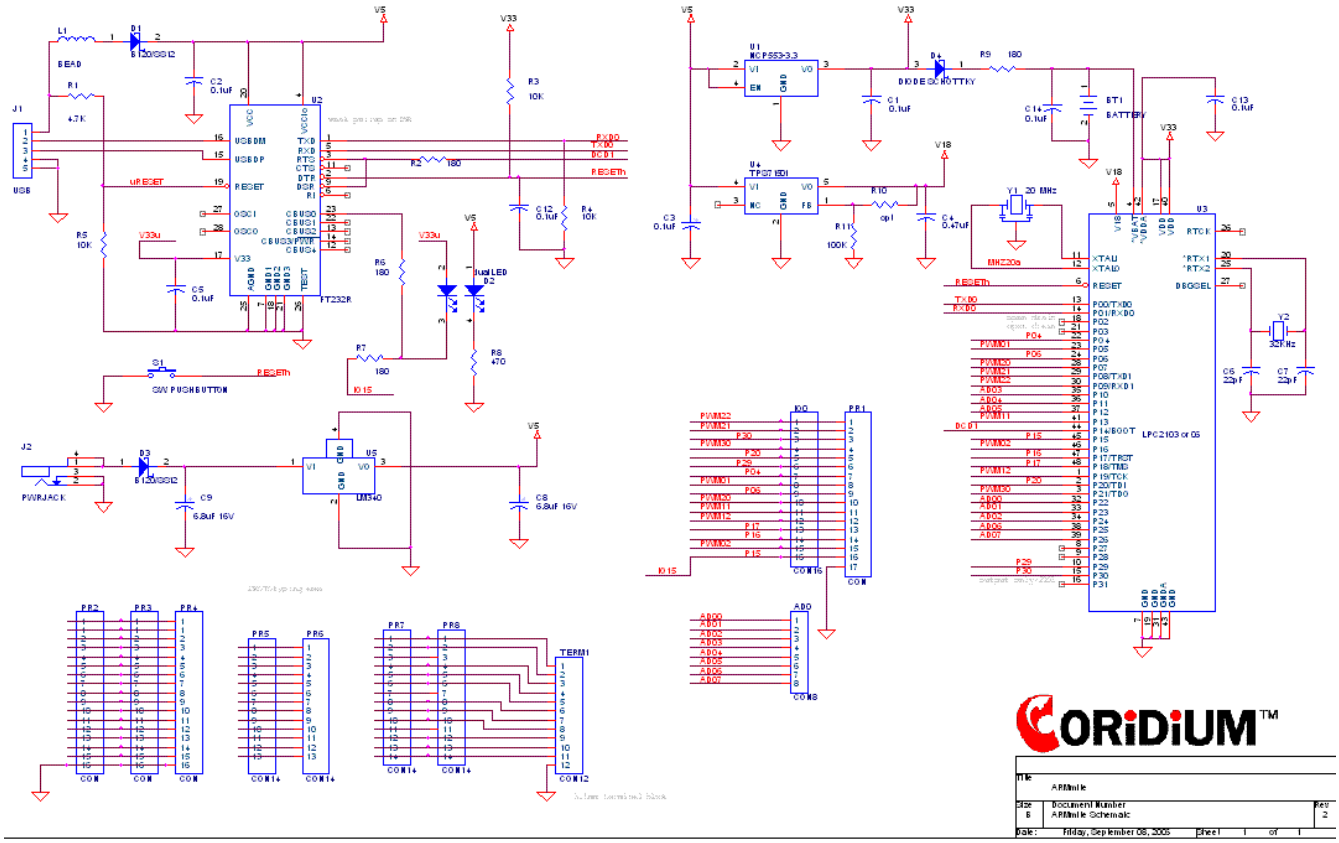

DB style
7-16V
+center
ML2020
BT1
3906
TERM1
RESET
SW1
www.CORIDIUMCORP.com
3.5mm terminal

Prototype Connections



ARMmite Schematic

For a finer image see ARMmite.pdf in your install directory (C:\Program Files\Coridium\Schematics).



ARMexpress Pin Diagram

ARMexpress Schematic

ARMexpress Eval Kit Schematic

Serial Configuration

Though we recommend using TclTerm to talk to the ARMstamp, here are settings for other terminal programs.

Baudrate

115.2 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 ARMstamp, 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 about 1 second. A total of 40K is available to the user for code, constant strings and DATA.

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) where the ARMstamp 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 ARMstamp is ready to be reprogrammed. Or the user can restart the program by typing RUN or using the RUN button in TclTerm.

Program Running Signaling (ver 6.11 and after)

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 TclTerm needs to know when the user code is running. ARMstamp versions starting with 6.11 also support this.

When TclTerm appears to be deaf

There are cases where the USB driver and TclTerm 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 TclTerm and then restart it, and in some cases reboot the system.

TclTerm configuration settings

The configuration of TclTerm is saved in a file TclTerm.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.

USB use

During programming TclTerm 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 ARMstamp 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.

Baudrate

Baudrate will remain at 19.2Kb, unless changed by the user program which can be done with

BAUD(16)=newrate

SEROUT 16,newrate,...

SERIN 16,newrate,...

Output of Data to PC

The ARMbasic program can use PRINT, SEROUT 16,... , or TXD(16)=

Input of Data from PC

An ARMbasic program should use SERIN 16,... or = RXD(16). These routines will time out after 500 msec 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 RXD(16).

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 and while using TclTerm.

Suggested RS232 connection

TTL interfacing

The ARMstamp 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 ARMstamp 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 ARMstamp inputs may be connected directly to a GND pin, but if connecting to a fixed hight level, then it may be connected to a 5V supply line with a 1K or greater resistor. This is the same recommendation for any TTL compatable device. The reason being is that the 5V supply may exceed the 5V at times.

Power

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.

ARMstamp 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.

Once the programming is completed, the ARMstamp may be run without a connection to a PC. In this case an alternate power supply has been provided in the evaluation kit. This supply will generate an un-regulated 6-8V which is connected to pin 24. Onboard the ARMstamp this will be regulated to 3.3V and 1.8V for use by the ARM CPU.

Initial Power on conditions

On power up all pins are tri-stated on the ARMstamp.

Restarting the program

If the user has programmed the ARMstamp, 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 ARMstamp 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 ARMstamp 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 ARMstamp is ready to be reprogrammed. Or the user can restart the program by typing RUN or using the RUN button in TclTerm.

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 ARMstamp 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.

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 ARMstamp 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 or greater on pin 21. Also for future expansion, pin 21 should not be connected for new designs. The maximum voltage that may be applied to either pin 21 or 24 is 16V, but this is not a recommended continuous voltage, as it will cause extra heat to be generated by the ARMstamp 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 ARMstamp 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 ARMstamp 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, PULSIN, and COUNT.

Other operations such as I2CIN, I2COUT, SPIIN, SPIOUT, SHIFTIN, SHIFTOUT, and PWM are "bit-banged" loops that are calibrated to the speed of the CPU.

Interupts

The serial port for control and the timer are the only interupts currently used by the ARMstamp. The service routines for these actions have been minimized so that the user program is only interupted for TBD microseconds.

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 ARMstamp receives serial input it will interput 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 PWM. In normal operation this should not be a problem.

AD timing (ARMmite only)

The 8 analog inputs can do a conversion in 5.5 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.

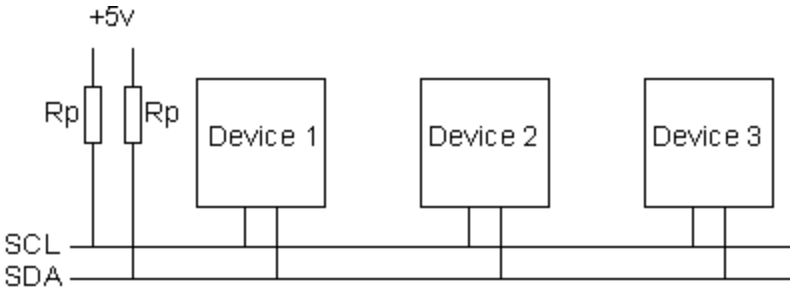
SPI looks at first like a non-standard. However, many programmers that develop embedded systems have a software module somewhere in their past that drives such a bus from a few general-purpose I/O pins, often with the ability to run different clock polarities, select polarities and clock edges for different 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

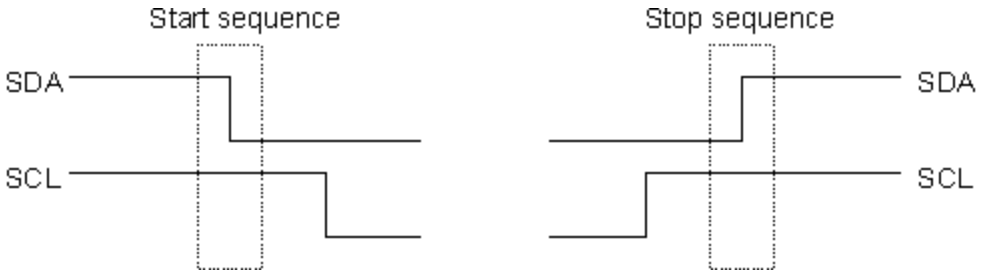
The physical I2C bus This is just two wires, called SCL and SDA. SCL is the clock line. It is used to synchronize all data transfers over the I2C bus. SDA is the data line. The SCL & SDA lines are connected to all devices on the I2C bus. There needs to be a third wire which is just the ground or 0 volts. 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:



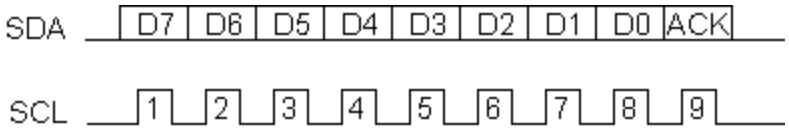
The value of the resistors should be from 1.8K (1800 ohms) to 4.7k (4700 ohms). It depends on the length of the I2C bus, the longer the bus, the smaller value should be used. If the value is too large, the rise time of the signals will be too slow and the bus may not work properly. If the resistors are 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 ARMstamp 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. ARMstamp 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 ARMstamp 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.

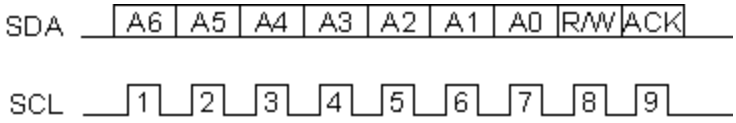


Data is transferred in sequences of 8 bits. The bits are placed on the SDA line starting with the MSB (Most Significant Bit). The SCL line is then pulsed high, then low. Remember that the chip cannot really drive the line high, it simply "lets go" of it and the resistor actually pulls it high. For every 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? ARMstamp 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

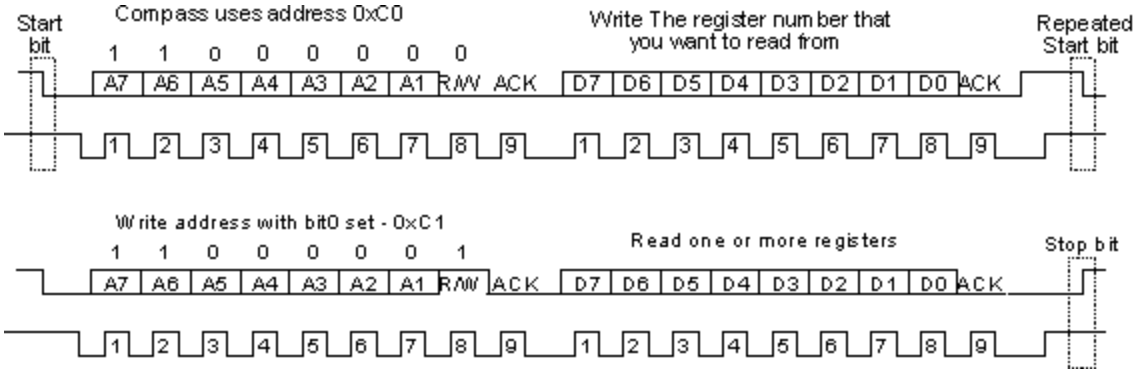
registers because the slave will automatically increment the internal register address after each byte. When the master has finished writing all data to the slave, it sends a stop sequence which completes the transaction. So to write to a slave device:

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)
5. 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:

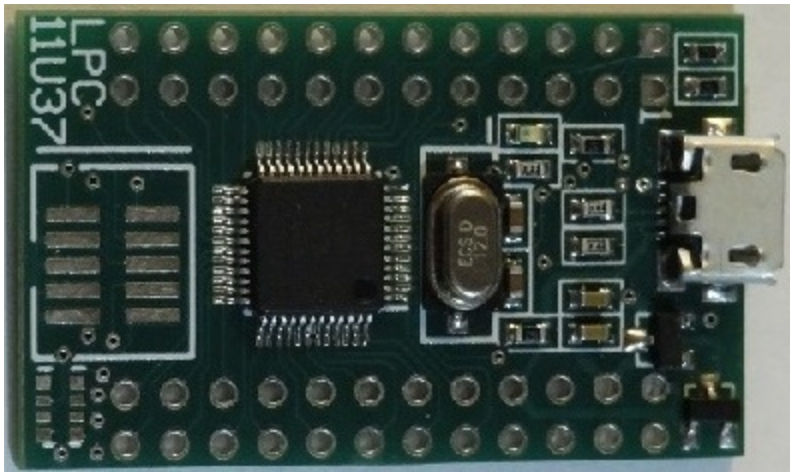


Wait a moment The ARMstamp does not support slaves that use clock stretching. The result is that erroneous data is read from the slave. Beware! Luckily this function is relatively rare these days. **Example Master Code** `DIM A$(10) I2CIN 1,$30,$10, [STR A$ \10] ' read 10 bytes from slave $30 register $10 connected on pins 1,2 I2CIN 5,$40,$20, [X] ' read a single byte from slave $40 register $20 on pins 5,6 FOR I=0 TO 9 A$(I) = $30 + I NEXT I I2COUT 1,$30,$10, [STR A$ \10] ' send 10 bytes toslave $30 register $10 connected on pins 1,2 X=$55 I2COUT 5,$40,$20,[X] ' send a single byte to slave $40 register $20 on pins 5,6 I2COUT 5,$50,$20,[$AA] ' send $AA to slave $50 register $20 on pins 5,6` 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

Miscellaneous



PreProcessor

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 ARMstamp.
Debugging Functions @ CLEAR RUN

> (execute immediately)

. (print now)

@ (dump memory)

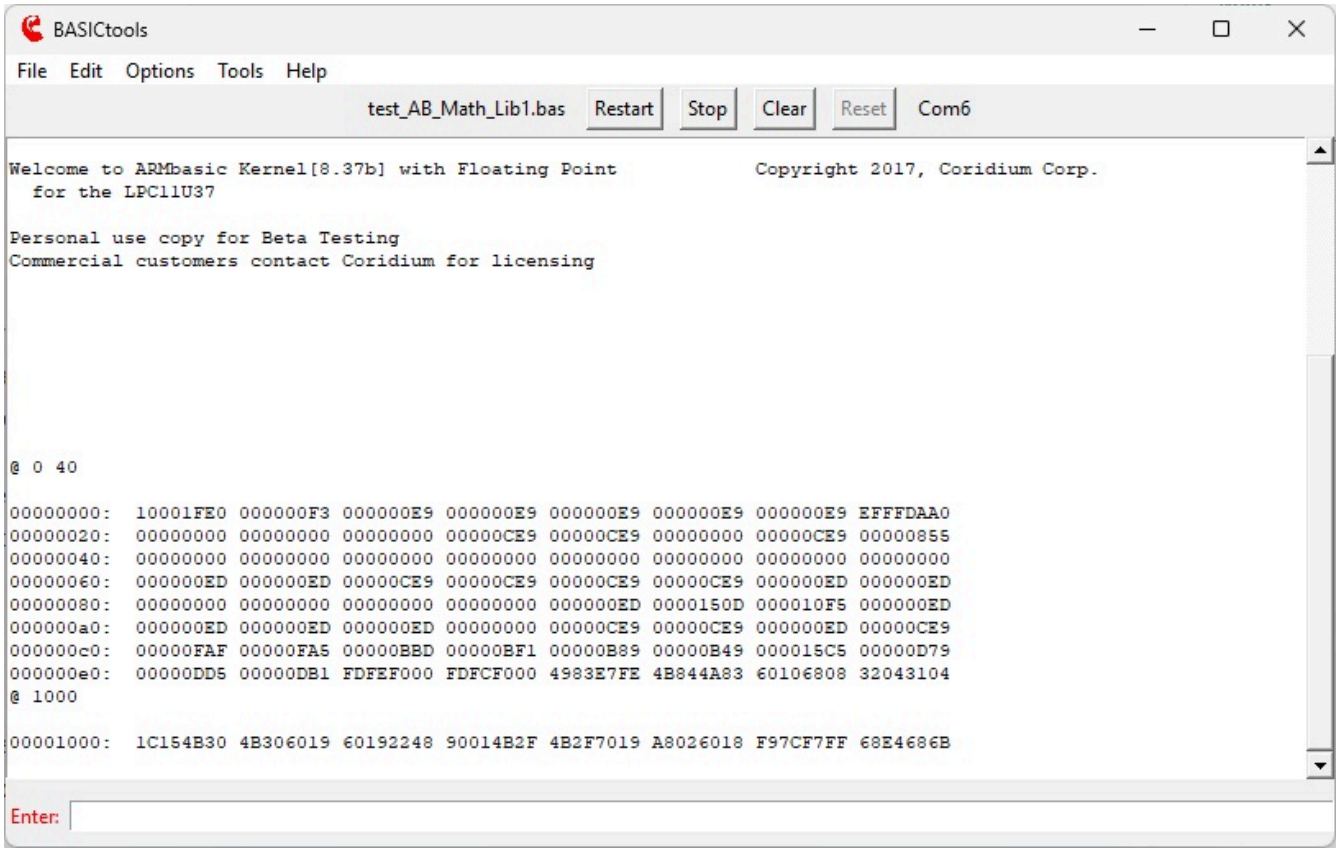
Syntax

`@ address count`

`@ address` -- *displays 8 words*

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.



CLEAR

DEC

HEX

RUN

Tables

ASCII Character Codes

ARMbasic uses the standard "ASCII extended" character set. The compiler uses character values 32 to 126, which correspond to SPACE through TILDE (`~`).

Characters outside this range may have a special meaning and are interpreted by the terminal emulation program controlling the ARMstamp — for example BACKSPACE, TAB, CR, and LF. These characters cause changes in the stream of characters going to or from the ARMstamp module, and 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 ARMstamp module, the module may require a delay when writing code into Flash memory. During these writes, an XOFF character is sent to the PC indicating it should pause sending data. After the block is written (about 0.4 second) an XON is sent to resume communication.

When using `SERIN` or `SEROUT`, there is no special interpretation of characters, so all codes 0 to 255 may be sent without any change.

When a program starts, a **SOH** (`001`) character is sent, and when it finishes an **EOT** (`004`) is sent — BASICtools uses these to know whether the user **ARMbasic** code is running. User code should avoid these character codes when communicating through BASICtools.

Printable characters (32–126)

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
32	0x20	(space)	56	0x38	8	80	0x50	P	104	0x68	h
33	0x21	!	57	0x39	9	81	0x51	Q	105	0x69	i
34	0x22	"	58	0x3A	:	82	0x52	R	106	0x6A	j
35	0x23	#	59	0x3B	;	83	0x53	S	107	0x6B	k
36	0x24	$	60	0x3C	<	84	0x54	T	108	0x6C	l

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
37	0x25	%	61	0x3D	=	85	0x55	U	109	0x6D	m
38	0x26	&	62	0x3E	>	86	0x56	V	110	0x6E	n
39	0x27	'	63	0x3F	?	87	0x57	W	111	0x6F	o
40	0x28	(	64	0x40	@	88	0x58	X	112	0x70	p
41	0x29	)	65	0x41	A	89	0x59	Y	113	0x71	q
42	0x2A	*	66	0x42	B	90	0x5A	Z	114	0x72	r
43	0x2B	+	67	0x43	C	91	0x5B	[	115	0x73	s
44	0x2C	,	68	0x44	D	92	0x5C	\	116	0x74	t
45	0x2D	-	69	0x45	E	93	0x5D	]	117	0x75	u
46	0x2E	.	70	0x46	F	94	0x5E	^	118	0x76	v
47	0x2F	/	71	0x47	G	95	0x5F	_	119	0x77	w
48	0x30	0	72	0x48	H	96	0x60	`	120	0x78	x
49	0x31	1	73	0x49	I	97	0x61	a	121	0x79	y
50	0x32	2	74	0x4A	J	98	0x62	b	122	0x7A	z
51	0x33	3	75	0x4B	K	99	0x63	c	123	0x7B	{
52	0x34	4	76	0x4C	L	100	0x64	d	124	0x7C	|
53	0x35	5	77	0x4D	M	101	0x65	e	125	0x7D	}
54	0x36	6	78	0x4E	N	102	0x66	f	126	0x7E	~
55	0x37	7	79	0x4F	O	103	0x67	g			

Common control characters (0–31, 127)

Dec	Hex	Code	Meaning
1	0x01	SOH	Start of program (sent by ARMbasic)
4	0x04	EOT	End of program (sent by ARMbasic)
8	0x08	BS	Backspace
9	0x09	TAB	Tab
10	0x0A	LF	Line feed
13	0x0D	CR	Carriage return
17	0x11	XON	Resume transmission
19	0x13	XOFF	Pause transmission

Dec	Hex	Code	Meaning
27	0x1B	ESC	Escape
127	0x7F	DEL	Delete

Bitwise Operators

ARMbasic operators that work bit-by-bit on integer operands. The truth tables below describe each operator one bit at a time. When applied to multi-bit integers, the operation is performed independently on each bit position.

AND

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

OR

A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

XOR

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

NOT

A	NOT A
0	1
1	0

See also

- AND
- OR
- XOR
- NOT

Operator Precedence

Variable Types

ARMbasic supports a small, deliberately simple set of types.

Type	Size	Range / contents
INTEGER	32 bits	Signed whole number, −2,147,483,648 to 2,147,483,647 (default type)
SINGLE	32 bits	IEEE-754 single-precision floating point
BYTE	8 bits	Unsigned 0 to 255
STRING	declared	Array of bytes terminated with a 0; size set in the DIM

Defaults

- Scalar variables are INTEGER by default. They do not need to be declared with DIM — they spring into existence on first use, with an initial value of 0.
- Strings and arrays must be declared with DIM, which sets their length.
- The legacy `$` suffix on a variable name (e.g. `A$`) declares a STRING. The suffix is accepted for backward compatibility but is not required.

Example

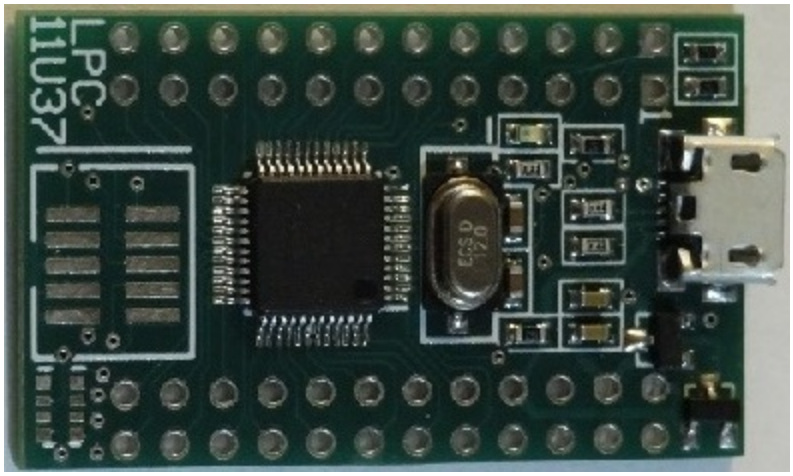
```
DIM Name$(20)            ' STRING, up to 20 chars
DIM Pi AS SINGLE         ' floating point
DIM Counts(8) AS BYTE    ' array of 8 bytes
DIM Total                ' implicit INTEGER

Name$ = "World"
Pi = 3.14159
Counts(0) = 255
Total = 0
```

See also

- DIM
- CONST
- Variables
- Arrays
- Strings

Support



How to contact the developers

You should contact the **ARMbasic** developers through Coridium Corp.

- Website: coridium.us
- Tech support email: techsupport@coridium.us
- Community: the **BASIC** group on Facebook is monitored by tech support

Coridium has done custom ports of ARMbasic to other platforms — contact tech support to discuss.

See also

- Reporting a bug

How to report a bug

Before reporting a bug, try to make sure it's actually a bug in **ARMbasic** and not a bug in your own code. Write a small test that reproduces the problem and read any relevant documentation. If you show people that you have tried to solve your own problem rather than immediately running for help, you'll be more likely to find people willing to help.

Be as specific as you can — "The PWM runtime library function fails when it is called with a value of 1234" is much better than "It crashes."

The first place to ask is the **BASIC** group on Facebook, which Coridium tech support monitors.

If you have isolated a compiler bug, with steps to reproduce and a small piece of sample code, file a bug report by emailing techsupport@coridium.us.

Please do **not** file general "it doesn't work!" reports in the public group — only isolated, reproducible bugs.

Contributors

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 ARMstamp system.

Freewrap is used to generate TclTerm from a Tcl/Tk script (source available from Coridium website).

The PBASIC translator uses GNU sed v3.02.80 and MinGW cpp.

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.

This documentation site was rebuilt from the original CHM help file in 2026 with help from Anthropic Claude and is published with Docusaurus, Meta's open-source documentation framework.