EzSBC.

Language Reference
Control BASIC Reference Manual

CdB

v 1.07

Table of Contents

The aim of this document is serve as a reference manual for the EzSBC1 and Control BASIC.

Conventions

Programs are displayed as follows:

```
REPEAT
    OUTD 41,0
    WAIT ontime
    OUTD 41,1
    WAIT offtime
UNTIL INKEY > 0
```

Keystrokes and Keys to press are shown as [Ctrl-W] and it should be understood to mean press the Ctrl key followed by the W key and letting go of the keys in the reverse order.

Concepts

To distinguish between different parts of the controller we need to use the same terminology. EzSBC1 refers to the entire module, all the components on the PC board, the pins. Control BASIC or sometimes just BASIC refers to the language used to program the EzSBC1. It is likely to be slightly different from other BASIC dialects that you may be familiar with. The details of Control BASIC and its keywords and reserved words are described the Control BASIC Language Reference. All user programs executed on the EzSBC1 are written in Control BASIC.

Input and Output can be very confusing because the direction depends on the viewpoint of the author or reader. In this document direction is described from the view of the EzSBC1. Pretend you are sitting on the EzSBC1 then IN or Input is towards you and OUT or Output is away from you (and the EzSBC1). When a signal is described as an Output then the EzSBC1 will drive the pin in some way. It will source (or sink) some current and try to force a voltage on the pin. The EzSBC1 is in control of the state of the pin.

When a signal or a pin is described as an Input the EzSBC1 will try to 'read' or 'evaluate' the pin. In most cases the pin will approximate an open circuit; the EzSBC1 will draw little or no current from the pin or signal. Something other than the EzSBC1 should drive the pin or control the voltage on the pin.

When a BASIC program is not running on the EzSBC1 then it is waiting for user input. A program is still running on the EzSBC1 and it is referred to as the EZmon. It is a "monitor program", a program which is loaded onto a Single Board Computer (SBC) at the time of manufacture. It provides a few features and controls over the SBC. It provides a functionality that is always there and can be relied upon. It makes it easier to develop software or programs on the Single Board Computer. Output to and from EZmon occurs via USB that emulates a serial port and will appear on the PC as a serial port.

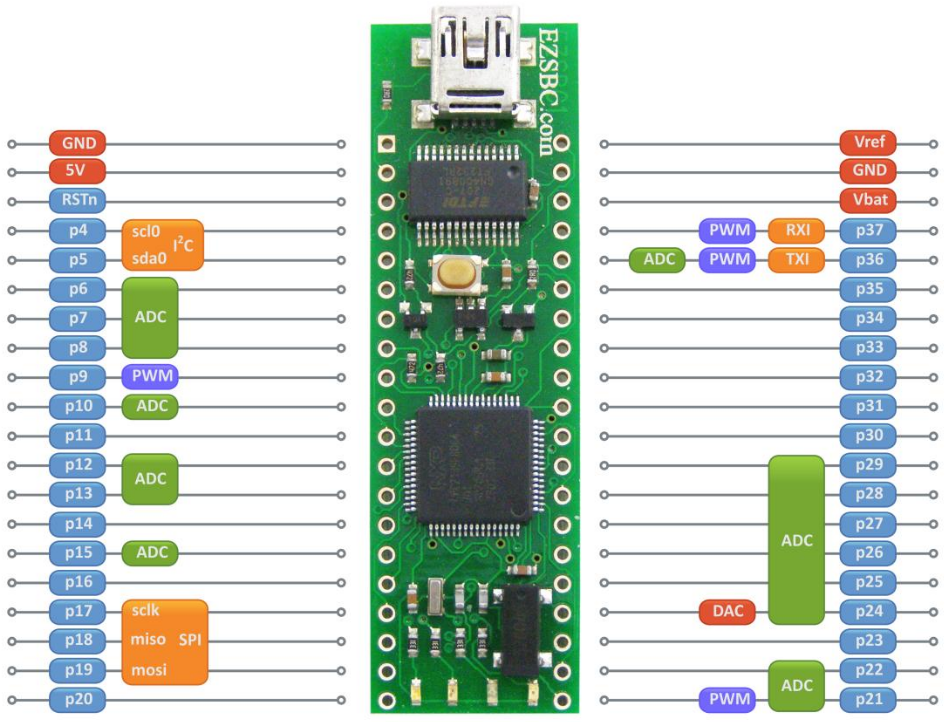
Introduction to the EzSBC1

The Name EzSBC1

The name is pronounced E zee S B C one and is an acronym for Easy Single Board Controller. The EzSBC1 is the first in a range of Single Board Controllers all aimed at simplifying the development of embedded controllers.

Physical Design

The EzSBC1 is approximately the same size as a 40 pin DIP IC and it can be plugged directly into a 40 pin IC socket or solder-less breadboard. See the picture below. The EzSBC1 has two connectors, a mini-USB plug for communicating with a computer during program development and two rows of 20 pins each that connect to the on board micro-controller. The EzSCB1 is a modern lead free design, 100% surface mount and assembled on automatic equipment for consistent quality. All parts used on the EzSBC1 is rated for the full industrial temperature range of -40 to 85 degrees Celsius. Unless the EzSBC1 is soldered to another PCB it is recommended that the environment not be allowed to go below the freezing point of water as this may damage the socket leading to poor contact. The dimensions of the EzSBC1 printed circuit board is 2.38 x 0.74 inch or 18.8 x 60.5 mm.



Hardware

The USB interface is an FTDIchip FT232RL USB 2.0 Slave to UART converter providing a rock-solid USB interface with bullet proof drivers for all modern operating systems. The EzSBC1 connects to the FT232RL through a hardware UART that defaults to 57600 bits per second. The bit rate can be set to any standard speed.

The EzSBC1 has a 5V to 3.3V regulator and a reset generator on the module. The 3.3V output of the voltage regulator is available on pin 40 as Vref. All the I/O pins are 3.3V pins that are 5V tolerant. They will not be damaged by the application of signals up to 5.5V. When pins are configured as digital inputs they will operate correctly when driven by TTL or CMOS signals with supply voltages up to 5.5V. Analog inputs will not be damaged by 5V signals but the presence of 5V signals on analog pins my cause incorrect reading on nearby analog inputs.

With a few exceptions all pins can sink and source 4mA continuously without damage. The DAC output should be buffered to preserve accuracy but can drive a load resistance as low as 1kΩ. The I^2C pins can sink 4mA each.

There are two timing elements, a 32768Hz crystal and a 12 MHz oscillator on the board to provide timing references for the Real Time Clock and micro-controller. There is a circuit to allow an external coin cell to power the RTC when the main power is not present. The RTC keeps track of the date and time and is leap year aware.

In the diagram above the pins with from pin 4 to pin 37 are I/O pins under control of the micro-controller. Where pins have multiple functions the alternate functions are identified by colored boxes.

The pins with green ADC boxes can be used as inputs to the 10-bit analog to digital converter of the EzSBC1. Pin 24 can be driven by the internal 10-bit digital to analog converter.

The pins with PWM next to them can generate pulse width modulation signals as with 32-bit resolution to generate square waves with very precise frequencies and duty cycles. The TX1 and RX1 pins can be controlled by an integrated hardware UART for serial communication with other devices.

Pin 4 and Pin 5 can be controlled by a hardware I^2C interface and are Open Drain pins. The pins require external pull up resistors and may be pulled to 3.3V or 5V depending on the requirement of the I^2C bus. The EzSBC1 has 4.7k Ohm pull up resistors to 3.3V on pins 11 and 14 which are also Open Drain pins. Take care not to pull pin 14 low during power up as it may cause the EzSBC1 to malfunction if the pin is not high during reset events.

Companion Boards

We offer boards specifically designed for the EzSBC1 that contain additional components for functions such as motor drivers, temperature sensors, relay drivers and breadboard areas. Please visit EzSBC.com for the currently available companion boards. If you have specific requirements please contact us as we may develop a board to suit your needs.

Architecture

The EzSBC1 is based on a 32-bit ARM7 core running at 60MHz. The actual controller used is a device by NXP, the LPC2136 and it has 256k of Flash memory and 32k of RAM. The internal details of the LPC2136 are invisible to the user of the EzSBC1 and most of the functions of the LPC2136 is accessible through BASIC commands.

The EzSBC1 has many features built that on competing products require external ICs to implement. Many competing devices do not have ADC's, DAC's or background PWM generators. Few other devices of this class offer 33 I/O pins as standard. Most competing devices require external USB adapters to connect to laptop or desktop computers since modern computers hardly ever have serial ports.

Many features are unmatched by other devices of this class such as an integrated Real Time Clock and full 64-bit floating point arithmetic. Another feature that is unmatched is the ability to protect the program stored on the EzSBC1 so that you can sell your design knowing that your program cannot be stolen by unscrupulous companies and individuals. It is possible to build USB powered devices with the EzSBC1.

Memory Organization

The EzSBC1 has two types of memory; Flash memory and static Random Access Memory, commonly referred to as RAM or SRAM. RAM loses its contents every time the power is removed. Flash memory is persistent or 'non-volatile' and maintains its contents even without power.

On embedded controllers programs are usually stored in Flash Memory and variables in SRAM. The EzSBC1 is no different, your program is stored in Flash and the variables used for calculations are stored in RAM. Usually you do not need to know anything more about the memory organization of the EzSBC1 since everything is automatically placed in the correct memory location.

The only additional detail that you need to be aware of is that the Flash memory for program storage is divided into two equal sized blocks called Bank 0 and Bank 1. You can choose to store your program in either bank; they are equivalent in every way. You can store two different programs in the two banks and run either one on start up. The EzSBC1 has a small amount of Flash set aside (4k) to store configuration parameters to control or influence the behavior of the program or configure items that change often.

Each Flash bank is 64k bytes in size allowing programs with thousands of lines of code to be stored and executed. The exact size of the program that can be stored depends on many parameters such as the length of the variable names and the mix of instructions and commands used. Each command consumes only one byte but parameters and variables can increase the amount of storage required per line of code.

The CPU has 32k bytes of static RAM and about 2k is used by the BASIC interpreter and buffers for the I/O devices that need buffers. Variables use four to eight bytes per variable with string variables requiring four bytes more than the number of characters in the string.

Control BASIC Reference

Programming is the art and science of achieving a desired result by decomposing a problem into a series of steps that can be executed by a computer. Computers and embedded controllers have very limited actions that they can perform. The entire list boils down to Repetition, Conditional Execution, Input, Output and some form of Arithmetic. Any programming language that provides facilities for performing these five basic operations can be used to execute any program that can be conceived. This does not imply that all programming languages are equally suitable to solving all problems, just that a program that can be written in one language can be written in any other computer language.

The design of the EzSBC1 and Control BASIC involved compromises and tradeoffs to make an Embedded Controller that is easy to use; fast enough for most applications and that does not need a 500 page manual. Most (all?) of the complexity of the underlying ARM central processing unit and its peripherals have been hidden from the user. The user of the EzSBC1 does not need to know how to initialize the ARM CPU, configure the ADC or DAC or deal with the complexities of the I^{2}C controller.

To allow you to control your world the EzSBC1 and Control BASIC provides peripherals to perform Input and Output of Analog and Digital signals and instructions and commands to perform Arithmetic, Looping and Conditional Execution of program sections.

Input

To perform useful control the embedded controller needs to observe its operating environment. The EzSBC1 with Control BASIC has up to 32 digital input pins for observing the state of switches and other digital signals. Up to 16 of the pins can be used as Analog input pins and analog values can be measured with 10-bit resolution from 0V to 3.3V. There are BASIC commands to configure the pins and read the state of the digital pins or the analog values.

The relevant BASIC words for digital input are IN, INPIN, IND, PINMODE, COUNT and PULSIN.

More complex functions such as Serial, I^{2}C and SPI IO are performed by dedicated instructions that hide most of the complexity of the protocols from the user.

The EzSBC1 has two hardware supported serial ports. One is permanently connected to a Serial to USB interface IC to make connection to modern computers easier. Programs can be downloaded via the serial port and user input and output to a VT100 terminal emulator program is supported. Input from the hardware serial ports can be accessed with the following commands: INPUT, INSTR, INKEY and SERINP$. The serial ports can be controlled and configured with the SERINIT command.

The EzSBC1 has one hardware supported I^2C bus. The I^2C bus does not have pull-up resistors on the module and must be externally pulled high to either 3.3V or 5V with suitable resistors. This allows the use of 5V or 3.3V I^2C devices. The I^2C control commands are I2CINIT, I2CWR, I2CRD$, I2CRDS$, I2CBUSY, I2CERR, I2CER$ and I2CTIME.

Serial Peripheral IO (SPI) can be used on any group of digital pins and the SPI function is implemented in software. See the SERSPI command.

Analog input is performed with any one of the 16 possible ADC input pins. To configure and read the pins use the instructions PINMODE and INADC. The analog inputs are designed to operate from 0V to 3.3V but will not be damaged if a 5V signal is applied to the pins. If a 5V signal is present on one of the analog input pins then measurements on the other analog pins may be inaccurate.

Output

The EzSBC1 has up to 32 digital outputs and one analog output. The analog output is a 10-bit Digital to Analog Converter (DAC), available on pin 24 of the EzSBC1. The most of the digital outputs have push-pull drivers implying that they can actively source and sink current. The exceptions are pins 4, 5, 11 and 14 which are open drain pins. Open Drain pins can sink current but are pulled high by a resistor external to the micro-controller. Pins 4 and 5 don't have pull-up resistors on the EzSBC1 and need external pull-up resistors. Pin 4 is the clock pin (SCL) of the I^2C controller and pin 5 is the data pin (SDA). The user must select appropriate values for these resistors and tie them to 3.3V or 5V depending on the supply voltage of the external devices on the I^2C bus. A good starting value is 2.2kΩ. The I^2C control commands are I2CINIT, I2CWR, I2CRD$, I2CRDS$, I2CBUSY, I2CERR, I2CER$ and I2CTIME.

Pins 11 and 14 have 4.7kΩ resistors to 3.3V on the EzSBC1 PCB. Pin 14 is a special pin on the LPC2136 micro-controllers and must NOT be low during power-up for normal operation of the EzSBC1. If pin 14 is sensed low during power-up then the micro-controller enters a special mode to allow code to be loaded into the internal Flash memory without the use of special programming circuitry.

The relevant BASIC words for digital output are OUT, OUTPIN, OUTD, PINMODE, PWM and PULSOUT. Digital output pins can sink and source 4mA each while keeping the voltages with 0.4V from each supply voltage. The rise and fall times of the digital transitions are typically 10ns.

Serial Peripheral IO (SPI) can be used on any group of digital pins and the SPI function is implemented in software. See the SERSPI command.

Analog output is performed with the PINMODE, DAC and OUTDAC BASIC word. The DAC is a voltage mode DAC and can drive pin 24 to 0 and 3.3V in 1024 steps and can drive a resistive load of 1kΩ or more without loss of accuracy.

Strings can be printed and formatted on an external VT100 type terminal with the BOLD, CLREOL, CLRSCR, LOCATE, PRINT and TAB commands.

Repetition

Control BASIC has three means of repeating a block of instructions

- REPEAT/UNTIL
- WHILE/WEND
- FOR/NEXT

These control structures may be nested inside each other or themselves to perform complex iterations over code. The number of program lines in each block is limited only by the available memory for program storage.

REPEAT UNTIL

The REPEAT UNTIL block repeats the code between the REPEAT keyword and the UNTIL line as long as the Boolean condition after the UNTIL keyword remains false.

```
sw1=11          ‘IO pin to connect to Switch 1, Switch pulls pin high.
PINMODE sw1, IN 'Configure as digital input

REPEAT
UNTIL IND(sw1)=1

PRINT "Switch 1 pressed."

END
```

The short program above shows a REPEAT/UNTIL loop that does all the work in the conditional statement after the UNTIL keyword. The first line defines sw1 as (pin) 11 and the second line configures the pin as an input. The instruction IND(sw1) reads the state of the pin and returns either 1 or 0 depending on whether the pin is found high or low. If a pull down resistor holds the pin low if the switch is not pressed then the interpreter will execute the loop as fast as it can as long as the switch is not pressed. As soon as the switch is pressed the program will print the message on the terminal and end.

To make an infinite loop a condition that can never be true is used; a favorite is UNTIL 1=2.

An important feature of a REPEAT/UNTIL loop is that the body of the loop (the instructions between REPEAT and UNTIL) will always be executed at least once. Another feature of the REPEAT UNTIL loop is that the starting point of the loop is known when the loop reaches the conditional test at the end of the loop. The interpreter stores the position of the start of the loop when the REPEAT keyword is encountered and if the condition is false the code resumes at the start of the loop without having to search for the start of the loop. This is not true for the WHILE WEND loop. Large loops with lots of code in the body of the loop are best implemented with REPEAT UNTIL loops.

WHILE WEND

The WHILE loop will execute the instructions between the WHILE and the WEND keywords as long as the condition following the WHILE keyword remains true. It is possible to write a loop where the body of the loop may never be executed. If the condition after the WHILE keyword is false the interpreter searches forward for the next WEND statement and executes the code following the WEND statement.

FOR NEXT

The FOR loop is a very convenient way of executing some action a defined number of times. The FOR loop starts at the FOR instruction and continues to the matching NEXT instruction.

There are four LEDs on the EzSBC1 that are connected to four pseudo pins numbered 41 through 44. The pins are referred to as ‘pseudo’ pins because, unlike the other IO pins, they are not present as physical pins on the EzSBC1 module. To use the LEDs the pins first have to be configured as output pins and driven to a known state. The instruction `PINMODE 41, OUT` can be repeated four times, once of each pin 41 through 44. The instruction OUTD 41, 1 can then be used for each pin to turn the corresponding LED off. The code fragment below illustrates a better way by using of a FOR loop to initialize the pins that drive the onboard LEDs.

```
FOR i=41 TO 44
    PINMODE i, OUT
    OUTD i, 1
NEXT i
END
```

Since we know exactly how many pins need to be initialized we can write line 1 immediately. Line2 configures the pin as an output. Line 3 drives the pin high and the next line tells the interpreter that it is time to repeat the loop with the next value of the loop counter.

The code listed above will not produce any visible output since the LEDs are turned off. By changing line 3 to `OUTD I, 0` the LEDs will turn on one after another. This can easily be seen by single stepping through the code.

The FOR loop can take a modifier which alters the step size that the loop counter takes as the loop is executed.

```
FOR i=41 TO 44
    PINMODE i, OUT
    OUTD i, 0
NEXT i

FOR i=44 to 41 STEP -1
    OUTD I, 1
NEXT i

END
```

In the program above the first loop initializes the pins and turns the LEDs on. The second loop then turns the LEDs off in the reverse order in which they were turned on.

This program implements a famous algorithm for finding prime numbers and illustrates the use of the FOR/NEXT loop.

```
TRUE=1
FALSE=0
SIZE=1000

DIM flags(SIZE+1)

PRINT "Sieve of Eratosthenes - EzSBC1"

'Find all he prime numbers below 1000 by using the Sieve of
'Eratosthenes.

time = GETTICK
pCOUNT = 0                                         'initialize prime
counter
FOR i = 2 TO SIZE                                  'set all flags true
    flags(i) = TRUE
NEXT i
'SSTEP
prime=2
FOR i = 2 TO SIZE
    IF flags(i)=true THEN                          'found a prime
       prime = i
       FOR k = i + prime TO SIZE STEP prime
             flags(k) = FALSE                      'kill all multiples
       NEXT k
       pCOUNT=pCOUNT+1                             'primes found
    ENDIF
NEXT i
time = GETTICK-time
PRINT pCOUNT;
PRINT " primes."
PRINT "Runtime = ";
PRINT time/10;
PRINT" ms"
END
```

The program prints:

```
Sieve of Eratosthenes - EzSBC1
168 primes.
Runtime = 924.7 ms
```

Conditional Execution

In Control BASIC the most important instruction for conditional execution is the IF/THEN/ELSE/ENDIF statement. The IF statement is block structured and can contain any number of instructions between the THEN, ELSE and ENDIF statements. The IF keyword must be followed by an expression that can be evaluated by only referring to variables that already have values. Valid expressions contain one of the five relational operators =, <, >, <= and >= and variables or constants. Tests can be combined by the AND and OR keywords.

```
IF x>1 AND x<20 THEN
```

A special feature of the IF statement is that the rest of the line after the THEN keyword cannot contain any other instructions. The instructions to execute when the condition is true must be on the following lines.

There are additional forms of the IF statement to allow for cases where the full IF/THEN/ELSE/ENDIF is overkill. See the IIF and II$ instructions.

Arithmetic

Because of the large amount of RAM present on the EzSBC1 the BASIC interpreter does not use bit or byte variables. All numeric variables offer at least 32-bit precision including the sign bit. Numbers as large as +-2,147,483,648 (2 billion) will not cause overflows. You do not need to learn how to use obscure manipulations to extend the range of variables so that you can count to a million. If you perform an operation that exceeds the range that can be represented in 32 bits the interpreter will automatically convert the numbers to 64-bit floating point numbers and perform the calculations correctly. One divided by two will not return an answer of zero but of 0.5, as you would expect.

Defining and Using Variables

Most variables are automatically created when they are first assigned a value. The only variables that need to be declared are arrays. If you want to use two variables named x and y to store two values you can just make the assignment.

```
x=1
y=2
```

To help guard against hard to find bugs the use of a variable that has not been assigned a value on the right hand side of an = sign, causes an error message to appear. For example

```
y=2*z
```

will cause an error since z has not been assigned a value. These error messages can be suppressed with the ONERROR command in which case z will be assumed to be 0. In general, variables should be assigned values before they are used.

Defining Array Variables

Arrays or dimensioned variables must be declared but may be declared and initialized in one instruction.

```
DIM A(10)=20,18,16,14,12,10,8,6,4,2
DIM NAME$(7)="Joe","Frank","John","Bill","Bert","George","Simon"
x=0
y=1
z=2
DIM position(3)=x,y,z
```

are valid ways of declaring and initializing dimensioned variables. See the **DIM** command for complete details.

The Rules for Variable Names

Variable names can start with an alphabetic character or underscore and can contain any alphabetic or numeric character and the underscore (_). They may be up to 32 characters long. Variable names may not start with a number. Variable names are not case sensitive, so that "joe", "Joe" and JOE all refer to the same variable.

There are two types of variable: numeric and string variables. Numeric variables store integers and floating point numbers. String variables store a string of characters such as “Joe”. String variable names must end the $ symbol (e.g., name$) while numeric variables must not end with $.

Variable names may not contain spaces and cannot be the same as a command name or keyword.

Constants

Numerical constants may begin with a numeric digit (0-9) for a decimal constant or 0x for a hexadecimal constant. For example 0x8 is the same as the decimal constant 8. Decimal constants may be preceded with a minus (-) or plus (+) and may terminated with 'E' followed by an exponent number to denote exponential notation. For example 1.0E+6 is the same as 1000000. The command PI returns the value of PI as represented internally.

PI = 3.1415926535897932384626433

String constants must be enclosed by double quote (“) marks for example “Hello World.”.

Order of Operations

The order of operations is set by the precedence of the operators in an expression. The entire line is read before evaluation starts to ensure that the orders of operations are correct. The order may be forced to any desired order by placing brackets around sections of the expression.

Integer Arithmetic Rules

If you wish to keep the value of an expression as an integer value then you must take care not to use commands that return fractional values (such as Sin, Cos, Sqrt) or perform division in the expression. It is much easier to simply force the value of an expression to be an integer by by placing the entire expression inside the INT( ) command. The INT command truncates the fractional value of an expression or number.

If you wish to round a (floating point) number X to the closest integer then the expression

```
X=INT(X+0.5)
```

will do the rounding correctly. There is a small speed benefit to keeping expressions to integer values.

Floating Point Arithmetic

Floating point arithmetic has the benefit that huge ranges of numbers can be represented with a relatively small number of bits (64) but floating point arithmetic can lead to some unpleasant surprises.

It is dangerous to compare two floating point number for equality. It is much safer to compare floating point numbers for relative size such as x > y or x >= y. These comparisons never fail in unexpected ways but tests for equality may fail unexpectedly.

Floating point numbers are subject to approximation errors that depend on the exact value of the number and even the history of how the result was reached. We tend to think in terms of decimal values but the calculations are performed in binary. This leads to numbers such as 2 and 4 being represented 'more' accurately than numbers such as 0.1. Some numbers that can be represented exactly in decimal notation such as 1/5 = 2E-1 cannot be represented exactly as a binary floating point number (or a binary rational number). The consequence is that 5*(1/5) may or may not be equal to 1.0 depending on rounding. Even worse, 5*(1/5) may not equal (1/5)*5.

If you perform these calculations on the EzSBC1 both will print 1 (or 1.0) but that is due to details of how the calculations are preformed and stored <u>BUT IT MAY NOT ALWAYS BE TRUE</u>. Care has been taken in how the floating point calculations and comparisons are performed but floating point numbers are not exact so please be careful.

Where possible you should use integers as loop counters instead of floating point numbers. Instead of writing

```
FOR i=0 to 1.0 STEP 0.1
  'Some calculation involving i
NEXT i
```

```
write
FOR i=0 to 10 STEP 1
  'Some calculation involving i/10
NEXT i
```

Then the loop will never surprise you by being an infinite loop instead of executing precisely 11 times before ending.

The prior examples are well behaved on the EzSBC1 but see what happens when you perform

```
PRINT 0.6/0.2
PRINT 0.6/0.2 - 3
END
```

Unary Operators

Control_BASIC has two unary operators - and ~. - negates a value and works on all arithmetic types, integer and floating point.

Inverse (~)

~ performs a bit-wise inversion of an integer, replacing 0 with 1 and 1 with 0 in the binary representation of the number or variable following the ~. The ~ operator expects an integer operand and will convert floating point numbers to integers prior to performing the bit inversion.

Negative (-)

- negates a value and works on all arithmetic types, integer and floating point.

Absolute Value (ABS)

ABS takes any number and returns the positive value of the number. Algorithmically it can be described as:

ABS(x) = x if x >= 0

ABS(x) = -x if x < 0

Cosine (COS), Sine (SIN), Tangent (TAN)

COS(x) returns the cosine of x as a floating point number where x is interpreted as an angle in radians.

SIN(x) returns the sine of x as a floating point number where x is interpreted as an angle in radians.

TAN(x) returns the tangent of x as a floating point number where x is interpreted as an angle in radians.

The inverse functions of **SIN**(x), **COS**(x) and **TAN**(x) are available as **ASIN**(y), **ACOS**(y) and **ATAN**(y) respectively. These functions return an angle (in radians) in the first two quadrants of the unit circle. A special function **ATAN2** is included to calculate directions while preserving the sign of the direction.

Square Root (SQRT)

SQRT returns the square root of the argument. If the argument is negative an error will be reported.

Since most square roots are irrational numbers $x \neq [SQRT(x)]^2$ once any rounding takes place during the square root operation or the multiplication.

BINARY OPERATORS

Add (+)

The Add operator adds the value to the left and right of the '+' together and returns the result to be assigned to a variable or tested for equality. If the values added together are sufficiently large so that the result will exceed the range of a 32-bit integer, the result will be a floating point number. The values to the left and right of the '+' may be integers or floating point numbers.

Subtract (-)

The subtract operator subtracts the value to the right of the '-' from the value on the left of the'-' and returns the result to be assigned to a variable or tested for equality. If the values subtracted are sufficiently large so that the result will exceed the range of a 32-bit integer, the result will be a floating point number. The values to the left and right of the '-' may be integers or floating point numbers.

Multiply (*)

The multiply operator multiplies the value to the left and right of the '*' together and returns the result to be assigned to a variable or tested for equality. If the values multiplied together are sufficiently large or small so that the result will exceed the range of a 32-bit integer, the result will be a floating point number. The values to the left and right of the '*' may be integers or floating point numbers.

Divide (/)

The divide operator divides the value to the left of the '/' by the value to on the right of the '/' and returns the result to be assigned to a variable or tested for equality. If the values subtracted are sufficiently large or small so that the result will exceed the range of a 32-bit integer, the result will be a floating point number. The values to the left and right of the '/' may be integers or floating point numbers.

Modulus (MOD)

The modulus operator divides the value to the left of the 'MOD' by the value to on the right of the 'MOD' and returns the remainder of the division operation to be assigned to a variable or tested for equality. The values to the left and right of the 'MOD' may be integers or floating point numbers. If one or both of the numbers are floating point numbers then the FMOD function is performed instead of MOD.

Minimum (MIN)

The MIN function returns the smaller of its two arguments to be assigned to a variable or tested for equality.

Maximum (MAX)

The MAX function returns the larger of its two arguments to be assigned to a variable or tested for equality.

Shift Left (LSL)

The LSL function shifts the first argument to the left by the number of bits specified by the second argument. The bit positions on the right of the integer being shifted are filled in with zeros. The bits shifted out the left are discarded.

Shift Right (LSR)

The LSR function shifts the first argument to the right by the number of bits specified by the second argument. The bit positions on the left of the integer being shifted are filled in with zeros. The bits shifted out to the right are discarded.

And (&)

The bitwise AND function '&' performs a bit by bit AND of the bits of the two integers to the left and right of the '&'.

Or (|)

The bitwise OR function '|' performs a bit by bit OR of the two integers to the left and right of the '|'.

Xor (^)

The bitwise Exclusive OR function '^|' performs a bit by bit Exclusive OR of the two integers to the left and right of the '|'.

STRINGS

Strings are special variables do deal with text in the form of ASCII characters. Strings are normally used for interaction with humans but may also be used as control commands for other machines. Control-BASIC has powerful string processing commands and keywords. String variable names always end with the $ sign (e.g., myString$) while numeric variables must not end with $. String constants must be enclosed by double quote (") marks for example "Hello World.". Strings may be printed to the terminal in the obvious way. String operations are easy to show as examples.

```
A$ = "Hello "
B$ = "world."
PRINT A$
PRINT B$
'Print on one line
PRINT A$, B$
'Concatenate strings
myStr$ = A$ + B$
PRINT myStr$
```

This program produces the following output:

```
Start program
Hello
world.
Hello world.
Hello world.
```

```
Program Ended.
```

The LEFT$ and RIGHT$ functions can be used to get only the beginning or end of a string. They are often used in combination with the LEN function to reduce the size of a string. The following example should make it clear.

```
A$="abcdefghijklmnopqrstuvwxyz"
LenA = LEN( A$)
PRINT LenA
'Remove right most character
ShortA$ = LEFT$( A$, LenA-1)
PRINT ShortA$
EvenShorterA$= RIGHT$( ShortA$, LenA-2)
PRINT EvenShorterA$
END
```

prints

```
Start program
26
abcdefghijklmnopqrstuvwxy
bcdefghijklmnopqrstuvwxy

Program Ended.
```

The TRIM$ function removes spaces and other non-printing characters from the end of a string. UPPER$ can convert strings to all uppercase or all lowercase letters.

The MID$ string function can be used to break strings into smaller strings or to insert one string into another at any position.

The TIME$ string function returns the current date and time as a string by reading the value of the built in Real Time Clock.

The BYTE$, HEX$ and BIN$ functions provide conversion from numbers to strings; VAL, VALLEN, BSTR and BYTESTR functions convert from strings to numbers.

Strings can contain any value from 0 to 255 in any position making them useful as buffers for binary data.

Control-BASIC Reference

The keywords are listed alphabetically.

ABORT

ABORT <label:>

Jumps to line denoted by <label:>. Unwinds all GOSUB, REPEAT/UNTIL, and FOR/NEXT loops. The ABORT command is intended for graceful error recovery to ensure the program will continue to run.

```
Start:
PINMODE 6, ADCPIN
ONERROR EH1:
time= INADC(6)
PRINT "Speed = ", 120/time
END
EH1:       'Error Handler
ABORT Start:
END
```

In the program listed above the program will jump to the Start label if the value measured on pin 6 by the analog to digital converter is zero.

ABS

x=**ABS**(<number>)
ABS takes any number and returns the positive value of the number. Algorithmically it can be described as:

ABS(x) = x if x >= 0

ABS(x) = -x if x < 0

```
x = SQRT( ABS( Number) )
```

The code above prevents a runtime error if Number is ever negative.

ACOS

Theta=**ACOS**(x)

The **ACOS** function returns the ArcCosine of x and is the inverse of the **COS** function. Theta is in radians.

AND

The AND keyword is a logical operator for use with the IF, WHILE and UNTIL keyword to build more complex conditions under which statements are executed. Also see the OR, IF, WHILE or UNTIL keywords.

ATAN

Theta=**ATAN**(x)

Calculates the ArcTangent of x and returns an angle in radians. **ATAN** is the inverse of the **TAN** function.

ATAN2

dir=**ATAN2**(Y,X)

The ATAN2 function calculates the direction angle in radians of the numbers (X,Y) on the unit circle. Note the order of the parameters. **ATAN2** is a specialized version of the **ATAN** function that preserves the sign of the angle to make direction calculations easier. See http://en.wikipedia.org/wiki/Atan2 for more information

ASCII

x=**ASCII**(<string>)

Returns an integer 0 to 255 giving the character code value of the first character of <string>. An alternative x=ASCII(<string>,n) gives the code of the nth character or -1 if the string does not an nth character.

```
PRINT ASCII( “ABC”, 1)
PRINT ASCII( “ABC”, 2)
```

Prints

```
65
66
```

ASIN

Theta=**ASIN**(x)

The **ASIN** function returns the ArcSine function of x. The returned angle Theta is in radians. The **ASIN** function is the inverse of the **SIN** function.

BIN$

x$=**BIN$**(x,<digits>)
Compute the string equivalent of the integer x. <digits> is the number of characters to put in x$. Negative <digits> is big-endian, positive <digits> is little endian. This is the inverse function of **BSTR**.

BOLD

BOLD <value>
A non-zero value outputs a VT100/ANSI code to turn on bold type, a zero value turns off bold.

BSTR

x=**BSTR**(x$,<digits>)
Return the integer equivalent of the binary value of x$. <digits> is the number of digits to use in x$. For a positive value of digits the binary is interpreted from right to left. For a negative value of digits the binary string is interpreted from left to right. Example:

```
x$="1101"
PRINT BSTR(x$, 4)
PRINT BSTR(x$, -4)
```

prints

```
13
11
```

Negative <digits> is big-endian, positive <digits> is little endian. This is the inverse function of BIN$.

BYTE$

x$=**BYTE**$(x,<bytes>)
Returns the string equivalent of x. <bytes> is the length of x$ in bytes. For a positive value of bytes x is converted from right to left. For a negative value of bytes x is converted from left to right.

BYTESTR

x= **BYTESTR**(x$, <bytes>)

Returns an integer value when x$ is interpreted as a numerical value. <bytes> is the number of bytes to use in x$. For a positive value of bytes the string is converted from right to left. For a negative value of bytes the string is converted from left to right. BYTESTR is very useful in converting data read in the I2C or SPI bus to numerical values.

```
x=BYTESTR(x$, 2)       "Converts string x$, right to left."
x=BYTESTR(x$, -2)     "Converts string x$, left to right."
```

CHR$

CHR$(<decimal value>)

Returns a string with a single character with character code <decimal value> (from 0 to 255).

```
PRINT CHR$(65)
PRINT CHR$(90)
END
```

prints

```
A
Z
```

CLREOL

CLREOL

Outputs a VT100/ANSI code to clear the current line from the cursor position to the end of the line.

CLRSCR

CLRSCR

Outputs a VT100/ANSI code to clear the screen and place the cursor at the upper left corner.

CONF$

x$=**CONF$**(<confstring>)

CONF$ reads the configuration variable out of the Configuration Flash and places it into x$. For example if there was a line in the Configuration Flash such as NAME=John Do then Name$=**CONF$**("NAME") places "John Do" into Name$. The configuration space is in Flash memory and is 4k byte in size. The space is organized as an array of strings in the form Identifier="Content String" CR LF where the CR LF characters mark the end of a string. The Configuration Flash is not intended as a general purpose EEPROM replacement and cannot be written to from within the BASIC program.

COS

X=**COS**(Theta)

The **COS** function returns the Cosine value of the angle Theta where Theta is expressed in radians. See **ACOS** for the inverse function.

COUNT

x=**COUNT**(< pin, duration>)

The COUNT instruction counts the number of transitions on input pin in the number of 100µs intervals specified by duration. For example x=COUNT(21, 10000) sets x to the number of edges on pin 21 in the next second. The pin must be configured as a digital input with **PINMODE** before COUNT is called.

DAC

DAC is used with **PINMODE** to configure a pin as the DAC output. On the EzSBC1 only pin 24 may be configured as a DAC pin. See **PINMODE** and **OUTDAC**.

```
PINMODE 24, DAC
```

DELAY

DELAY <100microseconds>
The **DELAY** instruction waits an integer number of 100microsecond ticks. See **WAIT**

DHTHUM

Humidity=**DHTHUM**(<PinNo>, <Type>)

The DHT11 and DHT22 temperature and humidity sensors are popular humidity sensors that have an awkward serial protocol. This instruction returns the Humidity reading from the sensor connected to the pin <PinNo>. The <Type> parameter must be 11 for a DHT11 and 22 for a DHT22 sensor. The AM2302 sensor is also accessed using 22 as the type parameter. The DHT11 and DHT22 sensors require that the pin specified in PinNo have a pull up resistor installed. The DHTHUM instruction will switch the pin from input to output as required by the protocol for the sensor. For the DHT11 sensor the humidity is always a whole number. For the DHT22 (and AM2302) the humidity includes a decimal part. The DHT type sensors may only be read once per second. The DHTHUM and DHTTEMP instructions will always return the latest available reading. If you read the temperature immediately followed by the humidity (or in the reverse order) the sensor will only be read once. Once a second has elapsed from the last read of the sensor a new operation will take place to get the latest values for temperature and humidity.

DHTTEMP

Temperature=**DHTTEMP**(<PinNo>, <Type>)

This instruction returns the Temperature reading from the sensor connected to the pin <PinNo>. The <Type> parameter must be 11 for a DHT11 and 22 for a DHT22 sensor. The AM2302 sensor is also accessed using 22 as the type parameter. The DHT11 and DHT22 sensors require that the pin specified in PinNo have a pull up resistor installed. The DHTTEMP instruction will switch the pin from input to output as required by the protocol for the sensor. For the DHT11 sensor the temperature is always a whole number in Celsius degrees. For the DHT22 (and AM2302) the temperature includes a decimal part down to a tenth of a Celsius degree. The DHT type sensors may only be read once per second. The DHTHUM and DHTTEMP instructions will always return the latest available reading. If you read the temperature immediately followed by the humidity (or in the reverse order) the sensor will only be read once. Once a second has elapsed from the last read of the sensor a new operation will take place to get the latest values for temperature and humidity.

DIM

DIM A(<index 1>)
DIM A(<index 1>,<index 2>)
DIM A$(<index 1>)
DIM A$(<index 1>,<index 2>)

These commands create arrays of strings or numbers, one, two or three dimensional. These can be initialized to particular values by using the =, e.g.

```
DIM A(10)=20,18,16,14,12,10,8,6,4,2
DIM
NAME$(7)="Joe","Frank","John","Bill","Bert","George","Simon"
x=0
y=1
z=2
DIM position(3)=x,y,z
```

declares an array that hold three numbers and initializes it to 0,1,2.

END

END terminates the program. An **END** instruction may be placed before the subroutine definitions and labels used for the ABORT command.

ENDIF

ENDIF
Terminates the **IF** statement.

ERC$

x$=**ERC$**(<error code number>)

Return a string describing the error associated with error code number. If the number is negative, then the function returns the string representing the last error.

ERR

x=**ERR**(0)
The function retrieves the error code of the last error for **ONERROR**. Reading this clears the error code so subsequent accesses return 0 until another error occurs. x=**ERR**(1) This retrieves the line number of the last error for **ONERROR**. Reading this clears the line number of the error so subsequent access return 0 until another error is thrown. In the editor, the line number of a particular statement can be jumped to by using the CTRL-G command.

FOR

FOR <var>=<begin> **TO** <end> **STEP** <step>

Iterates <var> starting at <begin> until it reaches <end>, each time through the loop incrementing the step size the value <step>. <step> may be negative if <end> is smaller than <begin>. The end of the loop is **NEXT** <var>.

e.g.

```
FOR I=99 TO 1 STEP -1
    PRINT I," days till Christmas."
NEXT I
```

GETTICK

GETTICK
Returns a number that increments every 100us. It is very useful for doing timeouts.

```
start = GETTICK
REPEAT
   ‘ Execute some code for a second
UNTIL GETTICK > ( start + 10000 )
```

Can also be used to determine how long a piece of code takes to execute. The number will eventually overflow, after a few days. See **SETTICK**

GOSUB

GOSUB <label:>

Calls a subroutine at the line labeled with <label:>. Control returns to the statement after the **GOSUB** command when a **RETURN** is encountered. Note that the destination label must end in a colon.

GOTO

GOTO <label:>
Jumps program execution to the line labeled by <label:>. Note that the destination label must end in a colon. Use sparingly.

HEX$

x$=**HEX$**(<decimal number>)

Returns a string representation of the unsigned hexadecimal equivalent of <decimal number>. Base 10 decimal numbers can be returned with **STR$**.

```
x$=HEX$(26)
PRINT x$
END
```

Prints

```
1A
```

HI

HI <pin_no1>, <pin_no2>, ...

HI sets the digital output pins in the list of pins to a HIGH state. See LO, OUTD and PINMODE.

I2CBUSY

x=**I2CBUSY** (< slave address>)

I^{2}C Routine, Return Slave Status

Some I^{2}C devices such as EEPROM's need a variable time to complete commands. This instruction provides an easy way to poll the status of such a device.

I2CER$

I2CER$(<error code>)

I^{2}C Routine, Return error or status string

This instruction is used in conjunction with the **I2CERR** function to print error messages when the I^{2}C bus transactions fail. The I2CERR instruction is used to retrieve the error code from the I^{2}C subsystem. The code that is used as a parameter to the I2CER$ instruction to get an ASCII string that can be printed directly to indicate the source of trouble on the I^{2}C bus.

I2CERR

x=**I2CERR**
I^{2}C Routine, Return error or status code.

The I2CERR function returns the last status or error code reported by the I^{2}C controller function. The same numerical values are used as defined in the LPC2136 User Manual, Chapter 13, Table 167 and Table 168. This is for advanced debugging of bus problems. See the **I2CER$** instruction for more readable error and status reporting. I2CERR returns 0 if there was no error.

I2CINIT

I2CINIT(<clock speed>)

I^{2}C Routine, Initialize the I^{2}C subsystem before use

Sets Pin 4 and Pin 5 as Open Drain pins and sets the frequency of SCLK in Hz. I2CINIT(100000) initializes the controller for 100kHz devices. The Pins are 5V tolerant and may be pulled up to 5V to directly drive 5V IC's.

I2CRD$

I2CRD$ (<slave address>, < register to read>, <number of bytes>)

I^{2}C Routine, Random Read

Instruction to read data from register in a slave device.

If the number of bytes to read is greater than 1 the read is from consecutive addresses (registers) in the slave device. The maximum number of bytes that can be read is 64 (limited by the internal buffer of the EZSBC1) or the number of bytes supported by the slave device.

I2CRD16$

I2CRD16$ (<slave address>, < register to read>, <number of bytes>)

I^{2}C Routine, Random Read

Instruction to read data from register in a slave device.

If the number of bytes to read is greater than 1 the read is from consecutive addresses (registers) in the slave device. The maximum number of bytes that can be read is 64 (limited by the internal buffer of the EZSBC1) or the number of bytes supported by the slave device.

I2CRDS$

I2CRDS$ (<slave address>, <number of bytes>)

I^{2}C Routine, Sequential Read

Instruction to read data from the I^{2}C slave device starting from the byte following the previous address or register read by the I2CRD$ instruction.

I2CTIME

I2CTIME(< time to wait >)

I^{2}C Routine, Set Bus Timeout

When the I^{2}C bus is accessed it is possible for the transaction to fail due to noise or protocol failure. <time to wait> specifies the time in units of milliseconds that the EZSBC1 allows before abandoning an I^{2}C transaction.

I2CWR

I2CWR(<slave address>, < register to write>, <data string>, <number of bytes>)

I^{2}C Routine, Write to one or more bytes to a slave device

This instruction writes the data in <data string> to a register or consecutive registers or addresses of a slave device. The internal buffer of the EZSBC1 used for the write is 64 bytes.

I2CWR16

I2CWR16(<slave address>, < register to write>, <data string>, <number of bytes>)

I^{2}C Routine, Write to one or more bytes to a slave device with a sixteen bit address register such as an 24LC512 or 24LC256. <register to write> will be interpreted as a 16-bit address and will be sent high byte first followed by the low byte.

This instruction writes the data in <data string> to a register or consecutive registers or addresses of a slave device. The internal buffer of the EZSBC1 used for the write is 64 bytes.

IF

IF <condition> **THEN**
  <program lines>
ENDIF
IF <condition> **THEN**
  <program lines>
ELSE
  <program lines>
ENDIF

If the condition is true, then the program executes the program lines up to **ENDIF**. If **ELSE** is included, then execute the program lines between **ELSE** and **ENDIF** if the condition is false. **IF** statements may be nested.

<condition> may contain the following test for equality: **=, <, >, <=, >=** and they may be combined with **AND** and **OR** to form complex conditionals.

IIF

x=**IIF**(<condition>,<truevalue>,<falsevalue>)
If the condition is true, then return <truevalue> otherwise return <falsevalue>. Both <truevalue> and <falsevalue> may be expressions and both <truevalue> and <falsevalue> are evaluated even though only one result is used. <condition> may include AND and OR to build more complex conditions.

```
x=1
y=2
PRINT IIF (x<y,x,y)
PRINT IIF(x>y,x,y)
```

prints

```
1
2
```

II$

x$=**II$**(<condition>,<truestring>,<falsestring>)
If the condition is true, then return <truestring>, otherwise return <falsestring>. Both <truestring> and <falsestring> may be string expressions and both <truestring> and <falsestring> are evaluated even though only one of the results is used. <condition> may include AND and OR to build more complex conditions.

IN, INPIN

The **IN** or **INPIN** keyword is used to set the direction of an IO-pin. See the **PINMODE**, **IND** and **OUTD** keywords.

INADC

x=**INADC**(<pin #>)

INADC reads the analog value present on a pin that is connected to the internal Analog to Digital Converters. You must use **PINMODE** first to the appropriate pin to enable up the Analog to Digital Converter. (See **PINMODE** examples).

The following pins are valid for use as analog input pins: 6, 7, 8, 10, 12, 13, 15, 21, 22, 24, 25, 26, 27, 28, 29 and 36.

The Analog to Digital Converter has 10-bit resolution and an input range of 0-3.3V.

IND

x=**IND**(<pin #>)
Get the digital input at the pin number denoted by <pin #>
Use the **PINMODE** <pin #>, **IN** command first to set up the pin as a GPIO input.

INKEY

x=**INKEY**
Returns the character code of the currently entered character on the terminal UART0, or -1 if no character is available.

INPLEN

x=**INPLEN**(<length>)
Sets the maximum number of characters to accept for subsequent **INPUT** statements. Returns the old **INPLEN** value. The input length maximum defaults to 80 characters.

INPUT

INPUT <numerical variable>
INPUT <string variable>

Takes input from the terminal UART0 and places a numerical representation in <numerical variable> of the number entered at the terminal, or into <string variable$> the string entered at the terminal. The entered strings have a maximum size set by the INPLEN commands, which defaults to 80 characters.

INSTR

x=**INSTR**(<search string>,<substring>,<index>)

Searches for the first occurrence of string <substring> in the string <search string> that is at or after the character index <index>, with <index>=1 being the first character. If the substring is not found, zero is returned.

INT

INT(<number>)
Returns the integer representation of the floating point number <number>. If <number> is an integer hen the instruction has no effect.

LEFT$

x$=**LEFT$**(<string>,<N>)
Returns the first N characters of string <string>.

LEN

x=**LEN**(<string>)
Returns the number of characters in <string>.

LET

LET <variable>=<expression>
Assigns the variable <variable> to the expression <expression>. If the expression is a string, then the variable must be a string variable. or if the expression is a number then the variable must be a numeric variable. **LET** is optional, e.g.

```
X = 5
```

is a valid statement (the **LET** may be omitted).

LO

LO <pin_no1>, <pin_no2>, ...
LO sets the digital output pins in the list of pins to a LOW state. See **HI**, **OUTD** and **PINMODE**.

LOCATE

LOCATE <y cursor position>,<x cursor position>

LOCATE sends the VT100/ANSI code to the terminal to position the cursor at y position <y cursor position> and x position <x cursor position>.

LSL

The **LSL** function shifts the first argument to the left by the number of bits specified by the second argument. The bit positions on the right of the integer being shifted are filled in with zeros. The bits shifted out the left are discarded.

LSR

The **LSR** function shifts the first argument to the right by the number of bits specified by the second argument. The bit positions on the left of the integer being shifted are filled in with zeros. The bits shifted out to the right are discarded.

MAP

scaled_x= **MAP**(x, in_min, in_max, out_min, out_max)

This function maps x that lies in the interval (in_min, in_max) to a scaled position on the new scale (out_min, out_max)

As an example; you have a sensor with an input range from 0.25V to 3.0V. If it is connected to an ADC channel you will get a range of values from (approximately) 77 to 930 instead of from 0 to 1023. With the **MAP** function you can scale it to 0 to 1023 like this:

```
x=INADC( Ch1 )
scaled_x= MAP( x, 77, 930, 0, 1023 )
```

or this function

```
scaled_x = MAP( x, 77, 930, 0.25, 3.3 )
PRINT scaled_x
```

will print the sensor reading directly in Volts. Note that **MAP** can't improve accuracy, it just performs scaling.

MAX

x= **MAX** (expression1, expression2)

MAX evaluates both expressions and assigns the larger result to x.

MID$

x$=**MID$**(<string>,<index>,<num characters>)

Returns the <num characters> characters starting at index <index> in string <string> as a new string. If <num characters>=-1 or is omitted then the remainder of the string to the end is returned.

MID$(<string>,<index>,<num characters>)=<substring>

Substitutes characters from <substring> into <string> starting at index <index>. If <num characters> is specified, then it substitutes only <num characters> into <string> (if that many are available), or if not specified, substitutes all the characters from <substring>. The string <string> is extended if the <substring> overwrites the end of <string>.

MIN

x=**MIN**(expression1, expression2)
MIN evaluates both expressions and assigns the smaller result to x.

NEXT

NEXT <variable>
NEXT ends a **FOR**/**NEXT** loop. The variable name after NEXT is not optional, omitting the name of the variable will result in a syntax error. See the see **FOR** instruction.

ONERROR

ONERROR <label:>
On an error, jump to the line labeled by <label:>. Note that after an error is trapped, **ONERROR** must be reissued to trap another error (to prevent a potential infinite loop). Omitting the label restores built-in error handling (which terminates program execution). This is useful with **ERR**() to get the error code, and **ABORT** to unwind the **GOSUB**/**REPEAT-UNTIL**/**FOR-NEXT** stacks. Note that the destination label must end in a colon. See ERR.

OR

OR is used in conjunction with comparison operators such as =, >, >=, <, <= to build complex conditional tests for the IF, WHILE and UNTIL commands. See AND

OUT, OUTPIN

OUT, OUTPIN
Used to set the direction of a general purpose pin to Output in preparation for using the OUTD keyword. See the **PINMODE**, **IND** and **OUTD** keywords.

OUTD

OUTD <pin #>,<state>

Output a low for state=0, or a high for state=1 to the pin number denoted by <pin #>. The <pin #> refers to the pin number on the DIP40 module. Pin4 and Pin5 is normally used for I^{2}C and is open drain, requiring external pull-ups for use as digital outputs.

Use the command **PINMODE** <pin #>, **OUT** first to set up the pin as a GPIO output. See PINMODE and OUT.

OUTDAC

OUTDAC <value>

Outputs an analog signal on the DAC proportional to <value>. Since there is only one DAC on the EzSBC1 you must use **PINMODE** 24, DAC to set up the DAC as an output pin first. For an alternative "analog" output signal, see PWM.

The DAC has 10 bit resolution and voltage output. It can only drive a mA or so, if you need more you have to add an external buffer. OUTDAC (0) will place 0V on pin 24; OUTDAC(1023) will output 3.3V - 1 LSB or about 3.296V. The step size of the DAC and hence is 1 LSB or about 3.3/1024 V or roughly 3.22mV per step. The DAC is not perfect and OUTDAC(0) does not necessarily produce exactly 0V nor is the step size identical for each step but the errors are generally less than 1 LSB in size.

```
'Put 1.5V on pin 24
PINMODE 24, DAC
OUTDAC( (1.5*1023)/3.3 )
END
```

See **PINMODE**.

PEEK

x=**PEEK**(<address>)
Reads the memory location <address> and returns the 32-bit value. PEEK and POKE to addresses that don't exist causes a 'bus fault' requiring a reset to recover from the fault state. If you want only a byte you can do bitwise-AND for example:

x=PEEK(<address>) & 0xFF

PINMODE

PINMODE <pin #>,<function>

PINMODE sets up the pin <pin #> to have function denoted by number <function>. The <pin #> is the pin on the 40 pin DIP module. The actual port address corresponding to <pin #> may be found on the EzSBC1 Schematic.

If <pin #> is not available as an IO pin the interpreter will print an error message. The meaning of the function number depends on the pin. Here are the examples of how to use **PINMODE**:

```
PINMODE x, IN                'set up any pin as digital input
PINMODE x, INPIN             'set up any pin as digital input
PINMODE x, OUT               'set up any pin as digital output
PINMODE 21,PWMPIN            'sets pin 21 as a PWM output
PINMODE 7, ADC               'set up pin 7 as ADC input
PINMODE 24,DAC               'set up pin 24 as DAC output
```

See **IN**, **INPIN**, **OUT**, **PWM**, **ADC** and **DAC** for more details.
The phantom 'pin' numbers 41, 42, 43 and 44 give access to the on board LED's. The mapping is as follows:

Pin 41	Red LED
Pin 42	Yellow LED
Pin 43	Green LED
Pin 44	Blue LED

The LEDs turn on when the corresponding ‘pin’ is driven low. To turn the Blue LED on use these two commands:

```
PINMODE 44, OUT
OUTD 44, 0
```

To turn the BLUE LED off use this command:

```
OUTD 44, 1
```

To use the built in 10-bit Analog to Digital converter see the **INADC** function. To use the built in 10-bit Digital to Analog converter see the **OUTDAC** instruction.

POKE

POKE <address>,<value>

Writes the memory location at location <address> with the 32-bit value <value>. PEEK and POKE to addresses that don't exist causes a 'bus fault' requiring a reset to recover from the fault state.

PORT, PORT0, PORT1

PORT(<port number>)

x=PORT(0) returns the value of all the pins of Port 0. Legal values for <port number> is 0 and 1. Bit 0 in PORT(0) corresponds to P0.0 ... Bit 31 in PORT(0) corresponds to P0.31.

Bit 0 in PORT(1) corresponds to P1.0 ... Bit 31 in PORT(1) corresponds to P1.31.

PORT0, PORT1 = <bits to set>

PORT0 and PORT1 gives direct access to the register controlling the digital output pin values.

x=**PORT**(<0 | 1>)

Reads all 32 bits of input port 0 or 1 at once. Pin direction must still be set with PINMODE
PORT0 (<pins>)

PORT1 (<pins>)

Write pins to the register that controls the digital output pins on port 0 or port 1. Pin direction must still be set with PINMODE. There is on = sign in the syntax.

```
X=b101011
PORT0 x    'Set port 0 pint to x
PORT1 y
```

PRINT

PRINT <number>

PRINT <string>

PRINT <v1>,<v2>,...

PRINT <v1>,<v2>,TAB(20),... ;

? "X= ", x

PRINT writes characters to the terminal. The ASCII representation of a number in decimal is output for a number, or the characters of a string for a string. Multiple values can be output to the terminal in the same **PRINT** statement if separated by commas. By placing a semicolon at the end of the line, no linefeed is output at the end of the **PRINT** statement. The "?" character can be used as an alias for **PRINT**. **TAB(x)** produces space characters until the cursor is at column x.

PULSIN

PULSIN(pinno, level, timeout)

The **PULSIN** instruction measures the duration of a pulse on the specified pin. The instruction waits for the pin to go to the opposite of the specified level and starts counting on the next edge and keeps counting till the edge of the opposite polarity appears. If the edges take too long to appear then the instruction will return with a value of zero. Timeout is specified in increments of 100µs i.e.,

pulsewidth=PULSIN(20, 1, 10000)
will measure the positive width of the pulse on pin 20 and return 0 if it did not find a pulse within 1 second. The value returned is in microseconds. The resolution of the measurement is 2µs. The following program generates an accurate square wave using the PWM function on pin 21 and measures the result on pin19 using the PULSIN function.

```
' Create 1kHz pulse on pin 21 with 500us high time
scale= 30000000/1000
timeout=10000
PINMODE 21,PWMPIN
PWM 21, scale, 0.5*scale
'Now measure the pulse width on pin 19
PINMODE 19,IN
REPEAT
    pw=PULSIN(19,1, timeout)
    PRINT "Width=",pw,"us"
UNTIL INKEY > 0
PINMODE 21, IN  'Turn off the PWM signal
END
```

The output of the program is:

```
Width=506us
Width=508us
Width=508us
Width=506us
Width=506us
Width=506us
Width=506us
Width=508us

Program Ended.
```

And from the output it is clear that the PULSIN function returns a value that is greater than the actual pulse width on the pin. The difference is due to a small calibration error and it can be removed by scaling the measurement by 5000/5056 as in the instruction below.

```
pw=INT(5000/5056*PULSIN(19,1, timeout))
```

After the resolution of the measurement is 2µs but the accuracy is ±2 µs or +-0.5%, whichever is bigger.

PULSOUT

PULSOUT pin,duration, level

The **PULSOUT** instruction generates a pulse on the specified pin with a duration specified in microseconds. The instruction will leave the pin in the opposite state of level but will not change it before the pulse is generated. The pin must be configured as a digital output pin before using the PULSOUT instruction. The minimum pulse width is around 5µs, the maximum is 2100 seconds. Increments of 2µs in the duration causes a change in duration of about 2µs, 1µs increments may not produce an increment in pulse width but may affect the jitter in pulsewidth. Pulses produced by this instruction are always slightly longer than the specified length. Very precise signals can be generated by the **PWM** instruction on a limited number of pins.

PWM

PWM <pin #>,<total count>,<fractional count>

Outputs a pulse width modulated output signal on PWM <pin #>. <pin #> is 9,21,36 or 37 on the EzSBC1. **PWM** Must prepare the pin first using the appropriate **PINMODE** command. The duty cycle set up on the pin is given by <fractional count> divided by <total count>. The <total count> divides a frequency of 30MHz to get the PWM frequency. By using a low pass filter on the pin one can obtain an analog signal proportional to the duty cycle. NOTE: all of the **PWM** channels have the same <total count>, so changing the <total count> for one channel changes <total count> for the rest of them. Therefore one should usually pass the same total count value to all pins when using the PWM command.

REM, '

REM <remark>

Allows a comment to be placed in the code. The comment ends at the end of the line. If the comment is not in quotes, it will be tokenized e.g.: **REM** I want to print the value will turn into **REM** I want to **PRINT** the value Alternatively, **REM** "I want to print the value" is not changed. Use **'** instead.

' <remark>
Allows a comment to be placed in the code. The comment ends at the end of the line.

REPEAT

Repeats a section of code until a corresponding **UNTIL** condition is satisfied. Do not **GOTO** out of a **REPEAT**/**UNTIL** loop. If you skip over the **UNTIL** the loop with not be properly unwound. Always use the **UNTIL** to exit the **REPEAT**/**UNTIL** loop.

Be aware that it is possible for an **IF/THEN/ELSE/ENDIF** instruction to straddle the **UNTIL** condition in such a way that the loop is not correctly unwound. This will lead to hard to find bugs.

RIGHT$

x$=**RIGHT$**(<string>,<num characters>)

Returns the last <num characters> characters of <string>.

RND

x=**RND**(<number>)
Returns a pseudorandom number between 0 and <number>-1 if <number> is positive. If <number> is negative, the value -<number> is used to seed the random number generator. See http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html for the algorithm employed.

RETURN

Return to the statement after the calling **GOSUB** command. See **GOSUB**.

SERBNG

SERBNG <pin #>, <speed>, <string>

SERBNG outputs an RS232 compliant serial stream on the <pin #> pin with 8 bits, 1 stop bit, no parity. This is bit-banged so any pin can be used to output data. The <speed> is how long each bit should be delayed. For some standard Baud Rates the Speed Values are shown in the table below. The <string> is the string data to output to the pin as serial data.

The SERBNG instruction is a 'blocking' instruction and the interpreter waits at that line till the data is transmitted. Once the data has gone then the interpreter advances to the next line. The serial IO to the hardware ports (using SEROUT) are non-blocking and the data gets dumped in a buffer and transmitted in the background so in principal, those instructions can have timing issues but the buffers are large and errors will show up as incomplete messages or dropped characters. The SERBNG instruction does not care about the content of the string and it does not need a terminating character. The strings in the EzSBC1 interpreter can contain any data including the 0x0 byte and it will be processed correctly.

```
PINMODE 30, OUT
```

```
SERBNG 30, 467, "Testing"
```

The instructions above will transmit “Testing” on pin 30 at approximately 9600 bits per second.

Baud Rate	Speed Value
110	41020
300	15040
600	7520
1200	3760
2400	1880
4800	934
9600	467
19200	235
38400	118
57600	76
115200	37
230400	17
460800	7

See **SERINIT**, **SEROUT**, **SERINP**$.

SERINIT

SERINIT <port #>,<baud rate>,<databits>,<stopbits>,<parity>

Initializes the serial port denoted by <port #>. Currently two serial ports are supported denoted by <port #>=0 and <port #>=1. <baud rate> is in bps. <databits>=7 or 8 for the number of data bits, <stopbits>=1 or 2 for the number of stop bits, and <parity>=0 for no parity, <parity>=1 for odd parity, and <parity>=2 for even parity.

```
SERINIT 1,9600,8,1,0  ‘port 1, 9600 bps, 8 bits,1 stop bit, no parity
```

Port 0 is used for the USB-serial connection. Port 1 is on EzSBC1 pin 37 (RXD1) and pin 36 (TXD1).

SERINP$

<instring>=**SERINP$**(<port #>,<numchars>,<endchar>,<timeout>)

Reads up to <numchars> from serial port <port #>. If <numchars> is negative, then -<numchars> characters will be read, but discarded, which is useful for clearing the receive queue. It will stop receiving characters if the character denoted by <endchar> is received and <endchar> is not a negative number. The number <timeout> is proportional to the amount of time to wait for the characters to arrive before the **SERINP$** function terminates. Currently a value <timeout>=50000 produces about a one second wait before timeout. Set <timeout>=1 to return with only the queued characters. A string is returned with the characters received. See **SERINIT** to set the serial port parameters.

SEROUT

x=**SEROUT**(<port #>,<string>)

Send the characters in <string> out serial port number <port #>. The return code currently is zero, but would be negative for an error code. See **SERINIT** to set the serial port parameters.

```
x=SEROUT(1, “string”)       ‘send string to pin 36, TXD1 on the EzSBC1
x=SEROUT(0, “string”)       ‘send string to the USB serial port
```

If you are using the SEROUT to send to a RS-232 serial printer and you want to send a CR & LF the following program will work.

```
‘First use SERINIT to initialize the port.
SERINIT 1,9600,1,0     ‘port 1, 9600 bps, 8 bits, 1 stop bit, no parity
CRLF$ = CHR$(0X0D) + CHR$(0X0A)  ‘Hex for CR & LF
SOUT$ = ”Some text” + CRLF$      ‘Combines text & CRLF
x = SEROUT (1, SOUT$)            ‘Sends SOUT$ to serial port 1 (pin 36)
END
```

SERSPI

SERSPI <spibytein>=**SERSPI**(<MOSI pin>,<MISO pin>,<CLK pin>,<output data>, <delay>)

Clocks out an 8-bit byte using the SPI master protocol with CPOL=0 and CPHA=0 (Type 0). <MOSI pin> is used as the MOSI pin (must be an output pin), <MISO pin> is used as the MISO pin (must be set as an input pin), <CLK pin> is the clock pin (an output pin), and <output data> is a byte to output. <delay> controls the speed at which the clock is toggled, larger values result in lower clock rates. The data received on MISO is returned by the function. The chip select lines must be controlled by **OUTD** commands.

SERVO

SERVO Pin, Pulsewidth

The **SERVO** command is a specialized PWM command to simplify the control of hobby servos that use Pulse Width Modulation. The Pin parameter specifies which pin to use; 9, 21, 36 or 37. Pulsewidth is the desired pulse width in micro-seconds.

SERVO 21, 1500 sets a 1.5ms pulse width with a 50Hz repetition rate on pin 21. The allowable range for the pulse width is 500 to 2000 microseconds.

SETTICK

SETTICK(NewValue)

SETTICK is used to set the value of the 100µs per tick background timer to a known value NewValue,. Normally NewValue will be zero to reset the background timer to prevent it from overflowing or to measure a time interval in the program. See **GETTICK**

```
' Gettick, Settick Test
CLRSCR
PINMODE 41,OUT  'Red LED pin
SETTICK(0)
REPEAT
    now = GETTICK
    REPEAT
        ' Execute some code for a tenth of a second
        OUTD 41,0
    UNTIL GETTICK > ( now + 1000 )
    'Flash the Red LED
    now = GETTICK
    REPEAT
        OUTD 41,1
    UNTIL GETTICK > ( now + 1000 )
    ch=INKEY
UNTIL GETTICK > 100000  'Program runs for ten seconds
END
```

SHIFTIN

X = **SHIFTIN**(DataPin, ClkPin, Mode)

The SHIFTIN instruction shift eight bits of data in from an external shift register. The DataPin is the pin number of the pin where the data will be read, The ClkPin is the pin number where one low to high to low transition will be made for every data bit read. The Mode parameter can take only two values, 0 and 1. When Mode=0 the Most Significant Bit is shifted in first. When Mode=1 the Least Significant Bit is shifted in first. The clock pulse is 1µs wide.

SHIFTOUT

SHIFTOUT(DataPin, ClkPin, Mode, Data)

The SHIFTOUT instruction shifts the lower eight bits of Data out to an external shift register (latch). The DataPin is the pin number of the pin where the data will be driven, The ClkPin is the pin number where one low to high to low transition will be made for every data bit sent. The Mode parameter can take only two values, 0 and 1. When Mode=0 the Most Significant Bit is shifted out first. When Mode=1 the Least Significant Bit is shifted out first. The clock pulse is 1µs wide.

SIN

X=**SIN**(Theta)

The SIN function returns the value of the trigonometric Sinus function of angle Theta where Theta is expressed in radians. ASIN is the inverse of the SIN function

STEP

STEP

See the **FOR** keyword.

SSTEP

SSTEP

Breaks program execution and Invokes the single stepping function. Can be used anywhere in the program where an instruction is allowed. To break into the debugger when a variable, lets say x, has a particular value is easy: Place this code where the value of x is changed:

```
IF x=10 THEN
    SSTEP    'Breakpoint
ENDIF
```

and when x=10 the program will start in single step mode.

STR$

Convert numbers to a string representation.

```
X=100
Num$ = STR$(X)                      'Make a string out of the value of X
CRLF$ = CHR$(0X0D) + CHR$(0X0A)     'Hex for CR & LF
SOUT$ = Num$ + CRLF$                'Combines Num$ with CRLF
x = SEROUT(1, SOUT$)                'Sends SOUT$ to serial port 1 (pin 36)
```

STRING$

x$=**STRING$**(<repeats>, <string>)

Return a string with the string <string> repeated <repeats> times.

TAB

TAB(x) produces space characters until the cursor is at column x. TAB is only useful in the **PRINT**

command.

TAN

Y=**TAN**(x)

The TAN function calculates the Tangent of angle x. X is in radians. See the ATAN and ATAN2 functions.

TIME$

x$=**TIME$**
Gets the current time from the RTC to a string in the format "YYYY-MM-DD HH:MM:SS" where YYYY is the year, MM is the month, DD is the day, HH is the hour (in 24 hour format), MM is the minute, and SS is the second. For the RTC to maintain the date and time when the main power is removed a Lithium coin cell must be connected between pin 38 and ground. There is an on board switch over circuit and the battery will only be used to power the RTC when the main power is unavailable or below 3V.

TIMESET

x=**TIMESET**(<setstring>)

This command adjusts the date and time of the integrated Real Time Clock. **TIMESET** sets the time, with a string in the format "YYYY-MM-DD HH:MM:SS". It returns zero for no error, or a negative number for error. The only error condition detected by this command is an improperly formatted string.

TO

The **TO** keyword is used to set the upper boundary of iteration for the **FOR** loop.

TONE

TONE <pin #>,<duration>,<frequency>

Toggle the pin <pin #> high and low for a duration of <duration> milliseconds at a frequency of <frequency> cycles per second (Hz). The pin must be set as a GPIO output first, e.g. **PINMODE** <pin #>, OUT. The frequency is not very accurate. Very precise frequencies can be generated with the PWM output pins. See the **PWM** command.

TRIM$

x$=**TRIM$**(<string>,<mode>)
Returns a new string with the spaces, tabs, newlines, and carriage returns removed from the end of the string if mode=0, or the beginning of the string if mode=1. If mode is omitted, mode=0 is assumed.

UNTIL

UNTIL <condition>

If the <condition> is false, return to the last **REPEAT**, otherwise continue with the next statement. Do not skip over an **UNTIL** with a **GOTO** statement, the **UNTIL** must always evaluate to true to remove **UNTIL** from **REPEAT/UNTIL** stack.

<condition> may contain the following test for equality: **=, <, >, <=, >=** and they may be combined with **AND** and **OR** to form complex conditionals. See **REPEAT**.

UPPER$

x$=**UPPER$**(<string>,<mode>)
Returns a new string with the letters a-z converted to uppercase if mode=0, or lowercase if mode=1. If mode is omitted, uppercase conversion is assumed.

VAL

x=**VAL**(<string>)
Returns a numerical representation of the string <string>.

VALLEN

x=**VALLEN**(<string>)
Returns the number of characters in the numerical representation of the string <string>. e.g.

```
x=VALLEN("100")
```

assigns 3 to x.

```
x=VALLEN("100blah")
```

assigns 3 to x

WAIT

WAIT <milliseconds>

Wait delays program execution by an integral number of milliseconds Also see **DELAY**.

WEND

WEND
End of a WHILE loop. See **WHILE**.

WHILE

WHILE <condition>

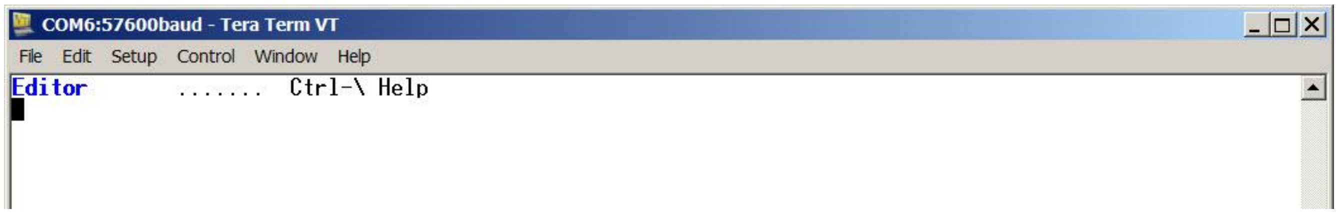
If the <condition> is true then the instructions between the WHILE and the matching WEND keyword is executed. If the <condition> is false then the program execution resumes at the line following the matching WEND keyword. WHILE can be nested inside other WHILE blocks. <condition> may contain the following test for equality: **=, <, >, <=, >=** and they may be combined with **AND** and **OR** to form complex conditionals.

Entering a Program

Blink A LED

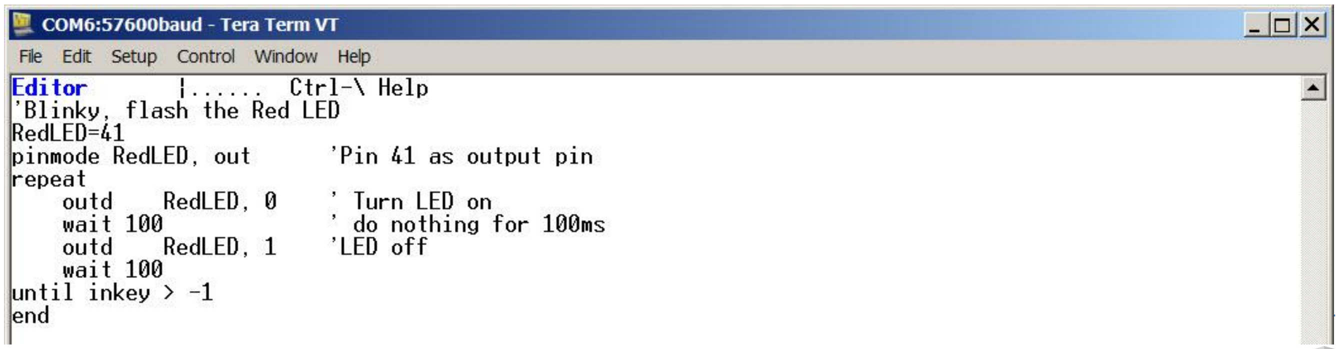
Traditionally blinking an LED is the first program on an embedded controller, much like 'Hello World.' in a desktop programming language.

Type **'e'** to enter the editor and type the code listed below. If the screen does not appear as in the screen capture below then another program has been loaded earlier. Skip to the section on 'Deleting an Entire Program' and follow the instructions before returning here.


COM6:57600baud - Tera Term VT
File Edit Setup Control Window Help

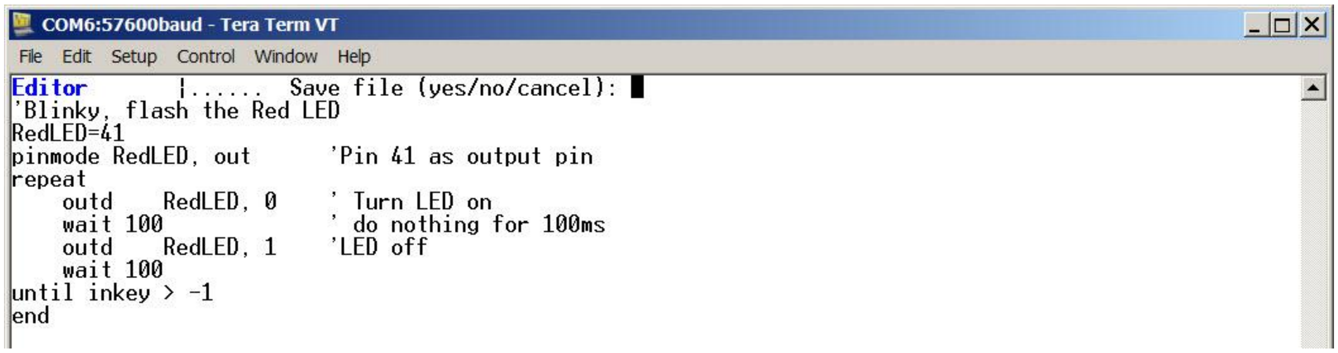
```
Editor       .......  Ctrl-\ Help
```

```
'Blinky, flash the Red LED
RedLED=41
pinmode RedLED, out       'Pin 41 as output pin
repeat
    outd    RedLED, 0     ' Turn LED on
    wait 100              ' do nothing for 100ms
    outd    RedLED, 1     'LED off
    wait 100
until inkey > -1
end
```


COM6:57600baud - Tera Term VT
File Edit Setup Control Window Help

```
Editor       |......  Ctrl-\ Help
'Blinky, flash the Red LED
RedLED=41
pinmode RedLED, out       'Pin 41 as output pin
repeat
    outd    RedLED, 0     ' Turn LED on
    wait 100              ' do nothing for 100ms
    outd    RedLED, 1     'LED off
    wait 100
until inkey > -1
end
```

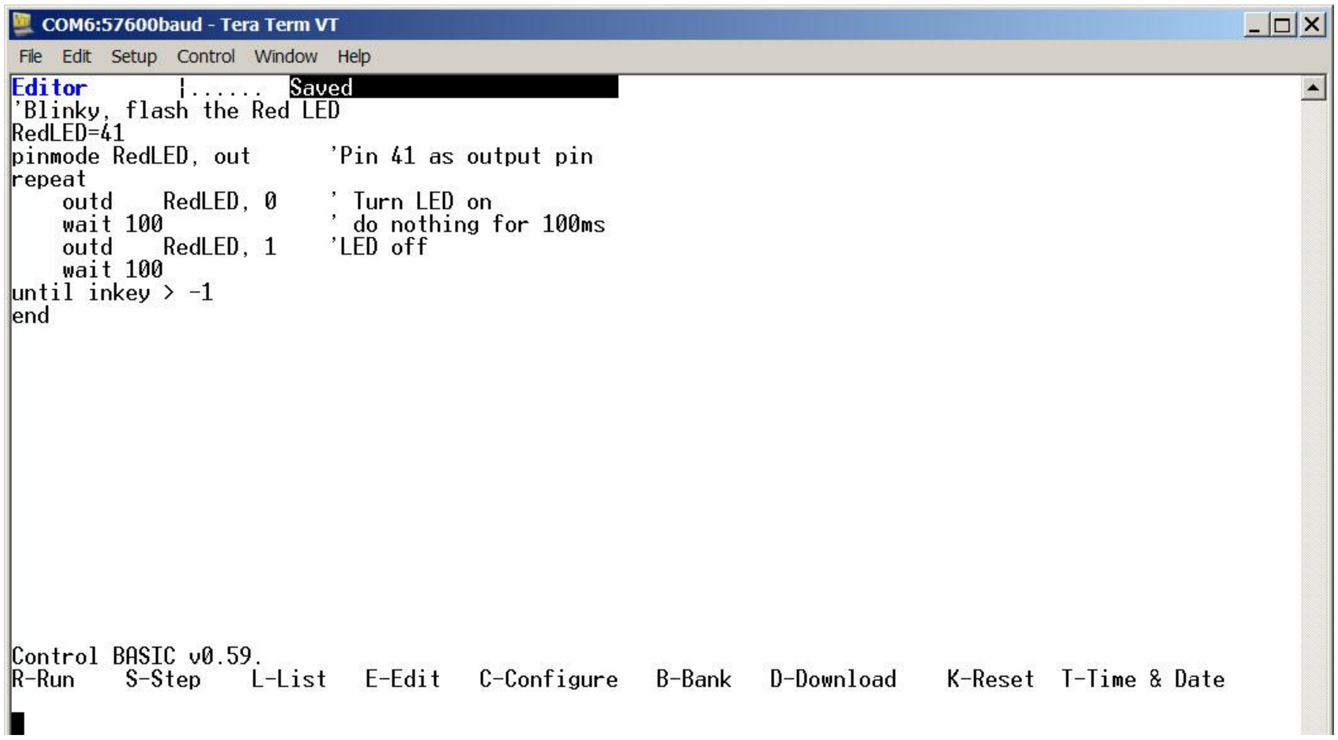
When the screen looks like the screen capture shown above press **Ctrl-W** and note that the top of the screen has changed as in the screen capture below.



```
Editor       |......  Save file (yes/no/cancel):
'Blinky, flash the Red LED
RedLED=41
pinmode RedLED, out      'Pin 41 as output pin
repeat
    outd    RedLED, 0    ' Turn LED on
    wait 100             ' do nothing for 100ms
    outd    RedLED, 1    'LED off
    wait 100
until inkey > -1
end
```

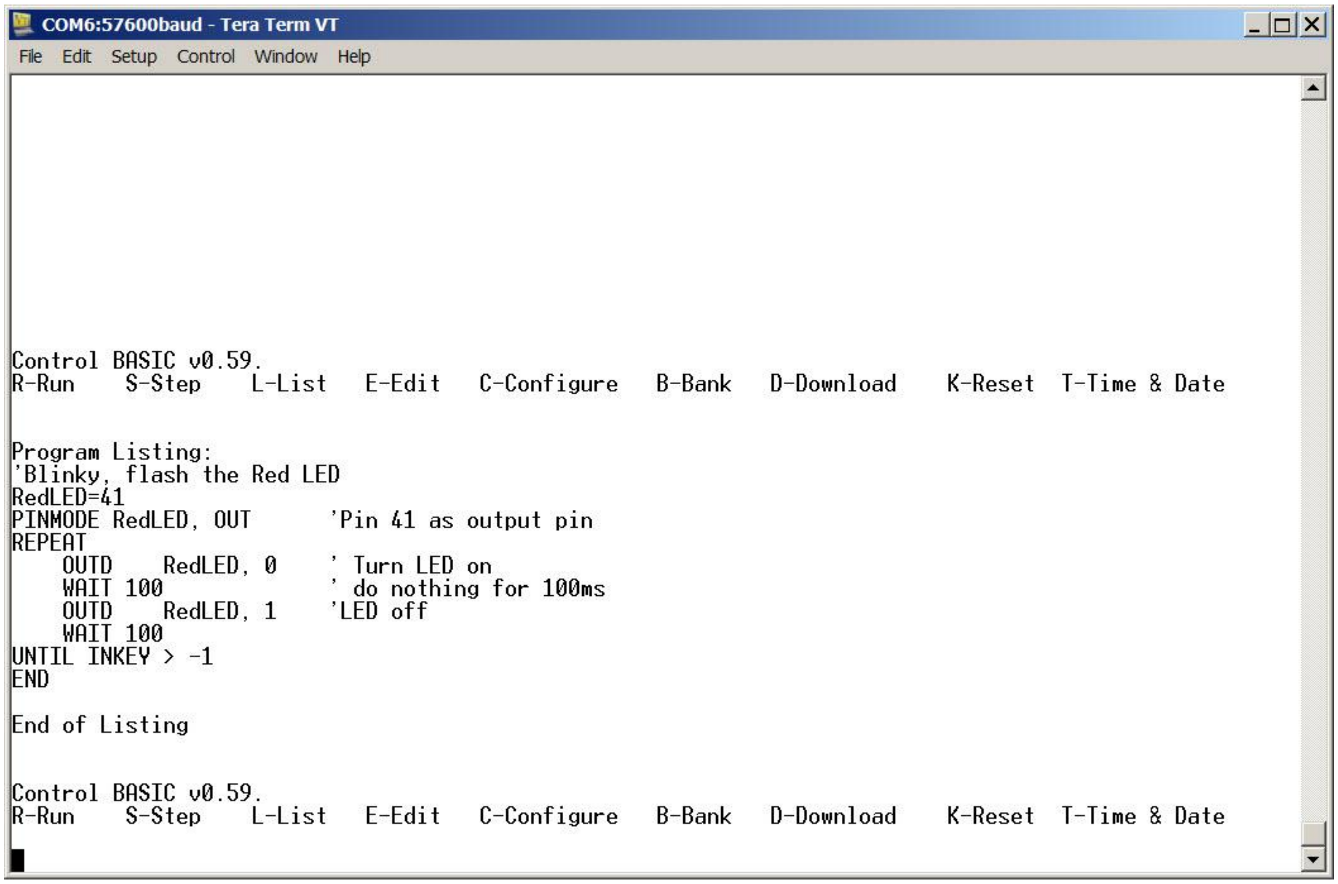
The **'Save file (yes/no/cancel):**' message is asking for permission to write the program into the memory on the EzSBC1. **Press y** to save the program. If you press n the editor will quit without saving anything and c will not save the program but you will remain in the editor.

This is what you should see:



```
Editor       |......  Saved
'Blinky, flash the Red LED
RedLED=41
pinmode RedLED, out      'Pin 41 as output pin
repeat
    outd    RedLED, 0    ' Turn LED on
    wait 100             ' do nothing for 100ms
    outd    RedLED, 1    'LED off
    wait 100
until inkey > -1
end









Control BASIC v0.59.
R-Run    S-Step    L-List   E-Edit   C-Configure   B-Bank   D-Download    K-Reset  T-Time & Date
```

Now press l and EZmon will list the program to the screen. It should look like this:


COM6:57600baud - Tera Term VT
File Edit Setup Control Window Help

```
Control BASIC v0.59.
R-Run    S-Step    L-List    E-Edit    C-Configure    B-Bank    D-Download    K-Reset   T-Time & Date


Program Listing:
'Blinky, flash the Red LED
RedLED=41
PINMODE RedLED, OUT       'Pin 41 as output pin
REPEAT
    OUTD    RedLED, 0     ' Turn LED on
    WAIT 100              ' do nothing for 100ms
    OUTD    RedLED, 1     'LED off
    WAIT 100
UNTIL INKEY > -1
END

End of Listing


Control BASIC v0.59.
R-Run    S-Step    L-List    E-Edit    C-Configure    B-Bank    D-Download    K-Reset   T-Time & Date
```

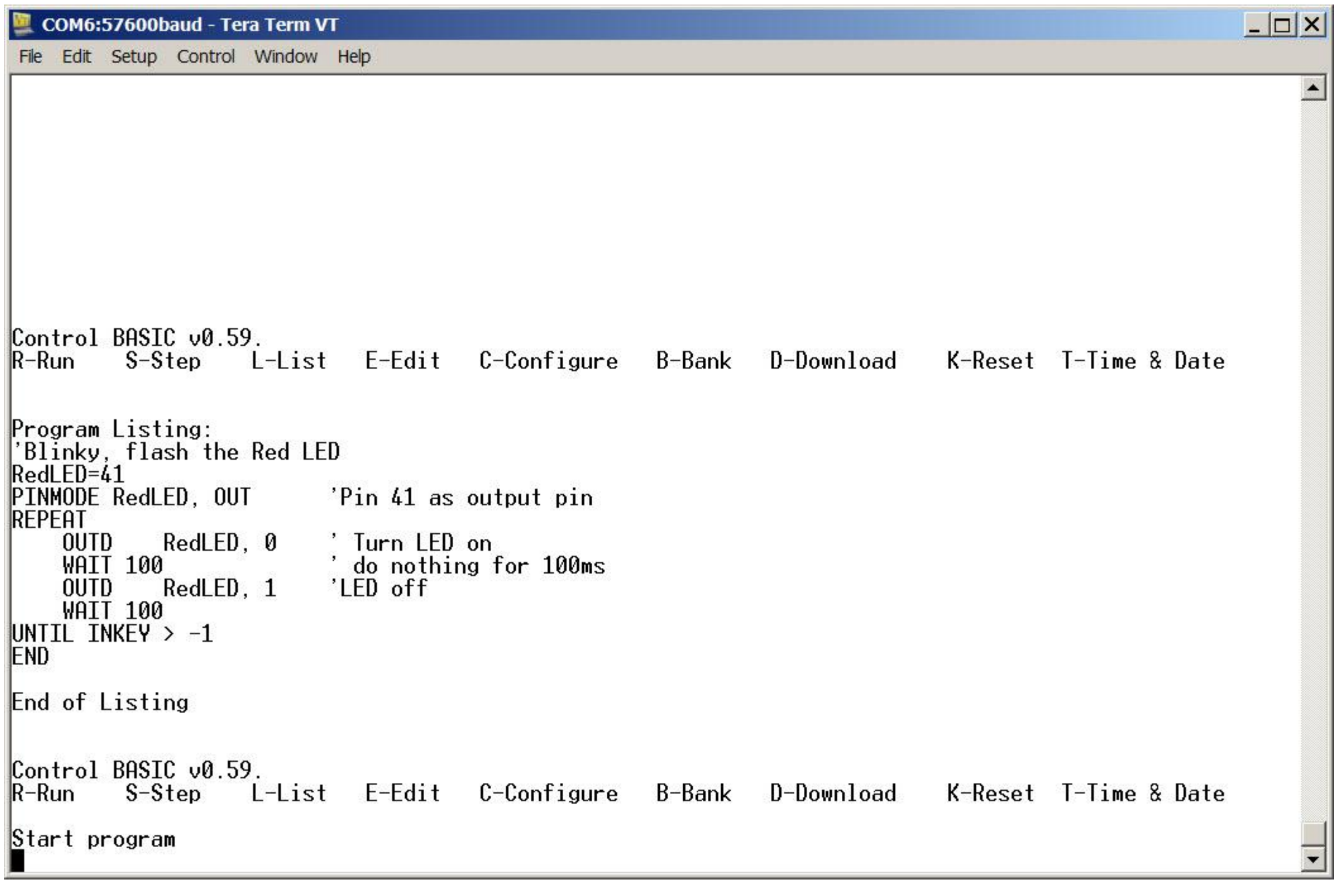
See how many words are now in upper case. The interpreter (actually the tokenizer) has recognized these words as valid Control BASIC reserved words and they will now be displayed in upper case. The variables were not changed; they are exactly as you typed them. If a variable changed case to upper case then you accidentally chose a variable name that is the same as a reserved word of Control BASIC. This is an error check of your program and it is recommended that variable names not be typed in all upper case letters. In fact, typing in lower case allows many errors to be found by listing the program and paying attention to the capitalization of the text on the screen.

We are now ready to run our first program. Press r to run the program and see the red LED flash on and off 5 times per second.

When the program is running you will see **'Start program'** displayed on the terminal screen (unless the program cleared the screen). This is an indication that the program has started and is running. If the program encounters an error then it will stop running and display an error message on the terminal.

Press any key on the keyboard to end the program.

The terminal display should like like this:


COM6:57600baud - Tera Term VT
File Edit Setup Control Window Help

```
Control BASIC v0.59.
R-Run    S-Step    L-List    E-Edit    C-Configure    B-Bank    D-Download    K-Reset   T-Time & Date


Program Listing:
'Blinky, flash the Red LED
RedLED=41
PINMODE RedLED, OUT      'Pin 41 as output pin
REPEAT
    OUTD    RedLED, 0    ' Turn LED on
    WAIT 100             ' do nothing for 100ms
    OUTD    RedLED, 1    'LED off
    WAIT 100
UNTIL INKEY > -1
END

End of Listing


Control BASIC v0.59.
R-Run    S-Step    L-List    E-Edit    C-Configure    B-Bank    D-Download    K-Reset   T-Time & Date

Start program
```

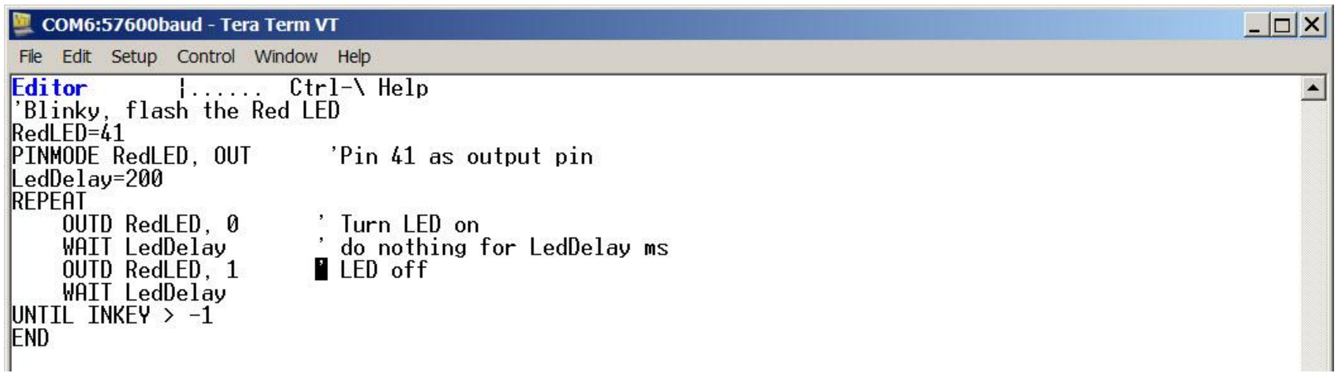
List the program again. Note that there are no 'GOTO' commands or line numbers in the program. The version of BASIC used on the EZSBC does not require line numbers or the use of GOTO commands. The GOTO command is supported but should be used sparingly. The targets of GOTO or GOSUB commands are labels that can appear at the beginning of any line or on lines by themselves.

The first line 'Blinky, flash the Red LED is a comment. Comments start with ' and end at the end of the line. The next line RedLED=41 assigns the value 41 to a variable called RedLED. Pin 41 on the EzSBC1 is a phantom pin to give easy access to the onboard red LED. Three more phantom pins exist; 42, 43 and 44 for controlling the yellow, green and blue LEDs. Setting these pins to output pins and low turns on the LED associated with the phantom pin.

The line pinmode RedLED, out sets the phantom pin as an output pin, as the comment suggests. The lines between the repeat and until inkey > -1 are executed repeatedly until a key is pressed on the terminal emulator keyboard. The keyword INKEY returns the ASCII value of a key and -1 if a key has not been pressed. OUTD is the digital output command of Control BASIC. IND is the input command. The WAIT 100 line causes the program to do nothing for 100 milliseconds.

```
repeat
    outd    RedLED, 0    ' Turn LED on
    wait 100             ' do nothing for 100ms
    outd    RedLED, 1    'LED off
    wait 100
until inkey > -1
end
```

Now for a few more features of the EZmon editor. Press e to edit the program. You see the (old) program on the screen. Let's change the program so it is easy to change the rate at which the LED flashes. Change the program to the program in the screen capture.


COM6:57600baud - Tera Term VT
File Edit Setup Control Window Help

```
Editor       |......  Ctrl-\ Help
'Blinky, flash the Red LED
RedLED=41
PINMODE RedLED, OUT      'Pin 41 as output pin
LedDelay=200
REPEAT
    OUTD RedLED, 0      ' Turn LED on
    WAIT LedDelay       ' do nothing for LedDelay ms
    OUTD RedLED, 1      ' LED off
    WAIT LedDelay
UNTIL INKEY > -1
END
```

The word LedDelay is a new variable and it is assigned the value 200. The delay can now be changed in a single location. Make the changes by navigating with the cursor keys and typing. You will see that typing inserts new characters. The up and down cursors change lines and left and right cursors move to the left and right. Home, End, PageUp, PageDown, Delete, Tab and Backspace all work if Tera Term or your favorite terminal emulator is configured correctly. Pressing the Insert key changes the mode to Overstrike and back to Insert mode if you press it again.

Ctrl-W y will save the changes and quit. Just remember, this is a remote terminal, it is completely unaware of the mouse, so use only keystrokes.

Run the program again. The LED now flashes at half the earlier rate. While the program is running press Ctrl-C and see the text **'Ctrl-C detected'** followed by a line that starts with a number followed by a colon maybe

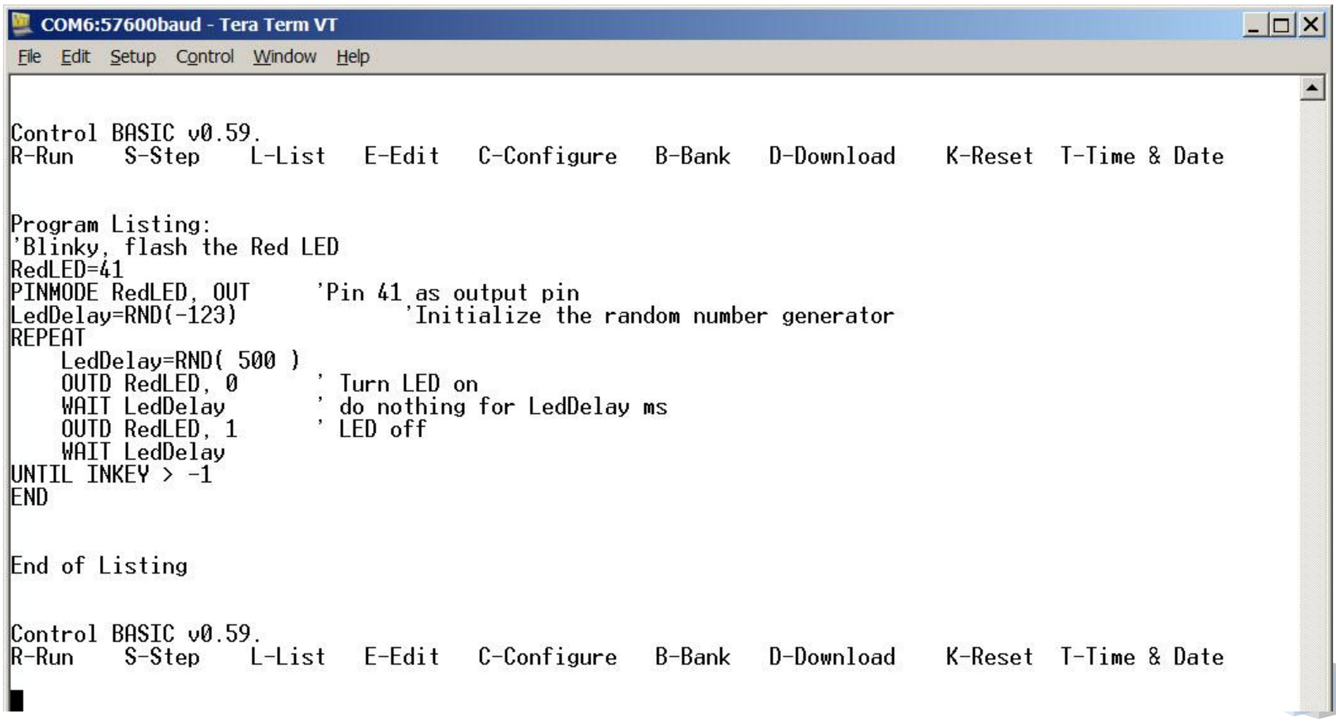
```
10: UNTIL INKEY > -1
```

Now press the Space Bar and see the code appear a line at a time like this:

```
Ctrl-C detected
10: UNTIL INKEY > -1
6:     OUTD RedLED, 0      ' Turn LED on
7:     WAIT  LedDelay       ' do nothing for LedDelay ms
8:     OUTD RedLED, 1      ' LED off
9:     WAIT LedDelay
10: UNTIL INKEY > -1
6:     OUTD RedLED, 0      ' Turn LED on
7:     WAIT  LedDelay       ' do nothing for LedDelay ms
8:     OUTD RedLED, 1      ' LED off
9:     WAIT LedDelay
10: UNTIL INKEY > -1
6:     OUTD RedLED, 0      ' Turn LED on
7:     WAIT  LedDelay       ' do nothing for LedDelay ms
8:     OUTD RedLED, 1      ' LED off
9:     WAIT LedDelay
10: UNTIL INKEY > -1
6:     OUTD RedLED, 0      ' Turn LED on
```

You are stepping through the program a line at a time. The number before the colon is the line number to be executed next and the text is the program code to be executed. To continue running the program at full speed type g.

Change the program again so that it looks like the screen capture below

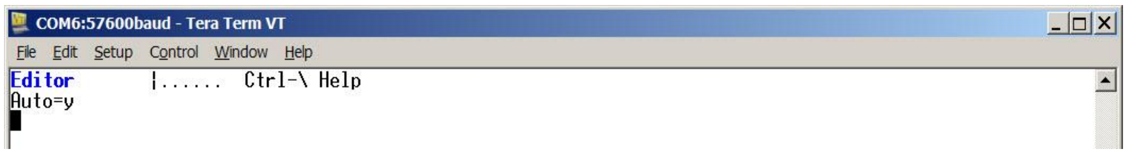

COM6:57600baud - Tera Term VT
File Edit Setup Control Window Help

```
Control BASIC v0.59.
R-Run    S-Step    L-List   E-Edit   C-Configure   B-Bank   D-Download    K-Reset   T-Time & Date


Program Listing:
'Blinky, flash the Red LED
RedLED=41
PINMODE RedLED, OUT     'Pin 41 as output pin
LedDelay=RND(-123)             'Initialize the random number generator
REPEAT
    LedDelay=RND( 500 )
    OUTD RedLED, 0       ' Turn LED on
    WAIT LedDelay        ' do nothing for LedDelay ms
    OUTD RedLED, 1       ' LED off
    WAIT LedDelay
UNTIL INKEY > -1
END


End of Listing


Control BASIC v0.59.
R-Run    S-Step    L-List   E-Edit   C-Configure   B-Bank   D-Download    K-Reset   T-Time & Date
```

and run the program. The LED now flashes in some unpredictable way, not very useful but good for demonstrating another feature of EZmon. LedDelay = RND(500) assigns a random value between 0 and 500 to LedDelay. Press s and note that the program starts in single step mode. After every line press v before pressing the Space Bar. As the variables are defined and their values change they can be examined. Since the delay changes randomly the only way to know the value is to print it to the screen or view it by single stepping the program.

Having an embedded controller is not much use if it does not run automatically when the power is applied or the reset button is pressed. That leads to the next section.

Running a Program Automatically


COM6:57600baud - Tera Term VT
File Edit Setup Control Window Help

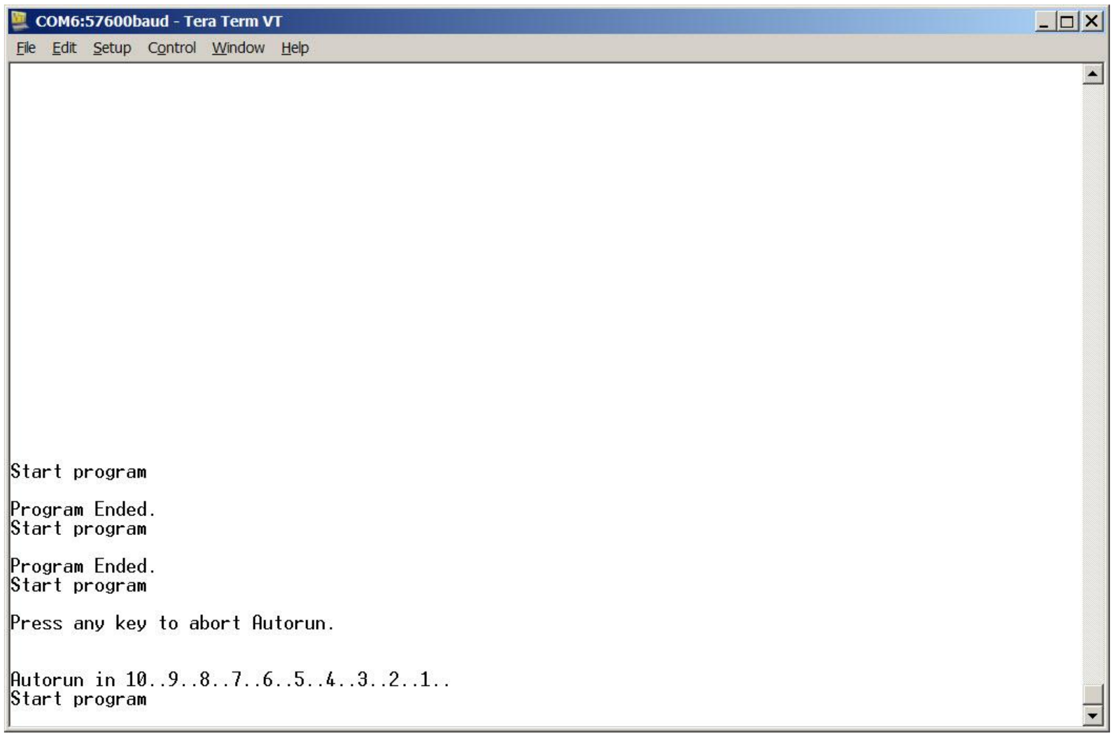
```
Editor      |......  Ctrl-\ Help
Auto=y
```

Stop the program if it is running. Press c and you will see the editor open up but it will not show the program. Type Auto=y and save the edit **(Ctrl-W y).** See that the program started running immediately when you quit the editor. When you press a key on the keyboard you see the lines

```
Start program


Program Ended.
Start program
```

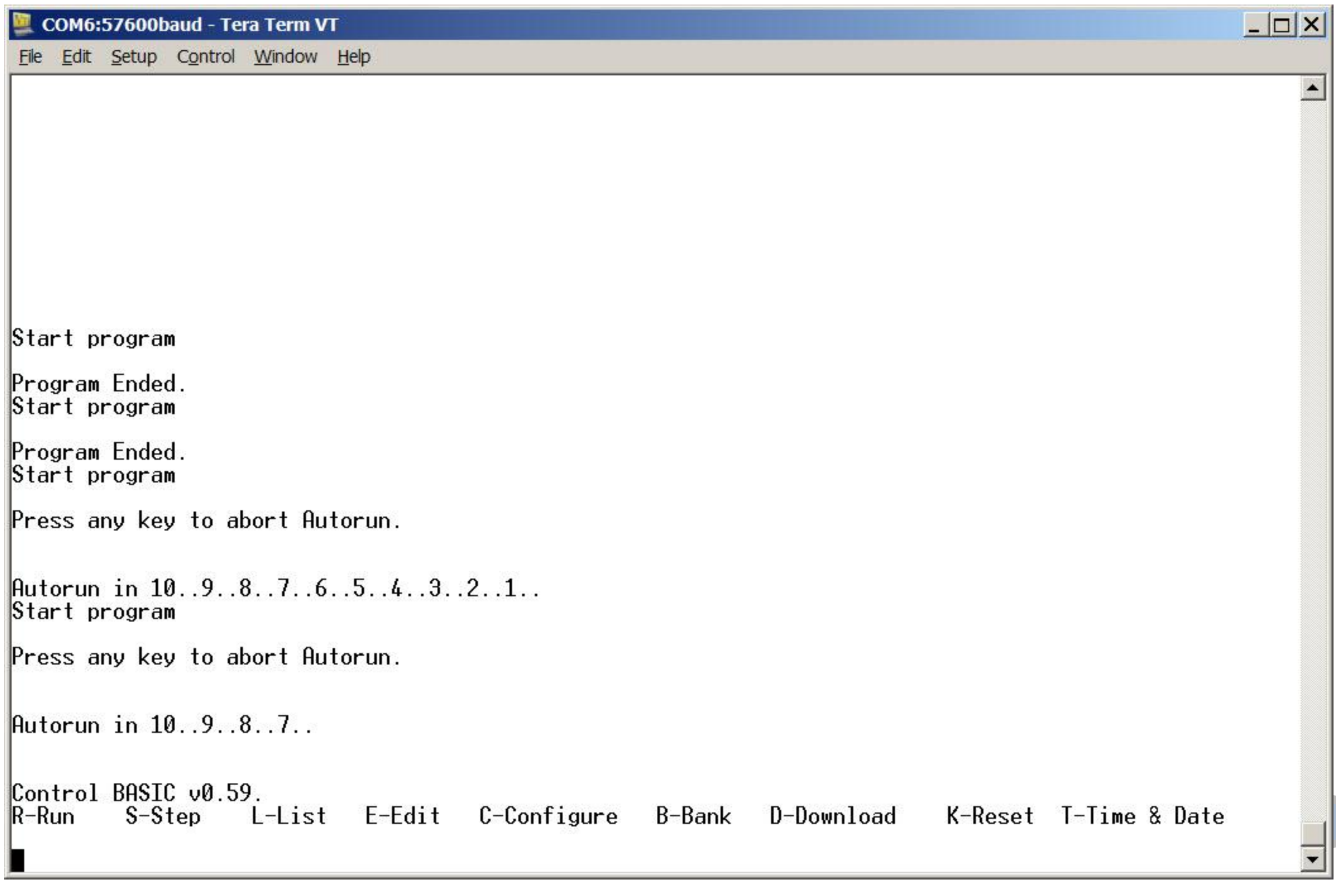
appear on the display. The program now runs automatically on start up and will restart if it ends, like a real embedded controller should.

Press the Reset button on the EzSBC1 and you will see this message appear on the terminal window:


COM6:57600baud - Tera Term VT
File Edit Setup Control Window Help

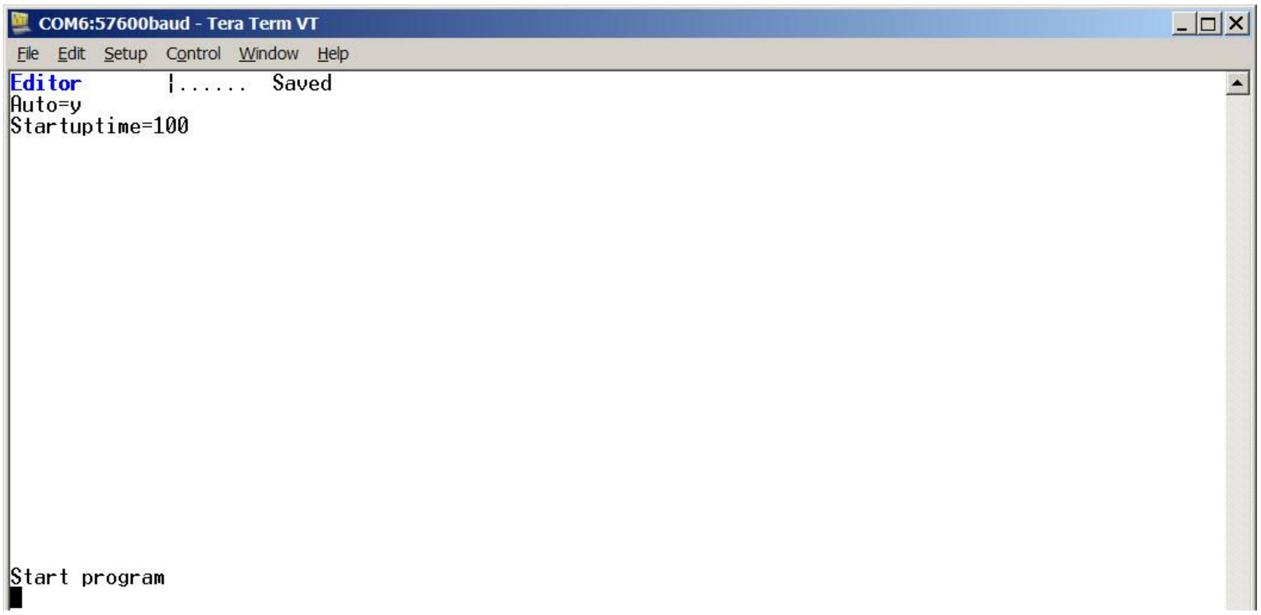
```
Start program

Program Ended.
Start program

Program Ended.
Start program

Press any key to abort Autorun.


Autorun in 10..9..8..7..6..5..4..3..2..1..
Start program
```

Now press the reset button again and before the countdown gets to zero press a key on the terminal emulator keyboard and this is what you will see:


COM6:57600baud - Tera Term VT
File Edit Setup Control Window Help

```
Start program

Program Ended.
Start program

Program Ended.
Start program

Press any key to abort Autorun.


Autorun in 10..9..8..7..6..5..4..3..2..1..
Start program

Press any key to abort Autorun.


Autorun in 10..9..8..7..


Control BASIC v0.59.
R-Run    S-Step    L-List    E-Edit    C-Configure    B-Bank    D-Download    K-Reset   T-Time & Date
```

Change the Configuration parameters by pressing c and then typing the new entry till the screen looks like the first three lines on the screen capture. Save the changes to the Configuration parameters by typing **Ctrl-W**.


COM6:57600baud - Tera Term VT
File Edit Setup Control Window Help

```
Editor       |......   Saved
Auto=y
Startuptime=100




Start program
```

Now the delay before the program starts is very long, around 50 seconds. As a precaution against never being able to regain control of the EzSBC1 values of x<5 are ignored 5 is used. The default value is 10 when the parameter is not specified. Specifying Bank=0 (the default) or Bank=1 selects from which bank the program is started.

See the CONF$ keyword for more uses of the configuration area.

Memory Bank Selection

The EzSBC1's program memory is divided into two almost equal sized banks of memory, the first one Bank0 being 64k Byte in size and the second bank being 60k Byte in size. Four kilobytes of flash is 'stolen' from the second bank to implement the Configuration Area. The memory is divided into two banks for a few reasons. It is very useful to have a place to test pieces of code without going through the effort of saving the program to the host computer, loading a test program and the downloading the saved program again.

Some systems have two distinct behaviors. A good example of such a system is a data logger. When the system is in the field collecting data it is busy with one very specific task. When the system is retrieved and the data needs to be downloaded to a host computer the behavior and requirements are entirely different from the logging requirements. By having the option to split the program into the two banks the two resulting programs are much simpler to design and implement than one big program that has to perform both sets of tasks. By loading one program in each bank and selecting which bank's program executes, the system has two entirely different personalities, each one fairly easy to implement.

When EZmon is active, press **the B key** and a three lines of text will appear:

```
Current bank: 1
Bank #0 contents: 'Blinky, flash the Red LED
Bank #1 contents:
```

The first line of code from each bank is displayed as well as the currently active bank. Pressing 0 or 1 selects the appropriate bank, any other key causes no change. Since the first line of code is displayed it is good practice to make the first line of any program a comment describing the function of the program.

Specifying Bank=0 (the default) or Bank=1 in the Configuration area selects from which bank is active after power-up or reset.

Download a Program

It gets very tiresome to type long programs using only the editor function of EZmon. It is intended for making quick changes to programs in the field or while debugging. Writing programs is best done on the PC using your favorite programmers' editor. Programmers Notepad and CodeLite are both free editors for writing programs. Don't write the program using WordPad or similar word processors; they do not store the programs in ASCII format.

By pressing D at the prompt the Controller will wait for a program to save into the current bank. Immediately pressing Ctrl-D will clear the bank. Using the 'Send File' option on the TeraTerm **File** menu will download a program to the controller. The program must be in ASCII. For large programs the download operation can take some time. The controller writes the Flash in 256 byte blocks and each block is erased before the new code is written into the block of Flash. Also, the program in not stored in ASCII in the Flash but rather in a tokenized format to save space and increase execution speed. Tokenization occurs after the block is downloaded but before it is programmed into the Flash. When the controller is busy writing to the Flash, it cannot save data arriving on the serial port. When the EzSBC1's buffer is nearly full or prior to programming or erasing the Flash, the controller sends the X-Off character to the host to temporarily suspend serial communication via the USB. When the controller is ready for more data it sends the X-On character to allow the host to resume sending data. To enable this feature to work Xon/Xoff flow control must be enabled on the serial port connecting to the controller

Deleting an Entire Program

A quick way to get a blank page (bank/program) is to 'Download' nothing, an empty program. Get to the EZmon prompt:

```
Control BASIC v0.59.

R-Run S-Step L-List  E-Edit  C-Configure  B-Bank  D-Download  K-Reset
T-Time & Date
```

```
Type 'd'
```

You should see:

```
Send program.

Use XON/XOFF flow control.

Press Ctrl-D after download completes.
```

Now type **Ctrl-D** (Control key and D simultaneously). You should see:

```
Programmed bytes 0x100
```

Meaning that 256 bytes of the program memory was written to (out of 65535 bytes).

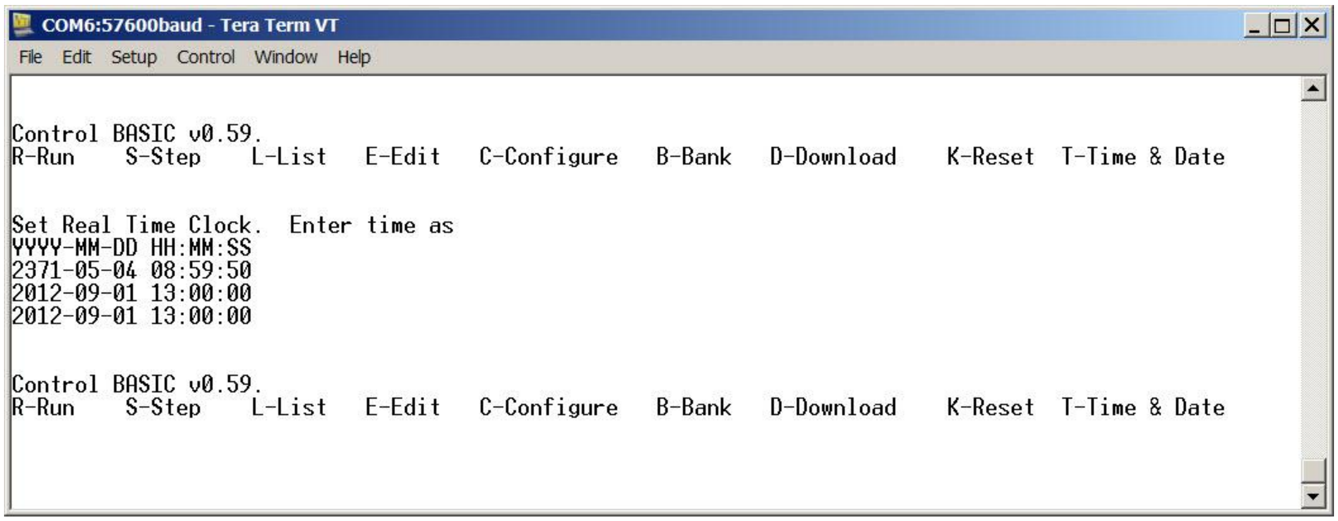
Setting the Time

```
Control BASIC v0.59

R-Run S-Step L-List  E-Edit  C-Configure  B-Bank  D-Download  K-Reset
T-Time & Date
```

Press T and you should see something like this:

```
Set Real Time Clock.  Enter time as
YYYY-MM-DD HH:MM:SS
2371-05-04 08:54:61
```

Now type the date and time exactly as shown using the - key next to the 0 on the keyboard, not the keypad -. Hours must be entered in 24 hour format so to specify 1PM, enter 13 in the hour position. Leading zeros must be entered. Any error will not update the time. A successful update will look like the screen capture below


COM6:57600baud - Tera Term VT
File Edit Setup Control Window Help

```
Control BASIC v0.59.
R-Run    S-Step    L-List   E-Edit   C-Configure   B-Bank   D-Download    K-Reset  T-Time & Date


Set Real Time Clock.  Enter time as
YYYY-MM-DD HH:MM:SS
2371-05-04 08:59:50
2012-09-01 13:00:00
2012-09-01 13:00:00


Control BASIC v0.59.
R-Run    S-Step    L-List   E-Edit   C-Configure   B-Bank   D-Download    K-Reset  T-Time & Date
```

If you **press the T-key** again you will see that the time has advanced.

The Real Time Clock can be powered by connecting a Lithium coin cell to pin 38 of the EzSBC1 and it will keep time while the main power is off. There is an automatic power changeover circuit included on the EzSBC1.

Resetting the EzSBC1

There are two ways of resetting the EzSBC1. Pressing the reset button on the EzSBC1 performs a hardware reset on the module and the RSTn pin on the module will be driven low to reset external device which are tied to the pin. The USB-bus does not enumerate again when the reset button is pressed.

By pressing K at the prompt causes a hardware reset of the CPU on the EzSBC1. The USB bus is not reset allowing the terminal connection to be maintained. Also, the Reset pin (RSTn) on the controller board does not toggle when the Reset command is used. Some BASIC keywords have effects even after the program ends. A good example of such an instruction is the PWM command. Once the PWM command is given, the pin will be driven even after the program ends. The Reset command is useful to restore the IO-pins to their power on state without having to reset the entire system.

The RSTn pin has a pull-up resistor on the EzSBC1 and can be driven low by external components to reset the entire system. The reset controller used on the EzSBC1 will hold the reset pin low for a time (30ms) after the reset pulse disappears to ensure a valid reset condition for the EzSBC1. External components should not drive the RSTn pin high.

Revisions

1.07

25 January 2014

Corrected the syntax for the SERVO command by removing the brackets from the Manual.

14 December 2014

1.06

Applies to Firmware 0.79 and above.

The PULSIN function was modified to provide more consistent readings.

Applies to Firmware 0.78 and above.

The SERBNG instruction had a bug fixed and the syntax is different. It no longer returns a meaningless error code. Since it does not return a value, the brackets around the parameters are not required.

18 November 2013

1.05

Applies to Firmware 0.76 and above.

Added I2CWR16 and I2CRD16$ instructions.

17 June 2013

1.04

Applies to Firmware 0.75 and above.

Added ATAN2 function for calculating headings.

Added SIN, COS, TAN, ACOS, ASIN, ATAN, ATAN2 to the keywords.

15 May 2013

1.03

Added a section on Strings and related keywords.
Added AND and OR to the keyword list.
Corrected the description of INADC.
Added a hyperlink to OUTDAC and expanded the description of the command.
Added description of STR$ function.
New instructions DHTTEMP, DHTHUM, SHIFTIN and SHIFTOUT.
Corrected the Pin Diagram on p9.
Added more examples.

26 Feb 2013:

1.02

Changed Analog to Digital to Digital to Analog on p10 describing the DAC output pin.
Moved some text on page 46 to the correct place showing the effect of pressing Ctrl-W.

1 Dec 2012:

1.01

Removed a duplicated paragraph in the Arithmetic section on p18.
Fixed grammar on a few pages.
Added the Revisions section (these pages).