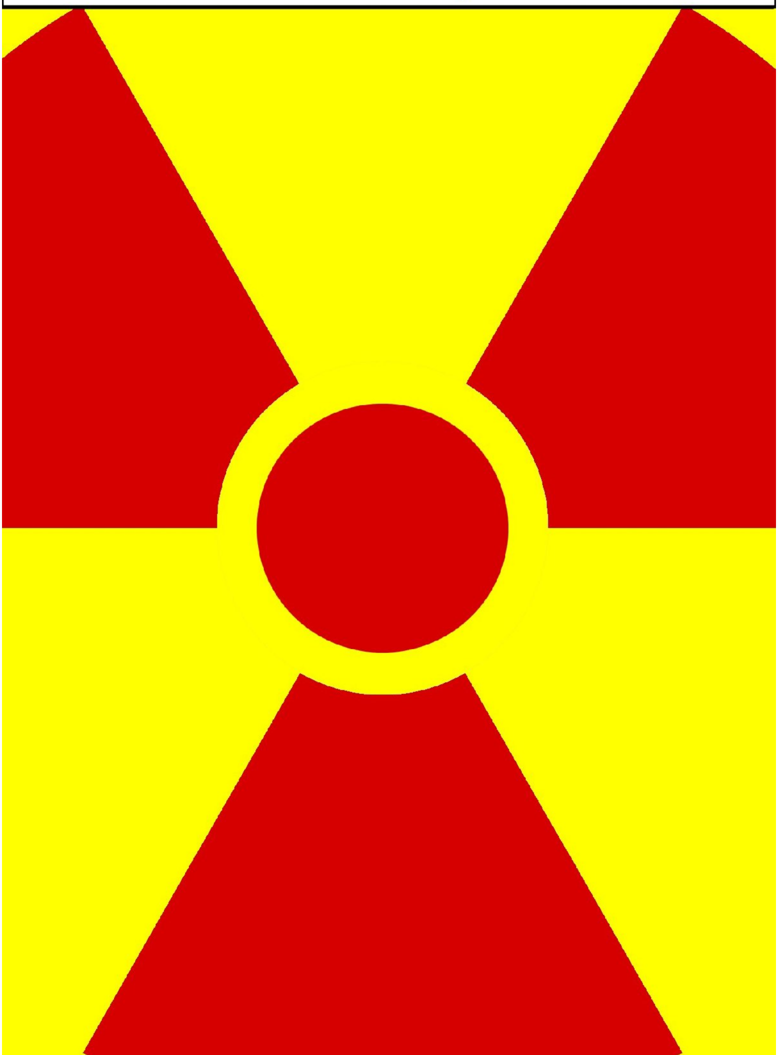
BASIC MICRO
TECHNOLOGY AT WORK

BasicATOM Pro Syntax Manual



Unleash The Power Of The Basic Atom Pro

Version 8.0.0.0

Warranty

Basic Micro warranties its products against defects in material and workmanship for a period of 90 days. If a defect is discovered, Basic Micro will at its discretion repair, replace, or refund the purchase price of the product in question. Contact us at support@basicmicro.com. No returns will be accepted without the proper authorization.

Copyrights and Trademarks

Disclaimer

Basic Micro cannot be held responsible for any incidental, or consequential damages resulting from use of products manufactured or sold by Basic Micro or its distributors. No products from Basic Micro should be used in any medical devices and/or medical situations. No product should be used in a life support situation.

Contacts

Email: sales@basicmicro.com
Tech support: support@basicmicro.com
Web: http://www.basicmicro.com

Discussion List

A web based discussion board is maintained at http://www.basicmicro.com

Updates

In our continuing effort to provide the best and most innovative products, software updates are made available by contacting us at support@basicmicro.com or via our web site.

Table of Contents

Table of Figures

SECTION 1:
Learning about the Atom Pro

Introduction

Thank you for purchasing the BasicATOM Pro; an advanced microcontroller. This manual will help you to set up, program, and test your BasicATOM Pro. Some procedures described assume the use of a suitable development kit, available from Basic Micro.

What is a BasicATOM Pro?

The BasicATOM Pro is a self contained microcontroller; essentially a microcomputer with memory and support circuitry in a single plug-in package. The Atom Pro's built-in command language is programmed using a convenient BASIC-like compiler which runs on a PC. This special version of BASIC is very powerful and easy to use, and runs from an Integrated Development Environment (IDE) offering programming and debugging tools.

This Manual

This manual is designed for both first time and experienced microcontroller users. It describes setup and programming of the BasicATOM Pro. Hardware details and schematics are provided separately in the form of data sheets.

Models covered by this manual are BasicATOM Pro 24-M, 28-M and BasicATOM Pro-X modules.

The Integrated Development Environment (IDE) is described in overview form, with further details available from the on-screen help provided with the program.

For more information about a particular device refer to its Data Sheet. Printed data sheets are included with each product, and all data sheets are available from the download section of the Basic Micro web site at http://www.basicmicro.com.

> Data sheets for other products mentioned in this manual are available from the manufacturers, and can usually be found easily online using a search engine such as Google.

We continually update and improve this manual. All updates will be made available for download from our web site.

On-line Discussion Forums

We maintain discussion forums at http://www.basicmicro.com in order to facilitate information exchange among users. The discussion forums are free and will help you to find information and assistance quickly.

Information Resources

In addition to other resources mentioned in this manual, you can also find useful information by using the search feature of the online discussion forums at the Basic Micro web site.

Updates

Atom Pro software updates will be available to new and current customers. We recommend joining the discussion forums at http://www.basicmicro.com where update announcements will be posted.

Technical Support

Technical support is provided via email. When technical support is required please send email to support@basicmicro.com. In order to assure a proper response. Please include a copy of the program you are having problems with, the hardware you are using, ATOM Pro revision number, prototyping board and so on. By including this information with your email, you can help us to answer your questions quickly.

The BasicATOM Pro

Overview

The BasicATOM Pro is a complete microcontroller with a wide range of programmable functions. User-programming is done with a BASIC-like compiler running on a PC, and the resulting object code is downloaded to the ATOM Pro.

For development and testing, Basic Micro provides development boards which have computer and power connections, as well as a breadboard area for circuit design (see page 5). Once your circuit is finalized, you can use a Basic Micro prototyping board (see page 6) or design your own board to accommodate the BasicATOM Pro module. One-time projects can be left on the prototyping board permanently.

Software

The ATOM Pro Language

The ATOM Pro language is a version of BASIC designed for control applications. It's based on Basic Micro's Mbasic, with added functions to support the BasicATOM Pro's hardware capabilities.

How the ATOM Pro Supports Software

The BasicATOM Pro includes the following software support facilities:

- a microprocessor to run your program
- a program loader to install programs developed using the Basic compiler
- 32K or 56K* bytes of flash program ROM, for storing programs and constants. Flash can be erased and rewritten many times.
- 2K or 4K* of RAM for calculations and variable storage (minus system overhead).

  * see *Characteristics* on page 5 for specifications.

The runtime environment[1] is not permanently stored in the CPU. This gives greater flexibility in that commands and functionality can be added and modified without changing hardware. The runtime environment is automatically generated by the compiler, and can vary in size between as little as 250 bytes for a very simple program to a maximum of around 3000 bytes of program memory. As more functions are used in a program, the runtime environment will automatically expand to include support for these functions. The runtime environment is optimized and only includes support for functions actually used in your program. The support code for a function used in a program is only included once, even if the function is used many times.

Hardware

The BasicATOM Pro is an integrated microcontroller based on a Hitachi[2] H8/3664F processor chip. The Atom Pro Plus is based on the H8/3687F.

> Users are strongly advised to obtain a copy of the H8/3664F or H8/3687F hardware manuals, available at http://www.renesas.com which give important details for the chips used to build the ATOM Pro module.

The BasicATOM Pro modules add support circuitry (RS-232, voltage regulation, oscillator, etc.) and a program loader to the H8/3664F or H8/3687F chip. For more specific information, refer to the data sheet supplied with each BasicATOM Pro module (data sheets are also available on our web site, in the Download section, as PDF documents).

The BasicATOM Pro is programmed by means of a serial data stream at 115 kb/s. This data format is supported by all modern PCs (since 1996 or before). The BasicATOM Pro and AtomPro development boards provide an RS-232 connector that can be connected by a standard cable to your PC's serial port.

---

[1] The runtime environment is sometimes known as a “software brain”, runtime library, or runtime module.

[2] Hitachi's semiconductor division is now known as Renesas

Different models of BasicATOM Pro

The BasicATOM Pro models differ as follows;

Atom Pro 24 pin	16 I/O pins (P0 to P15). Based on Hitachi H8/3664F
Atom Pro 28 pin	20 I/O pins (P0 to P19). Based on Hitachi H8/3664F
Atom Pro Plus 40 pin	32 I/O pins (P0 to P31). Based on Hitachi H8/3687F. Offers additional features as described below.

Characteristics

	Atom Pro	Atom Pro Plus
Clock speed	16 MHz	20 MHz
RAM	2 kB	4 kB
EEPROM	none	4 kB
Flash (program memory)	32 kB	56 kB
UARTS	1	2
HPWM pins	3	6

Available Development Boards

Basic Micro supplies the following development boards[3]. Data sheets are available on our website (download area) for review.

Development boards include an experimenter "breadboard" area for easy project development.

Board	**Description**
Atom Universal Development Board	For BasicATOM Pro 24-M, 28 and Atom Pro Plus 40 pin modules. Can also be used with BasicATOM Pro-X (no longer available) to provide a breadboard development area. *Note: this board also works with BasicATOM 24, 28 and 40 pin modules.*

[3] New products are frequently added. Please visit our website for the latest list of available development boards.

Available Prototyping and Enclosure Boards

Basic Micro supplies the following prototyping and enclosure boards. Data sheets are available on our website (download area) for review.

Prototyping and enclosure boards include a circuit area with plated through holes for permanent projects.

Board	**Description**
BasicATOM Pro Enclosure Board	For BasicATOM Pro 24-M module. Fits in plastic case for finished projects. *Also works with BasicATOM 24-M.*
BasicATOM Pro LCD Enclosure Board	For BasicATOM Pro 24-M modules. Fits in plastic case. Standard LCD connector provided. *Also works with BasicATOM 24-M.*
BasicATOM Pro Mini Prototyping Board	For BasicATOM Pro 24, and Pro 28 modules. Small size, low cost for simpler applications.

Getting Started

This section explains in simple terms how to get started using your BasicATOM Pro. While some hardware basics are explained and a simple example given, general hardware design for BasicATOM Pro controlled devices is beyond the scope of this manual.

What You Will Need

Project development is normally done using hardware prototypes. Basic Micro provides Development Boards for this purpose. If you're using your own development environment suitable power and RS-232 connections will be needed.

You will need:

- An Atom Pro 24 or 28 pin, or Atom Pro Plus 40 pin module.
- An Atom or Atom Pro development board (or your own suitable hardware development environment).
- A suitable power source.
- An RS-232 connector and cable to connect to a PC serial port.
- BasicATOM Pro software (Integrated Development Environment)
- A PC running Windows 9x, 2000, NT4 or XP. A CD drive is required to install the software included with the development kit; software may also be downloaded from our website.

We recommend a Pentium 266 or faster: operation may be quite slow with lesser computers, though they should work.

All items except the PC are supplied with BasicATOM Pro development kits.

Follow These Steps

Designing a project is as simple as following these steps.

1. Set up the BasicATOM Pro software (IDE).
2. Build your circuit on a development board (these have "breadboards" to allow easy wiring and frequent changes).
3. Write the software to control your circuit and download it to the BasicATOM Pro.

4. Debug and revise your hardware and software.

What you do next depends on what you need. If you're building:

- a one-time project, and won't need the development board for future projects, you can leave the circuit on the breadboard. *Note that the long-term stability of breadboard projects may not be as good as those with soldered connections.*
- a one-time or limited production project, but want to re-use the breadboard for other projects, or want smaller size and the permanence of soldered connections, transfer your project to a BasicATOM Prototyping board.
- a project for production, or just want to do your own board, design a suitable board incorporating your circuit.

Note: For production lots use a development or prototyping board to program BasicATOM Pro modules or interpreter chips, then transfer the programmed modules or chips to your production boards. You don't need the circuit for programming on the production board unless you want to allow in-field reprogramming.

Software Setup

Software setup is easy and follows standard Windows practice.

1. With Windows running, insert the CD into the CD drive.[4]
2. If the installation program doesn't automatically start, open an Explorer or My Computer window, navigate to the CD, and double click the "setup" icon or the file "setup.exe".
3. Installation from this point is automatic. Once done, there will be an ATOM icon on your desktop.
4. Double click the ATOM icon to open the Integrated Development Environment (IDE).
5. Go to Tools : System Setup and choose the serial port to be used for connecting the development board (see Figure 1). Click Find AtomPro if it is connected but you don't know which port to choose. (The other tabs aren't needed at this point.) Close the program.

[4] Current software is also available for download at our web site.

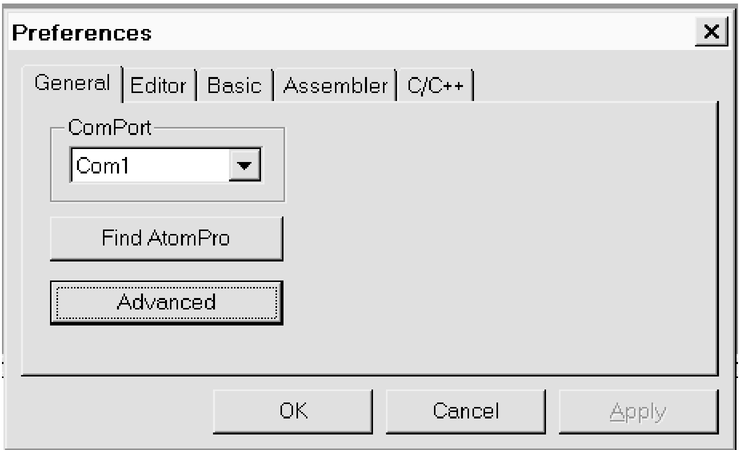


Figure 1 - Setting the serial port

That's it! You're ready to start programming and working with your BasicATOM Pro as soon as it's connected and ready.

Hardware Setup

Note: Some instructions in this section are provided for the convenience of Pro-X owners. The Pro-X is now discontinued.

For circuit and program development you will probably want to use a Universal Development Board which provides power, RS-232 programming connector, and a hardware breadboard area.

The ATOM Pro-X has built-in power and RS-232 connectors, however it may be used with the Universal Development Board to provide a experimenter's "breadboard" area. We will use it this way when describing our sample circuits.

Setup is easy (see Figure 2, Figure 3 and Figure 4):

1. Make sure you're in a static free working environment. Microprocessors of all types are sensitive to static electricity and can be damaged if not properly handled.
2. **For ATOM Pro 24 and 28 pin and Atom Pro Plus 40 pin modules:** Plug the module into the development board socket, making sure to align it at the end marked Pin 1 (see Figure 3). Be sure not to bend any of the pins.

> Important Note: If the pins seem too widely spaced to fit into the socket, hold the module by the ends, and gently “roll” the leads against a tabletop. Do this for each side until the pins slide easily into the socket.

__For ATOM Pro-X:__ Plug the module into the development board inline connectors as shown in Figure 4.

3. **For ATOM Pro 24 and 28 pin and Atom Pro Plus 40 pin modules:** Connect the provided RS-232 cable (9 pin connectors) to the 9 pin socket on the development board.

   __For ATOM Pro-X:__ Connect the provided RS-232 cable (9 pin connectors) to the 9 pin socket on the Pro-X module. Leave the socket on the development board unconnected.

4. Plug the other end of the cable into an available serial port on your PC.[5]

5. When you’re ready to begin programming and experimentation, plug the 9VDC power adapter into the socket on the development board *(you may use either power connector with the Pro-X)*.

> Important note: Never make hardware changes in the prototype area or plug in or unplug the BasicATOM Pro module with the power connected!

If you’re using an ATOM Pro or Pro Plus module without a development board, you’ll need to provide your own RS-232 and power connections. The BasicATOM Pro has a voltage regulator and an RS-232 level converter built-in. See the data sheet for connection details.

Setup sequence for 24, 28 and 40 pin modules

The diagram on the next page shows the setup sequence if you're using an ATOM Pro 24, 28 or Pro Plus 40 pin module with the Universal Development Board.

[5] If the PC uses a 25 pin connector, use a 9 to 25 pin adapter, available at most computer stores. PCs without serial ports can be connected using a USB to RS-232 adapter.

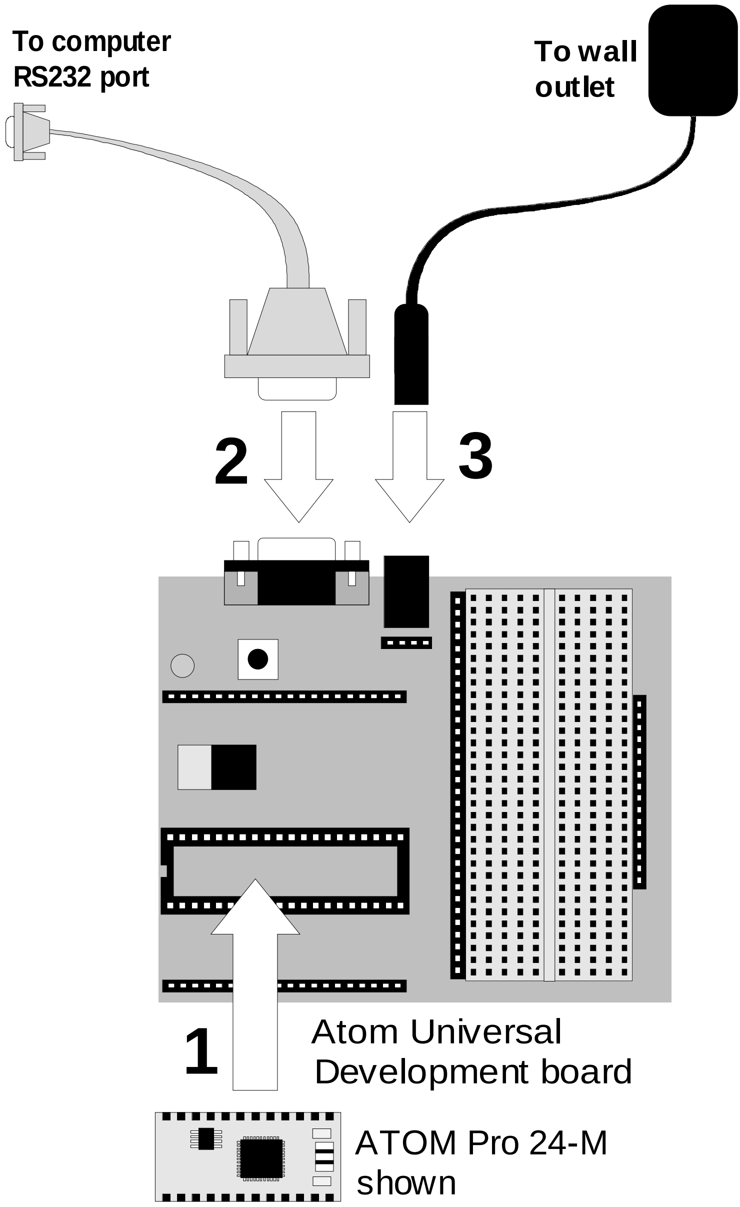


Figure 2 - Hardware Setup Sequence

Make sure to orient the module correctly before plugging it in to the development or prototyping board. Look for the notch at one end of the socket, then align Pin 1 of the module with Pin 1 of the socket. Pin 1 on the module can be found by referring to the diagram below.

Note: If the module has fewer pins than the socket, it must be plugged in at the Pin 1 end of the socket.

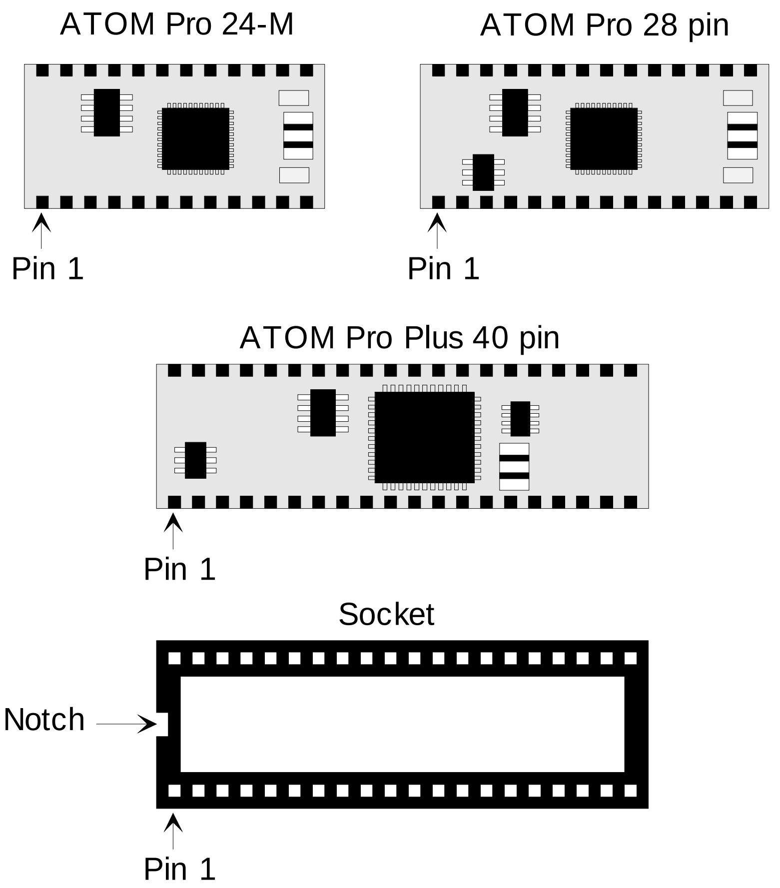


Figure 3 - Orienting the Module

Setup Sequence for Pro-X

This section is provided for the convenience of Pro-X owners. The Pro-X is now discontinued.

The diagram below shows the setup sequence if you're using an ATOM Pro-X with the Universal Development Board to allow breadboarding.

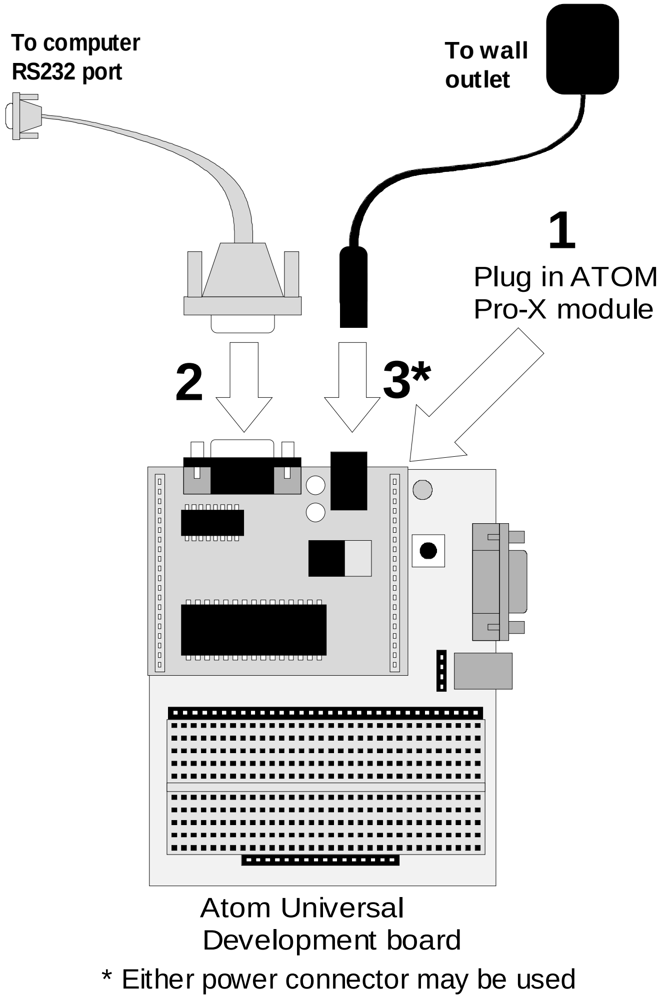


Figure 4 - Hardware Setup Sequence for Pro-X

Building Your Prototype or Project

While it's beyond the scope of this manual to discuss hardware design in detail, here are a few pointers to get you started.

The best way to design your hardware prototype or project is to use a Basic Micro development board (see our web site and the list on page 5 for available models). Development boards include a breadboard area for easy experimentation and circuit development.

Figure 5 shows a typical breadboard (Atom Universal Development board shown). Connections for microcontroller I/O, Vss and Vdd are provided. The board is marked to indicate voltages and pin numbers.

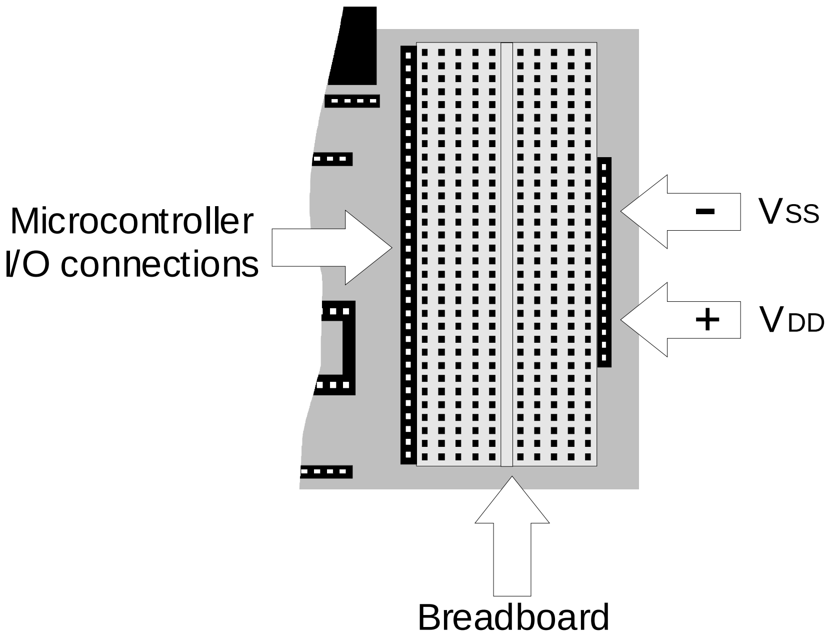


Figure 5 - Typical Prototyping Area

Component leads can be inserted directly into the holes in the breadboard. Jumpers can be made with #22 AWG or #24 AWG wire.

Sockets in the breadboard are grouped together as shown below in Figure 6. This makes it easy to connect components together or to "fan out" voltages or I/O pins to multiple connections.

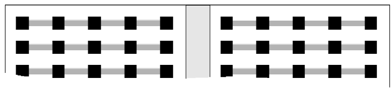


Figure 6 - Breadboard internal connections

The breadboard uses standard 0.10" spacing so small integrated circuits or other DIP (dual inline package) or SIP (single inline package) components can be inserted directly. Make sure that DIP components are inserted so as to "bridge" the central area, as shown in Figure 7.

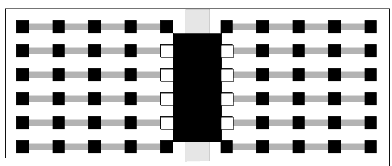


Figure 7 - IC orientation on Breadboard

Once you've designed your experimental circuit or prototype, you're ready to program the BasicATOM Pro and try it out!

Running the IDE Software

This section gives a brief overview of the IDE to help you get started. More complete information, including how to use the debugger, is included in the online help file.

Double click the Atom icon (on your desktop or in your Start menu), then click on File **|** New to open the CreateFile dialog. It defaults to bas type files but can be set for C/C++,Asm or Txt files as well from the dropdown list. After choosing the directory to create the file in, type in the filename you want to use. We chose to use a Basic file with "test.bas" for our filename. Your screen should now look like this:

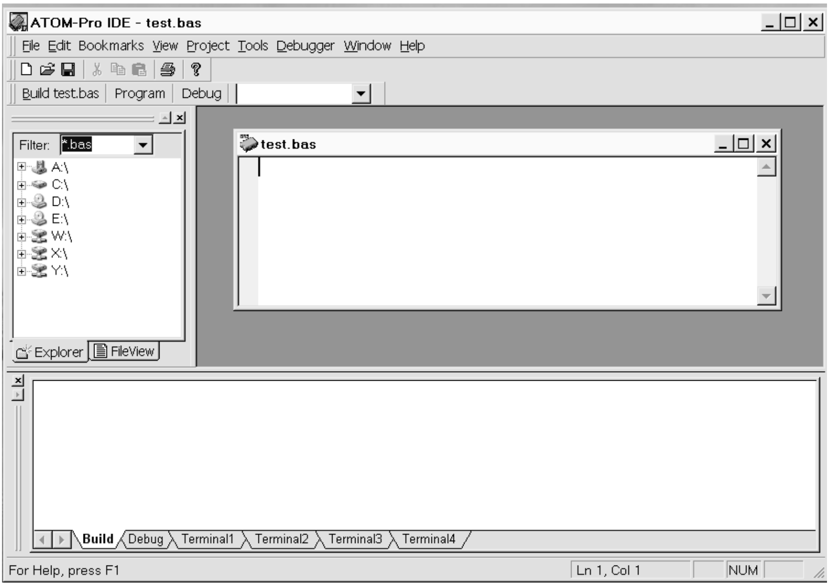


Figure 8 - IDE screen

The window is divided into three main areas: The File Explorer Window (at top left) displays directories and files.

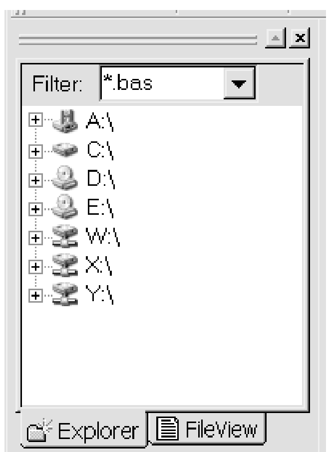


Figure 9 - IDE Workspace

You can get more window room for your program by turning off the File Explorer Window: either click the X at its top right corner, or from the View menu choose Toolbars, and uncheck the “Workspace” toolbar.

The Build area (at bottom) shows error messages and compile time messages. Turn it off by clicking the X in its top right corner, or via the View | Toolbars menu (“Results” toolbar). *Once you’re done writing your program, you should turn it back on again before compiling (so you can see messages).*

You’ll notice that the programming area isn’t maximized; just click the maximize button (on the “test.bas” bar) and your window will look like this:

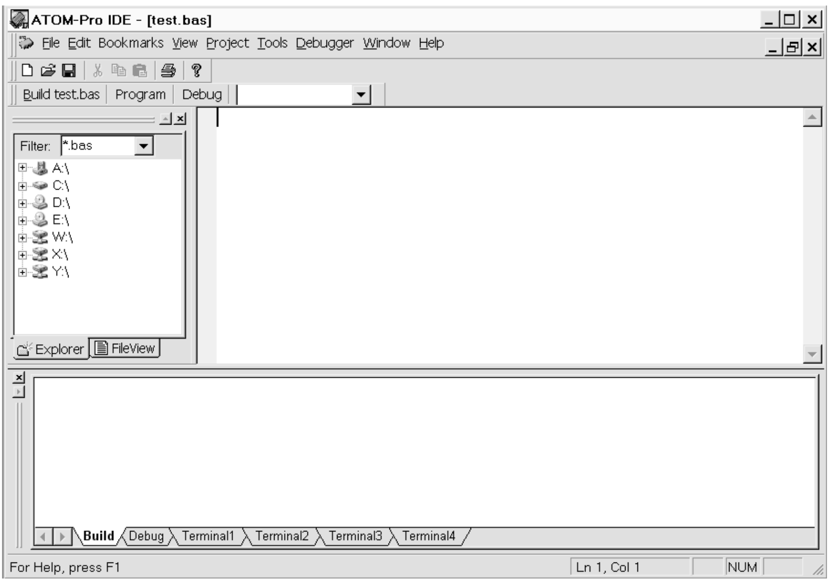


Figure 10 - IDE Screen with program space maximized

Once you’ve typed in your program, you can test it for errors by clicking the Compile button. This will compile your program, but will not write the output to the BasicATOM Pro.

Alternately, you can simply click the Program button, which will compile your program and send it to the BasicATOM Pro. The program will start running on the BasicATOM Pro immediately. Now let’s use a concrete example to help you figure all this out.

Let's Try it Out

To help you get started on your project, we'll start by setting up a simple circuit or two, writing programs to operate them, and testing to see that they work. Once you've been through the procedure, you'll be all set to work on your own.

Here's what you'll learn:

1. How to set up a simple circuit on a breadboard.
2. How to write a simple program to control the circuit.
3. How to install and run the program on the Atom Pro.
4. A few troubleshooting pointers.

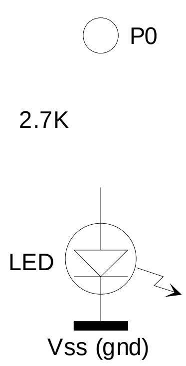
Once you're done, we'll make a slightly more complex circuit, with a few additional programming details, and after that you're on your own.

Your First ATOM Pro Project

We'll start with a simple project to flash a LED (light emitting diode). Here's everything you'll need:

- A BasicATOM Pro.
- A development board (we used the Atom Universal Development Board).
- Power supply, cables, etc. supplied with the development board.
- A PC with the BasicATOM Pro software installed.
- A red LED with wire leads.
- A 2.7 kΩ ¼ W resistor. (Other values in the range of 1 kΩ to 4.7 kΩ should work).
- Some #22 AWG or #24 AWG solid insulated wire for jumpers.

The circuit is very simple, just a resistor and the LED in series. Note that the anode end of the LED (connected to the resistor) may be marked by a longer lead or a flat side on circular LEDs.



> Before going any further, make sure you've followed all the steps for *Getting Started*, beginning on page 7.

You can wire this circuit on the breadboard area of the development board (see Figure 11).

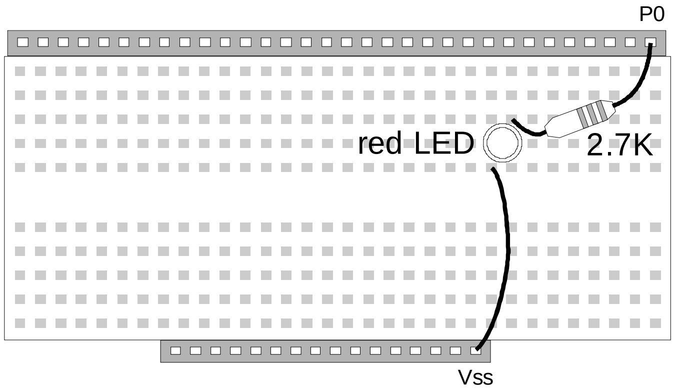


Figure 11 - Blinker circuit on breadboard

Don’t worry too much about the orientation of the LED at this point; it if doesn’t blink after programming, try reversing it.

Writing the Program

By this time you should have the BasicATOM Pro IDE software installed and running on your computer. Next:

1. Plug in the power for the development board.
2. Using the IDE, click on File | New, and choose a Basic file.
3. Type in the following short program;

```
Main
  high P0
  pause 200
  low P0
  pause 200
goto main
End
```

   Use the TAB key to indent lines (this is only needed to make the program easier to read; the compiler doesn’t care).

4. Click on the Program button on the IDE. The program should compile, and be downloaded to the BasicATOM Pro without errors.
5. Watch the LED: it should be flashing 2.5 times per second.

Troubleshooting

If the IDE shows errors, recheck that you've typed the program correctly; it's not likely that there would be some obscure, hard to find error in such a simple program.

If the LED doesn't flash, it is probably plugged in "backwards" Unplug it and plug it in with the leads reversed. Still doesn't flash? Check the voltage on both resistor leads: it should alternate between 5V and 0V on the end connected to P0, and between about 1.2V and 0V on the end connected to the LED.

Program Notes

Let's take another look at that program and add some comments.

```
Main                ;Start of program
  high P0           ;Set P0 to "high" (5V)
  pause 200         ;Wait for 200 ms
  low P0            ;Set P0 to "low" (0V)
  pause 200         ;Wait another 200 ms
goto main           ;Do it again, forever
End
```

"Main" is a label; in this case it's at the beginning of the program. We need it so that the "goto" can find its way back to restart the program, making operation continuous.

"End" is not a label, it's an instruction to the compiler, telling it that the program code is now finished.

Permanency

Once you've programmed the BasicATOM Pro, the program remains permanently in memory until you overwrite it with another program. Try this:

1. Unplug the power from the development board.
2. Disconnect the RS-232 programming cable.
3. Reconnect the power to the development board.

Programs in memory start automatically as soon as the power is applied and your LED is flashing again; just as it was before. Note that you can restart a "stuck" program by pressing the RESET button on the development board.

Making a Traffic Light

As a second project we'll wire up a miniature "traffic light". The idea is to show some slightly more complex programming techniques, as well as more sophisticated use of the breadboard.

The traffic light uses three LEDs, one red, one yellow and one green. We'll set it up to follow this sequence:

- A 10 second red light, followed by
- A 10 second green light, followed by
- A flashing green "priority" light, and finally
- A 3 second yellow (amber) light

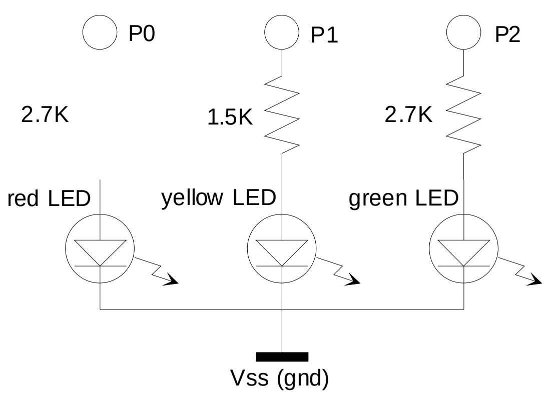
The cycle repeats forever until you turn off the power.

You'll need the following parts *in addition to* the ones from our first project:

- A yellow LED
- A green LED
- A 2.7 kΩ ¼ watt resistor
- A 1.5 kΩ ¼ watt resistor

(We're using a lower value resistor to make up for the yellow LED's reduced efficiency, which would otherwise make it too dim.)

The circuit is really just the first project repeated 3 times:



Wire it up on the breadboard as shown in Figure 12.

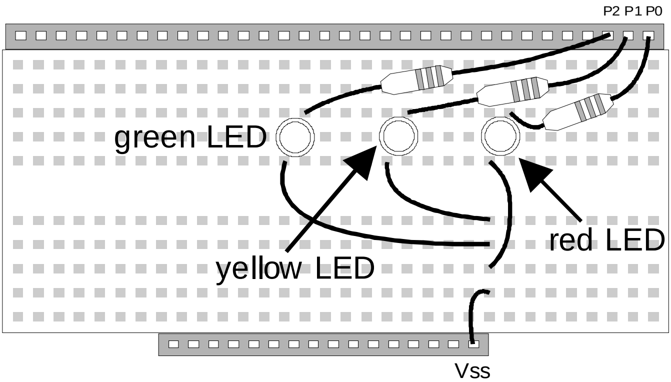


Figure 12 - Traffic light

If you're using the Atom Pro development board, the P0, P1 and P2 connections are available at the edge of the breadboard area.

The Traffic Light Program

Open a new basic file in the IDE and type in the following program:

```
main
   counter var word       ;must define variables
   red con P0             ;red LED on P0
   yellow con P1          ;yellow LED on P1
   green con P2           ;green LED on P2
   low red                ;turn all the LEDs off
   low yellow
   low green
loop                      ;main program loop
   high red               ;turn on red LED
   pause 10000            ;wait 10 seconds
   low red                ;turn off red LED
   high green             ;turn on green LED
   pause 10000            ;wait 10 seconds
   low green              ;turn off green LED
   for counter=1 to 10    ;This loop flashes the
      high green          ;green LED 10 times
      pause 300
      low green
      pause 300
   next
   high yellow            ;turn on yellow LED
   pause 3000             ;wait 3 seconds
```

```
   low yellow          ;turn off yellow LED
goto loop              ;start over again
end
```

Once you're done, make sure the BasicATOM Pro is connected to the computer, and click the Program button. Your program should compile and download without errors, and your LEDs should light in the sequence described above. Compile time errors are probably a result of typing mistakes; the printed program above is taken directly from a working model without retyping.

The IDE window should look much like this once you're done:

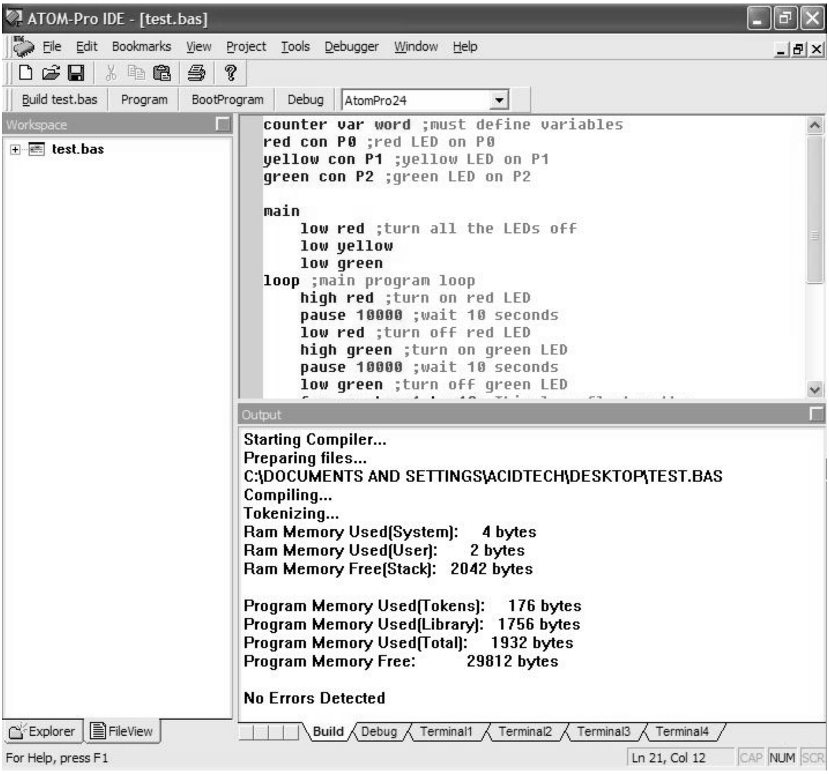


Figure 13 - IDE Screen while compiling Traffic Light program

Program Notes

This program has a couple of additional features that are worth discussing.

The section following Main defines variables and constants. Unlike many basics, all variables used in Atom Basic must be defined before they're used. So a simple "for x = 1 to 10" won't work unless you've defined "x" at the beginning of your program.

Here we've defined the variable "counter" as a "word" variable (i.e. 16 bits) in the line: `counter var word.`

We've also defined some constants. For convenience, instead of having to remember that P0 is red, P1 yellow, etc. we've defined the constants "red", "yellow", and "green" (e.g. `red con P0`).

Definitions are followed by a label "loop". This provides a re-entry point that does not set the lights to OFF each time the program cycles.

Understanding the Build Window

The bottom pane in Figure 13 shows the results of compiling your program. In our example, scrolling the Build Window will show the following lines:

```
Starting Compiler...
Preparing Files...
C:\MY DOCUMENTS\TRAFFIC.BAS
Compiling...
Tokenizing...
Ram Memory Bytes Used(System): 4
Ram Memory Bytes Used(User): 2
Ram Memory Bytes Free(Stack): 2042

Program Memory Bytes Used(Tokens): 176
Program Memory Bytes Used(Library): 1756
Program Memory Bytes Used(Total): 1932
Program Memory Bytes Free: 29812

No Errors Detected
Programming...
```

Most of the lines are self-explanatory, a few notes will help to clarify others.

“Tokenizing” is the process of converting input words into numbers (tokens) recognized by the compiler. The number of “tokens compiled” is the number of unique commands in your program.

RAM Memory Bytes are used by the system, stack and User. In our example we defined the variable “counter” as a “word”, i.e. two bytes, as shown above.

Programming Multiple BasicATOM Pros

Multiple BasicATOM Pro modules can be programmed in sequence from the same source program; just connect each one in turn to the computer and click the Program button.

Note that the program will be recompiled for each BasicATOM PRO Programmed, the object code (i.e. the code that’s downloaded to the BasicATOM Pro module) is not retained by the IDE for re-use.

Summary

You’ve learned about the components that make up a BasicATOM Pro development system. You’ve learned to run the software, write a program, set up the hardware, and test a couple of simple projects. By now you should be feeling quite at home with the BasicATOM Pro.

Now it’s time to plan your own applications and projects. The rest of this manual is devoted to Atom BASIC details, compiler directives, syntax, etc.

SECTION 2:
Atom BASIC

This group of sections includes all Atom BASIC commands, functions, keywords, etc. grouped in logical sequence.

> Important: While the commands, functions and keywords described in this manual are similar to those for other Basic Micro products, there are some detail differences. Please refer to the correct manual for the Basic Micro product you are using.

This page intentionally left blank

Compiler Preprocessor

In most cases the operation of the compiler is automatic and transparent: you simply write your program and click the Program button. However, the Atom BASIC compiler includes a “preprocessor” that accepts instructions (“directives”) to control compiling according to rules you specify.

The preprocessor has two basic functions:

1. It lets you keep a collection of frequently used program modules (“snippets” of code) and include them whenever you need them.
2. It provides conditional tests (IF ... THEN) to modify the code you compile, allowing you to use the same basic code to compile different versions of a program.

Including Files

Perhaps you’ve written a subroutine to control an LCD display, and you’d like to use this subroutine in various different programs. You can save the subroutine on disk, and “include” it whenever you need it.

#include

The #include directive is used to “paste” program modules at compile time. Modules are pasted at the location of the #include directive.

Syntax

#include “*filename*”

#include “*partial or complete path to file*”

Example

Assume that your LCD subroutine called “lcd.bas” and is in the same directory as your main program. The subroutine contains the label “displaywrite”. You can include this in your program like this:

```
main
(some code)
   gosub displaywrite
(some more code)
#include “lcd.bas”
end
```

The #include directive simply pastes in the code from lcd.bas as if it was part of your program.

If lcd.bas is in a subdirectory of your program directory, just put the partial path, for example:

```
#include “modules\lcd.bas”
```

If it’s in another directory, you can include the relative or absolute path, using normal Windows notation.

Conditional Compiling

Sometimes the same program may be used for slightly different applications. For example, if you’ve written a program to display temperature from a sensor, you may want versions for Celsius and Fahrenheit degrees, or perhaps you want one version to use an LCD display and a different one to output serial data to your computer. Most of the code is identical, but some constants, variables and subroutines may differ.

Conditional compiling lets you set a “switch” in your program (usually a constant, but not necessarily) that controls compiling. You can have different constants, variables, or even different sections of code compiled depending on the switch or switches that you set.

#IF ... #ENDIF

Similar to the usual BASIC if ... then conditional branch. Specifies code to be compiled if the expression is true.

Syntax

#IF *constant expression*

Example

```
fahr con 1
#IF fahr=1
... some code ...
#ENDIF
... rest of program ...
```

This will compile the code between #if and #endif only if fahr=1. If fahr has any other value, the code between #if and #endif will be skipped.

#IFDEF ... #ENDIF

Compiles the following code (up to #ENDIF) only if the constant or variable is defined, or if the label appears previously in the code.

Syntax

#IFDEF *name*   ; name is a variable, constant or label

Example 1

```
temperature var byte
#IFDEF temperature
... some code ...
#ENDIF
... rest of program ...
```

This will compile “some code” because “temperature” has been defined.

Example 2

```
main
... some code ...
#IFDEF main
... conditional code ...
#ENDIF
... rest of program ...
```

This will compile “conditional code” because the label “main” precedes the #IFDEF condition.

#IFNDEF ... #ENDIF

Compiles the code between #IFNDEF and #ENDIF only if the constant or variable has **not** been defined, or the label has **not** been previously used in the program. In effect, it’s the inverse of #IFDEF.

#ELSE

Allows you to have two code snippets, and compile one or the other depending on the result of the #IF, #IFDEF or #IFNDEF directive.

Syntax

#ELSE

Example

```
fahr con 1
#IF fahr=1
... some code ...
#ELSE
... some other code ...
#ENDIF
... rest of program ...
```

Compiles “some code” if fahr = 1 and “some other code” if fahr has any other value.

#ELSEIF

Allows multiple snippets of code to be compiled based on multiple tests. Regard this as an extension of the #ELSE directive.

Syntax

#ELSEIF *constant expression*

Example

```
screentype con 1
#IF screentype=1
... some code ...
#ELSEIF screentype=2
... some other code ...
#ELSEIF screentype=3
... yet more code ...
#ENDIF
... rest of program ...
```

Compiles “some code”, “some other code”, or “yet more code” respectively when *screentype* is 1, 2 or 3. If *screentype* has some other value, compilation simply continues with “rest of program” and none of the snippets is compiled.

#ELSEIFDEF, #ELSEIFNDEF

Equivalents of #ELSEIF for the #IFDEF and #IFNDEF directives.

Syntax

#ELSEIFDEF *name*

#ELSEIFNDEF *name*

Example

Similar to the example given for #ELSEIF.

Note: All compiler preprocessor directives must start with the # sign. If you forget this, results will not be what you expect.

This page intentionally left blank

Hardware, Memory, Variables, Constants

Built-in Hardware

The BasicATOM Pro has hardware functions that are independent of the main microcontroller. Examples include:

- Analog to digital converters
- Pulse width modulators
- UARTS
- Timers
- Capture/Compare modules

Built in hardware runs independently of the microcontroller, and can be set up via your program then allowed to run independently.

RAM

RAM is Random Access Memory, which is “volatile” (i.e. the contents are lost if power is removed). The Atom Pro has 2 kbytes of RAM, the Atom Pro Plus has 4 kbytes. RAM is used to store:

- Variables
- Values used by the system (registers) – 0 to 256 bytes
- Program stack, counters, etc. – approx. 80 to 120 bytes.

Remaining RAM is available for user data.

Registers

The microcontroller on which the Atom Pro is based, uses a number of *registers* to control its internal operation, set status, etc. These registers occupy memory addresses in RAM. Registers are accessible by using their numeric addresses (see the H8/3664F or H8/3687F hardware manuals for these addresses) or by using the register name.

All H8 register names are pre-defined in Atom BASIC and may be used as variables to access and modify register bits.

EEPROM

EEPROM is Electrically Erasable Programmable Read Only Memory. It is similar to RAM in that numbers can be stored in EEPROM and retrieved from EEPROM, but it is slower than RAM (particularly for writing values) and has a limited number of write-cycles (about 10 million) before it fails.

- The Atom Pro has no built-in EEPROM. An external I^2C EEPROM may be connected, see *Notes* on page 172.
- The Atom Pro 28 and Atom Pro Plus 40 have 4 kbytes of EEPROM.

EEPROM is normally used for storing constants that don't change frequently. It is accessed by means of the DATA, READ, READDM, WRITE and WRITEDM commands (see page 170).

Program Memory (Flash)

Programs are stored in “Flash EEPROM” which can be re-written many times. Flash is non-volatile, so your programs will be saved during power off periods. More complex programs require more memory. Program memory can also be used to store constants (see Tables on page 45). The ATOM Pro has 32 kbytes of program memory, the ATOM Pro Plus has 56 kbytes. Flash provides fewer rewrite cycles before failure than EEPROM.

Number Types

Atom BASIC lets you store numbers with varying precision to minimize memory space requirements. The following number types are used:

Table 1 - Number Types

Type	**Bits**	**Value range**
Bit	1	1 or 0
Nib	4	0 to 15
Byte	8	0 to 255
Sbyte	8	-127 to +128
Word	16	0 to 65535
Sword	16	-32767 to +32768

Type	Bits	Value range
Long	32	0 to 4294967295
Slong	32	-2147483647 to +2147483648
Float	32	$\pm 2^{-126}$ to $\pm 2^{127}$

Use the smallest number type that gives the range and precision you need. This will minimize the use of RAM.

Variables

Variables are used to store temporary values while your program runs. Like most compiled BASICs, with Atom BASIC you must define your variables before using them. Variables are always stored in user RAM.

Defining variables

Variables are defined with the VAR keyword. They may be defined as any of the number types described in Table 1. Remember to define variables as SBYTE, SWORD, SLONG or FLOAT if they may have negative values.

Syntax

variable name VAR *number type*

Examples

```
red var byte
tick var nib
switch var bit
totals var long
fraction var float
```

Note that the compiler will automatically “pack” *nib* and *bit* variables to use as few bytes as possible.

Variable Names

Variable names must start with a letter, but they can contain letters, numbers, and special characters. They are case-independent so RED and red are the same variable. Variable names may be up to 1024 characters long. We recommend that they be made long enough to be meaningful, but short enough for easy reference. The length of a variable name does not affect the length of your compiled program. You may not define the same variable twice within a program.

The following may not be used for variable names:

- Atom BASIC reserved words (see page 191)
- label names used within your program

> Important: Please note that the following single letters are reserved and may not be used as variables: b, c, d, p, r, s, z.

Note: Some of the examples in this manual use selected single letter variables for brevity and simplicity in short code segments.

Array variables (strings)

Arrays are special variables used in BASIC to hold a number of related values. Atom BASIC provides for **linear** arrays (also known as one-dimensional arrays or “subscripted” variables).

> Important: Array variables should not use the same names as existing simple variables. See page 184 for more information about this.

Syntax to Define an Array

variable name VAR *number type(number of cells)*

Example

Suppose you want to store values for 5 temperature sensors in an array. Each sensor requires a WORD value. Define the array as:

```
temp var word(5)
```

This sets aside 5 “words” in RAM for your array, numbered 0 to 4.

Note: Atom BASIC numbers arrays starting from 0, so the first value is arrayname*(0) etc. The last value is* arrayname*(4) in this case.*[6]

You can access each value as if it was a separate variable, for example you can access the third value as:

```
word(2)
```

[6] If you find counting from 0 confusing, simply define your array to have one more cell than needed, and ignore cell 0. This wastes one cell, of course.

The number “2” in this case is called the “index”. The third value has an index of 2 because the array is numbered 0, 1, 2, 3, 4.

When accessing array values, the index can be a variable:

```
cntr var byte
temp var word(5)
for cntr=0 to 4
temp(cntr)= cntr+15
next
```

This will assign the values of 15, 16, 17, 18 and 19 to the array cells, so that temp(3) would have a value of 18, etc.

Using Array Variables to Hold Strings

A common use of array variables is to hold "strings" of ASCII characters. When used for this purpose, arrays should be defined as byte variables.

> Important: Bounds checking is not performed. It is the programmer's responsibility to make sure that the string to be stored does not exceed the length of the array.[7]

Example

The following program excerpt will illustrate this use of array variables.

var byte mystring(20)

mystring = "This is a test"

will assign "T" to mystring(0), "h" to mystring(1), etc.

Aliases

An alias gives you a means of assigning more than one name to the same variable. You may need temporary variables at different points in your program, but not want to use too much RAM to store all of them.

Syntax

new variable VAR *existing variable*

[7] If the string length exceeds the array length, characters will simply overwrite the next variable(s) in the order that they were originally defined.

Example

```
sensor VAR byte
eye VAR sensor
```

This will create a variable “eye” that points to the same RAM location as the variable “sensor”. In your program you might have a loop:

```
for sensor = 1 to 10
(some code or other)
next
```

Then later in your program you could have a loop:

```
for eye = 2 to 8
(some code or other)
next
```

Sensor and eye would use the same RAM location, thus saving 1 byte of RAM.

Variable Modifiers

Variable modifiers are used to access parts of a variable. For example, you could access the high or low nibble of a “byte” variable. Modifiers can be used both when variables are defined, and “on the fly” during program execution.

Note: Modifiers won’t work with FLOAT type variables.

Syntax

variable.modifier

A complete list of modifiers appears on page 41.

Example 1

Example 1 shows an alias that accesses only part of a variable. The modifier is used to define a variable.

```
sensor VAR byte
eye VAR sensor.highnib
```

Sensor is a byte (8 bit) variable. Eye is defined as the high nibble (most significant 4 bits) of Sensor. Changing the value of Eye will change only the top 4 bits of Sensor.

Example 2

Example 2 shows the use of modifiers during execution of a program. First, the variables are defined:

```
maxval VAR word
topval VAR byte
```

Then, at a later point in the program, the statement:

```
topval = maxval.highbyte
```

Assigns the value contained in the high 8 bits of maxval to topval. (We could also have used maxval.byte1 to get the same value in this example.)

Note: when using variable modifiers, be sure to keep track of the length of each variable as defined by its number type.

Example 3

Variable modifiers can be used in conditional statements:

```
if maxval.bit0 = 0 then even
```

Where “even” is a label. Note that if bit0 = 0 the number must be an even number, so this test can be used to determine if a number is even or odd.

List of Modifiers

The following modifiers are allowed. Note that in some cases different modifiers will give the same result (for example, in a “word” variable, “highbyte” and “byte1” are the same).

Modifier	**Notes**
lowbit	returns the low bit (least significant bit) of a variable.
highbit	returns the high bit (most significant bit) of a variable.
bit*n*	returns the “nth” bit of a variable. n can have a value of: 0 to 3 for a NIB variable 0 to 7 for a BYTE variable 0 to 15 for a WORD variable 0 to 31 for a LONG variable
lownib	returns the low nibble (4 bits) of a variable
highnib	returns the high nibble (4 bits) of a variable
nib*n*	returns the “nth” nibble of a variable. n can have a value of: 0 or 1 for a BYTE variable

Modifier	Notes
	0 to 3 for a WORD variable 0 to 7 for a LONG variable
lowbyte	returns the low byte of a variable
highbyte	returns the high byte of a variable
byte*n*	returns the “nth” byte of a variable. n can have a value of: 0 or 1 for a WORD variable 0 to 3 for a LONG variable
lowword	returns the low word (16 bits) of a variable.
highword	returns the high word of a variable.
word*n*	returns the “nth” word of a long variable. n can have a value of 0 or 1 (which are equivalent to lowword and highword respectively).

Pin Variables (Ports)

Pin variables are special bit-mapped variables used to set the direction and set or read the state of any I/O pin. Pin Variables are also known as Ports.

Note: The tables below show pins 0 to 31. Please refer to the data sheet for your module to determine which pins are actually available.

Direction

At program start all I/O pins are set as inputs. Directions can be set using the Input and Output commands, or using a Pin variable from the following list:

Variable/port	bits	pins	Variable/port	bits	pins
DIRE	32	P0 – P31	DIRB	4	P4 – P7
DIRS	16	P0 – P15	DIRC	4	P8 – P11
DIRES	16	P16 – P31	DIRD	4	P12 – P15
DIRL	8	P0 – P7	DIREA	4	P16 – P19
DIRH	8	P8 – P15	DIREB	4	P20 – P23
DIREL	8	P16 – P23	DIREC	4	P24 – P27
DIREH	8	P24 – P31	DIRED	4	P28 – P31
DIRA	4	P0 – P3	DIR#	1	P#
			(where # is a number from 0 – 31)		

Each direction bit must be set to 0 for input and 1 for output.

State

The state of an I/O pin can be read or set using the IN and OUT pin variables.

Note: IN and OUT are interchangeable; the two names are provided for compatibility with other BASIC implementations and for clarity in programming. Whether a state is set or read is determined by the direction assigned to the pin.

The table below shows all IN and OUT Pin variables:

Variable/port	bits	pins	Variable/port	bits	pins
INE or OUTE	32	P0 – P31	INB or OUTB	4	P4 – P7
INS or OUTS	16	P0 – P15	INC or OUTC	4	P8 – P11
INES or OUTES	16	P16 – P31	IND or OUTD	4	P12 – P15
INL or OUTL	8	P0 – P7	INEA or OUTEA	4	P16 – P19
INH or OUTH	8	P8 – P15	INEB or OUTEB	4	P20 – P23
INEL or OUTEL	8	P16 – P23	INEC or OUTEC	4	P24 – P27
INEH or OUTEH	8	P24 – P31	INED or OUTED	4	P28 – P31
INA or OUTA	4	P0 – P3	IN# or OUT#	1	P#
			(where # is a number from 0 – 31)		

The state bit is 0 for low and 1 for high.

Examples

As an example, let’s say we want to set P0 to P3 as inputs and P4 to P7 as outputs. Here are two different ways to accomplish the same thing.

Using 4 bit variables:

```
...
dira = 15            ; 15 decimal = 1111 binary
dirb = 0             ;  0 decimal = 0000 binary
...
```

Using an 8 bit variable:

```
...
dirl = 15            ; 15 decimal = 00001111 binary
...
```

Now we want to set and read the I/O pins we've set up. Let's say we expect the following, where 0 = low and 1 = high:

pin	P0	P1	P2	P3	P4	P5	P6	P7
out	1	0	1	1				
in					0	1	1	0

Using two 4 bit variables:

```
...
outa = 11            ; 11 decimal = 1011 binary
status = inb         ; status will equal 6 decimal
                     ; or 0110 binary
...
```

Note: We've used "out" for output and "in" for input to make it easier to understand the program snippet. In fact, any combination of "out" and "in" would work equally well (but could be confusing to read).

Constants

Constants are similar to variables except that the values are set at compile time and can't be changed by the program while it's running. Think of a constant as a convenient way to give a name to a numeric value so you don't have to remember what it is each time it's used.

Note: Constants are stored in program memory (see page 36), not in RAM. This frees valuable RAM for variables as well as making constants non-volatile. The values of constants are stored when the program is downloaded from the IDE to the Atom Pro.

Defining Constants

Constants are defined with the CON keyword. The numeric type is automatically set by the compiler based on the value you enter.

Syntax

constant name CON *value of constant*

Examples

```
temperature_adjust CON 24
kilo CON 1000
true CON 1
false CON 0
```

```
endpoint CON –3442567
```

Constant Names

Names of constants follow the same rules as names of variables, see page 45.

Tables

Tables are to constants as Arrays are to variables. A table can be used to store a number of related constants which are then referred to using the index number.

Note: Tables are stored in program memory (see page 36), not in user RAM. This frees valuable RAM for variables and makes Tables non-volatile.

The table contents are written to program memory while the Atom Pro is being programmed, therefore defined constants and variables can NOT be used to populate tables. However, the table is accessed from within a running program, so variables can be used to "index" into a table.

Syntax

tablename TableType *data, data, data...*

Table names follow the same rules as variable and constant names.

TableType can be any of the following:

- ByteTable (8 bit data)
- WordTable (16 bit data)
- LongTable (32 bit data)
- FloatTable (floating point data)

Byte Tables can also be used to store “string” values.

data is a constant, or a series of constants or constant expressions. Variables and named constants can't be used as data because the table is populated before your program starts running. However, expressions such as 3 * 10 are legal.

Examples

```
adjust WordTable 100,350,5678,73,9,8133*3,0
```

creates a constant array with values as shown. Cell numbering starts with 0, so the value of “adjust(3)” would be 73. As with arrays, when referring to the table the index can be a variable.

```
letter ByteTable “This is a test”,0
```

creates a constant array containing a string value, terminated with a zero. In this example the value of “letter(0)” would be “T”, the value of “letter(3)” would be “s”, etc.

Terminating the table with 0 allows you to test for the end of the table, without knowing how many characters it contains.

Pin Constants

Note: This section shows pins P0 to P31. Please refer to the data sheet for your module to determine which pins are actually available.

Pin constants are pre-defined for easy reference to I/O pins.

```
P0 = 0
P1 = 1
...
P31 = 31
```

Example

I/O pin 8 could be set to Output state and to High (5V) using either:

```
high 8
```

or

```
high P8
```

Another example of the use of Pin constants can be found in the Traffic Light program on page 23.

Each Atom Pro module also has two special serial I/O pins used for programming and serial data communications. They can be referred to in your program as:

S_IN and S_OUT

See the appropriate data sheet to identify these pins.

Math and Functions

As with most BASIC implementations, Atom BASIC includes a full complement of math and comparison functions. For the most part, Atom BASIC performs *integer* arithmetic, however floating point variables and functions are provided for use where required.

Integer arithmetic is preferred for most control functions because it is fast and economical of memory. The descriptions in this chapter show some of the techniques that can be used to deal with fractional values in an integer-only environment.

Users should remember that variables must be defined before use, and that each definition should be of the appropriate number type (e.g. byte, word or long) for the functions used. Most functions will work with all variable types; exceptions are noted in the function descriptions.

Number Bases

Although all calculations are handled internally in binary, users can refer to numbers as decimal, hexadecimal or binary, whichever is most convenient for the programmer. For example, the number 2349 can be referred to as:

```
2349 or d’2349’      decimal notation
$092D or 0x092D      hexadecimal notation
%100100101101        binary notation
```

Note: Leading zeros are not required for hex or binary numbers, but may be used if desired.

If you’re planning to use signed integers (sbyte, sword, slong) it’s probably a good idea to stick to decimal notation to avoid confusion.

Integer Math Functions

The math functions described in this section all use integer arithmetic (unless otherwise stated). For floating point functions see Floating Point Math on page 64.

Operator Precedence

The ATOM-Pro uses standard algebraic syntax. In the ATOM-Pro 2+2*5/ 10 = 3. This is because in the ATOM-Pro each math operator has a precedence. The multiply and divide operators have equal precedence. In the above calculation 2*5 will be calculated first (equaling 10), then the divide by 10 (equals 1), then the addition of 2 (equaling 3). You can use parenthesis to force specific orders (e.g. ((2+2)*5) /10 would yield a value of 2).

<table>
<tr><th>Order</th><th>Operation</th></tr>
<tr><td>1st</td><td>NOT, ABS, SIN, COS, - (NEG), DCD, NCD, SQR, RANDOM, TOINT, TOFLOAT, BIN2BCD, BCD2BIN, ~(Binary NOT), !(Binary NOT), NOT(Logical NOT), FSQRT, FSIN, FCOS, FTAN, FASIN, FACOS, FATAN, FSINH, FCOSH, FTANH, FATANH, FLN, FEXP</td></tr>
<tr><td>2nd</td><td>Rev, Dig</td></tr>
<tr><td>3rd</td><td>MAX, MIN</td></tr>
<tr><td>4th</td><td>*, **, */, /, //</td></tr>
<tr><td>5th</td><td>+, -</td></tr>
<tr><td>6th</td><td><<, >></td></tr>
<tr><td>7th</td><td><, <=, =, >=, >, <></td></tr>
<tr><td>8th</td><td>&, |, ^, &/, |/, ^/</td></tr>
<tr><td>9th</td><td>And, Or, Xor</td></tr>
</table>

Out of Range Values

> Warning: Out of range values can occur if the limitations of number types (see page 36) are exceeded. The Atom BASIC compiler does not warn the user of out of range values. It’s your responsibility to make sure variables are appropriately defined.

Out of range conditions can occur if, for example, executing a function produces a result with a value greater than the target variable is capable of storing. For example, in the following program segment:

```
ant var byte
bat var byte
cat var byte
bat = 12
cat = 200
ant = bat * cat     ; * is the multiply function
```

"ant" will not have the expected value. This is because 12 x 200 = 2400, a value too large to fit into a byte variable. To see what actually happens, look at the numbers in binary form:

bat = 12 decimal = 1100 binary

cat = 200 decimal = 11001000 binary

bat * cat = 2400 decimal = 100101100000 binary (12 bits)

Since "ant" can't hold a 12 bit number, **the lowest 8 bits** will be stored in "ant", and the result will be a = 01100000 binary = 96 decimal, which is incorrect. You won't be warned of this, so take care to make sure the target variable is large enough to handle the full range of expected results.

Unary Functions

Unary functions have only one *argument* and produce one result. In the list below, "expr" is any variable, constant or valid mathematical expression.

> Important: These functions don't work with floating point numbers.[8]

Function	**Description**
– expr	negates the value of expr
ABS expr	returns the absolute value of expr
SIN expr	returns the sine of expr
COS expr	returns the cosine of expr
DCD expr	returns 2 to the power of expr
NCD expr	returns the smallest power of 2 that is greater than expr
SQR expr	returns the square root of expr
BIN2BCD expr	converts expr from binary to packed BCD format
BCD2BIN expr	converts expr from packed BCD to binary format
RANDOM expr	returns a random number (32 bit) generated with seed expr

– (negate)

Negates the value of the associated expression. The result will be a signed value; the *target* variable should be defined as such.

[8] See the floating point functions on page 64.

Example

If temp is a signed variable, and “mark” has a value of 456, the statement

```
temp = –mark
```

will assign the value of –456 to “temp”

SIN, COS

Consider using floating point equivalents for these functions. See page 64.

For use with integer arithmetic, some modifications to the usual use of sine and cosine are made. For example, in floating point math, the expression:

```
ans = sin(angle)
```

where *angle* is 1 radian, would return a value of 0.841... for *ans*. In fact, the sine of an angle must always be a fractional value between -1 and 1. Atom BASIC can’t deal with fractional values, however, so we’ve modified the use of SIN and COS to work with integers.

Because we are dealing with binary integers, we divide the circle into 256 (rather than 360) parts. This means that a right angle is expressed as 64 units, rather than 90 degrees. Thus, working with Atom BASIC angular units gives you a precision of about 1.4 degrees.

The result of the SIN or COS function is a signed number in the range of –127 to +128. This number divided by 128 gives the fractional value of SIN or COS.

Real World Example

In most “real world” applications, the angle need not be in degrees, nor need the result be in decimal form. The following example shows a possible use of SIN with the Atom BASIC values.

Suppose that a sensor returns the angle of a control arm as a number from 0 to 64, where 0 is parallel and 64 is a right angle. We want to take action based on the sine of the angle.

```
   limit var byte
   angle var byte
loop
```

```
   (code that inputs the value of "angle")
   limit = sin angle
   if limit > 24 then first
   if limit > 48 then second
   goto loop
first
   code to warn of excessive angle
   goto loop
second
   code to shut down equipment
   etc...
```

This will warn the operator if the arm angle exceeds approximately 8 units (11.25 degrees) and shut down the equipment if the arm angle exceeds approximately 16 units (22.5 degrees).

Theoretical Example

Although most control examples don't need to work in actual degrees or decimal values of sine or cosine, this example will show how that can be accomplished. To find the sine of a 60 degree angle, first convert the angle to Atom BASIC units by multiplying by 256 and dividing by 360. For example,

```
angle = 60 * 256 / 360
```

which will give a value of 42. (It should actually be 42.667, which rounds to 43, but with integer arithmetic the decimal fraction is ignored, and the integer is not rounded up.)

Then find the sine of this angle:

```
ans = sin angle
```

This will give the value 109. Dividing this value by 128 will give the decimal value of 0.851 (compared to the correct floating point value which should be 0.866).

Note: You can't directly get the decimal value by doing this division within Atom BASIC (you would get a result of 0). However, you could first multiply by 1000, then divide by 128 to get 851 as your result.

DCD

Similar to the "exp" function in some other BASIC implementations. Returns 2 to the power DCD.

Example

If the value of “num” is 7, the statement

```
ans = dcd num
```

will return a value of 2^7, or 128. Since the returned value increases exponentially, make sure your target variable (“result” in this case) is correctly defined to accommodate the largest value anticipated. If the target variable is too small, only the low order bits of the result will be stored.

NCD

This function returns the smallest power of 2 that is greater than the argument.

Example

If the value of “num” is 51, the statement

```
ans = ncd num
```

will return the value of 6. Since 2^5 = 32 and 2^6 = 64, 6 is the smallest power of 2 greater than 51.

SQR

Returns the integer portion of the square root of the argument. Increased precision can be obtained by multiplying the argument by an even power of 10, such as 100 or 10000.

Consider using FSQRT with floating point numbers, see page 64.

Example 1

If the value of “num” is 64, the statement

```
ans = sqr num
```

will return the value of 8 (which is the square root of 64).

Example 2

If the value of “num” is 220, the statement

```
ans = sqr num
```

will return the value 14, which is the integer portion of 14.832..., the square root of 220.

Example 3

If more precision is required, multiply the argument by 100 or 10000. Again, using the example where “num” = 220:

```
ans = sqr (num * 100)
```

will return the value 148, which is 10 times the square root of 220.

Alternately,

```
ans = sqr (num * 10000)
```

will return the value 1483, which is 100 times the square root of 220.

Note: If you subsequently divide these results by 10 or 100, the precision gained will be lost because of the integer division. You should convert the numbers to floating point first. *See page 64.*

BIN2BCD, BCD2BIN

These commands let you convert back and forth between binary and “packed” binary coded decimal (BCD). A BCD number is one in which each decimal digit is represented by a 4 bit binary number (from 0 to 9). Packed BCD packs two 4 bit decimal digits in a single byte of memory.

For example, the decimal number 93 is represented in binary as:

	128	64	32	16	8	4	2	1
binary	0	1	0	1	1	1	0	1

The same number is expressed in packed BCD as:

	8	4	2	1	8	4	2	1
packed BCD	1	0	0	1	0	0	1	1
	9				3			

Example

Assuming that “ans” is a byte variable and “num” has the decimal value of 93, the statement

```
ans = bin2bcd num
```

will set *ans* to a binary value of 10010011 (which is 93 in packed BCD).

Note: if you choose to interpret "ans" as a decimal number, the value will seem to be 147 decimal, which is an incorrect interpretation of the result.

RANDOM

The RANDOM function generates a 32 bit random number (LONG) from the *seed* value.

Syntax

random *seed*

"seed" is a variable, constant or expression having any value from 0 to 2^{32}. The seed will be treated as an unsigned number.

As with most random number generators, the random numbers generated will follow a predictable pattern, and each time the program is run the random number sequence will be the same. Two steps can avoid this problem and generate a usefully random sequence of numbers;

1. Generate the original seed from, say, a timer value so that the result will not be the same twice in succession.
2. Use the returned value as the seed for the next RANDOM statement.

Example

Timer A is always running, so we can read its value at any time to get a pseudo-random seed number.

```
   a var long
   a = random TCA
      (code using the value of a)
loop
   a = random a ; uses "a" to generate a new "a"
      (code using the value of a)
goto loop
```

Binary Functions

Binary functions have two arguments and produce one result. In the list below, "expr" is any variable, constant or valid mathematical expression.

Note: The word "binary" means that these functions have two arguments, not that they are specifically designed for use with the bits of binary numbers.

Function	Syntax	Comment
+	expr1 + expr2	addition
–	expr1 – expr2	subtraction
*	expr1 * expr2	multiplication
/	expr1 / expr2	division *
The expressions below do not work with floating point numbers.		
**	expr1 ** expr2	return high 32 bits of a multiplication *
*/	expr1 */ expr2	fractional multiplication *
//	expr1 // expr2	mod *
MAX	expr1 max expr2	returns the smaller expression *
MIN	expr1 min expr2	returns the larger expression *
DIG	expr1 dig expr2	returns the digit from expr1 in the position determined by expr2 *
REV	expr1 rev expr2	reverses the value of expr2 bits of expr1 starting with the LSB *

** These functions are further described in the following sections.*

*** is read as GET HIGH BITS*

If two *long* variables or constants are multiplied, the result may exceed 32 bits. Normally, the multiply function will return the least significant (lowest) 32 bits. The ** function will, instead, return the most significant 32 bits.

You can use both functions to retrieve up to 64 bits of a multiplication, however two long variables will be needed to store this result.

Note: The returned value does not represent the decimal digits of the beginning of the product; it is best to work with binary or hexadecimal when using this function.

---

*/ (fractional multiplication)

Consider using floating point math instead. See page 65

Fractional multiplication lets you multiply by a number with a fractional part. The multiplier must be a *long* number, and it is handled in a special fashion. The high 16 bits are the integer portion of the multiplier, the low 16 bits are the fractional part (expressed as a fraction of 65535). The result, of course, will be an integer; any fractional part is discarded (not rounded).

Example

Let us say we want to multiply the number 346 x 2.5. The multiplier must be constructed as follows:

The high 16 bits will have a value of 2. We can do this with:

```
mult.highword = 2
```

The low 16 bits will have a value of half of 65535, or 32782, so:

```
mult.lowword = 32782
```

Then we do the fractional multiply:

```
a = 346 */ mult
```

which will give “a” the value 865

A similar procedure will let you multiply by any fraction; simply express that fraction with a denominator of 65535 as closely as possible.

Note: Astute readers will notice that half of 65535 is actually 32782.5; a number we can’t enter as the fractional part. This means that multiplication by exactly ½ is not possible. However, the difference is so small that it has no effect on the actual outcome of the integer result.

/ (division)

Integer division discards fractional results. For example:

```
result = 76/7
```

will set the variable “result” to a value of 10. (The actual decimal result should be 10.857... but the decimal part is discarded, rounding is not done.) If your application requires fractional results, there are two possible solutions:

1. Multiply the dividend by 10, 100, 1000 etc. before dividing. The result will gain extra digits of precision but must be interpreted correctly.

   Using the previous example we can gain three digits of precision as follows:

   ```
   temp = dividend * 1000  ;dividend is now 76000
   result = temp/7
   ```

   Which sets "result" to a value of 10857.

2. Use floating point numbers (see page 65)

// (mod)

The *mod* function (short for “modulo”) returns the remainder after an integer division. So, for example, 13 modulo 5 is 3 (the remainder after dividing 13 by 5).

The mod function can be used to determine if a number is odd or even, as shown here:

```
   x var word
   y var word
   (code that sets the value of x)
   y = x//2
   if y=0 goto even  ;zero indicates an even number
   if y=1 goto odd   ;one indicates an odd number
even
   (more code)
odd
   (more code)
```

Similarly, the mod function can be used to determine if a number is divisible by any other number.

Note: Of course there are other ways to determine if a number is odd or even, this is just one example.

MAX

The MAX function returns the smaller of two expressions. For example:

```
x var word
```

```
y var word
code to set value of y
x = y max 13
```

will set x to the value of y or 13, whichever is smaller. Think of this as "x equals y up to a maximum value of 13".

MIN

The MIN function returns the larger of two expressions. For example:

```
x var word
y var word
code to set value of y
x = y min 9
```

will set y to the value of x or 9, whichever is larger. Think of this as "x equals y down to a minimum value of 9".

DIG

The DIG (digit) function is used to isolate a single digit of a decimal number. For example:

```
x var word
y var byte
(code to set y)      ;say the result is y=17458
x = y dig 4          ;gives the 4th digit of y, which is 7
```

Digits are counted from the right, starting with 1. The DIG function will work with numbers in decimal format only. If you need to find a specific digit in a hex or binary number, use a *variable modifier* (see page 40).

hexadecimal

Use the "nib" modifier. Each nibble is a hexadecimal digit. Counting is from the right starting with 0. For example, to find the 3rd hex digit of the number "y" you could use:

```
x = y.nib2
```

(it's nib2 because counting starts from 0, not 1).

binary

Use the “bit” modifier. Each bit is a binary digit. Counting is from the right, starting with 0. For example, to find the 3rd bit of the binary number “y” you could use:

```
x = y.bit2
```

REV

The REV function works directly in binary, but the results may be expressed in any form. It is used to “reverse” the value of the low order bits of a number (i.e. change 0’s to 1’s and vice versa). For example:

```
x var byte
y var byte
x = %101110          ;this is decimal 46
y = x rev 3          ;gives g a value of %101001 or 41
```

Bitwise Operators

Bitwise operators are designed to work with the bits of binary numbers. In the list below, “expr” is any variable, constant or valid mathematical expression.

> Important: These functions don’t work with FLOAT variables or constants. Since they don’t automatically preserve the sign bit, they should be used with caution for signed numbers.

Function	Syntax	Comment
&	expr1 & expr2	AND the bits of the expressions
|	expr1 | expr2	OR the bits of the expressions
^	expr1 ^ expr2	XOR (exclusive OR)
>>	expr1 >> expr2	Shift right the bits of expr1 by expr2 places
<<	expr1 << expr2	Shift left the bits of expr1 by expr2 places
~	~ expr1	Invert the bits of expr1
!	! expr1	Invert the bits of expr1

The examples below will use 8 bit (BYTE) values for simplicity.

& (AND)

The AND function compares two values bit by bit and sets the equivalent bit of the result to 1 if both matching bits are 1's, to 0 if either or both bits are 0's. For example:

expr1	0	1	0	1	1	1	0	1
expr2	1	0	0	1	0	0	1	1
expr1 AND expr2	1	0	0	1	0	0	0	1

Using AND for masking

One useful function for AND is to "mask" certain bits of a number. For example, if we are interested only in the low 4 bits of a number, and want to ignore the high 4 bits, we could AND the number with 00001111 as shown here:

expr1	0	1	0	1	1	1	0	1
expr2 (mask)	0	0	0	0	1	1	1	1
expr1 AND expr2	0	0	0	0	1	1	0	1

As you can see, the high 4 bits are now all set to 0's, regardless of their original state, but the low 4 bits retain their original state.

| (OR)

Note: The **|** *symbol is usually found on the same key as the backslash \.*

The OR function compares two values bit by bit and sets the equivalent bit of the result to 1 if either or both of the matching bits are 1, and to 0 if both bits are 0's. For example:

expr1	0	1	0	1	1	1	0	1
expr2	1	0	0	1	0	0	1	1
expr1 OR expr2	1	1	0	1	1	1	1	1

^ (Exclusive OR)

The Exclusive OR function compares two values bit by bit and sets the equivalent bit of the result to 1 if either *but not both* of the matching bits are 1, and to 0 otherwise. For example:

expr1	0	1	0	1	1	1	0	1
expr2	1	0	0	1	0	0	1	1
expr1 XOR expr2	1	1	0	0	1	1	1	0

>> (Shift Right)

The Shift Right function shifts all the bits of expr1 to the right by the number of places specified by expr2. Zeros are added to the left of the result to fill the vacant spaces. (In some versions of BASIC this is called a “logical shift right”). For example:

expr1	0	1	0	1	1	1	0	1
expr2	0	0	0	0	0	0	1	1
Shift right 3	0	0	0	0	1	0	1	1

Important: The sign bit is not preserved so this function should not normally be used with signed numbers.

<< (Shift Left)

The Shift Left function shifts all the bits of expr1 to the left by the number of places specified by expr2. Zeros are added to the right of the result to fill the vacant spaces. (In some versions of BASIC this is called a “logical shift left”). For example:

expr1	0	1	0	1	1	1	0	1
expr2	0	0	0	0	0	0	1	1
Shift left 3	1	1	1	0	1	0	0	0

Important: The sign bit is not preserved so this function should not be used with signed numbers.

~ or ! (NOT)

The NOT function inverts the value of each bit in a number. For example:

expr1	0	1	0	1	1	1	0	1
NOT expr1	1	0	1	0	0	0	1	0

Comparison Operators

Comparison operators let you compare the values of two expressions for such things as conditional tests (e.g. IF...THEN).

Note: Comparison operators also work with floating point numbers.

Operator	Description
=	is equal to
<>	is not equal to
<	is less than
>	is greater than
<=	is less than or equal to
>=	is greater than or equal to

Comparisons include signed numbers, so –2 is less than +1, etc.

Logical Operators

> These operators don't work with floating point numbers.

Logical operations are used to make logical comparisons. These allow you to set ranges for conditional tests. The following operators are available:

Operator	Description
AND	Logical AND
OR	Logical OR
XOR	Logical Exclusive OR
NOT	Logical NOT

> Important: Do not confuse logical operators with similar bitwise operators. Logical operators return a TRUE or FALSE value that can be tested with a conditional test. They do not operate on individual bits of an expression.

Example of Use

Logical operators link two comparisons. For example:

```
if (a < 100) AND (a  > 10) then label
```

This will branch program execution to “label” if the value of a is between 11 and 99, or go on to the next step if it is outside these limits.

In the following sections, “comp” refers to a comparison test between two expressions.

AND (logical AND)

(comp1) AND (comp2)

Returns a value of TRUE if both comp1 and comp2 are true.

OR (logical OR)

(comp1) OR (comp2)

Returns a value of TRUE if either comp1 or comp 2 or both are true.

Example

```
if (a  < 10) OR (a > 100) then label
```

This will branch program execution to “label” if the value of a is less than 10 or greater than 100, i.e. if the value of a is **not** between 10 and 100).

XOR (logical exclusive OR)

```
if (a < 50) XOR (a > 40) then label
```

This will branch program execution to “label” if the value of “a” is less than 50 or if it is greater than 40, but **not** if it is between 41 and 49. In other words, the branch to “label” will take place if “a” is less than 41 or greater than 49.

NOT (logical NOT)

This unary operator works with a single argument, and returns the reverse of its truth value. So if a comparison is TRUE, NOT(comp) will be FALSE.

Example

```
if NOT(a > 20) then label
```

This will branch to “label” if a is **not** greater than 20, i.e. if a is less than or equal to 20.

Floating Point Math

Floating point numbers are those which are capable of including decimal fractions. They are saved internally as a *mantissa* (the decimal part) and an *exponent* (a multiplier). For example, the number 39.456 would be saved as 0.39456 (mantissa) x 100 (exponent).

Atom Pro BASIC has built-in floating point capability, including a range of functions designed specifically for use with floating point numbers.

Operator Precedence

See the table on page 48 which includes floating point operators.

Unary Functions (Floating Point)

Unary functions have only one *argument* and produce one result. In the list below, “expr” is any variable, constant or valid mathematical expression.

Important: These functions don't work with integer values.

Function	Description
TOINT expr	Converts a Floating Point value to an Integer value.
TOFLOAT expr	Converts an Integer value to a Float ing Point value.
FSQRT expr	Square Root (range: all positive values)
FSIN expr	Sine (range: approx. pi/2 to -pi/2)*
FCOS expr	Cosine (range: approx. pi/2 to -pi/2)*
FTAN expr	Tangent (range: approx pi/2 to -pi/2)*

Function	Description
FASIN expr	ArcSine (range: 1 to –1: accuracy suffers beyond ±0.98)
FACOS expr	ArcCosine (range: same as FASIN)
FATAN expr	ArcTangent (range: all)
FSINH expr	Hyperbolic Sine (range: 1.13 to -1.13)
FCOSH expr	Hyperbolic Cosine (range: 1.13 to -1.13)
FTANH expr	Hyperbolic Tangent (range: 1.13 to -1.13)
FATANH expr	Hyperbolic ArcTangent (range: 1.13 to -1.13)
FLN expr	Natural Log (range: 9.58 to 0.1)
FEXP expr	Exponent, i.e. e to the power of expr (range: all values)

* range is actually ± 1.74 radians

One example will suffice to show the use of these functions.

Example: TOINT

```
x var long
y var float
(code)
x = TOINT y
```

Converts the floating point number "y" to a long integer.

Note: "x" should be a long integer for full accuracy. If "x" is a byte or word only the least significant 8 or 16 bits will be saved.

Binary Functions

Binary functions have two arguments and produce one result. In the list below, "expr" is any variable, constant or valid mathematical expression.

Note: The word "binary" means that these functions have two arguments, not that they are designed for use with the bits of binary numbers.

Function	Syntax	Comment
+	expr1 + expr2	addition
–	expr1 – expr2	subtraction
*	expr1 * expr2	multiplication
/	expr1 / expr2	division *

Comparison Operators

Comparison operators let you compare the values of two expressions for such things as conditional tests (e.g. IF...THEN).

Note: Comparison operators also work with integer math.

Operator	Description
=	is equal to
<>	is not equal to
<	is less than
>	is greater than
<=	is less than or equal to
>=	is greater than or equal to

Comparisons include signed numbers, so –2 is less than +1, etc.

Floating Point Format

This description is provided for comparison with other systems that use IEEE floating point math. It is not necessary to understand this format to successfully use floating point numbers.

The floating point math used by the BasicATOM Pro is similar to the IEEE 754 floating point standard with the exception of the position of the sign bit (S).

IEEE 754 format:

Bit 31	sign bit (S)
Bits 30 – 23	exponent (E)
Bits 22 – 0	mantissa (M)

ATOM PRO format:

Bits 31 – 24	exponent (E)
Bit 23	sign bit (S)
Bits 22 – 0	mantissa (M)

In graphical form:

	31	bits 30 - 23	bits 22 - 0
IEEE	S	exponent (E)	mantissa (M)

	bits 31 - 24	23	bits 22 - 0
ATOM	exponent (E)	S	mantissa (M)

Command Modifiers

All internal calculations in the BasicATOM Pro are done in binary arithmetic. On the other hand, much I/O (input/output) is done in the form of ASCII characters. This includes such things as keyboard input, output to displays, etc.

Atom Pro Basic provides *command modifiers* to convert between numeric values and ASCII, in a variety of formats. So, for example, the numeric value 21 (00010101 binary) could be output as

```
 00010101 (binary)
%00010101 (binary with indicator)
 21 (decimal)
 15 (hexadecimal)
$15 (hex with indicator)
```

There is also provision for signed and floating point numbers. All of the above are output as ASCII characters, so the number %00010101, which is a single byte in memory, is actually output as nine ASCII characters to the display, debugger, etc.

Modifiers can also be used to input ASCII characters and convert them to binary numbers.

Commands that Use Modifiers

The following commands accept modifiers in their arguments. See the listing for each individual command for details. Modifiers can be used wherever {mods} or {modifiers} are shown in the command syntax.

The Examples in this Chapter

To avoid confusion, for the examples in this chapter we'll use HSERIN and HSEROUT which have the simplest syntax.

Conventions Used in this Chapter

{ ... } represent **optional** components in a command. The { } are **not** to be included.

[...] used for lists – the [ ] are **required**

(...) used for some arguments. The ( ) are **required**

Available Modifiers

The following modifiers are available in Atom Basic:

I/O modifiers

dec	decimal
hex	hexadecimal
bin	binary
str	input or output array variables

Signed I/O modifiers

sdec	signed decimal
shex	signed hexadecimal
sbin	signed binary

Indicated I/O modifiers

ihex	hexadecimal with $
ibin	binary with %

Combination I/O modifiers

ishex	hex with sign and $
isbin	binary with sign and %

Output-only modifiers

rep	output character multiple times
real	output floating point number

Input-only modifiers

waitstr	waits until values match array
wait	waits until values match constant string
Skip	skip multiple values

Output type command modifiers can also be used with the equal command and an array variable. For example:

```
string var byte(100)
string = "Hello World"  ;string(0-10)="Hello World"
string = dec 1234567    ;string(0-6)="1234567"
string = ihex 0x3456    ;string(0-4)="$3456"
```

etc...

I/O Modifiers (HEX, DEC, BIN)

Input

Convert input ASCII characters to a numeric value. Input must be in hex, decimal or binary format.

> Important: ASCII characters will continue to be input until an "illegal" character is received or until the specified length is reached. Any illegal character will be discarded and will terminate input. An "illegal" character is one not appropriate for the type of input, e.g. anything other than 0...9 for DEC, 0 or 1 for BIN, etc.

Output

Convert a numeric value to ASCII characters in hex, decimal or binary format.

Syntax

modifier{#max} argument{\#min}

#max: optional maximum number of digits to pass
#min: optional minimum number of digits to pass

Examples - Input

If the input is 56 (in ASCII characters)

```
hserin [dec a]
```

assigns the numeric value 56 (binary 00111000) to the variable “a”.

If the input is 3456 (in ASCII characters)

```
hserin [dec2 a]
```

takes the two least significant digits (56) of the input and assigns the numeric value 56 (binary 00111000) to the variable “a”.

Examples - Output

If the numeric value of variable “a” is 1234

```
hserout [dec a]     ; output is 1234 in ASCII
hserout [dec2 a]    ; output is 34 in ASCII
```

If the value of variable “x” is 5

```
hserout [dec x\2]   ; output is 05 in ASCII
```

Note: See also the Special Note re. Output Modifiers on page 77.

I/O Modifier (STR)

Input

Accept a variable number of values and store them in a variable array. Input is in numeric format, undelimited (i.e. bytes are expected to simply follow each other sequentially) and is not converted from ASCII.

Output

Output the elements of an array in numeric format. Output is not converted to ASCII characters and bytes are not delimited but simply follow each other sequentially.

Syntax

str arrayname{\length{\eol}}

length	optional maximum number of values to pass
eol	optional end of line (EOL) character to terminate

Examples – Input

```
hserin [str temp\5]
```

will accept the next 5 numeric input values and assign them to temp(0)... temp(4)

```
hserin [str temp\100\”x”]
```

will accept up to 100 input values, stopping when an “x” is input, and assign them to array “temp”. Similarly,

```
hserin [str temp\100\13]
```

will accept up to 100 input values, stopping when a carriage return (ASCII 13) is input.

Example – Output

```
hserout [str temp\8]
```

will output the first 8 values of the array “temp”, beginning with temp(0). Remember that output is in numeric form, not converted to ASCII.

Note: See also the Special Note re. Output Modifiers on page 77.

Signed I/O Modifiers (SHEX, SDEC, SBIN)

Input

Convert input ASCII characters to a signed numeric value. Input must be in hex, decimal or binary format.

> Important: ASCII characters will continue to be input until an "illegal" character is received. That character will be discarded and will terminate input. An "illegal" character is one not appropriate for the type of input, e.g. anything other than 0...9 for DEC, 0 or 1 for BIN, etc.

Output

Convert a signed numeric value to ASCII characters in hex, decimal or binary format.

Syntax

modifier{#max} argument{\#min}

#max: optional maximum number of digits to pass
#min: optional minimum number of digits to pass

Examples - Input

If the input is –56 (in ASCII characters)

```
hserin [sdec a]
```

assigns the numeric value –56 to the variable “a”.

If the input is –3f (in ASCII characters)

```
hserin [shex a]
```

assigns the numeric value –56 (expressed in decimal) to the variable “a”.

Examples - Output

If the value of variable “a” is –1234 (decimal) or –4d2 (hex)

```
hserout [sdec a]    ; output is –1234 in ASCII
hserout [shex2 a]   ; output is –D2 in ASCII
```

If the value of variable “x” is 5

```
hserout [sdec x\2]  ; output is +05 in ASCII
```

Note: See also the Special Note re. Output Modifiers on page 77.

Indicated I/O Modifiers (IHEX, IBIN)

Indicated I/O modifiers are almost identical in both syntax and operation to the unsigned and signed modifiers described on the previous two pages.

Input

ASCII characters are converted to an unsigned numeric value. Input characters are ignored until a valid indicator ($ for hex, % for binary) is received.

> Important: ASCII characters will continue to be input until an "illegal" character is received. That character will be discarded and will terminate input. An "illegal" character is one not appropriate for the type of input, e.g. anything other than 0...9 for DEC, 0 or 1 for BIN, etc.

Output

An unsigned numeric value is converted to ASCII characters preceded by an indicator ($ for hex, % for binary).

Syntax

modifier{#max} argument{\#min}

#max:	optional maximum number of digits to pass
#min:	optional minimum number of digits to pass

Examples – Input

If the input is $A3FC

```
hserin [ihex a]
```

will assign the numeric value $A3FC (41980 decimal) to variable “a”.

Note that an input of “The value is $A3FC” will ignore all characters prior to the $ and give the same result as above.

```
hserin [ihex2 a]
```

will assign the numeric value $FC (252 decimal) to variable “a”.

Examples – Output

If the numeric value of “a” is 41980 (decimal) or $A4FC (hex):

```
hserout [ihex a]    ; output is $A4FC in ASCII
hserout [ihex2 a]   ; output is $FC in ASCII
```

If the numeric value of “x” is 5

```
hserout [ibin x]    ; output is %101 in ASCII
hserout [ibin x\8]  ; output is %00000101 in ASCII
```

Note: See also the Special Note re. Output Modifiers on page 77.

Combination I/O Modifiers (ISHEX, ISBIN)

Combination I/O modifiers have the characteristics of both Indicated and Signed modifiers, as described in the previous section.

Syntax

For syntax see the previous sections.

Examples – Input

If the input is $-3FC

```
hserin [ishex a]
```

will assign the numeric value $-3FC (-1020 decimal) to variable “a”.

Examples – Output

If the numeric value of “a” is -41980 (decimal) or $A4FC (hex):

```
hserout [ishex a]   ; output is $-A4FC in ASCII
hserout [isbin8 a\8]    ; output is $-11111100 in ASCII
```

(these are the low 8 bits i.e. 2 hex digits of the value).

Note: See also the Special Note re. Output Modifiers on page 77.

Output Only Modifiers (REAL, REP)

REAL

Converts a floating point value to ASCII characters, including sign and decimal point.

Syntax

real{#maxb} argument{\#maxa}

#maxb: optional maximum number of digits to pass **before** decimal point (default 10)

#maxa: optional maximum number of digits to display **after** decimal point (default 10)

Examples

If variable "y" contains the floating point value 123.45:

```
hserout [real y]
```

will send the ASCII characters 123.4500000000 to the hardware serial port.

```
hserout [real2 y]
```

will send the ASCII characters 23.4500000000 to the hardware serial port

```
hserout [real y\2]
```

will send the ASCII characters 123.45 to the hardware serial port.

```
hserout [real y\1]
```

will send the ASCII characters 123.4 to the hardware serial port (the number is truncated, not rounded).

Note: See also the Special Note re. Output Modifiers on page 77.

REP

Repeats a character multiple times.

Syntax

rep argument\n

n is the number of repetitions

Example

```
hserout [rep "–"\20]
```

will output the – character 20 times (could be used for underlining, as an example).

Special Note re. Output Modifiers

In addition to the examples given previously in this section, output modifiers can be used in assignment statements (e.g. a = b) to assign decimal, hex or binary ASCII characters or strings of characters to a variable or array.

Examples

If "x" and "y" are byte variables, and "y" contains the numeric value 8, then:

```
x = hex1 y
```

will assign the value 56 (which is the ASCII character for the number "8") to "x".

> Important: If the numeric value would result in a 2 or more digit ASCII number (including sign or indicator), the target variable ("x" in the above example) should be defined as an array.

For example,

```
var byte x(3)
x = dec3 y
```

would allow "x" to hold up to 3 decimal digits, covering all possible values of a single byte variable. Add an extra byte to "x" if you use a signed or indicated modifier, and two extra bytes if you use a combination modifier. So if "y" held the hex value $-4D

```
var byte x(4)
x = ishex2 y
```

would assign the values $24, $2D, $34 and $44 to successive locations of "x" (these are the ASCII values corresponding to $-4D).

Using REP to Preset an Array

The REP modifier can also be used to pre-set an array to any desired value. For example,

```
var byte a(20)
a = rep 0\20
```

will set all 20 elements of "a" to the numeric value 0 (zero). If you were, instead, to use

```
var byte a(20)
a = rep "0"\20
```

all 20 elements of "a" would be set to the numeric value 48 (i.e. the ASCII value of the character "0" (zero).

Input-only Modifiers (WAITSTR, WAIT, SKIP)

WAITSTR

Receives data until a continuous group matches the string contained in an array variable.

Syntax

waitstr string\length{\eol}

> *string* is the array to use for matching purposes
> *length* is the number of characters to match
> *eol* is the End Of Line character to watch for

Example

The following program excerpt will accept input from the hardware serial port until the characters "e", "n", "d" are received in sequence, or until an end of line (Carriage Return) character is received.

```
var byte x(30)
var byte y(3)
y = "end"           ; this puts the ASCII characters
                    ; e, n, and d into y(0), y(1) and
                    ; y(2) respectively
hserin [waitstr y\3\13 x]  ; 13 is the CR character
```

WAIT

Receives data until a continuous group matches the string constant included in the modifier.

WAIT is similar to WAITSTR except a string constant, rather than a pre-defined array, is used for matching.

Syntax

wait("constant string")

Example

The following program excerpt has a similar function to that shown in the previous section, except that an EOL test is not performed.

```
var byte a(30)
hserin [wait("end")]    ; parentheses are required
```

This program will accept input data for the array "a" until the three characters "e", "n" and "d" are received in sequence.

SKIP

Skips a specified number of input values. This is useful if your data is preceded by a label that should not be input.

Syntax

skip count

> *count* is the number of bytes to skip

Example

The following program excerpt inputs a two digit temperature and saves it as a numeric value.. Actual input has the format:

temperature: *nn*

where *nn* is the two digit temperature in decimal ASCII.

```
var byte temp
hserin [skip 13,a]      ; "temperature: " is 13
                        ; characters long
hserin [dec2 a]
```

This page intentionally left blank

Core BASIC Commands

This chapter includes the "normal" BASIC commands that are included with most versions of BASIC, as well as commands specific to Atom BASIC. Read it carefully: some familiar commands may be defined somewhat differently in Atom BASIC.

This chapter is divided into the following sections:

Conventions Used in this Chapter

{ ... }	represent **optional** components in a command. The { } are **not** to be included.
[...]	used for lists – the [ ] are **required**
(...)	used for some arguments. The ( ) are **required**

Assignment and Data Commands

= (LET)

Assigns the value of an expression or a variable to the target variable.

Syntax

{let} *target variable* = *expression*

> *expression* is any valid numeric expression, variable, or constant. The word LET is optional and may be included to improve readability.

Note: The target variable should have a number type sufficient to store the largest expected result of the expression. If the target variable is too small only the lowest significant bits will be assigned. See the examples below.

Examples

If x is a byte variable, and y has the numeric value 13,

```
x=y*5
```

will assign the value 65 to variable x.

Note, however, that using the same values,

```
x=y*20
```

will assign the value 4 to variable x. This is because the actual result, 260, is too large to be stored in one byte so only the 8 lowest significant bits are stored.

CLEAR

Sets all user RAM to zeros. This can be used to clear RAM after a reset (so that its state will be known) or within a program to clear all variables to zeros.

Note: Atom BASIC uses CLEAR differently from other basics.

Syntax

clear

LOOKDOWN

Lookdown checks through items in a list of variables looking for the first one that matches the specified criterion. The index number (beginning with 0) of the matching list item is passed to the target variable. The scan goes from left to right and stops as soon as the operator condition is met.

Note: If the condition is not met, the target variable will be unchanged.

Syntax

lookdown *value*,{*operator*,} [*list*],*target*

value is the variable or constant to be compared
operator is a comparison operator (see page 62) – default is "="
list is a list of constants or variables, up to 16 bits each
target is a variable to store the resulting index value

Examples

```
x var byte
y var byte
x = 120
lookdown x,>,[3,10,18,36,50,66,100,130,150,200,240],y
```

will set y = 6

Instead of numeric constants, defined constants, variables or array elements may be used in the list.

However, in the case of:

```
x var byte
y var byte
x = 120
lookdown x,=,[3,10,18,36,50,66,100,130,150,200,240],y
```

the condition is not met so the value of "y" will be unchanged.

LOOKUP

Uses an index number to select a value from a list. Each item in the list corresponds to a position of 0, 1, 2, 3... etc. in the list.

Note: If the index number exceeds the number of items in the list, the target variable will be unchanged.

Syntax

lookup *index*,[*list*],*target*

> *index* is the position (constant or variable) to be retrieved from the list
> *list* is a list of constants or variables, up to 16 bits each
> *target* is a variable to store the resulting list item

Examples

```
temp var byte
value var byte
temp = 2
lookup temp,[10,20,30,40,50,60,70,80,90],value
```

will set "value" = 30 (remember that counting starts from zero).

Summary of LOOKUP and LOOKDOWN

LOOKUP is, in effect, the converse of LOOKDOWN. Lookup takes an index and returns the corresponding value from a table, lookdown compares a value to elements in a table, and returns the corresponding index.

SWAP

Exchanges the values of two variables.

Note: The variables should be of the same size to prevent possible errors.

Syntax

swap *variable1*,*variable2*

Examples

```
ant var word
bat var word
ant = 3800
bat = 27
swap ant,bat
```

Now ant = 27 and bat = 3800.

Swap eliminates the need for an intermediate variable and shortens program length when compared with the alternative method shown below:

```
ant var word
bat var word
cat var word
ant = 3800
bat = 27
let cat = ant          ; now cat = 3800
let ant = bat          ; now ant = 27
let bat = cat          ; now bat = 3800
```

PUSH, POP

PUSH Stores a 32 bit value on the stack.

POP Retrieves a 32 bit value from the stack.

> Important: PUSH must always be matched by a subsequent POP instruction before any other stack-oriented commands (e.g. GOSUB, RETURN, EXCEPTION) are used.

Syntax

push *variable*
pop *variable*

variable may be of any type. For *push* the value will be padded with high order zeros to fill 32 bits if necessary. For *pop* the high order bits will be truncated if necessary to fit the variable.

Variable types should be matched for predictability. While it's possible, for example, to PUSH a long variable, and subsequently POP a word or byte variable, it's less confusing to stick to matched types. If you POP a word or byte variable only the low order bits will be stored.

Branching and Subroutines

GOTO

Unconditionally forces program execution to jump to the supplied label. The line following the GOTO command is not executed unless it is a label referenced from elsewhere in the program.

Syntax

goto *label*

> *label* is the label at which program execution should continue

Examples

This sample program shows one of many possible uses of the *goto* command:

```
variables and constants defined here
start               ; beginning of program
   program code here
goto start          ; loop back to start of program
firstsub            ; beginning of subroutine section
   subroutines here
```

BRANCH

BRANCH is an indexed form of GOTO. Branch uses an index number to choose from a list of labels, then jumps to that label.

Syntax

branch *index*,[*label1, label2, label3,...]*

> *index* is a variable or constant pointing within the list of labels, with counting starting at zero.
> *labels* are any valid labels in your program

If "index" is greater than the number of labels in the list, no jump will occur and program execution will continue with the next line.

Examples

If the value of variable "test" is 3,

```
branch test,[hot, cold, raise, lower, adjust, terminate]
```

will cause program execution to jump to the line labeled "lower".

GOSUB... RETURN

> Atom Pro Basic now supports parameter passing to subroutines. See below.

GOSUB stores a return address on the "stack" and jumps to the specified label (which should be the label of a subroutine). The subroutine must end with a RETURN command.

RETURN retrieves and removes from the stack the address stored by GOSUB, and resumes program execution on the line following the original GOSUB command.

> Important: Subroutines should exit via the RETURN command which clears the saved address from the stack[9]. If multiple exit points are required from a subroutine, use the EXCEPTION command described below. Do not use BRANCH or GOTO to exit a subroutine.

Syntax

gosub *label*

label is the label of any valid subroutine in the program

return

or

gosub *label [parameters] [,returned value]*

parameters are values to be passed as arguments for the subroutine

returned value will contain the value passed back by the subroutine

return *returned value*

Note: All variables are global in scope.

---

[9] If a subroutine exits without using RETURN or EXCEPTION the saved address will remain on the stack. If such subroutines are executed many times the stack may overflow.

Example 1

```
val var word
weightmin var word
weightmax var word
start
   code to calculate minimum weight
   hserout ["Minimum "]
   val = weightmin
   gosub outvaldec
   code to calculate maximum weight
   hserout ["Maximum "]
   val = weightmax
   gosub outvaldec
goto start
outvaldec
   hserout ["weight is ",dec val," mg",13]
return
```

The program calculates a minimum and maximum weight (perhaps using sensors) and displays output on a serial terminal in the format:

```
   Minimum weight is 15 mg
   Maximum weight is 32 mg
```

The intermediate variable *val* is used to pass the output value.

Example 2

```
returnval var sbyte

gosub myfunction[3,1,5],returnval ;returnval = -1

arg1 var sbyte
arg2 var sbyte
arg3 var sbyte
result var sbyte
myfunction [arg1,arg2,arg3]
   result = arg1+arg2-arg3
   return result
```

This example calculates a result based on the passed parameters, and returns the result (-1 in the example) as *returnval*.

Note: result *and* returnval *need not have the same name.*

Example 3

You can use just the return argument:

```
gosub myfunc2 returnval
myfunc2
   return 3
```

Example 4

Or you can use just the arguments:

```
gosub myfunc3[3,2]
arg1 var sbyte
arg2 var sbyte
myfunc3[arg1,arg2]
   high arg1
   low arg2
   return
```

EXCEPTION

If multiple exit points are needed from a subroutine, all but the last should use the EXCEPTION command. EXCEPTION differs from RETURN as follows:

RETURN	Retrieves the saved address from the stack, clears the address from the stack, and sets program execution to the line following the GOSUB command.
EXCEPTION	Clears the return address from the stack, and resumes program execution at the label given.

Syntax

exception *label*

label is the label at which program execution should continue

Examples

```
val var word
weightmin var word
weightmax var word
start
   code to calculate minimum weight
   hserout ["Minimum "]
   val = weightmin
   gosub outvaldec
   code to calculate maximum weight
   hserout ["Maximum "]
   val = weightmax
```

```
   gosub outvaldec
goto start
outvaldec           ; start of subroutine
   if weightmin > 5 then continue
   exception start   ; value is too low – do again
continue
      hserout ["weight is ",dec val," mg",13]
return
```

This program is similar to the one under GOSUB but provides an "escape" from the subroutine if the minimum weight is too low.

IF... THEN... ELSEIF... ELSE... ENDIF

This set of commands provides conditional GOTO and/or GOSUB capability. The IF... THEN commands can be used in two formats: simple and extended.

Syntax – Simple Format

if *comparison* then *label*

> *comparison* is a statement that can be evaluated as true or false, for example x = 7, temp <> 13, etc.
> *label* marks the program line which will be executed next if the comparison is true

The comparison is evaluated. If it is true, program execution passes to the line marked by the specified label. If it is false program control continues with the next line following the IF... THEN line.

if *comparison* then gosub *label*

Behaves as above except that a GOSUB rather than a GOTO is performed if the comparison is true.

Example

```
a var byte
   statements to set value of a
if a > 35 then limit
   statements will execute if a <= 35
limit
   statements
```

If a <= 35 the statements immediately following the IF... THEN line will execute. Otherwise control will jump to the label "limit".

Syntax – Extended Format

if *comparison1* then

statements (executed if comparison1 is true, then jumps to the line following ENDIF. If false, jumps to ELSE or ELSEIF.)

elseif *comparison2* then

statements (executed if comparison 1 is false but comparison 2 is true, then jumps to the line following ENDIF. If false, jumps to the next ELSEIF or to ELSE.)

else

statements (executed if neither comparison1 nor comparison2 is true)

endif

Note that elseif and else are optional. See the examples below.

Examples

```
ant var byte
bat var word
array var byte(20)
{code setting value of a}
if ant < 5 then
   array = "small"
   bat = 100
elseif ant < 10
   array = "medium"
   bat = 1000
else
   array = "big"
   bat = 10000
endif
```

If the first comparison (ant < 5) is true the next two statements are executed and then program execution passes to the line following ENDIF.

If the first comparison is false, the next two statements are skipped and program execution passes to the "elseif" line.

Note: Multiple "elseif" lines may be included if necessary

If the second comparison is true, the next two lines are executed and program execution then passes to the line following ENDIF.

If the second comparison is false, the next two lines are skipped and program execution passes to the "else" line.

The statements following the "else" line are executed until the "endif" is reached.

```
ant var byte
bat var word
start
   code setting value of ant
if ant < 5 then small
elseif ant < 10 then medium
endif
goto big
small
   code to process small value
goto start
medium
   code to process medium value
goto start
big
   code to process big value
goto start
```

This code provides a 3 way "switch" depending on the value of "ant". The line following "goto big" should be a label referenced from elsewhere in the program or it will not be executed.

Looping Commands

Looping commands repeat a number of lines (instructions) multiple times, depending on certain conditions.

Command	Repeats	Condition tested
for... next	defined number of times	at beginning of loop
do... while	until false	at end of loop
while... wend	until false	at beginning of loop
repeat... until	until true	at end of loop

FOR... NEXT

Repeats the instructions between FOR and NEXT a predefined number of times.

Syntax

for *counter* = *startvalue* to *endvalue* {step *stepvalue*}

statements to be executed

next

counter is a variable used to hold the current counter value
startvalue is the initial value of the loop counter
endvalue is the final value of the loop counter
stepvalue is the optional increment or decrement

These values may be bit, nibble, byte, word, long or float. *Startvalue, endvalue* and *stepvalue* may be variables or constants.

If STEP is omitted a *stepvalue* of 1 is automatically assigned.

Stepvalue may be negative in which case the counter will be decremented rather than incremented. The loop will continue until the counter value falls outside the range set by *endvalue*.

Note: Unlike some BASICs, "next" does not have an argument in Atom BASIC, i.e. the form "next x" is not valid.

> Take care not to modify the value of *counter* using statements within the loop. This can cause unpredictable operation, and the loop may never end.

Examples

```
ant var byte
bat var byte(11)
for ant = 1 to 10
bat(ant) = ant * 20
next
```

This simple loop will store values in the array variable "bat" as follows:

bat(0) = unchanged, bat(1) = 20, bat(2) = 40... bat(10) = 200

```
a var word
for a = 10 to 20 step 5
{statements}
next
```

The statements will be executed 3 times with a = 10, a = 15 and a = 20. The value of "a" is incremented and tested at the end of the loop.

```
a var word
for a = 10 to 20 step 6
{statements}
next
```

The statements will be executed twice with a = 10 and a = 16.

```
a var sword
for a = 40 to 20 step -5
{statements}
next
```

The statements will be executed 5 times with a = 40, 35, 30, 25 and 20 respectively.

DO... WHILE

> Important: You must not nest a WHILE… WEND inside a DO… WHILE. A compiler error will result.

Repeats a set of instructions as long as a given condition remains true (i.e. until the given condition becomes false).

The condition is tested **after** the instructions have been executed. The instructions will be executed once even if the condition is initially false (see the second example below).

Syntax

do
    statements
while *condition*

> *condition* is any valid combination of variables, constants and logical operators.

Examples

```
a var word
a = 5
do
   a = a * 2
   hserout [dec a]
while a < 100
statements
```

The loop operates as follows:

Pass	Output (a)	Test result
1	10	true
2	20	true
3	40	true
4	80	true
5	160	false

Since the test is done at the end of the loop, the final value is output even though it is greater than 100. Program execution continues with the line following "while".

```
a var word
a = 150
do
   a = a * 2
   hserout [dec a]
while a < 100
statements
```

The loop will operate once, and output the value 300, even though the initial value is not less than 100. This is because the test is done at the end of the loop.

WHILE... WEND

> Important: You must not nest a WHILE… WEND inside a DO… WHILE. A compiler error will result.

Repeats a set of instructions as long as a given condition remains true (i.e. until the given condition becomes false).

The condition is tested **before** the instructions are been executed. If the condition is initially false, the instructions will never be executed.

Syntax

while *condition*
    program statements
wend

condition is any valid combination of variables, constants and logical operators.

Examples

```
a var word
a = 5
while a < 100
   a = a * 2
   hserout [dec a]
wend
program continues
```

The loop operates as follows:

Pass	Initial (a)	Test result	Output
1	5	true	10
2	10	true	20
3	20	true	40
4	40	true	80
5	80	true	160
6	160	false	none

On pass number 6 the test is false so the loop is not executed. Program execution continues with the line following WEND. The results are similar to the DO... WHILE loop shown above.

The following example illustrates a difference between the DO... WHILE and WHILE... WEND loops.

```
a var word
a = 150
while a < 100
   a = a * 2
   hserout [dec a]
wend
program continues
```

Unlike the DO... WHILE loop, the WHILE... WEND tests before the loop statements are executed. Since the condition is false initially, the loop is never executed and control passes to the statements following WEND. (Contrast this with the DO... WHILE loop which executes once in a similar situation.)

REPEAT... UNTIL

Repeats a set of instructions until a given condition becomes true (i.e. as long as the condition remains false).

The condition is tested **after** the instructions have been executed. The instructions will be executed once even if the condition is initially true. REPEAT... UNTIL is essentially the converse of DO... WHILE.

Syntax

repeat
    program statements
until *condition*

> *condition* is any valid combination of variables, constants and logical operators.

Examples

```
a var word
a = 5
repeat
   a = a * 2
   hserout [dec a]
```

```
until a > 100
program continues
```

The loop operates as follows:

Pass	Output (a)	Test result
1	10	false
2	20	false
3	40	false
4	80	false
5	160	true

Program execution then continues with the line following UNTIL.

If the initial value of a is greater than 100, the loop will be executed once because the test is at the end of the loop.

Input/Output Commands

Since the BasicATOM Pro is not normally used with a computer display, the input/output commands are highly specialized and do not duplicate those of conventional BASICs. In place of the usual PRINT, LPRINT, PRINT#, etc. commands, Atom BASIC provides a range of input/output commands for various devices commonly used with microcontrollers.

> Many of the I/O commands in this section accept the use of *command modifiers*. See 0Command Modifiers on page 69 for more information.

DEBUG

Sends output to the Debug Watch Window in the IDE.

Syntax

debug [{mod}expr1,{mod}expr2, … (mod)expr*N*]

> *mod* is any valid output modifier (*only* dec, hex, bin *and* real)
> *expr* is a variable, constant or expression generating data to be sent. The length of this list is limited only by available memory.

Notes

The *debug* command is useful only when your program is run in "debug" mode from the IDE. It provides an easy way to output the values of variables during program execution.

The debug watch window expects all output to be in ASCII characters. If variables are output directly without modifiers, their values will be interpreted as ASCII, which may give unexpected results. Word and long variables will output only the low order 8 bits unless a suitable modifier is used to convert to decimal, hex or binary.

The debug watch window accepts certain terminal commands including (but not limited to) the following:

Character	decimal value	function
NUL	0	clear screen
BEL	7	ring bell
BS	8	backspace

LF	10	new line
CR	13	new line

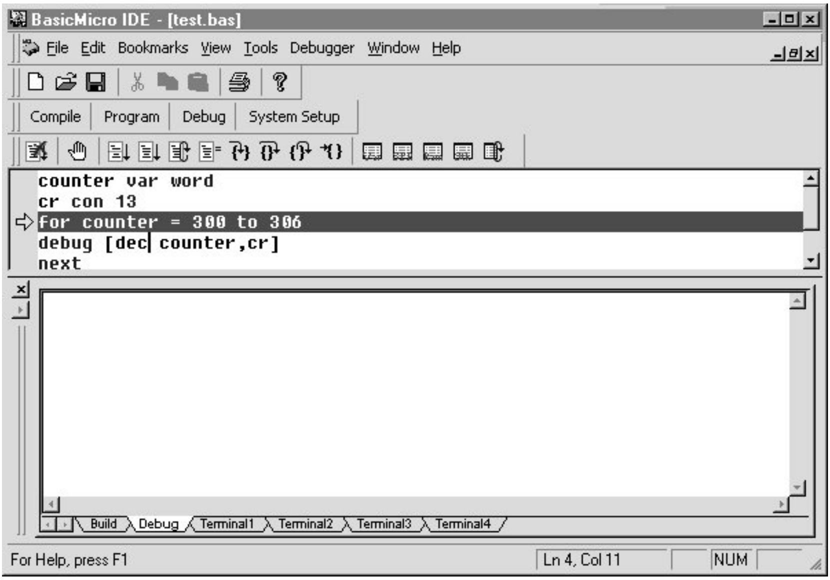
A more complete list will be found in the IDE documentation.

Example

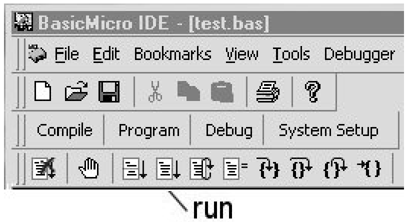
We'll use the following program to test the *debug* command.

```
counter var word
cr con 13
for counter = 300 to 306
debug [dec counter,cr]
next
```

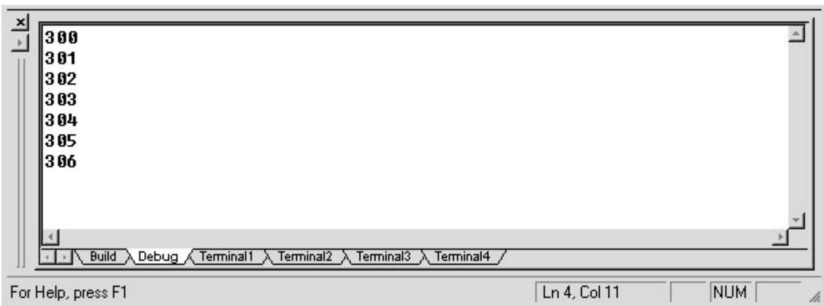
We first type in the program using the IDE (which must be connected to the BasicATOM Pro). Then click the DEBUG button. The program should compile with no errors, and the Atom Pro will be programmed. After this process you'll see a screen like this one:



Now click on the RUN button:



Your program should run and produce the following output:



After it runs, the Atom Pro will go to sleep and stop responding. To run your program again, simply press the RESET button on the Atom development board.

DEBUGIN

Accepts keyboard input from the Debug Watch Window. See the example under DEBUG which shows how to invoke this window.

Syntax

debugin [{mod}var1,{mod}var2, … (mod)var*N*]

> *var* is a variable that tells DEBUGIN what to do with incoming data. A comma delimited list or variables is supported.
>
> The list is of the form [{mod} var1, {mod} var2... {mod} varN] where *mod* is an optional input modifier (*only* dec, hex, bin *and* real) and *var* is a variable of the appropriate size.

Notes

In the absence of *modifiers* DEBUGIN assigns each keystroke to a single variable. Program execution will wait until all variables have input; there is no timeout with the DEBUGIN command.

See the example under HSERIN, below, for details about the use of input modifiers, delimiting characters, etc.

Example

```
counter var word
start var word
temp var byte
cr con 13
start=300
loop
   debugin [temp]       ; wait for any keystroke
   for counter = start to start+6
      debug [dec counter,cr]
   next
   start=counter
go to loop
```

In this example the DEBUGIN command is used simply to pause program execution. The *temp* value is echoed to the screen, but is otherwise ignored.

The program will output six numbers in sequence, starting with 300. Then it will wait for any key to be pressed before displaying the next six numbers.

Of course, DEBUGIN can be used to assign values to variables in exactly the same way as other input commands, such as HSERIN, SERIN, etc. The rest of this section has several helpful input examples.

HSERIN

This command accepts input via the hardware serial port. Before using this command the ENABLEHSERIAL compiler directive must be in the program, and you must use the SETHSERIAL command (see page 107) to set the correct baud rate, data bits, parity and stop bits. HSERIN is similar in operation to SERIN (see page 109) except that it uses the hardware serial input pin RXD (see pinouts on page 187) and the baudrate is set by the SETHSERIAL command.

Syntax

hserin {timeout,tlabel,}[{mod}var1,{mod}var2, … (mod)var*N*]

> *timeout* is an optional expression which specifies the time in milliseconds to wait for data to be received

tlabel is the target label where program execution continues if *timeout* is exceeded. *tlabel* must be specified if a *timeout* value is set.

mod is any valid input modifier (optional).

var is a variable or list of variables (comma delimited) where data will be stored.

Example

In the following example, "illegal" characters are used as delimiters in the input data stream.

```
ant var word
bat var word
cat var word
dog var word
sethserial h2400,h8databits,hnoparity,h1stopbits
hserin 10000,nomore,[dec ant,bat,cat,hex dog]
continues here once all data is received
nomore
continues here if no data is received for 10 seconds
```

The port is set to 2400 baud, 8 bit, no parity, with 1 stop bit.

Input data will be converted from ASCII decimal to numeric form and assigned to "ant" until a non-numeric character is received. That character will be discarded.

The next two input bytes will be assigned to "bat" and "cat" respectively. (Each unmodified input byte is assigned to one variable.)

Following data will be converted from ASCII hex to numeric form and assigned to "dog" until a non-hex character is received. That character will be discarded.

If 10 seconds elapses with no new data, program execution will jump to the label *nomore*.

For example, if the input data stream contains the following bytes starting at the left (shown in hex and ASCII format)

hex	31	38	2C	61	62	32	44	39	2C
ASCII	1	8	,	a	b	2	D	9	,

- "ant" will be assigned the numeric value 18
- the "," will terminate input for "ant" and be discarded

- "bat" will be assigned the numeric value 97 (i.e. 61 hex)
- "cat" will be assigned the numeric value 98 (i.e. 62 hex)
- "dog" will be assigned the numeric value 729 (i.e. 2D0 hex)
- the "," will terminate input for "dog" and be discarded

Example

In the following example, the input data stream must be pre-formatted into the correct number of bytes for each variable.

```
ant var word
bat var word
cat var word
sethserial h2400,h8databits,hnoparity,h1stopbits
hserin [dec4 ant\4, bat, hex3 cat\3]
```

This format expects exactly 4 ASCII decimal digits (which will be converted to a number and assigned to "ant"), followed by 1 numeric byte (which will be assigned directly to "bat" with no conversion), followed by exactly 3 ASCII hex digits (which will be converted to a number and assigned to "cat").

HSERIN2 (Atom Pro Plus only)

This command reads data from the second hardware serial port available on the Atom Pro Plus. In operation it is almost identical to the HSERIN command, and shares the same syntax.

Syntax

hserin2 {timeout,tlabel,}[{mod}var1,{mod}var2, … (mod)var*N*]

> *timeout* is an optional expression which specifies the time in milliseconds to wait for data to be received
>
> *tlabel* is the target label where program execution continues if *timeout* is exceeded. *tlabel* must be specified if a *timeout* value is set.
>
> *mod* is any valid input modifier (optional).
>
> *var* is a variable or list of variables (comma delimited) where data will be stored.

Notes

Before using this command the ENABLEHSERIAL2 compiler directive must be in the program, and you must use the SETHSERIAL2 command (see page 109) to set the correct baud rate, data bits, parity and stop bits. HSERIN2 uses the hardware input pin RXD_2 (see pinouts on page 187).

HSEROUT

This command sends output to the hardware serial port. Before using this command the ENABLEHSERIAL compiler directive must be in the program, and you must use the SETHSERIAL command (see page 107) to set the port parameters. HSEROUT is similar in operation to SEROUT (see page 111).

Syntax

hserout [{mod}exp1,{mod}exp, …{mod}exp*N*]

> *mod* is any valid output modifier
>
> *exp* is an expression or list of expressions (comma delimited) generating data to be sent.

Example

```
ant var byte
bat var byte
cat var byte
ant=65                  ; hex 41
bat=99                  ; hex 63
cat=66                  ; hex 42
sethserial h2400,h8databits,hnoparity,h1stopbits
hserout [dec ant,bat,hex4 cat\4]
```

The port is set to 2400 baud, 8 bit, no parity, with 1 stop bit.

The output will be the following:

hex	06	06	63	30	30	34	32
ASCII	6	5	c	0	0	4	2

Remember that "hex4 c\4" specifies that the output will be exactly 4 hex digits.

HSEROUT2

This command sends data to the second hardware serial port available on the Atom Pro Plus. In operation it is almost identical to the HSEROUT command, and shares the same syntax.

Syntax

hserout2 [{mod}exp1,{mod}exp, …{mod}exp*N*]

mod is any valid output modifier

exp is an expression or list of expressions (comma delimited) generating data to be sent.

Notes

Before using this command the ENABLEHSERIAL2 compiler directive must be in the program, and you must use the SETHSERIAL2 command (see page 109) to set the correct baud rate, data bits, parity and stop bits. HSEROUT2 uses the hardware output pin TXD_2 (see pinouts on page 187).

ENABLEHSERIAL

This is a compiler directive that enables the hardware serial port system (SCI3) on the Atom Pro and Atom Pro Plus (RXD/TXD).

Syntax

enablehserial

ENABLEHSERIAL2 (Atom Pro Plus only)

This is a compiler directive that enables the hardware serial port system (SCI3_2) on the Atom Pro Plus (RXD_2/TXD_2)

Syntax

enablehserial2

SETHSERIAL

Sets the baud rate, data bits, parity and stop bits for the hardware serial port (SCI3), initializes the serial buffers and enables the hardware serial port interrupt handler. This command must be executed before hserin or hserout are used.

> The ENABLEHSERIAL compiler directive must be in your program for this command to function properly.

Note: When using the hardware serial system the interrupts for the hardware serial port should not be used except by advanced programmers.

Syntax

sethserial baudrate, databits, parity, stopbits

baudrate is any of the following:[10]

H300	H12000	H26400	H57600
H600	H14400	H28800	H62500
H1200	H16800	H31200	H125000
H2400	H19200	H33600	H250000
H4800	H21600	H36000	H312500
H9600	H24000	H38400	H500000

databits can be either of the following constants:

H8DATABITS, H7DATABITS

parity can be any of the following constants:

HNOPARITY, HEVENPARITY, HODDPARITY

stopbits can be either of the following constants:

H1STOPBITS, H2STOPBITS

Examples

See hserin and hserout for examples.

---

[10] The values in this list are predefined constants having the appropriate numeric values for the respective baud rates.

SETHSERIAL2 (Atom Pro Plus only)

Sets the baud rate, data bits, parity and stop bits for the second hardware serial port (SCI3_2), initializes the serial buffers and enables the hardware serial port interrupt handler. This command must be executed before hserin2 or hserout2 are used.

> The ENABLEHSERIAL2 directive must be in your program for this command to function properly.

Syntax

Syntax is identical to SETHSERIAL.

Notes

The SETHSERIAL2 command is identical in operation to SETHSERIAL except that it applies to the second hardware serial port (TXD_2, RXD_2).

SERIN

This command receives serial input (i.e. asynchronous RS-232 data) through a specified I/O pin.

Syntax

serin rpin{\fpin},baudmode,{plabel,}{timeout,tlabel,}[InputData]

rpin is a variable or constant that specifies the I/O pin through which the serial data will be received. This pin will switch to input mode and remain in that state after the end of the instruction.

\fpin is an optional variable or constant that specifies the I/O pin that will be used for flow control (the "\" is required). This pin will switch to output mode and remain in that state after the end of the instruction.

Flow control is provided for use primarily with PCs and conforms to PC serial port standards.

baudmode is a 16 bit variable or constant that specifies serial timing and configuration. See the description under *Notes*.

plabel is an optional label. The program will jump to *plabel* if there is a parity error.

timeout is an optional 32 bit variable or constant that specifies the time to wait for incoming data in milliseconds. If data does not arrive within this time, the program will jump to *tlabel*.

InputData is a list of variables and modifiers that tells SERIN what to do with incoming data. See the examples under HSERIN (page 101) for a detailed description of this list.

Notes - Baudmode

The *baudmode* value is built as follows:

bit	15	14	13	12-0
function	not used for SERIN	polarity 0 = normal 1 = inverted	data/parity 0 = 8 bits, no parity 1 = 7 bits, even parity	bit period

Note: "polarity" applies to both data and flow control.

Programmers will not normally "build" this value themselves. The two preferred methods are:

- Use a predefined constant from the list below, or
- Use the SERDETECT command to automatically produce the required value as a variable.

Baudmode predefined constants

Note: You may equally well use the baudmode constant described under SEROUT for the SERIN command. The extra letter (O) will be ignored for SERIN.

The constants consist of 1 or 2 letters, in the order shown below, followed by the baud rate:

N indicates "normal" data and flow control[11]
I indicates "inverted" data and flow control
E indicates "even parity, 7 bits", else "no parity, 8 bits"

[11] Note: "normal" data for RS232 uses LOW (negative) for 1 and HIGH (positive) for 0.

Baud rate may be any one of 300, 1200, 2400, 4800, 9600, 14400, 19200, 28800, 33600, 38400 or 57600

Either N or I (not both) **must** be used as the first letter of the constant. E is optional. If E is not used, baudmode defaults to no parity, 8 bit data.

For example, the constant "NE2400" indicates non-inverted data, 7 bits even parity, 2400 baud. The constant "I19200" indicates inverted data, 8 bits no parity, 19,200 baud.

Note: You can confirm the syntax of your constant by checking the List of Reserved Words on page 191.

> Important: At least "N" or "I" must precede the baud rate or the constant will simply be taken as a number, which is invalid for this application.

Examples

This example is modified from the example given in HSERIN. See that example for detailed explanation of the data list.

```
ant var word
bat var word
cat var word
dog var word
serin P3\P4,NE2400,5000,expd,[dec ant,bat,cat,hex dog]
program continues here
...
expd ; jumps here if timeout
timeout processing
```

Serial input is on I/O pin 3, with pin 4 used for flow control. Data format is non-inverted, even parity, 7 bits, 2400 baud. Input will wait for 5 seconds (5000 ms) between bytes, and jump to "expd" if that time is exceeded with no data available.

SEROUT

This command sends serial output (i.e. RS232 asynchronous data) through a specified I/O pin. SEROUT can be used in two modes: with flow control or with timed intervals between bytes.

Note: Flow control is provided for use primarily with PCs and conforms to PC serial port standards.

Syntax

With timed intervals:

serout tpin,baudmode,{pace,}[OutputData]

With flow control:

serout tpin\fpin,baudmode,{timeout,tlabel,}[OutputData]

rpin is a variable or constant that specifies the I/O pin through which the serial data will be sent. This pin will switch to output mode and remain in that state after the end of the instruction.

\fpin is an optional variable or constant that specifies the I/O pin that will be used for flow control (the "\" is required). This pin will switch to input mode and remain in that state after the end of the instruction.

baudmode is a 16 bit variable or constant that specifies serial timing and configuration. See the description under *Notes*.

pace is an optional variable or constant (0 – 65535) that tells SEROUT how many milliseconds to wait between transmitting bytes. If *pace* is omitted, there will be no delay between bytes. Flow control is preferable to fixed output timing: *pace* is provided for use with peripherals that don't support flow control. Normally either flow control or delay is used, not both.

timeout is an optional 16 bit variable or constant that specifies flow control timeout in milliseconds. If data is halted by the receiving device for longer than this time, the program will jump to *tlabel*.

OutputData is a list of variables and modifiers that tells SEROUT what to do with outgoing data. See the examples under HSEROUT (page 106) for a detailed description of this list.

Notes - Baudmode

Baudmode for SEROUT is the same as *baudmode* for SERIN with the exception of bit 15. The *baudmode* value is built as follows:

bit	15	14	13	12-0
function	output state	polarity 0 = normal 1 = inverted	data/parity 0 = 8 bits, no parity 1 = 7 bits, even parity	bit period

Note 1: If the value of "output state" is 0, the output pin will be driven for both high and low states. If the value is 1, the output pin will be driven for low, and open drain for high (requires external pullup).

Note 2: "polarity" applies to both data and flow control.

Programmers will not normally "build" this value themselves. The two preferred methods are:

- Use a predefined constant from the list below, or
- Use the SERDETECT command to automatically produce the required value as a variable (this only works with bi-directional peripherals that can send as well as receive serial data).

Baudmode predefined constants

Note: The SEROUT baudmode *constants may also be used for SERIN. The "O", which sets bit 15, will simply be ignored for SERIN.*

The constants consist of 1, 2 or 3 letters, in the order shown below, followed by the baud rate:

N indicates "normal" data and flow control[12]
I indicates "inverted" data and flow control
E indicates "even parity, 7 bits", else "no parity, 8 bits".
O indicates open drain, else both high and low are driven.

Either N or I (not both) **must** be used as the first letter of the constant. E is optional. If E is not used, baudmode defaults to no parity, 8 bit data. O is also optional, if not used both high and low states are driven.

Baud rate may be any one of 300, 1200, 2400, 4800, 9600, 14400, 19200, 28800, 33600, 38400 or 57600

For example, the constant "NE2400" indicates non-inverted data, 7 bits even parity, 2400 baud. The constant "IO19200" indicates

[12] Note: "normal" data for RS232 uses LOW (negative) for 1 and HIGH (positive) for 0.

inverted data, 8 bits no parity, 19,200 baud, with open drain for the high state (which is data "1" in this case).

Note: You can confirm the syntax of your constant by checking the List of Reserved Words on page 191.

> Important: At least "N" or "I" must begin the constant or it will simply be taken as a number, which is invalid for this application.

Examples

This example is modified from the example given in HSEROUT. See that example for detailed explanation of the data list.

```
ant var byte
bat var byte
cat var byte
ant=65                 ; hex 41
bat=99                 ; hex 63
cat=66                 ; hex 42
serout P5\P6,NEO2400,5000,expd,[dec ant,bat,hex4 cat\4]
program continues here
...
expd ; jumps here if timeout
timeout processing
```

Serial output is on I/O pin 5, with pin 6 used for flow control. Data format is non-inverted, even parity, 7 bits, 2400 baud, open drain on high bits. The ATOM PRO will wait for a maximum of 5 seconds between bytes; if the receiving device is not ready (as determined by the flow control pin) after that time program execution will jump to "expd".

SERDETECT

Used to auto-detect baud rates and build the "baudmode" value used with SERIN and SEROUT

Syntax

serdetect pin,mode,var

> *pin* is a variable or constant that specifies the I/O pin that will be used to receive the sync character. This pin will switch to

input mode and remain in that state after the end of the instruction.

var is a word variable used to store the resulting baudmode value.

mode determines the setting for bits 15, 14 and 13 of the baudmode variable (see SERIN and SEROUT for details of these bits). For convenience, mode may use one of the following predefined constants:

bit 15	bit 14	bit 13	constant	description
0	0	0	NMODE	both driven, normal, 8 bit no par
0	0	1	NEMODE	both driven, normal, 7 bit even
0	1	0	IMODE	both driven, inverted, 8 bit no par
0	1	1	IEMODE	both driven, inverted, 7 bit even
1	0	0	NOMODE	open drain, normal, 8 bit no par
1	0	1	NEOMODE	open drain, normal, 7 bit even
1	1	0	IOMODE	open drain, inverted, 8 bit no par
1	1	1	IEOMODE	open drain, inverted, 7 bit even

Notes

SERDETECT is used to auto detect an incoming baud rate. This is ideal for applications or peripherals that can be used at different baud rates since it allows software switching of the Atom Pro. SERDETECT eliminates the need for switches or jumpers to select baud rates.

Note: For bi-directional devices, such as a PC serial port, the value may also be used for sending data after the detection is made.

SERDETECT works by measuring the length of one bit in the first received character. The sending device must send one of the following characters (X = don't care):

Normal data %XXXXX101 (binary)

Inverted data %XXXXXX01 (binary)

A short delay (or suitable flow control) after this byte will allow the SERDETECT command to be processed.

Once the time has been calculated, SERDETECT combines this with bits 15 – 13 as specified by the *mode* value to generate the correct value for use in *baudmode* with SERIN and SEROUT.

Examples

This example is the same as that given under SERIN except that SERDETECT is used to set baud rate.

```
ant var word
bat var word
cat var word
dog var word
baudset var word
serdetect P3,nemode,baudset
serin P3\P4,baudset,5000,expd,[dec ant,bat,cat,hex dog]
program continues here
...
expd ; jumps here if timeout
timeout processing
```

The SERDETECT command will "build" the correct value for baud rate and parameters, and save it as "baudset", which is then used in SERIN in place of a pre-determined *baudmode* parameter.

I2CIN

Receives data from an I^2C device such as an EEPROM, external A/D converter, etc.

Syntax

i2cin DataPin,ClockPin,{ErrLabel,}Control,{Address,}[varlist]

> *DataPin* is a variable or constant that specifies the I/O pin to use for SDA (serial data). This pin will switch to input mode and remain in that state after the end of the instruction.
>
> *ClockPin* is a variable or constant that specifies the I/O pin to use for SCL (serial clock). This clock is generated by the BasicATOM Pro. This pin will switch to output mode and remain in that state after the end of the instruction.
>
> *ErrLabel* is a label that the program will jump to if the I2CIN command fails (e.g. the device is disconnected, turned off, etc.)

Control is a variable or constant that specifies the I^{2}C device's control byte. This byte is defined as follows:

bits 7 – 4	Device type. For serial EEPROMs this should be %1010. For other I^{2}C peripherals, refer to the documentation of the peripheral.
bits 3 – 1	Device ID. You can address up to 8 devices on the same I^{2}C bus simultaneously. For example, if the address lines (A0 – A2) of a serial EEPROM are grounded, these bits should be %000.
bit 0	Addressing format. 0 = 8 bit addressing 1 = 16 bit addressing

Address is an optional variable or constant that specifies the starting address to read from (default is 0). This value should be 8 or 16 bits as set by bit 0 of the *Control* byte (see above).

Varlist is a list of modifiers and variables that tells I2CIN what to do with incoming data. See the examples under HSERIN (page 101) for a detailed description of this list.

Note: An EEPROM read address is automatically incremented with each byte read.

Notes

This manual does not attempt to document or describe the I^{2}C protocol in any detail. Users are advised to consult available sources for that information.

I^{2}C is a two-wire synchronous serial protocol used to communicate with a variety of peripherals such as EEPROMs, A/D converters, etc. I^{2}C is similar to SMBus and the two may normally be used interchangeably.

I^{2}C is a master/slave protocol with the master being able to address the various slave devices. This allows multiple slaves to share the same bus. Each slave must have a unique address.

In I^{2}C applications the BasicATOM Pro is always a Master.

Example

```
ant var byte
bat var byte
```

```
cat var byte
dog var byte
control var byte
address var byte
control=%10100000
address=$100
i2cin P3,P4,fail,control,address,[ant,bat,cat,hex dog]
program continues here
...
fail                ; jumps here if error
error processing
```

This program will read 4 bytes from an EEPROM, starting at address $100, and assign them to variables ant, bat, cat and dog. The fourth byte is assumed to be in ASCII hex format, and will be converted to numeric format. The other bytes are assumed to already be in numeric format.

The serial EEPROM has a device address of %000 (this is important if there are multiple serial EEPROMS on the same I^2C bus).

If communications fails for any reason (usually device not connected or powered on) program execution will jump to the label "fail".

I2COUT

Sends data to an I^2C device such as an EEPROM, external A/D converter, etc.

Syntax

i2cout DataPin,ClockPin,{ErrLabel,}Control,{Address,}[varlist]

> *DataPin* is a variable or constant that specifies the I/O pin to use for SDA (serial data). This pin will switch to output mode and remain in that state after the end of the instruction.
>
> *ClockPin* is a variable or constant that specifies the I/O pin to use for SCL (serial clock). This clock is generated by the BasicATOM Pro. This pin will switch to output mode and remain in that state after the end of the instruction.
>
> *ErrLabel* is a label that the program will jump to if the I2COUT command fails (e.g. the device is disconnected, turned off, etc.)

Control is a variable or constant that specifies the I²C device's control byte. This byte is defined as follows:

bits 7 – 4	Device type. For serial EEPROMs this should be %1010. For other I²C peripherals, refer to the documentation of the peripheral.
bits 3 – 1	Device ID. You can address up to 8 devices on the same I²C bus simultaneously. For example, if the address lines (A0 – A2) of a serial EEPROM are grounded, these bits should be %000.
bit 0	Addressing format. 0 = 8 bit addressing 1 = 16 bit addressing

Address is an optional variable or constant that specifies the starting address to write to (default is 0). This value should be 8 or 16 bits as set by bit 0 of the *Control* byte (see above).

Varlist is a list of modifiers and variables that tells I2COUT what data to output. See the example under HSEROUT (page 106) for a more detailed description of this list.

Note: An EEPROM write address is automatically incremented with each byte sent.

Notes

The I²C protocol is briefly described under *Notes* on page 117.

> In I²C applications the BasicATOM Pro is always a Master.

Serial EEPROMs use an input buffer to store data before it is written, since the writing process is typically slower than the I²C data transfer. The size of this input buffer is specified on the EEPROM data sheet. You **must not** exceed the buffer size in a single I2COUT command or data will be lost.

Once you have output one buffer's worth of data, you must wait the appropriate time for the data to be written before issuing another I2COUT command. This time is specified on the EEPROM data sheet.

> Refer to the EEPROM data sheet to determine buffer size and writing time.

See the examples below for one possible implementation of this procedure.

Example

```
a var byte(128)
control con %10100000
count1 var byte
count2 var byte
temp var byte
temp=0
code to populate a(0) to a(127)
for count1 = 1 to 8
   for count2 = temp to temp+16
      i2cout P3,P4,failed,control,[a(temp)]
   next
   pause 1600       ; delay to allow writing
   temp = count2
next
program continues here
failed
   executed if connection fails
```

This program first populates an array with 128 bytes of data, then writes the data to an external serial EEPROM.

The I^2C uses P3 for data, P4 for clock, and sends to an EEPROM with device number 0, using 8 bit data. The EEPROM has a 16 byte buffer and requires 100 ms to write each byte, or 1600 ms to empty the buffer.

The nested for... next loops output the array 16 bytes at a time, pausing for 1600 ms between each 16 bytes. If the connection fails program execution continues with the label "failed".

OWIN, OWOUT

OWIN receives data from a device using the 1-wire protocol.

OWOUT sends data to a device using the 1-wire protocol.

Syntax

owin pin,mode,{NCLabel,}[varlist]

owout pin,mode,{NCLabel,}[varlist]

> *pin* is a variable or constant that specifies the I/O pin to be used for 1-wire data transfer. This pin will switch to the

appropriate direction and remain in that state after the end of the instruction.

mode is a variable, constant or expression the specifies the data transfer mode as described in the table below.

Mode	**Reset**	**Byte/bit**	**Speed**
0	none	byte	low
1	before data	byte	low
2	after data	byte	low
3	before and after	byte	low
4	none	bit	low
5	before data	bit	low

> Refer to your device data sheet to determine the required settings. Data sheets can usually be found online using a search engine.

NCLabel is a label the program will jump to if communications fails (No Chip present).

varlist is a list of modifiers and variables that tells OWIN where to assign received data, or OWOUT what data to output. See the examples under HSERIN (page 101) and HSEROUT (page 106) for more detailed descriptions of this list.

Notes

The 1-wire protocol was developed by Dallas Semiconductor. It is a 1 wire asynchronous serial protocol that does not require a clock lead (as is the case with I^2C)

1-wire uses CMOS/TTL logic levels, open collector output. The data line requires an external pullup to the +5V supply of the Atom Pro. A value of 10K is suitable for short distances, 4.7K is better for longer runs. The master initiates and controls all activities on the 1-wire bus.

> In 1-wire applications the BasicATOM Pro is always a Master.

Example:

This example shows a sample program for reading a temperature sensor (Dallas DS1820):

See the DS1820 data sheet for further details on the commands used in this program and for the use of the 1-wire protocol.

```
temp  var word
convert var long
counter var byte
main
    owout P0,1,main,[$cc,$44]                           ;note 1
Wait
  owin P0,0,[temp]                                      ;note 2
  if temp = 0 then wait                                 ;note 3
  owout P0,1,main,[$cc,$be]                             ;note 4
  owin P0,0,[temp.byte0,temp.byte1]                     ;note 5
  convert = float temp fdiv 2.0                         ;note 6
  debug ["Temperature = ",real convert," C",13]  ;note 7
goto main
```

Note 1: Output is via I/O pin 0, byte mode, low speed, reset before data. $cc (Skip ROM) sets the DS1820 to accept commands regardless of its unique ID code, thus eliminating the need for the programmer to know that code. $44 (Convert T) initiates the temperature conversion and stores the result in the DS1820's scratchpad memory.

Note 2: Input is via I/O pin 0, byte mode, low speed, no reset. Input data will be 0 while conversion is in progress, 1 when data is ready in the scratchpad.

Note 3: Loop waiting for input data to be ready (i.e. data = 1).

Note 4: $cc is Skip ROM, as before. $be (Read Scratchpad) tells the DS1820 to send the two bytes from its scratchpad to the Atom Pro.

Note 5: Reads the two bytes from the DS1820's scratchpad and stores them in *temp*. Note the use of the variable modifiers *byte0* and *byte1* to "build" the word variable *temp*.

Note 6: Converts the temperature to floating point format. The division by 2 is required because the DS1820's output is in 0.5°C steps.

Note 7: Outputs the temperature to the debug watch window. Display is in the form "Temperature = 35 C" followed by a new line (13).

SHIFTIN

Reads data from a synchronous serial device (also knows as shifting in data). Unlike the previously described input commands (HSEROUT, SEROUT, I2COUT, OWOUT), SHIFTOUT operates on a bit, rather than a byte, basis.

Syntax

shiftin dpin,cpin,mode,[result{\bits}{result{\bits}...}]

dpin is a variable or constant that specifies the Data input pin. This pin will switch to input mode and remain in that state after the end of the instruction.

cpin is a variable or constant that specifies the Clock output pin. This pin will switch to output mode and remain in that state after the end of the instruction.

mode is a value (0 to 7) or a predefined constant that sets the incoming data conditions according to the following table:

Constant	value	speed	data order[13]	sampling
msbpre or msbfirst*	0	normal	msb first	before clock
lsbpre or lsbfirst*	1	normal	lsb first	before clock
msbpost	2	normal	msb first	after clock
lsbpost	3	normal	lsb first	after clock
fastmsbpre	4	fast	msb first	before clock
fastlsbpre	5	fast	lsb first	before clock
fastmsbpost	6	fast	msb first	after clock
fastlsbpost	7	fast	lsb first	after clock

* provided for backwards compatibility with previous versions.

Fast mode runs at the highest possible speed, normal is limited to 100 kb/s.

result is a variable where incoming data is stored. There can be multiple variables in a list, as shown in the examples.

[13] MSB means "Most Significant Bit", i.e. the highest order or leftmost bit of a nibble, byte, word or long number. LSB means "Least Significant Bit", i.e. the lowest order or rightmost bit of a nibble, byte, word or long number.

bits is an optional entry (1 – 32) defining the number of bits that will be stored in each variable in the list. Default is 8 bits.

> Refer to the data sheet for the peripheral device to determine the proper settings.

Notes

In synchronous serial communication, a clock signal (running at the bit rate) is provided by the master (the Atom Pro is configured automatically as the master) on a pin separate from the data signal. The remote device uses this clock signal to set the timing for transmitting bits to the Atom Pro.

When receiving bits, the Atom Pro expects one bit per clock pulse. The timing (set by the remote device) sends the bits either at the start (before) or end (after) each clock pulse.

When connecting the peripheral device, use the following pins:

Atom Pro	Peripheral
Data output	Data input
Data input	Data output
Clock	Clock

This form of communications is used by analog-digital converters, digital-analog converters, clocks, memory devices and other peripherals. Trade names include SPI and Microwire.

Example

```
ant var byte
bat var word
cat var long
shiftin P3,P4,msbpre,[ant,bat\16,cat\32]
```

This program will input 8 bits and store them in "ant", 16 bits and store them in "bat", and 32 bits and store them in "cat". Input will be at "normal" speed, msb first, and bits are expected at the start of clock pulses.

SHIFTOUT

Writes data to a synchronous serial device (also knows as shifting in data). Unlike the previously described input commands (HSEROUT, SEROUT, I2COUT, OWOUT), SHIFTOUT operates on a bit, rather than a byte, basis.

Syntax

shiftout dpin,cpin,mode,[var{\bits}{var{\bits}...}]

dpin is a variable or constant that specifies the Data output pin. This pin will switch to output mode and remain in that state after the end of the instruction.

cpin is a variable or constant that specifies the Clock output pin. This pin will switch to output mode and remain in that state after the end of the instruction.

Note: Since the Atom Pro is always the master device, the clock pin will always be an output for both SHIFTIN and SHIFTOUT.

---

mode is a value (0 to 7) or a predefined constant that sets the incoming data conditions. See the table under SHIFTIN.

var is a variable where incoming data is stored. There can be multiple variables in a list, as shown in the example.

bits is an optional entry (1 – 32) defining the number of bits that will be written from each variable in the list. Default is 8 bits.

> Refer to the data sheet for the peripheral device to determine the proper settings.

Notes

See the *Notes* under SHIFTIN.

Examples

```
ant var byte
bat var word
cat var long
code setting values for ant, bat, cat
shiftout P2,P4,msbpre,[ant,bat\16,cat\32]
```

This program will output 8 bits from variable "ant", 16 bits from variable "bat", and 32 bits from variable "cat". Output will be at "normal" speed, msb first, and bits are sent at the start of clock pulses.

Miscellaneous Commands

END, STOP

These commands stop program execution and place the Atom Pro in low power mode. All I/O pins will remain in their current state.

END and STOP are identical in function.

Syntax

END

STOP

Notes

To restart a stopped program, press the RESET button on the Atom Pro or power the Atom Pro OFF and back ON.

HIGH, LOW, TOGGLE

HIGH configures a pin as output and sets it high.

LOW configures a pin as output and sets it low.

TOGGLE configures a pin as output and switches its state from high to low or low to high.

Syntax

high pin

low pin

toggle pin

pin is a variable or constant that specifies the I/O pin to use.

Examples

```
high P4         ; makes P4 an output and sets it high (5V)

low P10         ; makes P10 an output and sets it low (0V)

high P4         ; makes P4 an output and sets it high.
toggle P4       ; switches P4 from high to low.
```

INPUT, OUTPUT, REVERSE

INPUT sets a pin to be an input.

OUTPUT sets a pin to be an output but does not set its state.

REVERSE reverses the direction of a pin.

Syntax

input pin

output pin

reverse pin

pin is a variable or constant specifying the I/O pin affected.

Notes

These commands let you set the direction of a pin. Note that several commands (e.g. high, low, etc. automatically set the direction of certain pins so it may not be necessary to set them using input, output or reverse. This behavior is documented for the individual commands.

Examples

```
input P8             ; sets I/O pin 8 as an input.

output P9            ; sets I/O pin 9 as an output

serout P5\P6,NEO2400,5000,expd,[dec ant,bat,hex4 cat\4]
program continues here
input P5             ; change P5 to an input
```

In the last example, the *serout* command has set P5 to an output. Later, the *input* command is used to change it to an input.

PAUSE

Pause execution for a specified number of milliseconds.

Syntax

pause milliseconds

milliseconds is a variable or constant specifying the number of milliseconds (up to 32 bits, or 4,294,967,295 ms).

Notes

Pause is used to delay program execution. The duration of the pause can be from 1 ms to 4,294,967 seconds (which is approximately 1193 hours, or 49.7 days).

While it is unlikely that longer pauses than this will be required, times shorter than 1 ms may be obtained with the *pauseus* and *pauseclk* commands.

Examples

See the traffic light program on page 23 for one example where *pause* could be used. Another example is shown under I2COUT on page 120.

PAUSECLK

Pause execution for a specified number of clock cycles. PAUSECLK is most useful for very short times, for longer times use PAUSE or PAUSEUS.

Syntax

pauseclk cycles

cycles is a variable or constant (up to 32 bits) specifying the number of clock cycles to pause.

Notes

Clock periods and timing are shown in the table below.

	Clock	Period	Minimum	Maximum
Atom Pro	16 MHz	62.5 ns	62.5 ns	268 s (4 m 28 s)
Atom Pro Plus	20 MHz	50.0 ns	50.0 ns	215 s (3 m 35 s)

Examples

With an Atom Pro

```
pauseclk 2000
```

will cause program execution to pause for 2000 x 62.5 ns, or 125 µs.

PAUSEUS

Pause execution for a specified number of microseconds.

Syntax

pauseus microseconds

> *microseconds* is a variable or constant specifying the number of microseconds to pause, up to 32 bits.

Notes

The *pauseus* command is used to pause program execution for short periods of time (from 1 µs to 4,294,967,295 µs, which is approximately 4,295 seconds, or 71.6 minutes, or 1.2 hours).

The resolution of the *pauseus* command is 1000 times smaller than that of the *pause* command and 5 times greater than the *pauseclk* command.

NAP

The NAP command executes the processor's *internal sleep* mode for the specified time period. Power consumption is reduced in sleep mode. NAP is essentially a simplified version of SLEEP (see page 131).

Syntax

nap period

> *period* is a variable or constant that determines the duration of the reduced power nap in units of 2 ms.

Notes

Times are approximate and may vary with temperature, supply voltage and manufacturing tolerances.

The Atom Pro will immediately wake up from a *nap* if an interrupt occurs.

The NAP command does not affect internal registers so your program will continue executing when the time expires.

Example

```
nap 2000
```

will put the Atom Pro in low power sleep state for 40 seconds

SLEEP

The SLEEP command is similar to the NAP command except that it adds a number of options. To achieve minimum power consumption set all I/O pins to *output* and in the low state[14].

Syntax

sleep time{,mode}

> *time* is a variable or constant (up to 32 bits) that specifies the duration of the sleep in 2 ms increments (see the *Important Note* below).
>
> *mode* is an optional parameter with several possible values as described below. If *mode* is not specified, the *sleep* command is equivalent to the *nap* command.

Modes

Standby mode puts the processor to sleep and turns off the oscillator to save power. An external interrupt or reset must be used to wake up the processor. The *time* parameter is ignored.

SLEEPSTANDBY	Enter standby, wake on external interrupt.

The following modes cause the clock multiplier to be set and puts the processor to sleep for the specified time.

SLEEPACTIVE	Normal sleep (default)
SLEEPACTIVE_8	1/8 system clock sleep
SLEEPACTIVE_16	1/16 system clock sleep
SLEEPACTIVE_32	1/32 system clock sleep
SLEEPACTIVE_64	1/64 system clock sleep

> Important Note: If the clock multiplier is set to 8, 16, 32 or 64 the 2 ms time increment is multiplied by the same factor.

[14] Don't set pins to "output" if they are connected to the outputs of other devices!

The following modes change the clock speed divisor without putting the processor to sleep. The *time* parameter is ignored.

DIRECTACTIVE	Normal system clock
DIRECTACTIVE_8	1/8 system clock
DIRECTACTIVE_16	1/16 system clock
DIRECTACTIVE_32	1/32 system clock
DIRECTACTIVE_64	1/64 system clock

Same as DIRECTACTIVE except that all system registers, TimerV, SCI3 and the AD hardware are reset. The *time* parameter is ignored.

DIRECTACTIVERES	Normal system clock
DIRECTACTIVERES_8	1/8 system clock
DIRECTACTIVERES_16	1/16 system clock
DIRECTACTIVERES_32	1/32 system clock
DIRECTACTIVERES_64	1/64 system clock

Notes

Examples

```
sleep 3000
```

will put the Atom Pro in low power sleep state for approximately 6 seconds.

```
sleep 3000,sleepactive_8
```

will reduce the clock rate by a factor of 8, and put the Atom Pro in low power sleep state for approximately 48 seconds.

```
sleep 1,sleepstandby
```

will put the processor to sleep and turn off the oscillator. The processor will start when a reset or external interrupt is received. The time parameter (1, in this case) is ignored.

This page intentionally left blank

Specialized I/O Commands

This chapter includes specialized input/output commands such as those for A/D conversions, generating audio tones, controlling LCD displays, stepper motors, servos, etc.

Conventions Used in this Chapter

{ ... } represent **optional** components in a command. The { } are **not** to be included.

[...] used for lists – the [ ] are **required**

(...) used for some arguments. The ( ) are **required**

> Several of the commands described in this chapter specifically address hardware features of the microcontroller chip. Descriptions of these hardware features are beyond the scope of this manual.
>
> Please refer to the H8/3664F or H8/3687F hardware manual, available at http://www.renesas.com for further details..

Waveform I/O Commands

DTMFOUT

Outputs a two frequency DTMF tone on a single pin of the Atom Pro. This tone can be used for dialing a telephone or operating remote devices with DTMF decoders.

Syntax

dtmfout pin,{ontime,offtime,}[tone1, tone2, ... toneN]

pin is a variable or constant that specifies the I/O pin to use. The pin will be set to an output during tone generation. After tone generation is complete the pin will be set to an input.

ontime is an optional variable or constant (0 – 65535) that specifies the duration of each tone in milliseconds. If not specified, default is 200 ms.

offtime is an optional variable or constatnt (0 – 65535) that specifies the length of silence after each tone in milliseconds. If not specified, default is 50 ms.

tone1 – toneN is a list of tones to be generated in the form of variables or constants defined by the list below:

Tone value	**DTMF pair**
0 to 9	0 to 9
10	*
11	#
12 to 15	fourth column tones A to D

Notes

DTMF tones consist of two sine waves at different frequencies.

	1209Hz	1336Hz	1477Hz	1633Hz
697Hz	1	2	3	A
770Hz	4	5	6	B
852Hz	7	8	9	C
941Hz	*	0	#	D

The DTMFOUT command causes the Atom Pro to create and mix two sine waves mathematically, then use the resulting data stream to control the duty cycle of a pulse width modulator (PWM). The resulting output must be filtered to remove the digitization "noise" and produce reasonable sine waves.

Note: The DTMFOUT2 command requires less complex filtering and is recommended if you have an extra I/O pin available.

The simplest circuit uses a resistor and capacitor as a low pass filter (shown connected to P1 in the diagram). Depending on the DTMF decoder used, this simple filter may be sufficient – it is not recommended for use with the PSTN (public switched telephone network). You may need to adjust the capacitor value for best results.

Design of more sophisticated filters, if required, is beyond the scope of this manual.

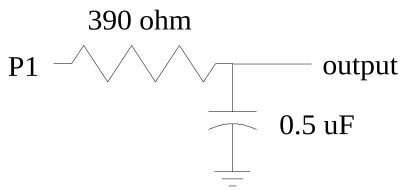


Figure 14 - Simple Low Pass Filter

Examples

```
dtmfout p1,100,50,[2,3,3,5,5,5,5]
```

This command will generate the DTMF pairs required to dial the number 233-5555 using 100 ms tones with 50 ms silent spaces between them.

DTMFOUT2

Outputs a two frequency DTMF tone on two pins of the Atom Pro, one frequency per pin. This tone can be used for dialing a telephone or operating remote devices with DTMF decoders.

Syntax

dtmfout2 pin1\pin2,{ontime,offtime,}[tone1, tone2, ... toneN] *pin1\pin2* are variables or constants that specifies the two I/O pins to use. The pins will be set to outputs during tone generation. After tone generation is complete the pins will be set to inputs.

ontime is an optional variable or constant (0 – 65535) that specifies the duration of each tone in milliseconds. If not specified, default is 200 ms.

offtime is an optional variable or constant (0 – 65535) that specifies the length of silence after each tone in milliseconds. If not specified, default is 50 ms.

tone1 – toneN is a list of tones to be generated in the form of variables or constants defined by the list below:

Tone value	**DTMF pair**
0 to 9	0 to 9
10	*
11	#
12 to 15	fourth column tones A to D

Notes

DTMF tones consist of two sine waves at different frequencies.

	1209Hz	1336Hz	1477Hz	1633Hz
697Hz	1	2	3	A
770Hz	4	5	6	B
852Hz	7	8	9	C
941Hz	*	0	#	D

The DTMFOUT2 command causes the Atom Pro to create two square waves, one at each of the required frequencies, and send them out on their respective pins. Filtering is required to create a reasonable approximation of sine waves, however the square waves have much less high frequency noise than the PWM tones generated by DTMFOUT and require less sophisticated filtering. The diagram assumes that the tones are generated on P1 and P2.

The capacitor value may require adjustment for best results. This circuit should work well on the PSTN if levels are correctly set.

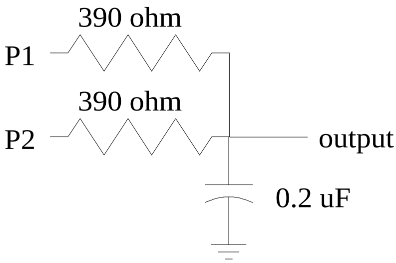


Figure 15 - Filter/combiner for DTMFOUT2

Examples

```
dtmfout2 p1\p2,100,50,[2,3,3,5,5,5,5]
```

This command will generate the DTMF pairs required to dial the number 233-5555 using 100 ms tones with 50 ms silent spaces between them. Output is on I/O pins P1 and P2 and should be combined using a circuit similar to that shown above.

FREQOUT

This command generates one or two tones that are output on a single I/O pin.[15] FREQOUT generates a pulse width modulated signal.

Syntax

freqout pin, duration, freq1{,freq2}

> *pin* is a variable or constant that specifies the I/O pin to be used. This pin will be set to output mode during tone generation and left in that state after output is completed.
>
> *duration* is a variable or constant that sets the duration of the output tone in milliseconds (0 – 65535).
>
> *freq1* is a variable or constant that specifies the frequency in Hz of the first tone (0 – 32767).

[15] FREQOUT generates a pulse width modulated signal designed to be filtered to create a sine wave. You may prefer to use one of the SOUND commands, which generate a square wave, if a single tone requiring less filtering is desired.

freq2 is an optional variable or constant that specifies the frequency in Hz of the second tone (0 – 32767).

Notes

The tone (or tones) is generated mathematically in the Atom Pro and output as a pulse width modulated (PWM) signal. The signal must be converted to a sine wave (or a pair of sine waves) by passing it through an integrator (low pass filter).

For non-critical applications, a simple filter such as the one shown below may suffice. You may need to experiment with the resistor and/or capacitor value for best results at the frequency you are using.

> Design of more sophisticated filter circuits is beyond the scope of this manual.

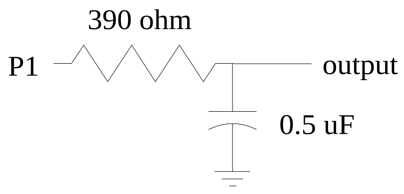


Figure 16 - Simple integrator/low pass filter

Examples

```
freq var word
dur var word
freq = 1000
dur = 5000
freqout p1, dur, freq
```

This will output a 1000 Hz tone for a duration of 5 seconds on Pin 1.

HPWM

This command gives access to the Atom-Pro's built-in hardware PWM generator. It allows you to output a PWM signal with any desired period and duty cycle (within the limits of the hardware).

A detailed description of the operation of this hardware is beyond the scope of this manual. Please refer to the H8/3664F or H8/3687F data sheet (see page 135 for availability).

Syntax

hpwm pin, period, duty

> *pin* is a constant or variable that specifies the pin to be used for PWM output. This pin will be switched into output mode for the duration of the pulse generation, then returned to input mode. Only pins with the FTIOB/C or D option can use the HPWM command. See page 187 for module pinouts.
>
> *period* is a constant or variable (0 to 524288) that specifies the period of the output signal in clock cycles.
>
> *duty* is a constant or variable (0 to 524288) that specifies the duty of the output signal in clock cycles.
>
> *Note:* duty *must be less than* period *for this command to work properly.*

Notes

The HPWM command uses the Atom-Pro's hardware module to generate a square wave signal with a definable duty cycle. Since it uses hardware generation the PWM signal may be output while program execution continues.

Once the HPWM command has executed, the PWM signal will be output continuously until cancelled, while the rest of your program will continue to execute.

To cancel the PWM signal.

Examples

```
select con 1                     ; 1 uses CCP2 on pin 9
period var word
duty var word
period=100                       ; 1000 us period
duty=25                          ; 25% duty cycle
main
```

```
   hpwm select, period, duty
   pause 5000                      ; wait 5 seconds
   ccp2con=0                       ; turn off output
   pause 5000                      ; wait 5 seconds
goto main                          ; repeat
```

This program generates a square wave of period 100 microseconds (frequency of 10,000 Hz) with a duty cycle of 25%. Output is on the Atom Pro module's pin 9. The signal will continue for 5 seconds, then be turned off (by setting register ccp2con to zero). After a further 5 seconds, the program will repeat.

PWM

The PWM command is used to generate an analog voltage from a digital calculation. The signal is generated in software; see HPWM (above) for a hardware generated PWM signal.

Syntax

pwm pin, period, duty, cycles

> *pin* is a variable or constant that specifies the Atom-Pro I/O pin to use. This pin will be set to output during pulse generation.
>
> *period* is a 32 bit expression that specifies the period of the pulse width signal in clock cycles.
>
> *duty* is a variable or constant (0 – 255) that sets the duty cycle from 0% (0) to 100% (255).
>
> *duration* is a 32 bit expression that specifies the number of pulses to output.

Notes

The PWM command generates a pulse width modulated signal with a specified duty cycle. Note that the frequency of the pwm signal is not fixed and varies with the duty cycle, therefore the primary use for this command is to produce a signal to be filtered for analog output.

The output of the PWM command must be integrated (using a low pass filter) to produce an analog voltage. A circuit such as the one shown below will suffice for most uses.

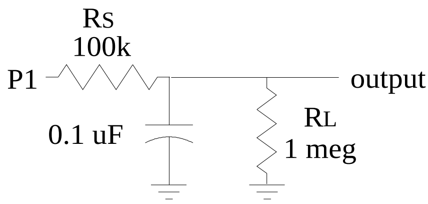


Figure 17 - Analog converter for PWM command

The values of capacitor, series resistor (R_S) and load resistor (R_L) may be varied to produce the desired output voltage and response time. The values shown produce adequate filtering. Note that response time is relatively slow, and the PWM command with a filter such as this is mainly suitable for producing steady-state voltages rather than rapidly varying waveforms.

The values shown will produce a voltage that varies linearly from 0V (with *duty* set to 0) to approximately 4.6V (with *duty* set to 255). The frequency of the pwm signal is approximately 125 kHz with *duty* set to 128 (50%) and drops significantly at both higher and lower duty cycles.

Examples

```
duty var byte
duration var word
duration=5000
main
   duty=0
   gosub generate
   duty=64
   gosub generate
   duty=128
   gosub generate
   duty=192
   gosub generate
   duty=255
   gosub generate
goto main
generate
   pwm p1, duty, duration
return
```

With a filter such as that shown above, this program will generate, in sequence, voltages of 0V, 1.15V, 2.3V, 3.45V and 4.6V for 5 seconds each, then repeat the same cycle indefinitely. (All voltages are approximate.)

PULSOUT

Generates a pulse on the specified pin. A "0" or "1" pulse will be generated, depending on the initial state of the pin.

Syntax

pulsout pin, time

> *pin* is a variable or constant that specifies the I/O pin to use. This pin will be placed in output mode immediately before the pulse, and left in that mode after the instruction finishes.
>
> *time* is a variable or constant (minimum value 8) that specifies the duration of the pulse in 0.5 microseconds increments.

Notes

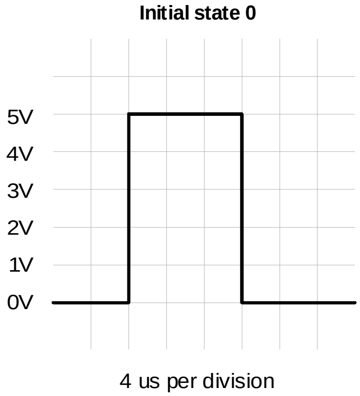
PULSOUT toggles the pin's high/low state twice to generate a pulse. You can use the *high* or *low* commands to set the initial state of the pin, which controls the polarity of the pulse.

Once the pulse is issued, the pin will remain in the final state (which is the same as its initial state prior to the PULSOUT command) until further commands affect that pin. Thus, successive use of the PULSOUT command will produce successive pulses of the same polarity.

Examples

```
time var word
time=12
low p0              ; set pin0 to output, low state
pulsout p0, time    ; generate a "high" pulse
program continues
```

This program will produce a pulse similar to that shown on the left in Figure 18. If the "low p0" command is replaced by "high p0" the pulse will be similar to that shown on the right in Figure 18.



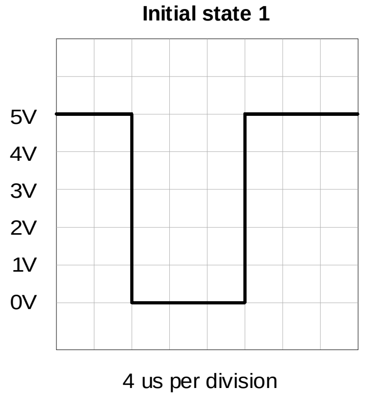


Figure 18 - Output of "pulsout" command

PULSIN

Measures the duration of an input pulse on a specified pin.

Syntax

pulsin pin, direction, {TimeoutLabel, TimeoutMultiplier,} duration

> *pin* is a variable or constant that specifies the pin to be used for the input pulse. This pin will be placed into input mode during the execution of this command and left in that state after the command finishes.
>
> *direction* is a variable or constant (0 or 1) that specifies the pulse direction. If *state* = 0, the pulse must begin with a 1-to-0 transition. If *state* = 1 the pulse must begin with a 0-to-1 transition.
>
> *TimeoutLabel* is an optional label that specifies the target if a timeout occurs. If the command times out before a pulse is detected, program execution will continue at this label. The default timeout value is 65535 μs. If no *TimeoutLabel* is specified, PULSIN will wait 65535 μs for a pulse to occur, then program execution will continue with the next instruction.
>
> *TimeoutMultiplier* is a variable or constant that specifies the multiplier to be used for the default 65536 μs timeout. a multiplier for the default 65535 μs timeout. For example, if

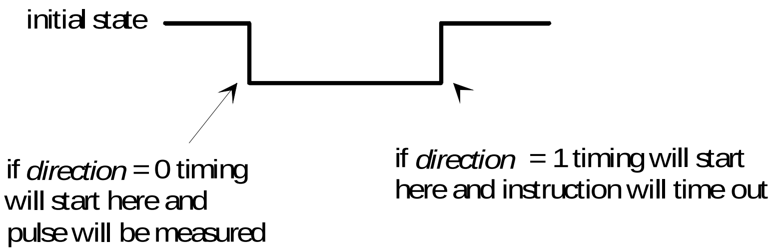
timeoutmultiplier = 10, the timeout will be 655350 µs or 0.655 seconds. *TimeoutMultiplier* is required if *TimeoutLabel* is specified.

duration is a variable that stores the pulse duration in µs. Make sure the variable is large enough to store the longest expected pulse time (either 65535 µs or that set by *TimeoutMultiplier*). If the variable is too small only the least significant bits will be stored. If no pulse is detected within the timeout value *duration* will be set to 0.

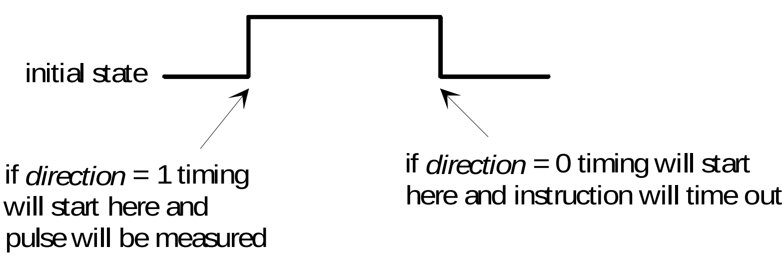
Notes

These illustrations will show the results of the PULSIN instruction in several situations.

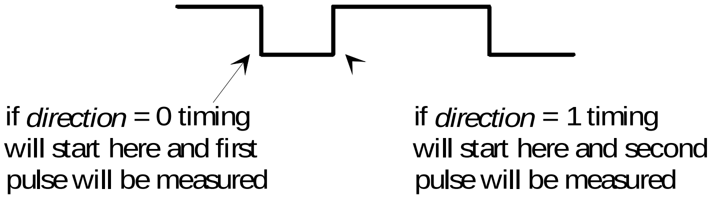
Pin is initially high, and a low pulse occurs:



Pin is initially low, and a high pulse occurs:



Pin is initially high, and a low pulse, followed by a high pulse, occurs:



Examples

```
pulsin p0,0,duration
```

will wait up to 65535 µs for a "low" pulse (i.e. a pulse starting with a falling edge) and measure its duration, saving the result in *duration*, then program execution will continue with the next instruction.

If there is no pulse within 65535 µs program execution will continue with the next instruction, and *duration* will be set to 0.

---

*Note: If a "high" pulse occurs, timing will start at the **end** of the pulse, and PULSIN will time out.*

```
pulsin p0,1,timeout,100,duration
```

will wait up to 6553500 µs (approx. 6.5 seconds) for a "high" pulse (i.e. a pulse starting with a rising edge) and measure its duration, saving the result in *duration*, then program execution will continue with the next instruction. *Duration* must be large enough to store a value up to 6553500.

If there is no pulse within 6553500 µs, program execution will jump to the label *timeout* and *duration* will be set to 0.

---

*Note: If a "low" pulse occurs, timing will start at the **end** of the pulse, and PULSIN will time out, jumping to label* timeout.

SOUND

Generates an audio tone or a sequence of tones on a specified I/O pin.[16] SOUND generates a square wave.

Syntax

sound pin,[dur1\note1, dur2\note2, ... durN\noteN]

> *pin* is a variable or constant specifying the output pin to use. This pin will be set to output mode during tone generation and will remain in that mode after the instruction is completed.
>
> *dur* is a constant or variable (or a number of constants or variables) that specify the duration, in milliseconds (1 – 65535) of each tone.
>
> *note* is a constant or variable (or a number of constants or variables) that specify the frequency in Hz (1 – 32767) of each tone to be generated.

Notes

The SOUND command generates a square wave output. If you are using it to drive a small speaker or amplifier no filtering may be needed. However, a low pass filter is recommended to convert the square wave to something resembling a sine wave.

A simple RC filter, such as that shown in Figure 16, can be used to approximate a sine wave. You may need to adjust the capacitor value for best results with the frequencies you are using.

Since a square wave contains all odd harmonics of the fundamental signal, the best filter would have a sharp cutoff at less that 3 times the frequency of the tones used. The design of such a filter is beyond the scope of this manual.

[16] If only a single tone is needed, you may prefer to use the FREQOUT instruction which generates a pulse width modulated signal designed to be filtered to create a sine wave.

Examples

```
note1 con 1000
note2 con 2000
note3 con 3000
dur con 1000
sound p1,[dur\note1,dur\note2,dur\note3]
```

will produce tones of 1000, 2000 and 3000 Hz in sequence, lasting 1 second each.

SOUND2

Generates two simultaneous tones, or a sequence of such tones, on two specified output pins. The tones generated are square waves.

Syntax

sound2 pin1\pin2,[dur1\note1\noteA,dur2\note2\noteB,... durN\note#\noteN]

> *pin1* and *pin2* are constants or variables specifying the two output pins to be used, one for each tone.
>
> *dur* is a constant or variable (or sequence of constants or variables) specifying the duration in milliseconds (1 – 65535) of each note pair. Both notes last the same duration in each case.
>
> *note1* to *note#* are constants or variables specifying the frequencies in Hz (0 – 32767) of the notes to be output on *pin1*.
>
> *noteA* to *noteN* are constants or variables specifying the frequencies in Hz (0 – 32767) of the notes to be output on *pin2*.

Notes

The SOUND2 command generates square wave output on each of the two pins. If you are using it to drive a small speaker or amplifier no filtering may be needed. The pins can be connected together as shown on the left in Figure 19. However, a low pass filter is recommended to convert the square wave to something resembling a sine wave, as shown on the right in Figure 19. You may need to adjust the capacitor value for best results.

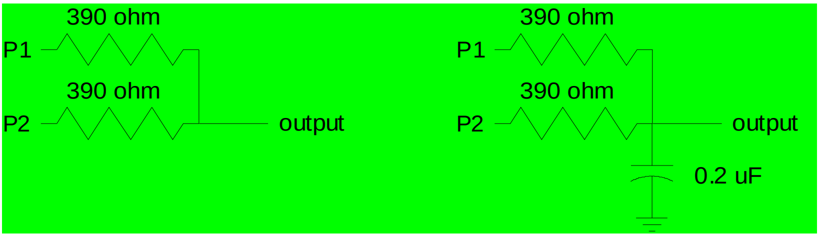


Figure 19 - Combining outputs for Sound2

Since a square wave contains all odd harmonics of the fundamental signal, the best filter would have a sharp cutoff at less that 3 times the frequency of the tones used. The design of such a filter is beyond the scope of this manual.

Examples

```
note1 con 1000
noteA con 1500
note2 con 1800
noteB con 2700
dur con 1000
sound2 p1\p2,[dur\note1\noteA,dur\note2\noteB]
```

This will output the frequency pair 1000/1500 Hz for 1 second, followed by the pair 1800/2700 Hz for 1 further second, on pins 1 and 2 respectively.

Special I/O Commands

ADIN

Sets up the hardware A/D converter and stores the resulting value in a variable. The resolution is 10 bits.

> The ADIN command uses the microcontroller's built-in A/D hardware. A description of this hardware is beyond the scope of this manual. Please refer to the H8/3664F or H8/3687F hardware manual (see page 135)

Syntax

adin pin,var

pin is a constant or variable specifying the analog pin to use for the A/D input. Available pins are :

AtomPro 24-M   P0 – P3
AtomPro 28-M   P0 – P3 and P16 – P19

var is a variable (word or long) that stores the returned value from the conversion (10 bit resolution).

Notes

The ADIN command converts an analog (0 – 5V) signal to a digital value from 0 – 1023. The input can be scaled by using a reference voltage (Vref).

Examples

```
volts var word
adin 1,volts
```

will convert an input voltage (0 – 5V) on pin 1 to a digital value of 0 – 1023 and store the result, right justified, in word variable "volts". The first 6 bits of "volts" will be zeros.

BUTTON

Processes a momentary switch contact, such as a button press. Includes debouncing and auto-repeat. The BUTTON command should be used in a program loop so that it is repeatedly accessed.

Note that all timing (except debounce) for the BUTTON command is set by counting program loops, so some experimentation may be required to find the best values.

Syntax

button pin, downstate, delay, rate, loopcounter, targetstate, target

pin is a variable or constant the specifies the I/O pin to be used. *Pin* will be set to an input automatically.

downstate is a variable or constant (value either 0 or 1) that specifies the logical state of the pin when the button is pressed. 0 = low, 1 = high. This lets you use normally open or normally closed buttons.

delay is a byte variable or constant (0 – 255) that specifies the number of program loops to execute before first entering the auto-repeat sequence.

- If *delay* is set to 0 both debounce and auto-repeat are disabled.
- If *delay* is set to 255 debounce is enabled, but auto-repeat is disabled.

rate is a byte variable or constant (0 – 255) that specifies the number of program loops to execute before auto-repeating, after the initial *delay* has expired.

loopcounter is a byte variable used to store the current number of program loops. This variable must not be used for any other purpose within this program loop. If more than one BUTTON command is used in your loop, you must specify a different *loopcount* variable for each.

targetstate is a variable or constant (0 or 1) that determines the logical state of the button for a branch to *target* to occur. 0 = not pressed, 1 = pressed. (Pressed and not-pressed are defined by *downstate.*)

target is a label to which execution will branch if the button is in *targetstate* and debounce, delay and rate conditions are met.

Notes

BUTTON checks the state of an I/O pin connected to a switch and branches according to the result. BUTTON is actually just a form of conditional branch; it does not produce any numeric result or save any values.

For the following notes, assume that the BUTTON parameters have been set to see a LOW state as *downstate* and to branch when the button is pressed (down). BUTTON is executed within a loop.

- **Not pressed.** If the button is not pressed, *loopcounter* is reset and control simply passes to the next program statement. After the following statements are executed, control must be returned to the start of the loop.

- **First press.** If the button is pressed for the first time (i.e. the previous loop pass showed it as not pressed), BUTTON first does a debounce check.
  - If debounce fails (i.e. the button wasn't actually pressed), control passes to the next program statement as above.
  - If debounce passes (i.e. the button is really pressed) BUTTON branches to the statement defined by *target*. *Loopcounter* is also incremented for use by the auto repeat function. The sequence of commands following *target* must return to the start of the loop.
- **Repeat delay.** If the button is still pressed the next time BUTTON is encountered (i.e. on the next pass through the loop), *loopcounter* is again incremented. If *loopcounter* has reached the value specified by *delay* execution will branch to *target* and *loopcounter* will be reset. If *loopcounter* has not reached this value, it will be incremented and execution will continue with the following program statement
- **Auto-repeat.**. If the repeat delay has expired, the sequence of steps under "Repeat delay" will be executed, but using the *rate* value for *loopcounter*, rather than the *delay* value. This lets you set the initial delay and the repeat rate independently.

Debouncing

Mechanical buttons often close and reopen many times (sometimes hundreds of times) before stabilizing in the pressed position. This is because of mechanical vibration of the components of the button. To avoid having BUTTON see these intermittent cycles as many button presses, it has a built-in debounce feature.

When BUTTON sees a valid *downstate* for the first time, it delays approximately 20 ms and checks again. If the *downstate* is still valid, it assumes that the button is really pressed and continues processing. If *downstate* is no longer valid, this could be the result of a bounce so it is ignored and control passes to the next program statement.

If the button really was pressed, the next execution of the BUTTON command will show a valid *downstate* and processing will continue as above.

Examples

```
delay var byte
rate var byte
count1 var byte
count2 var byte
delay=80
rate=40
startloop
   button P4,0,delay,rate,count1,1,right
   button P5,0,delay,rate,count2,1,left
goto startloop
right
   code to rotate to the right
goto startloop
left
   code to rotate to the left
goto startloop
```

This program will check two buttons, one for right on pin 4, the other for left on pin 5. The buttons are normally open, closed when pressed, so they pull the pins LOW when pressed. Depending on which button is pressed, a stepper motor (or other device) will be caused to turn left or right.

The two BUTTON commands use different *loopcount* variables, count1 and count2.

Note: This program does not include code to deal with simultaneous pressing of the two buttons.

COUNT

Counts the number of cycles (0 – 1 – 0) on an input pin during a specified time period. Used to determine frequency. The minimum pulse width that can be counted is 4 µs.

Syntax

count pin,period,cycles

> *pin* is a variable or constant that specifies the input pin to be used. This pin is automatically set to input mode.
>
> *period* is a variable or constant that specifies the counting time in milliseconds.

cycles is the variable in which the total count will be saved. *Cycles* must be large enough to store the highest expected number of cycles.

Examples

```
total var word
count p3,10,total
```

Will count the total number of 0-1-0 transitions on I/O pin 3 for 10 ms, and store the result in "total". For instance, if the input frequency was 50 kHz, the count would be 500 (± 1 count).

RCTIME

Measures short time intervals, such as the charge/discharge time of an R/C circuit.

Syntax

rctime pin,state,{timeoutlabel,timeout,}result

pin is a variable or constant that specifies the I/O pin to use. This pin will be placed into input mode and left in that mode when the instruction finishes.

state is a variable or constant (1 or 0) that specifies the state which will end the timing period.

TimeoutLabel is a program label. If the command times out before the pin state changes, execution will continue at this label. The default timeout is 65535 µs. If no *TimeoutLabel* is specified, program execution will continue with the next statement in the event of a timeout.

Timeout is a 32 bit value that specifies the timeout in 0.5 µs units.

ResultVariable is a variable in which the time measurement, in µs, will be stored. This variable must be large enough to store the maximum value set by *TimeoutMultiplier*.

Notes

RCTIME can be used to measure the value of capacitors or resistors, as well as to make other triggered timing measurements. One common use is to measure a potentiometer setting.

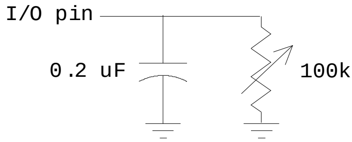


Figure 20 - Measuring time with RCTIME

With a circuit similar to that shown in Figure 20 the RCTIME command is used as follows:

1. Set the I/O pin to be an output and set it high.
2. Wait long enough for the capacitor to fully charge. About 5 time constants[17] (i.e. 5 x R x C) will do nicely.
3. Issue the RCTime command, which will switch the pin to an input and "watch" the voltage as the capacitor discharges through the variable resistor.

From the resulting time, the resistor or capacitor value can be calculated (you have to know at least one of them accurately if absolute, rather than relative, results are required).

To help in your calculations, here's some information about the BasicATOM Pro's I/O pins:

Approximate voltage to switch from low to high state on a rising input	2.05V
Approximate voltage to switch from high to low state on a falling input	0.80V

It takes about 1.83T (where T is the time constant) for the circuit to discharge enough to trigger the RCTime command. This means that the time constant = *ResultVariable* / 1.83. A sample calculation is shown below under Examples.

[17] The "time constant" of an R/C circuit is the time for it to charge to 63.2% of the applied voltage, or to discharge to 36.8% of the initial voltage. It is calculated using the equation T = R * C where R is in ohms and C is in farads.

Note: It's very convenient that the time constant equation also works perfectly if time is in µs and capacitance is in µF.

Examples

This program assumes that the circuit shown in Figure 20 is used, with the variable resistor set to about the mid point of its rotation. Some scaling is done to allow integer arithmetic to be used.

```
dtime var word
resist var word
high P3
pause 10            ; wait for capacitor to charge
rctime P3,0,dtime
resist = (10000*dtime)/(183*2)    ; see text below
```

The program sets P3 to output, high state, and then waits 10 ms for the capacitor to charge fully. The RCTIME command then changes P3 to an input and waits for a low state, which occurs when the capacitor discharges to about 0.8V. The time, in microseconds, is stored in *dtime*.

The resistance is then calculated using the formula:

resistance = dtime/(1.83 * capacitance)

However, to accommodate the integer arithmetic, the 1.83 and capacitance are each multiplied by 100 (giving 183 and 2, respectively), so the numerator must be multiplied by 10000 to compensate. These steps could be avoided by using floating point calculations, if desired, at the expense of program complexity and calculating time.

In the example, if *dtime* is 1830, the value of the resistance comes out to be 50000 ohms.

> Important: In most cases an absolute value won't be needed, only a relative position of the variable resistor, so the resistance calculation can be simplified.

ENABLEHSERVO

Enables the hardware servo control system and specifies its parameters of operation.

Syntax

ENABLEHSERVO

Notes

ENABLEHSERVO is a compile time directive that tells the compiler to add support for the hardware servo control system. The HSERVO system uses the TimerW (or TimerZ0 in ATOM-Pro Plus) to produce interrupt driven signals for up to 32 servos.

ENABLEHSERVO has no arguments.

HSERVO

Sets the position and speed of up to 32 servos. Before using this command the ENABLEHSERVO compile time directive must be included in your program.

Syntax

HSERVO [Pin\Pos\Spd....PinN\PosN\SpdN]

> *Pin...PinN* are expressions that specify the pin numbers connected to the servos whose position and speed are to be set.
>
> *Pos...PosN* are expressions that specify the desired positions for each of the specified servos (range –12000 to +12000).
>
> *Spd...SpdN* are optional expressions that specify the speed used to move each servo to its new position (default 255).

Notes

The HSERVO command is a background timer interrupt driven command. It allows you to set the new position and speed to move to that position of up to 32 servos at one time. Each servo set will start moving to its new position imediately after the HSERVO comand finishes.

To deactivate a servo that has been used set its *Pos* to –16000.

Example

```
HSERVO P1\2000\100,P2\-1000\100
```

WAITHSERVO

Waits for specified servos to reach position before continuing.

Syntax

WAITHSERVO [pin#,pin#,…]

> *Pin#* is an expression that specifies the I/O pin number to which the servo is connected.

Notes

WAITHSERVO pauses program execution until the specified servos have reached the postions set by the HSERVO instruction.

GETHSERVO

Get the current position of the specified servo. Optionally determine whether the servo is idle or not.

Syntax

GETHSERVO pin,position{,idle}

> *Pin* is an expression that specifies the I/O pin number on which to check servo position.
>
> *Position* is a variable where GETHSERVO will store the current position (-12000 to +12000, center = 0)[18] of the specified servo.
>
> *Idle* is an optional variable where GETHSERVO will store the current state of the servo. A value of $FFFFFFFF means the servo is idle (at its final position).

Notes

Gethservo is used to determine the current position of a servo and whether it has finished moving to its final position.

[18] Most servos will not reach the maximum values shown.

Example

```
getservo p0,pos
```

or

```
getservo p0,pos,idle
```

If the servo is at its new position "idle” will be a non-zero number. If the servo is still moving to its new position then "idle" will equal 0.

SERVO

Operates a servo motor.

Syntax

servo pin, rotation{, repeat}

> *pin* is a variable or constant that specifies the I/O pin used to control the servo.
>
> *rotation* is a variable or constant that specifies the position to which you want the servo to rotate. The value of *rotation* should fall within the limits of –2400 to +2400*, with 0 being the center position. See the Notes below for a discussion of servo motors.
>
> | * Exceeding these values could damage your servo. |
> |---|
>
> *repeat* is an optional variable or constant that specifies the number internal cycles the command runs (default = 20). This value must be high enough for the motor to reach the desired position, so higher values may be required for larger angles. *Repeat* allows the servo to reach the desired position before the program continues and perhaps sets a new position.

Notes

Servo motors are controlled by a pulse width modulated signal that is applied repeatedly at 20 ms intervals as long as the motor remains under control. The pulse width varies from 0 to 3 ms (this is standard for servo motors). Values from 0 to 1.5 ms rotate the motor to the left, values from 1.5 to 3.0 ms rotate it to the right. A pulse width of 1.5 ms sets the motor to the center of its rotation. These values are set by adjusting the *rotation* parameter (see above).[19]

The amount of rotation varies with different motors, from about 90 degrees to 270 degrees total. You may have to determine this amount by experiment if you don't have access to data sheets.

The control signal must be continuously applied or the motor will drift from its set position (i.e. it won't generate any torque with no signal). This implies that the SERVO command should be used in a program loop.

Since servo motors take time to reach the set position, the SERVO command repeats the pulse output for a sufficient time. Depending on the individual servo motor, and the amount of rotation change required, you may have to adjust the *repeat* parameter for the command.

For reference, the following wire colors are used by different manufacturers:

Manufacturer	Power (+5V)	GND (Vss)	Control
Airtronics	red	black	brown
Futaba J	red	black	white
KO Propo	red	black	blue
Kyosho/Pulsar	red	black	yellow
Japan Radio (JR)	red	brown	orange

Examples

```
pos var word
setpos
   code to determine desired position
   servo P4,pos,50
goto setpos
```

This simple program controls a servo connected to P4. The desired position may be determined by any number of different input factors, depending on the application. Since the 20 ms control sequence is repeated 50 times, the position may only be changed about once per second.

[19] Values of –1200 to +1200 don't actually cover the full 0 to 3 ms pulse width range. They are restrained at both ends to prevent over-rotation of the servo, which could cause damage.

MSERVO

Operates multiple servo motors. Similar to SERVO.

Syntax

MSERVO pin,
servo1{\servo2{\servo3{\servo4{\servo5{\servo6{\servo7{\servo8
{\servo9{\servo10{\servo11{\servo12}}}}}}}}}}}{, repeat}

> *Pin* is an expression that specifies the pin number controlling servo1. All other servos are controlled by the next pin (i.e. servo2, pin+1, servo3, pin+2 etc...)
>
> *Servo#* is an expression that specifies the position to which you want the servo to rotate. A value from -2400 to +2400* is used with 0 being center. The maximum +2400 and minimum -2400 will vary based on the servo being used.
>
> * Exceeding these values could damage your servo.
>
> *Repeat* (optional) Specifies the number of internal cycles the command runs (defaults to 20).

Notes

The MSERVO command is similar to the SERVO command except that it operates more than one servo motor. MSERVO is a foreground task.

See the SERVO command (page 160) for more detail on MSERVO parameters and the operation of servo motors.

Example

```
MServo p0,0,-1000,1000
```

Assumes three servos are attached to P0,P1 and P2. The pin argument specifies the first pin used.

SPMOTOR

Operates a stepper motor.

Syntax

spmotor pin,delay,step

> *pin* is a variable or constant specifying the lowest numbered of 4 output pins used. For example, if *pin* = P0, pins 0, 1, 2 and 3 will be used.
>
> *delay* is a variable or constant (0 – 65535) that specifies the delay in milliseconds between steps. The delay controls the speed at which the stepper motor operates.
>
> *step* is a variable or constant (-32682 to +32682) that specifies the number and direction of steps the stepper motor will execute. With correct wiring, positive values are clockwise, negative values are counterclockwise.

Notes

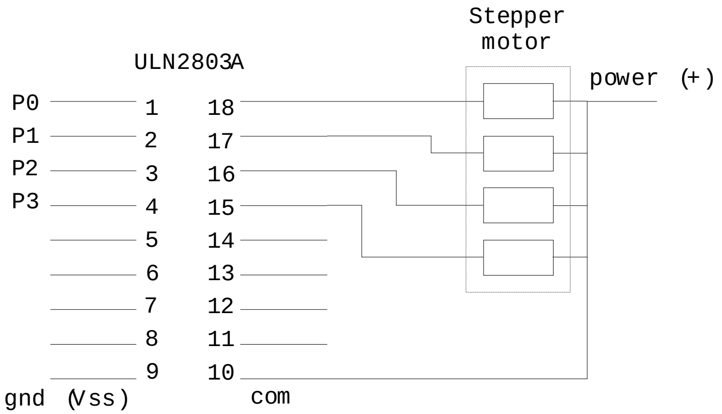
Stepper motors are precision devices that are used to control position or rotation in small increments. Each step moves the motor an absolute, predetermined amount (the amount varies with different stepper motors and may be determined by experiment, or by referring to the manufacturer's data sheets). Stepper motors are commonly found in XY positioning tables, graphing devices, disk drives, laser printers, etc.

Stepper motors may be unipolar or bipolar. Bipolar motors require slightly more complex control circuitry. Typically, unipolar steppers have 4 wires, bipolar have 5 wires.

The SPMOTOR command supports both unipolar and bipolar stepper motors. Wiring for bipolar motors is not described in this manual.

Since the inductive load of a stepper motor may exceed the ratings of the BasicATOM Pro, it should be driven using a buffer amplifier. The most common device for this purpose is the ULN2803A Darlington array, which includes protective diodes for inductive loads.

A sample circuit using a ULN2803A is shown below. Further circuit design and determining the correct wiring of the stepper motor are beyond the scope of this manual. Experimenting to find the right connections is a bit tedious, but you won't damage the motor by doing so.



Note: Data sheets and application notes for the ULN2803A are available on the internet: a quick search will let you find many resources.

Examples

```
delay var word
step var sword
delay = 50
step = -60
spmotor P0, delay, step
```

will cause the stepper motor to make 60 counterclockwise steps at intervals of 50 ms.

LCD Commands

The LCD commands in this section are specifically designed for use with the Hitachi 44780 controller (or equivalent). If you use an LCD module with a different controller, these commands probably will not work. In such cases, you can write your own subroutines to initialize your LCD and send output to it.

A detailed explanation of the Hitachi LCD controller is beyond the scope of this manual. The 44780 controller is use for text-mode displays, these commands don't apply to graphical LCD displays.

Note: Some LCD modules use serial I/O. If you have such a module, the normal SERIN, SEROUT, HSERIN and HSEROUT commands may be used to read from and write to the module.

LCDINIT

Initializes an LCD display. This command must be used before using the LCDREAD or LCDWRITE commands.

Syntax

lcdinit regsel\clk\D7\D6\D5\D4\RdWrPin

regsel is a constant or variable specifying the Atom Pro I/O pin connected to the LCD's R/S line.

clk is a constant or variable specifying the Atom Pro I/O pin connected to the LCD's E (Enable) line.

D7 – D4 are constants or variables that specify the Atom Pro pins to use for the LCD data lines.

RdWrPin is an optional constant or variable specifying the Atom Pro I/O pin connected to the R/W (read/write) line of the LCD.

Examples

If your LCD display uses the following connections:

LCD	Atom Pro
R/S	P4
E	P5
R/W	P6
I/O port (4-7)	P0, P1, P2, P3

The command:

```
lcdinit p4\p5\p3\p2\p1\p0\p6
```

will initialize the display for future LCDREAD and LCDWRITE commands.

LCDREAD

Reads the RAM on an LCD module using the Hitachi 44780 controller or equivalent.

> You must initialize the display with LCDINIT before using this command.

Syntax

ldcread regsel\clk\D7\D6\D5\D4,RdWrPin, address, [(modifiers} var]

regsel is a constant or variable specifying the Atom Pro I/O pin connected to the LCD's R/S line.

clk is a constant or variable specifying the Atom Pro I/O pin connected to the LCD's E (Enable) line.

D7 – D4 are constants or variables specifying the Atom Pro I/O pins connected to the LCD’s data lines (see *lcdinit*).

RdWrPin is a constant or variable specifying the Atom Pro I/O pin connected to the R/W (read/write) line of the LCD.

address is a constant or variable that specifies the RAM location to be read, according to this list:

address	contents
1 – 127	current character in display RAM
128 and above	character RAM values

modifiers are command modifiers (see 0Command Modifiers) used to modify *var*.

var is a byte variable (or a comma – separated list of variables) where the returned value will be stored.

Examples

If your LCD display uses the same connections as shown under LCDINIT, the program segment:

```
character var byte
lcdread p4\p5\p3\p2\p1\p0\p6,15,[character]
```

will read the character stored at RAM address 15 and save it in *character*.

LCDWRITE

Sends text output to an LCD display module that uses the Hitachi 44780 controller or equivalent.

> You must initialize the display with LCDINIT before using this command.

Syntax

lcdwrite regsel\clk\D7\D6\D5\D4,{RdWrPin,} [(modifiers} expr]

> *regsel* is a constant or variable specifying the Atom Pro I/O pin connected to the LCD's R/S line.
>
> *clk* is a constant or variable specifying the Atom Pro I/O pin connected to the LCD's E (Enable) line.
>
> *D7 – D4* are constants or variables specifying the Atom Pro I/O pins connected to the LCD's data lines (see *lcdinit*).
>
> *RdWrPin* is an optional constant or variable specifying the Atom Pro I/O pin connected to the R/W (read/write) line of the LCD.
>
> *modifiers* are command modifiers (see 0Command Modifiers) used to modify *expr*.
>
> *expr* is a variable, constant or expression that generates the data to be written. Data may be text characters or commands. A list of commands is given under Notes, below.

Notes

Here is a list of commands that can be used with the LCDWRITE command. Multiple commands may be included inside [...] if they are comma separated.

Command	**Value**	**Function**
initlcd1	$133	initialize LCD display
initlcd2	$132	initialize LCD display
Clear	$101	clear display
Home	$102	return to home position
Inccur	$104	auto increment cursor (default)
Incscr	$105	auto increment display

Command	Value	Function
Deccur	$106	auto decrement cursor
Decscr	$107	auto decrement display
Off	$108	display, cursor and blink OFF
Scr	$10C	display ON, cursor and blink OFF
Scrblk	$10D	display and blink ON, cursor OFF
Scrcur	$10E	display and cursor ON, blink OFF
Scrcurblk	$10F	display, cursor and blink ON
Curleft	$110	move cursor left
Currright	$114	move cursor right
Oneline	$120	set display for 1 line LCDs
Twoline	$128	set display for 2 line LCDs
cgram | address	$140	set CGRAM address for R/W
scrram | address	$180	set display RAM address for R/W

Examples

If your LCD display uses the same connections as shown under LCDINIT, the program segment:

```
printch var byte
printch = $41       ; ASCII value for "A"
lcdwrite p4\p5\p3\p2\p1\p0,p6, [clear,home,printch]
```

will clear the screen, move to the home position, and print an A on the display.

Memory, Interrupts, Timers, etc.

This chapter includes specialized commands used for accessing memory directly, responding to interrupts and using the Atom Pro's built-in timers.

Conventions Used in this Chapter

{ ... }	represent **optional** components in a command. The { } are **not** to be included.
[...]	used for lists – the [ ] are **required**
(...)	used for some arguments. The ( ) are **required**

Memory Commands

Memory commands may be used to access memory directly. RAM, EEPROM and even program memory may be accessed, depending on the command.

Note: The use of program memory for data storage is described on page 45 under Tables.

More details concerning various types of memory are found on page 35.

Memory commands are provided for advanced programmers, and memory usage and addressing is not discussed in detail in this manual. Users should note that all PIC16F876/7 registers are available via pre-defined variables, contents of variables are available via their names, etc. so in the majority of cases the commands described in this section are not essential. We have provided these commands for backwards compatibility, and for the convenience of advanced programmers with unusual applications.

> For details of register and memory addressing and use please consult the H8/3664F or H8/3687F hardware manual, available at http://www.renesas.com

PEEK, POKE

These commands are used to read and write to RAM locations.

Syntax

peek address, variable

poke address, expression

> *address* is a variable or constant that specifies the RAM location (see the Warning under Notes, below).
>
> *variable* is a byte variable used to store the contents of the RAM location for the PEEK command.
>
> *expression* is a variable, constant or expression that provides an 8 bit value to be stored in RAM with the POKE command.

Notes

PEEK and Poke start at the begining of RAM so the first ram location is 0 indexed. The actual ram location in the Hitachi memory map is 0xF780 to 0xFF7F

Registers start at 0xFF80, but because PEEK and POKE automatically offset 0 to 0xF780 the registers appear to start at 0x800

In most cases it is easier to use variable names (or register names, all of which are available in Atom BASIC) to access memory.

> Warning! Since RAM is used to store the Atom Pro's internal registers as well as user data, careless use of the POKE command could adversely affect the operation of the controller chip. Make sure you fully understand the memory map of the H8/3664 before using POKE.

Examples

```
regvalue var byte
peek $1F,regvalue   $1F is the address of register adcon0
```

will give the same result as

```
regvalue var byte
regvalue = adcon0   ; adcon0 is a pre-defined variable
```

READ, WRITE

These commands are used to read and write one byte at a time to EEPROM locations of Atom Pro Plus modules (see *Notes* below). In this respect they are equivalent to PEEK and POKE which read and write to RAM locations.

Note: Although PEEK and POKE are usually redundant because there exist more convenient ways to access RAM, these other ways don't exist for EEPROM, therefore READ and WRITE are quite useful.

Syntax

read address variable

write address expression

address is a variable or constant that specifies the EEPROM address (0 – 255) to read from or write to.

variable is a byte variable which will store the value read from EEPROM.

expression is a variable, constant or expression that generates the 8 bit value to store at *address*.

Notes

The Atom Pro 24 and 28 pin modules do not have EEPROM so these commands don't apply. The Atom Pro 40 pin (Atom Pro Plus) module has 4K EEPROM.

An external I^2C EEPROM may be connected to p10 (SCL)/p11 (SDA) of the Atom Pro modules to use the READ and WRITE commands.

READ and WRITE execute when your program is running, not when it is first written to program memory. This lets you change EEPROM values "on the fly". Users should note the following:

- EEPROM can be read an indefinite number of times at the same rate as RAM can be read.
- EEPROM can be written to only a limited number of times (around 10 million), so it's generally better to use RAM for values that change frequently.
- EEPROM is much slower to write than is RAM, so unnecessary use of WRITE can slow down program execution.

For example, if you change an EEPROM value once per second (about 86400 times per day) your Atom Pro's EEPROM could be worn out in about 116 days. This is to be avoided.

Examples

```
contents var byte
read 100,contents
```

will read EEPROM address 100 and store the 8 bit result in *contents*.

READDM, WRITEDM

Read or write a sequence of values from/to EEPROM locations of supported Atom Pro modules. Except that many values may be read or written by one command, these commands are essentially identical to READ and WRITE, above.

Syntax

readdm address,[{modifier} var, ... {modifier} var]

writedm address,[{modifier} expr, ... {modifier} expr]

> *address* is a variable or constant that specifies the first EEPROM address to read from or write to. Subsequent reads or writes within the same command will be sequential.
>
> *modifier* is any valid command modifier (see page 69). See the HSERIN and HSEROUT commands for examples of the use of these modifiers.
>
> *var* is a variable, or sequence of byte variables, in which the results of the EEPROM reads will be stored.
>
> *expr* is a byte variable, constant or expression (or a sequence of such) that generates data to be stored in EEPROM. Values are stored sequentially beginning at *address*.

Notes

See the READ, WRITE commands in the previous section for supported models and other information.

Examples

```
temp var byte(5)
readdm 100,[temp(0),temp(1),temp(3)]
```

will read the values at addresses 100, 101 and 102 and store them in temp(0), temp(1) and temp(2) respectively.

```
temp var byte(5)
code to set values of temp(n)
writedm 100,[dec1 temp(0), dec1 temp(1)]
```

will write the values of temp(0) and temp(1), converted to decimal ASCII characters, in EEPROM addresses 100 and 101, respectively. Only one decimal digit (the least significant) is written in each case.

Interrupts

Interrupts allow immediate processing of high-priority tasks. The Atom Pro uses a number of interrupts, based on timers and other events, as well as providing for external (hardware generated) interrupts.

Atom BASIC allows access to the microcontroller's built-in interrupt processing via BASIC commands. Explanation of uses and operation of interrupts is beyond the scope of this manual.

> For more information on the operation of interrupts, refer to the H8/3664F or H8/3687F hardware manual, available at http://www.renesas.com

The following interrupts are available for use in the Atom Pro:

All Modules

Name	Description
IRQ0INT	Irq0 pin interrupt
IRQ1INT	Irq1 pin interrupt
IRQ2INT	Irq2 pin interrupt
IRQ3INT	Irq3 pin interrupt
WKPINT_0	WKP0 pin onchange interrupt
WKPINT_1	WKP1 pin onchange interrupt
WKPINT_2	WKP2 pin onchange interrupt
WKPINT_3	WKP3 pin onchange interrupt
WKPINT_4	WKP4 pin onchange interrupt
WKPINT_5	WKP5 pin onchange interrupt
TIMERVINT_OVF	TimerV overflow interrupt
TIMERVINT_CMEB	TimerV compare match A int
TIMERVINT_CMEA	TimerV compare match B int
SCI3INT_TDRE	Transmit Data Register Empty interrupt
SCI3INT_RDRF	Read Data Register Full interrupt
SCI3INT_TEND	Transmit End interrupt
SCI3INT_OER	Overflow Error interrupt
SCI3INT_FER	Frame Error interrupt
SCI3INT_PER	Parity Error interrupt

Name	Description
IICINT	I2C interrupt
ADINT	Analog conversion complete int
HSERIALINT_TDRE	Transmit Data Register Empty interrupt
HSERIALINT_RDRF	Read Data Register Full interrupt
HSERIALINT_TEND	Transmit End interrupt
HSERIALINT_OER	Overflow Error interrupt
HSERIALINT_FER	Frame Error interrupt
HSERIALINT_PER	Parity Error interrupt.98
HSERVOINT_IDLE	Any Servo Idle interrupt
HSERVOINT_IDLE0-31	# Servo Idle interrupt
HSERVOINT_USER	HServo User Interrupt
HSERVOINT	Hservo Interrupt

ATOM-Pro only (H8/3664/3694)

Name	Description
TIMERAINT	Overflow interrupt
TIMERWINT_OVF	Overflow interrupt
TIMERWINT_IMIEA	Capture/Compare Match A int
TIMERWINT_IMIEB	Capture/Compare Match B int
TIMERWINT_IMIEC	Capture/Compare Match C int
TIMERWINT_IMIED	Capture/Compare Match D int

ATOM-Pro Plus only (H8/3687)

Name	Description
RTCINT	Real time clock interrupt
TIMERZ0INT_OVF	Overflow interrupt
TIMERZ0INT_IMIEA	Capture/Compare Match A int
TIMERZ0INT_IMIEB	Capture/Compare Match B int
TIMERZ0INT_IMIEC	Capture/Compare Match C int
TIMERZ0INT_IMIED	Capture/Compare Match D int
TIMERZ1INT_UDF	Underflow interrupt
TIMERZ1INT_OVF	Overflow interrupt

Name	Description
TIMERZ1INT_IMIEA	Capture/Compare Match A int
TIMERZ1INT_IMIEB	Capture/Compare Match B int
TIMERZ1INT_IMIEC	Capture/Compare Match C int
TIMERZ1INT_IMIED	Capture/Compare Match D int
TIMERB1INT	Overflow interrupt
SCI3_2INT_TDRE	Transmit Data Register Empty interrupt
SCI3_2INT_RDRF	Read Data Register Full interrupt
SCI3_2INT_TEND	Transmit End interrupt
SCI3_2INT_OER	Overflow Error interrupt
SCI3_2INT_FER	Frame Error interrupt
SCI3_2INT_PER	Parity Error interrupt
HSERIAL2INT_TDRE	Transmit Data Register Empty interrupt
HSERIAL2INT_RDRF	Read Data Register Full interrupt
HSERIAL2INT_TEND	Transmit End interrupt
HSERIAL2INT_OER	Overflow Error interrupt
HSERIAL2INT_FER	Frame Error interrupt
HSERIAL2INT_PER	Parity Error interrupt.99

Interrupts must be enabled before they can be used. You can enable and disable them globally or individually, using the ENABLE and DISABLE commands, described below.

ENABLE, DISABLE

Enables or disables one or all interrupts. ENABLE must be used before interrupts will work. DISABLE prevents the specified interrupt from working.

Syntax

enable {intname}

disable {intname}

> *intname* must be one of the interrupt names from the table on the previous page. *Intname* is optional, if it is omitted all interrupts will be enabled or disabled.

ONINTERRUPT

This is a compile time function that sets the label that the specified interrupt will jump to when it occurs. You must also enable the interrupt before it will work.

Syntax

oninterrupt intname, label

> *intname* is the name of the desired interrupt (see the table on page 174).
>
> *label* is the label to which program execution will jump when this interrupt occurs.

Note: You must use the RESUME command (see below) to return to normal program execution after your interrupt has been processed.

Example

```
ONINTERRUPT TIMERAINT,IntHandler
TMA = (TMA & 0xF0) | 0x4   ;Sets TimerA to increment once
                           ;every 256 clock cycles.
timer var long
ENABLE TIMERAINT

main
   serout S_OUT,i9600,[dec timer,13]
   pause 100
   goto main

IntHandler
   timer = timer + 1
   resume
```

RESUME

Return from interrupt. This command is used to return to the point in your program where execution was interrupted. It should be used at the end of the interrupt processing code for ONINTERRUPT.

Syntax

resume (no arguments)

Examples

See the example under ONINTERRUPT on page 177.

This page intentionally left blank

SECTION 3: Miscellaneous

This page intentionally left blank

Questions and Answers

These questions and answers summarize points discussed in the text.

How do I start the BasicATOM Pro's program running?

If you've already downloaded the program, just connect the power or press the RESET button on the development board.

If you're compiling a program using the Program button in the IDE, it will start running as soon as the download is complete.

If you're compiling a program using the Debug button in the IDE, it will start when you press the RUN button.

How can I run my program a second time with out re-compiling it?

Once your program ends, the Atom Pro goes to "sleep" and no longer responds to commands. Press the RESET button on the development board, or turn the power OFF and back ON again to re-run your program.

Do I lose my program if I disconnect the power from the BasicATOM Pro?

No. Your program is stored in semi-permanent "flash" memory and will be retained until it's overwritten by a new program.

Once I've compiled a program can I use the object code to program many BasicATOM Pros?

Although the object code is saved on your PC, it can't be re-used. Just re-compile the source program for each BasicATOM Pro you want to program.

Once I compile "traffic.bas" I see the files "traffic.bin" and "traffic.pdb" in the same directory. What are they?

These are files generated by the compiler; "traffic.bin" is the compiled object code that's downloaded to the BasicATOM Pro, and "traffic.pdb" is a file used in debugging.

You can safely delete both of these files once everything's working.

What's the difference between the 24, 28 and 40 pin BasicATOM Pro modules?

Mainly in the number and type of I/O pins. See the table on page 5 as well as the data sheets for more information. The 24 pin Atom Pro also has the analog pins available as solder pads rather than connected to pins.

Can I build my own board for a BasicATOM Pro module?

Yes. You might consider using a development board for designing your project and programming the BasicATOM Pro module. Then just transfer the programmed module to your own circuit. This has the added advantage that you don't need to include RS-232 circuitry on your project board.

When I define variables, how are they allocated in memory?

Variables are allocated sequential memory space, in the order in which they are defined. For example, if you define byte variables as follows:

```
a var byte
b var byte
c var byte
```

they will occupy three successive bytes in memory.

Tell me more about arrays.

Arrays in Atom Basic are simply a series of bytes, words, etc. assigned in sequence. For example, if you define an array as

```
a var byte(10)
```

this will set aside 10 sequential memory locations, numbered 0 to 9. If you subsequently define another variable, it will be assigned the next sequential memory location. So

```
b var byte
```

will assign b to the 11th memory location following a(0).

In fact, you can actually access "b" as "a(10)" since Atom Basic doesn't check for out of range array subscripts.

Does that mean that array and simple variables can overlap?

Yes, depending on how the variables are defined. For simplicity, let's assume that we're using byte variables, and we define a, b, c, d and e as successive byte variables.

You can now actually access, say, "c" as "a(2)" even though "a" was not defined as an array. Be careful doing this if your variables aren't all of the same type: it could get confusing.

Can you suggest any use for the above?

Yes, there are many possibilities. One example would be to output a number of variables by "pretending" that they are an array. So if we had the same five variables (a, b, c, d, e) as defined in the previous answer, they could be output with a single statement such as:

```
hserout str a\5
```

Note: The values will be output in numeric form without conversion to ASCII characters.

How do I find out more technical information about the Atom Pro?

Hardware technical information is can be found on the Atom Pro 24, 28 or 40 pin module data sheets. These are available online at http://www.basicmicro.com or may be ordered from Basic Micro.

Detailed hardware, feature and programming information can be found in the H8/3664F or H8/3687F hardware manuals. Both are available online at http://www.renesas.com

By a careful comparison among Atom BASIC commands, and the Atom Pro and PIC data sheets, you can learn a great deal about how the Atom Pro works.

Atom BASIC doesn't have a command to do exactly what I need. Can I address the microcontroller directly?

Yes. Atom BASIC includes pre-defined variables that let you access the microcontroller's registers directly. These registers are bit mapped so you should be familiar with this type of operation before attempting to use registers. **The use of registers directly is for advanced programmers only.**

Warning: Be sure you are thoroughly familiar with the operation of the microcontroller before trying to manipulate registers directly. Basic Micro neither documents nor supports (via technical support) such operation.

Module Pinouts

The following tables are provided so you can match H8 pins (as shown in the Renesas documentation) with Atom Pro and Pro Plus module pins. Note that in some cases two H8 pins are connected to one Atom Pro pin; such pins are tri-state and only one function can be used at a time.

<table>
<tr><th>Pro 24</th><th colspan="2">H8/3664F</th><th>Pro 24</th><th colspan="2">H8/3664F</th></tr>
<tr><td>P0</td><td>P50/WKP0</td><td>PB0/AN0</td><td>P8</td><td>P80/FTCI</td><td>P15/IRQ1</td></tr>
<tr><td>P1</td><td>P51/WKP1</td><td>PB1/AN1</td><td>P9</td><td>P81/FTIOA</td><td>P74/TMRIV</td></tr>
<tr><td>P2</td><td>P52/WKP2</td><td>PB2/AN2</td><td>P10</td><td>P82/FTIOB</td><td>P75/TMCIV</td></tr>
<tr><td>P3</td><td>P53/WKP3</td><td>PB3/AN3</td><td>P11</td><td>P83/FTIOC</td><td>P76/TMOV</td></tr>
<tr><td>P4</td><td colspan="2">P54/WKP4</td><td>P12</td><td>P84/FTIOD</td><td>P10/TMOW</td></tr>
<tr><td>P5</td><td colspan="2">P55/WKP5/ADTRG</td><td>P13</td><td>P85</td><td>P20/SCK3</td></tr>
<tr><td>P6</td><td>P56/SDA</td><td>P16/IRQ2</td><td>P14</td><td>P86</td><td>P21/RXD</td></tr>
<tr><td>P7</td><td>P57/SCL</td><td>P17/IRQ3</td><td>P15</td><td>P87</td><td>P22/TXD</td></tr>
</table>

<table>
<tr><th>Pro 28</th><th colspan="2">H8/3664F</th><th>Pro 28</th><th colspan="2">H8/3664F</th></tr>
<tr><td>P0</td><td>P50/WKP0</td><td>PB0/AN0</td><td>P8</td><td>P80/FTCI</td><td>P15/IRQ1</td></tr>
<tr><td>P1</td><td>P51/WKP1</td><td>PB1/AN1</td><td>P9</td><td>P81/FTIOA</td><td>P74/TMRIV</td></tr>
<tr><td>P2</td><td>P52/WKP2</td><td>PB2/AN2</td><td>P10</td><td colspan="2">P82/FTIOB</td></tr>
<tr><td>P3</td><td>P53/WKP3</td><td>PB3/AN3</td><td>P11</td><td colspan="2">P83/FTIOC</td></tr>
<tr><td>P4</td><td>P54/WKP4</td><td></td><td>P12</td><td>P84/FTIOD</td><td>P10/TMOW</td></tr>
<tr><td>P5</td><td colspan="2">P55/WKP5/ADTRG</td><td>P13</td><td>P85</td><td>P20/SCK3</td></tr>
<tr><td>P6</td><td colspan="2">P56/SDA</td><td>P14</td><td>P86</td><td>P21/RXD</td></tr>
<tr><td>P7</td><td colspan="2">P57/SCL</td><td>P15</td><td>P87</td><td>P22/TXD</td></tr>
<tr><td>P16</td><td>P11</td><td>PB7/AN7</td><td>P18</td><td>P16/IRQ2</td><td>PB5/AN5</td></tr>
<tr><td>P17</td><td>P12</td><td>PB6/AN6</td><td>P19</td><td>P17/IRQ3</td><td>PB4/AN4</td></tr>
</table>

ProPlus 40 – Pinouts to be determined.

Glossary

Argument	A constant, variable or other value used by a function to calculate a result.
Breadboard	Hardware development board with interconnected sockets for wire jumpers and component leads.
Compiler	A computer program that accepts commands in a source language (designed for easy human manipulation) and uses them to generate an “object” program which is then run on the same or another computer. The Basic Micro IDE includes a compiler.
Constant	A program item that has a fixed value that does not change during program operation.
decimal	Numbers based on the decimal system, i.e. powers of 10. Decimal numbers may or may not have a fractional part.
DIP	Dual in line package. A package for integrated circuits, resistor networks, opto-electronics, etc. with two rows of pins.
EEPROM	Electronically Erasable Programmable ROM. A type of read only memory that can be modified as often as needed. Contents are retained during power off periods.
Flash	A type of read-only memory that can be modified during programming. Contents are retained during power off periods until they are explicitly modified again. May have a limited number of write cycles.
IC	Integrated circuit
IDE	Integrated Development Environment – Basic Micro’s software development program.
integer	A positive, negative or unsigned number with no fractional part.
LED	Light Emitting Diode. A semiconductor device that radiates visible or infrared light when a current passes through it.
LSB	Least Significant Bit. The rightmost bit or bits in a number. For example, in the number %10001010 the LSB is "0". (Sometimes used as Least Significant Byte.)
Microcontroller	A special-purpose microcomputer chip designed for control applications.

MSB	Most Significant Bit. The leftmost bit or bits in a number. For example, in the number %10001011 the MSB is "1". (Sometimes used as Most Significant Byte.)
Object code	The compiled result of a BasicATOM Program which is downloaded to the BasicATOM Pro module.
PC	Personal Computer. For purposes of this manual a PC is an Intel (or similar) based computer running Windows 95, 98, ME, XP, NT4 or 2000.
PCB	Printed circuit board
Plated through hole	A hole in a PCB that’s metal plated on both sides and through the hole itself. Used as a solder point for one or more connections, and to connect traces on both sides of the PCB.
RAM	Random Access Memory. A memory area used for storage of variables during program operation. Contents are not maintained during power off periods.
ROM	Read Only Memory. Memory for storing programs and constants that are permanent or semi-permanent. See also “Flash”.
RS-232	Serial data interface standard used to interconnect computers and hardware.
Runtime library	That part of the object code that includes support for all the functions used in a program.
Serial data	Data that is sent in sequence, one bit at a time, over a single wire.
SIP	Single in line package. A package for integrated circuits, resistor networks, etc. with a single row of pins spaced 0.10 inch apart.
Stack	An area in RAM used to store temporary values or addresses that change during program operation.
Target variable	The variable used to store the result of a calculation.
Variable	A program item which has a value that may change during program operation.
Vdd	Positive voltage (drain voltage)
Vss	Negative or ground voltage (source voltage)

List of Reserved Words

Reserved words can not be used as labels, constants or variables. All command names are reserved words. The table below lists all Atom Pro BASIC reserved words. (Note that the list includes all Atom BASIC reserved words, as well as those reserved only for the Atom Pro and Atom Pro Plus.)

All math functions
Any name starting with a "_"
Any name starting with a number.
AAS (Pro only)
AASX (Pro only)
ABIE
ABIF
ABRKCR
ABRKSR
ACKB (Pro only)
ACKE (Pro only)
ACMP0
ACMP1
ACMP2
ADCR
ADCSR
ADDRA
ADDRB
ADDRC
ADDRD
ADF
ADIE
ADINT
ADST
ADZ (Pro only)
AL (Pro only)
B0WI
B2WI
B4WI
B6WI
BARH
BARH0 *to* BARH7
BARL
BARL0 *to* BARL7
BBSY (Pro only)
BC0 (Pro only)
BC1 (Pro only)
BC2 (Pro only)
BDRH
BDRH0 *to* BDRH7
BDRL
BDRL0 *to* BDRL7
BREAKINT
BRR
BRR_2 (Pro Plus only)
BRR0 *to* BRR7
BUFEA (Pro only)
BUFEB (Pro only)
CCLR0
CCLR1
CGRAM
CH0
CH1
CH2
CHR
CKE0
CKE1
CKS
CKS0
CKS1
CKS2
CKS3
CLEAR
CMFA
CMFB
CMIEA
CMIEB
COM
CSEL0
CSEL1
CURLEFT
CURRIGHT
DCMP0
DCMP1
DECCUR
DECSCR
DIR0 *to* DIR31
DIRA

DIRB
DIRC
DIRD
DIRE
DIREA
DIREB
DIREC
DIRECTACTIVE
DIRECTACTIVE_8
DIRECTACTIVE_16
DIRECTACTIVE_32
DIRECTACTIVE_64
DIRECTACTIVERES
DIRECTACTIVERES_8
DIRECTACTIVERES_16
DIRECTACTIVERES_32
DIRECTACTIVERES_64
DIRECTINT
DIRED
DIREH
DIREL
DIRES
DIRH
DIRL
DIRS
DTON
E
EB0
EB1
EB2
EB3
EB4
EBR1
ESTP (Pro only)
ESU
EV
FASTLSBPOST
FASTLSBPRE
FASTMSBPOST
FASTMSBPRE
FENR
FER
FLER
FLMCR1
FLMCR2
FLPWCR
FLSHE
FS (Pro only)
FSX (Pro only)
GRA (Pro only)
GRA_0 (Pro Plus only)
GRA_1 (Pro Plus only)
GRA0 *to* GRA15 (Pro only)
GRB (Pro only)
GRB_0 (Pro Plus only)
GRB_1 (Pro Plus only)
GRB0 *to* GRB15 (Pro only)
GRC (Pro only)
GRC_0 (Pro Plus only)
GRC_1 (Pro Plus only)
GRC0 *to* GRC15 (Pro only)
GRD (Pro only)
GRD_0 (Pro Plus only)
GRD_1 (Pro Plus only)
GRD0 *to* GRD15 (Pro only)
H*baud where baud is any of:*
300, 600, 1200, 2400, 4800
7200, 9600, 12000, 14400
16800, 19200, 21600, 24000
26400, 28800, 31200, 33600
36000, 38400, 57600, 62500
115200, 125000, 250000
312500, 500000
H1STOPBITS
H2STOPBITS
H7DATABITS
H8DATABITS
HEVENPARITY.144
HNOPARITY
HODDPARITY
HOME
HSERIAL2INT_FER (Pro Plus only)
HSERIAL2INT_OER (Pro Plus only)
HSERIAL2INT_PER (Pro Plus only)
HSERIAL2INT_RDRF (Pro Plus only)
HSERIAL2INT_TDRE (Pro Plus only)
HSERIAL2INT_TEND (Pro Plus only)
HSERIALINT_FER (Pro Plus only)

IOD1 (Pro only)
IOMODE
IRIC (Pro only)
IRQ0
IRQ0INT
IRQ1
IRQ1INT
IRQ2
IRQ2INT
IRQ3
IRQ3INT
IRR1
IRR2 (Pro Plus only)
IRRDT
IRRI0
IRRI1
IRRI2
IRRI3
IRRTA
IRTR (Pro only)
IWPF0
IWPF1
IWPF2
IWPF3
IWPF4
IWPF5
IWPR
LSBFIRST
LSBPOST
LSBPRE
LSON
LVDCR (Pro Plus only)
LVDSR (Pro Plus only)
MA0
MA1
MA2
MLS (Pro only)
MP
MPBR
MPBT
MPIE
MSBFIRST
MSBPOST
MSBPRE
MST (Pro only)
MSTAD
MSTCR1
MSTIIC
MSTS3
MSTTA
MSTTV
MSTTW
MSTWD
N*baud*
NE*baud*
NEO*baud*
NO*baud where baud is any of:*
300, 600, 1200, 2400, 4800
7200, 9600, 12000, 14400
16800, 19200, 21600, 24000
26400, 28800, 31200, 33600
36000, 38400, 57600, 115200
230400, 460800
NEMODE
NEOMODE
NESEL
NMIEG
NMODE
NOMODE
OCKS0
OER
OFF
ONELINE
OS0
OS1
OS2
OS3
OUT0 *to* OUT31
OUTA
OUTB
OUTC
OUTD
OUTE
OUTEA
OUTEB
OUTEC
OUTED
OUTEH
OUTEL
OUTES
OUTH
OUTL
OUTS
OVF
OVIE
P

SA1
SAR (Pro only)
SAR (Pro Plus only)
SARX (Pro only)
SCAN
SCI3_2INT_FER (Pro Plus only)
SCI3_2INT_OER (Pro Plus only)
SCI3_2INT_PER (Pro Plus only)
SCI3_2INT_RDRF (Pro Plus only)
SCI3_2INT_TDRE (Pro Plus only)
SCI3_2INT_TEND (Pro Plus only)
SCI3INT_FER
SCI3INT_OER
SCI3INT_PER
SCI3INT_RDRF
SCI3INT_TDRE
SCI3INT_TEND
SCP (Pro only)
SCR
SCR3
SCR3_2 (Pro Plus only)
SCRBLK
SCRCUR
SCRCURBLK
SCRLEFT
SCRRAM
SCRRIGHT
SLEEPACTIVE
SLEEPACTIVE_8
SLEEPACTIVE_16
SLEEPACTIVE_32
SLEEPACTIVE_64
SLEEPSTANDBY
SMR
SMR_2 (Pro Plus only)
SMSEL
SSBY
SSR
SSR_2 (Pro Plus only)
STOP
STS0
STS1
STS2
SVA0 (Pro only)
SVA1 (Pro only)
SVA2 (Pro only)
SVA3 (Pro only)
SVA4 (Pro only)
SVA5 (Pro only)
SVA6 (Pro only)
SVAX0 (Pro only)
SVAX1 (Pro only)
SVAX2 (Pro only)
SVAX3 (Pro only)
SVAX4 (Pro only)
SVAX5 (Pro only)
SVAX6 (Pro only)
SWE
SYSCR1
SYSCR2
TCA (Pro only)
TCA0 *to* TCA7 (Pro only)
TCB1 (Pro Plus only)
TCNT (Pro only)
TCNT_0 (Pro Plus only)
TCNT_1 (Pro Plus only)
TCNT0 (Pro only)
TCNT1 *to* TCNT15 (Pro only)
TCNTV
TCNTV0 *to* TCNTV7
TCORA
TCORA0 *to* TCORA7
TCORB
TCORB0 *to* TCORB7
TCR_0 (Pro Plus only)
TCR_1 (Pro Plus only)
TCRV0
TCRV1
TCRW (Pro only)
TCSRV
TCSRWD
TCSRWE
TCWD
TCWD0 *to* TCWD7
TCWE
TDR
TDR_2 (Pro Plus only)
TDR0 *to* TDR7
TDRE
TE
TEIE
TEND
TFCR (Pro Plus only)
TIE
TIER_0 (Pro Plus only)
TIER_1 (Pro Plus only)

X_Hail
X_I
X_J
X_K
X_L
X_Lights_Off
X_Lights_On
X_M
X_N
X_O
X_Off
X_On
X_P
X_Status_Off
X_Status_On
X_Status_Request
X_Units_On

This page intentionally left blank

Index of Commands

This page intentionally left blank

Main Index