CFSound-IV

BASIC Programming

7 May 2018

Table of Contents

Welcome Newbies!

If you don't know anything about programming computers, relax – we will try and make it simple. Using this manual as a guide, you'll be able to interact and use your Arbitrary Precision CFSound-IV right away. So sit down and spend a couple of hours with this manual and the display. Interact with it. Get comfortable with them. Once you learn how to program, the sky is the limit.

… And Hello Programmers

If you already know how to program, then turn to the BASIC Reference and BASIC Examples sections toward the end of this manual. There are summaries of the BASIC commands, statements, functions, operators and other programming elements that are supported.

To Get Started…

In order to start using the CFSound-IV you will need a power source and optionally a communications device. The power source provides the power that the display requires for its operation. The communications device allows you to interact with the built-in BASIC language to develop and test your applications.

Powering the CFSound-IV

The CFSound-IV can be powered in four different ways:

The CFSound requires a source of power to operate. There are four ways to power the CFSound:

1. Using a couple of pins on the MAIN connector to connect to an external power supply.
2. Using a couple of pins on the SERIAL connector to connect an external power supply.
3. Using a USB connection to supply power if USB power option is factory-installed.
4. Using the optional factory-installed Power Over Ethernet (POE) module and an external Ethernet power injection module.

Let's look at each of these options in more detail.

Via MAIN Connector to connect to an external power supply

The CFSound-IV can be powered through two pins on its MAIN connector.

Via SERIAL Connector with Arbitrary Precision Power Injector

This consists of a back-to-back pair of DB-9 serial connectors with a wall transformer. The CFSound-IV can be powered through two pins on its SERIAL connector. This injector supplies that power while allowing the other serial connector pins to pass through to your application cabling and hardware.

Arbitrary Precision Power Injector

Back to Table of Contents

Via USB DEVICE Connector with USB Power Option

This requires the optional USB Power Option to be factory-installed on the CFSound-IV. This consists of a USB A to Micro B cable that connects the CFSound-IV's DEVICE connector to your PC.

USB A to Micro B Cable

Via ETHERNET Connector with Power Over Ethernet Option

This requires the optional factory-installed POE module be installed on the CFSound-IV and an IEEE802.3af Ethernet power injector cabled to the ETHERNET jack on the CFSound.

PWR128RA 19W Power Over Ethernet Injector

The connection supports Power Over Ethernet (POE) if the optional POE module is installed on the controller and can be used to power the CFSound with remote power injection. The POE support is IEEE802.af compliant and provides a Class 0 signature. Both DC power on Spares (mode B) and DC power on Data (mode A) operation is supported:

ETHERNET Pin #	POE DC Power on Spares		POE DC on Data	
	MDI Signal	MDIX Signal	MDI Signal	MDIX Signal
1	TX+	RX+	TX+ PSE+	RX+ PSE+
2	TX-	RX-	TX- PSE+	RX- PSE+
3	RX+	TX+	RX+ PSE-	TX+ PSE-
4	PSE+	PSE+		
5	PSE+	PSE+		
6	RX-	TX-	RX- PSE-	TX- PSE-
7	PSE-	PSE-		
8	PSE-	PSE-		

More information about powering the CFSound can be found in the CFSound-IV User's Manual.

Communicating with the CFSound-IV

There are several options for communicating with the CFSound-IV. These are listed in order of decreasing preference with the last two somewhat limiting your ability to program the CFSound.

Serial Connection via RS-232

This requires either a stand-alone ANSI terminal device, or a PC running an ANSI terminal emulator program. The connection is made between the CFSound-IV and the terminal device using a cable between the SERIAL connector and the communications port on the terminal or PC. The PC communication port

Back to Table of Contents

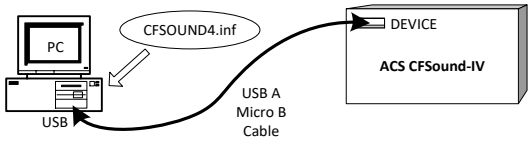
can be built-in or provided with an external USB Serial Adaptor. ***(See the CFSound-IV User's Manual appendix Wiring Harness Diagram)***

Serial Connection via USB

This requires a PC running an ANSI terminal emulator program. The CFSound-IV can be configured to use the USB as a serial communications device. A Micro B USB DEVICE connector is provided and the CFSound can be connected to a PC with a USB A to Micro B cable.

A **CFSound4.inf** file is available that identifies the CFSound-IV as a virtual serial port device that implements the Communications Device Class. (A Linux USB CDC configuration file is also available)

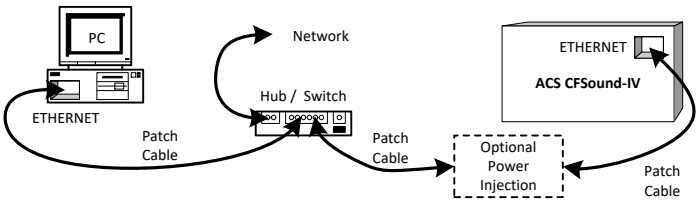
Connect the **USB A to Micro B cable** to the CFSound-IV **DEVICE** connector. Connect the other end of the cable into a USB port on your PC.



Windows will indicate that it has found new hardware and will eventually prompt for the location of a driver for the CFSound-IV. Browse to the location of the **CFSOUND4.inf** file that you have downloaded and select it. Windows should now finish installing the new hardware and it should be ready to use. The COM port identifier that Windows will assign to the CFSound-IV will depend upon what other communications devices are present in the system and can be determined using the Device Manager.

TCP/IP Raw Connection via Ethernet

This requires a PC running an ANSI terminal emulator program that is capable of communicating using TCP/IP Raw sockets – Tera Term, Hyperterminal or the PuTTY program are examples. The CFSound can be connected as an Ethernet device. A standard RJ-45 connector is provided and it can be connected to a network with a standard Ethernet cable – either straight or crossover, detection and correction is automatic via HP Auto MDI/MDI-X configuration. The network speed can be either 10 or 100 mbps with auto link negotiation. A link activity indicator is provided on the ETHERNET jack.



The CFSound supports a configurable MAC address and configurable static IP address and IP mask. Communication is performed using TCP/IP Raw Sockets with a configurable port address.

Terminal Emulator Programs

Your PC may have a terminal emulator program available. Windows XP systems have a program called Hyperterminal installed. It should be located at:

Start, All Programs, Accessories, Communications, Hyperterminal.

A slightly better version of this program, Hyperterminal PE (Private Edition), is also available for purchase from Hilgraeve:

Hyperterm PE: http://www.hilgraeve.com/hyperterminal/

Back to Table of Contents

The Hyperterminal PE edition can communicate with the CFSound-IV using your PC's serial or USB port or with TCP/IP Raw Sockets.

The following terminal emulator programs are also available for free download:

Tera Term: http://ttssh2.sourceforge.jp

PuTTY: http://www.chiark.greenend.org.uk/~sgtatham/putty/

Configuring the CFSound-IV

To perform the exercises outlined in this manual, the CFSound-IV must be started in BASIC mode. This mode of operation is achieved by powering the unit without a SD card or by inserting a SD card with no CFSound named .WAV files present.

If you're communicating using the CFSound SERIAL port, you should configure your terminal emulator with serial settings that matches your CFSound configuration. The CFSound defaults to 9600 baud, 8 data bits, 1 stop bit and no parity. Start Hyperterminal, Tera Term VT, PuTTY or other terminal emulator program on your connected computer (PC). Establish a connection to the connected serial port. Power-up or reset the CFSound-IV.

If you've configured the CFSound USB port to act as a serial communication device the baud rate does not matter. You must have installed the CFSound4.inf driver when the unit was first connected to the PC. Powering or resetting the CFSound-IV will disconnect and reconnect the USB requiring a re-establishing of the connection in your terminal emulator.

If you're communicating using the Ethernet and TCP/IP Raw Sockets you should configure the terminal emulator connection to use the IP address and port number that matches your CFSound configuration. The CFSound defaults to an IP address of 192.168.1.200 with a raw socket port of 23. Powering or resetting the CFSound-IV will disconnect from the terminal emulator requiring a re-establishing of the connection.

When connected with the SERIAL port this sign-on message should appear on the PC's terminal emulator program:

```
CFSound-IV #0 v1.2.11 on Mar 30 2015 12:50:47
DMX I/O via Art-Net

Scanning card directory for .WAV files...
Basic v3.1.5 Mar 30 2015 12:50:40
Ready
```

(the v1.2.11, v3.1.5 and date/time specify which version CFSound and BASIC firmware that the CFSound-IV is running)

When connected via USB or Ethernet you can press the 

key to receive BASIC's Ready prompt.

If you don't receive the signon message or Ready prompt, check your power supply and wiring. Try turning the CFSound off and on again and re-establishing the connection.

Once you see the message, press the 

key. You should see an additional Ready prompt.

Once you see this message on your PC and BASIC receives your key presses you're ready to begin.

Back to Table of Contents

What the CFSound-IV Understands

In this beginners section of the manual you will learn how to talk to the CFSound-IV. In Computer Terminology this is referred to as ***programming***. Once you learn how to program you can get your CFSound to do whatever you tell it to do – usually.

The CFSound-IV understands a language called BASIC. BASIC is a customized form of the "Beginners All-purpose Symbolic Instruction Code". The original BASIC was developed back in 1964 to provide computer access to people who didn't usually program. This version provides additional commands and features that make it useful to program an interactive CFSound.

Talking to the CFSound-IV

Try pressing the Enter key on your PC's keyboard. The CFSound-IV indicates that it is awaiting instruction by responding with Ready. Ready is the BASIC language prompt – it is waiting for a command from you.

Try typing the following as your first command – type this exactly as it is shown:

```
PRINT "Hello World"
```

When you reach the end of the line, review it for mistakes. Did you put the quotation marks where they were shown? If you made a mistake, simply press the backspace key and the last character that you typed will disappear. You can backspace over the entire line if necessary.

Your PC (and display screen) should look like this:

```
Ready
PRINT "Hello World!"
```

Now press the Enter key and see what happens. Your screen should now look like this:

```
Ready
PRINT "Hello World!"
Hello World!
Ready
```

The CFSound-IV executed the command that you entered by printing the message that you had in quotes.

Now let's try another command:

```
PRINT "2 + 2"  Enter
```

The CFSound-IV executes your command by printing:

```
2 + 2
Ready
```

If you expected to see the number four, then try removing the quotation marks:

```
PRINT 2 + 2  Enter
```

The CFSound-IV executes your command by printing:

```
4
Ready
```

The CFSound-IV sees everything that you type as either Strings or Numbers. If it's in quotes, it's a String and the CFSound sees it exactly as it was typed. If it's not in quotes it's a Number. The CFSound-IV

Back to Table of Contents

will figure it out like a numerical problem. It it's not in quotes the CFSound can interpret it by adding, subtracting, multiplying or dividing it.

Let's try a multiplication problem:

```
PRINT 1589 * 2 Enter
```

```
PRINT 1589 * 2
3178
Ready
```

The CFSound-IV uses an asterisk as a multiplication sign rather than an X since the X character is alphabetical and can be part of a name as we will see later.

There Are Rules…

The CFSound-IV is very literal. If it doesn't understand what you have typed it will produce an error message to let you know. Try typing this line deliberately misspelling the word **PRINT**:

```
PRIINT "HI" Enter
```

The CFSound-IV prints:

```
PRIINT "HI"
Illegal program command error
Ready
```

The CFSound doesn't understand what you have typed. The "Illegal program command error" message indicates that the command "PRIINT" is not one that it knows how to do.

Try leaving off the last quotation mark. Type:

```
PRINT "HI Enter
```

The CFSound-IV prints:

```
PRINT "HI
Mis-matched quotes error
Ready
```

The CFSound-IV will also give you error messages when it does understand what you have typed, but the command will result in incorrect operation. For instance, try typing:

```
PRINT 5 / 0 Enter
```

The CFSound prints:

```
PRINT 5 / 0
Divide by zero error
Ready
```

This error message indicates that you're asking it to divide by zero – which is not possible.

Back to Table of Contents

Remembering Numbers and Strings

One of the features of the CFSound-IV is the ability to remember things you ask it to. For example, to make the CFSound remember the number 13, type this:

```
A = 13 Enter
```

Now try the other **PRINT** commands that you did before. To see if the CFSound still remembers what A is equal to, type:

```
PRINT A Enter
```

And the CFSound should print:

```
PRINT A
13
Ready
```

The CFSound-IV remembers that the placeholder **A** has the number value of 13 until you turn it off or change it. Type:

```
A = 17 Enter
```

```
A = 17
Ready
PRINT A
17
Ready
```

So what happened? When you first set **A** equal to 13 the CFSound created a named memory location to hold that value. You can refer to that location by the name **A** to retrieve the current value of the memory location as with the **PRINT A** command above. You can also modify the value of the named memory location by setting it to a new value. You can use combinations of letters and numbers to name these memory locations – the only requirement is that the name must not start with a number. Try typing these commands:

```
B = 15 Enter

C2 = 20 Enter

LongName = 25 Enter
```

Now ask the CFSound-IV to retrieve all of these numbers by name. Type:

```
PRINT A, B, C2, LongName Enter
```

And the CFSound should show:

```
PRINT A, B, C2, LongName
17 15 20 25
Ready
```

To get the CFSound-IV to remember strings of letters or numbers, put a dollar sign at the end of the name. Type:

```
A$ = "Remember" Enter

B$ = "this for me" Enter
```

And then ask the CFSound to CFSound them:

```
PRINT A$, B$
Remember this for me
Ready
```


Back to Table of Contents

In Computer Terminology these named memory locations are referred to as ***variables***. The CFSound-IV keeps track of these variables and their current values for you. You can ask it for a list of what variables it is currently remembering by typing the command:

VARS Enter

And the CFSound will show the variables that we've used so far along with their current contents:

```
VARS
A                               R/W Int         = 17
B                               R/W Int         = 15
C2                              R/W Int         = 20
LongName                        R/W Int         = 25
A$                              R/W Str$        = "Remember "
B$                              R/W Str$        = "this for me"
Ready
```

The **R/W** indicates that the variable can be both **R**ead from and **W**ritten to. Later on we will see how to make variables **R**ead **O**nly. The **Int** indicates that the CFSound understands this variable to be for holding Numbers, **Str$** indicates that the variable holds strings.

And of course the CFSound-IV is picky about what is stored in the two types of variables. Try typing these lines:

D = "6" Enter

D$ = 6 Enter

With both of these lines the CFSound responds with an error message:

```
D = "6"
Wrong expression type error - can't assign string to numeric var
Ready
D$ = 6
Wrong expression type error - can't assign number to string var
Ready
```

In Computer Terminology, setting variables to values is referred to as ***assigning a value to a variable***.

There are four fundamental rules for variable value assignment:

Rules for Numeric Data

1. Numbers not in quotes are Numeric Data
2. Numeric Data can only be assigned to variables named without a trailing dollar sign

Rules for String Data

1. Any data in quotes is String Data
2. String Data may only be assigned to variable name with a trailing dollar sign

Variable Rules

1. You may use multiple characters from the upper case letters A-Z, the lower-case letters a-z, the numbers 0-9 and the '_' underscore character for variable names.
2. The first character of the name must be a letter, not a number or underscore.
3. Variable names are case-sensitive: **Aname** is not the same as **aNAME**.
4. Variables whose names end with a dollar sign can only hold String Data, otherwise they can only hold Numeric Data.
5. The list of current variables can be shown with the **VARS** command.
6. Short, concise variable names take less memory and work slightly faster.

Back to Table of Contents

Remembering Commands

First we need to erase everything that the CFSound-IV has remembered so far. Type:

NEW Enter

Now type this command line: Be sure and type the number 10 first:

10 PRINT "Hello from the CFSound-IV" Enter

Notice that this time, when you pressed Enter nothing appeared to have happened. Not that you can immediately see. What you did was to type your first program. Type:

RUN Enter

The CFSound now runs your program. You can type **RUN** again and again. You can also type run – BASIC's commands are not case-sensitive, just variable names:

```
10 PRINT "Hello from the CFSound-IV"
RUN
Hello from the CFSound-IV
Ready
run
Hello from the CFSound-IV
Ready
```

So the number at the beginning of the line tells BASIC to store the line instead of executing it. Let's add another line to the program. The number at the beginning of the line is the "line number". Type:

20 print "What's your name?" Enter

Now let's ask the CFSound-IV to show us the entire program. Type:

LIST Enter

The CFSound LISTs your entire program so far – notice how the lower-case print command was capitalized and how the lines are stored and shown in line number order:

```
LIST
10 PRINT "Hello from the CFSound-IV"
20 PRINT "What's your name?"
Ready
```

Now run the program: Type: **RUN** Enter The CFSound-IV prints:

```
RUN
Hello from the CFSound-IV
What's your name?
Ready
```

Try answering the question by typing your name and pressing Enter . . . there's that Illegal program command error again – the CFSound didn't understand what you meant when you typed in your name:

```
RUN
Hello from the CFSound-IV
What's your name?
Ready
Steve
Illegal program command error
Ready
```

In fact you have to instruct the CFSound-IV to accept your answer by giving it a command, **INPUT**, to do so. Add this line to the program:

30 INPUT Name$ Enter

Back to Table of Contents

The **INPUT** command tells the CFSound to stop and wait for you to type something which it will assign to the variable Name$. Add one more line to the program to show the Name that was entered:

```
40 PRINT "Hi, ", Name$ [Enter]
```

Now **LIST** the program. It should look like:

```
LIST
10 PRINT "Hello from the CFSound-IV"
20 PRINT "What's your name?"
30 INPUT Name$
40 PRINT "Hi,", Name$
Ready
```

Now let's **RUN** the program:

```
RUN
Hello from the CFSound-IV
What's your name?
? Steve
Hi, Steve
Ready
```

You can run the program many times, answering the question with different names – the CFSound doesn't care what name you use. After each **RUN** the variable Name$ holds the last name you entered.

You can make your program run over and over without having to type the **RUN** command each time. Add this line to the program:

```
50 GOTO 10 [Enter]
```

Now **RUN** the program. It runs over and over without stopping. The **GOTO** command told the CFSound-IV to go back up to line 10. Your program will repeat over and over because every time it executes line 50 it jumps back to line 10. In Computer Terminology this is referred to as a ***loop***. The only way to stop this endless loop is to press the [esc] key twice in a row followed by the [Enter] key. This is known as ESCaping the program and the CFSound tells you that you escaped:

```
Hello from the CFSound-IV
What's your name?
? [esc] [esc] [Enter]
 ESC at line 30
Ready
```

In this program the [Enter] key was required following the double [esc] key because the CFSound-IV was waiting for the [Enter] key at the **INPUT** command. If the CFSound is not waiting for **INPUT** the double [esc] key is sufficient to ESCape the program.

Back to Table of Contents

Changing your Program

So how can you modify your program without typing **NEW** and starting over each time? To replace an existing program line simply type the new line using the same line number at the beginning of the line you want to replace. Type:

```
50 GOTO 40 Enter
```

Your program should now look like:

```
50 GOTO 40
LIST
10 PRINT "Hello from the CFSound-IV"
20 PRINT "What's your name?"
30 INPUT Name$
40 PRINT "Hi,", Name$
50 GOTO 40
Ready
```

This program change modifies the loop to not ask the question over and over. Instead the program now just keeps repeatedly PRINTing the Name$ that you **INPUT** the first time. Press the esc key twice in a row to ESCape the program when you've seen enough. Notice that you don't need to press the Enter key to ESCape this time because the CFSound is not waiting for **INPUT**:

```
RUN
Hello from the CFSound-IV
What's your name?
? Steve
Hi, Steve
Hi, Steve
Hi, Steve
Hi, Steve
Hi, Steve
Hi, Steve
Hi, Steve
Hi, Steve
Hi, Steve
Hi, Steve esc esc
 ESC at line 40
Ready
```

To remove a line from your program simply type the line number followed by the Enter key. This erases that line from your program. Type:

```
50 Enter
```

This removes line 50 from your program:

```
50
LIST
10 PRINT "Hello from the CFSound-IV"
20 PRINT "What's your name?"
30 INPUT Name$
40 PRINT "Hi,", Name$
Ready
```

Put line 50 back into your program. Type:

```
50 GOTO 40 Enter
```

Now let's change the way that the **PRINT** shows your name. Replace line 40 in your program by typing it again, but add a semicolon at the end:

```
40 PRINT "Hi,", Name$; Enter
```

Now **RUN** the program. Notice how the trailing semicolon crams everything together?

Back to Table of Contents

```
40 PRINT "Hi, ", Name$,
LIST
10 PRINT "Hello from the CFSound-IV"
20 PRINT "What's your name?"
30 INPUT Name$
40 PRINT "Hi, ", Name$,
50 GOTO 40
Ready
run
Hello from the CFSound-IV
What's your name?
? Steve
Hi, SteveHi, SteveHi, SteveHi, SteveHi, SteveHi, SteveHi, SteveHi, SteveHi, SteveHi, SteveHi,
SteveHi,  SteveHi,  SteveHi,  SteveHi,  SteveHi,  SteveHi,  SteveHi,  SteveHi,  SteveHi,  SteveHi,
SteveHi,  SteveHi,  SteveHi,  SteveHi,  SteveHi,  SteveHi,  SteveHi,  SteveHi,  SteveHi,  SteveHi,
SteveHi,  SteveHi,  SteveHi,  SteveHi,  SteveHi,  SteveHi,  SteveHi,  SteveHi,  SteveHi,  SteveHi,
SteveHi,  SteveHi,  SteveHi,  SteveHi,  SteveHi,  SteveHi,  SteveHi,  SteveHi,  SteveHi,  SteveHi,
SteveHi,  SteveHi,  SteveHi,  SteveHi,  SteveHi,  SteveHi,  SteveHi,  SteveHi,  SteveHi,  SteveHi,
SteveHi, Ste ESC at line 40
Ready
```

Controlling the Program Execution

You've seen that a program can be **RUN** from lowest line number to highest. The program can also loop using the **GOTO** command. And a running program may be ESCaped to stop.

Another way to control the execution of a program is to **GOTO** only if a certain condition is met. Let's change our program so that it stops if we enter a blank name. Modify line 40 to remove the trailing comma by typing:

```
40 PRINT “Hi,“, Name$ [Enter]
```

Now replace line 50 with a new **IF / THEN** command. The new **IF / THEN** command evaluates a condition, and, **IF** the condition is True, **THEN** it executes the command after the **THEN**. Otherwise execution continues with the following program line. Type:

```
50 IF Name$ <> “” THEN GOTO 20 [Enter]
```

In this example the **IF** condition that is being evaluated is: **Name$ < > “”**. Name$ is the variable that receives your **INPUT** up until the [Enter] key. The < > (less than, greater than) stands for "not equal to", and the "" is an empty string – a string with no characters in it. So your program should now look like:

```
LIST
10 PRINT "Hello from the CFSound-IV"
20 PRINT "What's your name?"
30 INPUT Name$
40 PRINT "Hi,", Name$
50 IF Name$ <> "" THEN GOTO 20
Ready
```

Run the program. Try entering your name the first time your program asks for it, then the next time simply press the [Enter] key. Notice how the program loops back to line 20 when your name was **INPUT** otherwise the program stops:

```
RUN
Hello from the CFSound-IV
What's your name?
? Steve [Enter]
Hi, Steve
What's your name?
? [Enter]
Hi,
Ready
```

When you press the Enter key in response to the **INPUT** command without typing any characters first, the **INPUT** variable receives an empty string otherwise the variable holds any characters that were typed in before the Enter key.

You can think of line 50 as reading: IF variable Name$ not equal to empty string **THEN GOTO** line 20. And that is exactly what the program is doing. However there is an error. In Computer Terminology this is called a ***program bug***. Notice how the program still prints Hi, before it stops when nothing is ENTERed. This is because the program executes line 40 before it checks for Name$ being empty. In order to fix this 'bug' we have to check for the stop condition before we **PRINT** the Name$.

Replace line 40 with the new **IF / THEN** check. Type:

```
40 IF Name$ = "" THEN STOP Enter
```

And then replace line 50 with the **PRINT** command. Type:

```
50 PRINT "Hi,", Name$ Enter
```

The program should now look like:

```
LIST
10 PRINT "Hello from the CFSound-IV"
20 PRINT "What's your name?"
30 INPUT Name$
40 IF Name$ = "" THEN STOP
50 PRINT "Hi,", Name$
60 GOTO 20
Ready
RUN
Hello from the CFSound-IV
What's your name?
? Steve
Hi, Steve
What's your name?
?
STOP in line 40
Ready
```

The new **STOP** command does exactly that – it stops the program from running and shows you what line the program stopped on. You can also use the **END** command which stops without the message. Try it by changing line 40.

Program Rules

1. A program consists of one or more command lines that begin with a line number.
2. Commands are not case-sensitive but are converted by BASIC. Variables are case-sensitive.
3. Program lines and variables may be cleared from memory by the **NEW** command.
4. Program lines are kept in ascending numeric order by the CFSound.
5. A program is **RUN** to execute the lines in numeric order starting with the lowest numbered line.
6. Program lines may be shown with the **LIST** command.
7. The execution order of program lines may be changed by the **GOTO** command.
8. A program which is executing repeatedly in a loop may be ESCaped by pressing the esc key twice in a row.
9. A program line may be replaced by typing a new program line with the same line number.
10. A program line may be deleted by typing the line number only followed by Enter.
11. Program line execution may be conditioned by the **IF / THEN** command.
12. Program execution may be stopped with the **STOP** or **END** commands.

Back to Table of Contents

Learning How to Count

Most programs require the ability to count – lines, key presses, characters, loops or other things. With the commands that you already know you can write a program that counts.

First we need to learn a fundamental concept. In order to add to a variable you can use it on each side of the assignment equals sign. For example, type:

```
A = 1 Enter
PRINT A Enter
A = A + 1 Enter
PRINT A Enter
```

With assignment **A = A + 1**, the current value of **A** has one added to it then it becomes the new value of **A**. The value to add doesn't have to be 1, it can also be the value of another variable. You can read this statement as "Variable **A** equals the value of Variable **A** plus 1".

So in order to count from 1 to 10 we can write the following short program. Start fresh by typing:

```
NEW Enter
```

First we set the value of the counter, **A**, equal to 1. Type:

```
10 A = 1 Enter
```

Then we print it. Type:

```
20 PRINT A Enter
```

Then we add one to it. Type:

```
30 A = A + 1 Enter
```

Then we check to see if A is less than or equal to 10… and, if so, we loop back to print the new value. Otherwise we're done counting:

```
40 IF A <= 10 THEN GOTO 20 Enter
```

Now **LIST** the program. It should look like:

```
LIST
10 A = 1
20 PRINT A
30 A = A + 1
40 IF A <= 10 THEN GOTO 20
Ready
```

Now **RUN** it:

```
RUN
1
2
3
4
5
6
7
8
9
10
Ready
```

Simple – right? However, BASIC provides another way to count with fewer commands that's a little easier to read.

Back to Table of Contents

Start fresh by typing **NEW** Enter again. Now we will work with a new two command combination: **FOR / NEXT**.

The **FOR** command replaces two commands: line 10 - the setting of the initial count value, and line 40 – the check for the end count value. The **NEXT** command replaces two commands: line 30 – the advance of the count value, and the **GOTO** portion of line 40 – the counting loop. Let's see how this works.

First the **FOR** command. This command determines the count variable, its initial value and the ending value. Type:

```
10 FOR A = 1 TO 10 Enter
```

Now as before we **PRINT** the count value. Type:

```
20 PRINT A Enter
```

And finally the **NEXT** command. This command advances the count value and executes the loop if more counting is necessary. This is referred to as closing the loop. Type:

```
30 NEXT A Enter
```

Now **LIST** the program. It should look like:

```
LIST
10 FOR A = 1 TO 10
20 PRINT A
30 NEXT A
Ready
```

Now **RUN** it:

```
RUN
1
2
3
4
5
6
7
8
9
10
Ready
```

This program is one line shorter, but does the same thing. It is also easier to read and comprehend. Try changing the initial and ending value to see the results.

Great! But what if we want to count by two – how do we do that. It seems like the A = A + 1 is implied.

The BASIC language designers thought of that and have provided another keyword that can be added to the end of the **FOR** command to specify the value to be added each time around the loop – **STEP**.

The **STEP** keyword goes at the end of the **FOR** command – after the ending value. Let's replace line 10 with this new **FOR** command that has a **STEP 2** on the end:

```
10 FOR A = 1 TO 10 STEP 2 Enter
```

Your program should now look like:

```
LIST
10 FOR A = 1 TO 10 STEP 2
20 PRINT A
30 NEXT A
Ready
```

Now **RUN** it:

```
RUN
1
3
```


Back to Table of Contents

```
5
7
9
Ready
```

The initial value, ending value and optional **STEP** value don't have to be numbers – they can also be numeric variables. And the **STEP** value can be negative to. Let's replace line 10 again to try this. Type:

10 FOR A = 10 TO 1 STEP -1 Enter

Your program should now look like:

```
LIST
10 FOR A = 10 TO 1 STEP -1
20 PRINT A
30 NEXT A
Ready
```

and when you **RUN** it:

```
RUN
10
9
8
7
6
5
4
3
2
1
Ready
```

There's a few more important things to know about **FOR / NEXT**. First, you shouldn't try to **GOTO** into the middle of a **FOR / NEXT** loop. When the CFSound-IV encounters the **NEXT** command without having seen the matching **FOR** first it gets confused and gives you an error. Here's an example of that error – we add a line at the beginning of the program to **GOTO** line 20 which is inside the **FOR / NEXT** loop:

```
LIST
5 GOTO 20
10 FOR A = 10 TO 1 STEP -1
20 PRINT A
30 NEXT A
Ready
RUN
 0
Nesting error in line 30 - NEXT without preceding FOR
```

The second thing to remember is that if you nest **FOR / NEXT** loops – one inside of another, you must close the inner loop before closing the outer loop. Here's the right way to nest loops:

```
LIST
10 FOR X = 1 TO 3
20 FOR Y = 3 TO 1 STEP -1
30 PRINT "X = ",X, " Y = ",Y
40 NEXT Y
50 NEXT X
Ready
RUN
X =  1 Y =  3
X =  1 Y =  2
X =  1 Y =  1
X =  2 Y =  3
X =  2 Y =  2
X =  2 Y =  1
X =  3 Y =  3
X =  3 Y =  2
X =  3 Y =  1
Ready
```

And here's the wrong way to nest loops:

Back to Table of Contents

```
LIST
10 FOR X = 1 TO 3
20 FOR Y = 3 TO 1 STEP -1
30 PRINT "X = ",X, " Y = ",Y
40 NEXT X
50 NEXT Y
Ready
RUN
X =  1 Y =  3
Nesting error in line 40 - NEXT var doesn't match FOR var
Ready
```

The third thing to remember is that you shouldn't try to **GOTO** out of a **FOR / NEXT** loop. While you won't receive an immediate error, the cleanup performed by the **NEXT** statement isn't performed and if you do it enough times you will get an error when there isn't sufficient memory left for the CFSound to remember more **FOR / NEXT** commands. Exiting a **FOR / NEXT** loop early is a fairly common programming requirement however so BASIC provides a system form of **GOTO** to jump out – **BREAK**. To try it, modify our existing program. First remove line 5 by typing:

```
5 Enter
```

And then add line 40 to indicate that we're done counting and provide a target line for the **BREAK** command:

```
40 PRINT "Done" Enter
```

And finally add a condition check for exiting the **FOR / NEXT** loop – let's say when the count variable is equal to 5 we want to leave the loop. Type:

```
15 IF A = 5 THEN BREAK Enter
```

Your program should now look like:

```
LIST
10 FOR A = 10 TO 1 STEP -1
15 IF A = 5 THEN BREAK
20 PRINT A
30 NEXT A
40 PRINT "Done"
Ready
```

And when you **RUN** it:

```
RUN
10
9
8
7
6
Done
Ready
```

Counting Rules

1. A variable can appear on both sides of the assignment equal sign. The variable on the right hand side refers to its existing value and the left hand side variable will receive the new value.
2. You can implement a count sequence by assigning an initial value to a variable, advancing it by adding or subtracting to it then checking if it is at the ending value and looping back if it is not.
3. BASIC provides a **FOR / NEXT** command with an optional **STEP** clause to simplify counting.
4. Nested **FOR / NEXT** loops must be closed from the inside out – inside loops first.
5. You can't jump into the middle of a **FOR / NEXT** loop – an error will result on the **NEXT** command.
6. You should only jump out of a **FOR / NEXT** loop using the system **BREAK** command.

Back to Table of Contents

Remembering your Programs

The CFSound-IV wouldn't be very useful if you always had to type in your programs every time you turned it on. The longer your programs are the more of a problem this would become.

The CFSound-IV has a slot for a SD memory card. You can **SAVE** your programs by name on the card and **LOAD** or **RUN** them later by referring to the same name.

Think of the SD card as a file drawer that holds many files. You can create a new file and put it in the drawer, or retrieve an existing file from the drawer. You can't retrieve a file that hasn't been created first. The CFSound-IV will keep a directory listing of what files are in the drawer (on the SD card).

Let's try it. Put a fresh card into the slot and ask for a directory of files. Type:

```
DIR Enter
```

On a fresh card you should see the following:

```
DIR
-----------------------------------------------
                                          0 files
                                          0 directories
Ready
```

The CFSound is showing you that there are currently no files on the card. Now type in one of your earlier programs – remember to start with **NEW** Enter. Let's start with the simple **FOR / NEXT** program:

```
NEW
10 FOR A = 1 TO 10
20 PRINT A
30 NEXT A
```

You can **RUN** it to verify that it works as before. Now let's **SAVE** it in a file. Let's call it **FORNEXT**. Type:

```
SAVE FORNEXT Enter
```

Did it save it? Let's ask for a directory again:

```
DIR
FORNEXT.BAS                              43 A      02-26-2012 04:57:02 PM
-----------------------------------------------
                                          1 files
                                          0 directories
Ready
```

There it is. The CFSound is telling us that it's 43 characters long, it was created on 2-26-2012 at 4:57PM and that there is one file on the card. What's the .BAS on the end of the file name? There can be many types of files in the drawer (on the card) and this is BASIC's way of telling us that this file is a BASIC program. Great! Now let's try to get it back.

First clear out the program and variables with **NEW**. Try a **LIST** and **VARS** to see that there's actually nothing in memory:

```
NEW
Ready
LIST
Ready
VARS
Ready
```

Now let's retrieve it. Type:

```
LOAD FORNEXT Enter
```

and then **LIST** it:

Back to Table of Contents

```
LOAD FORNEXT
Ready
LIST
10 FOR A = 1 TO 10
20 PRINT A
30 NEXT A
Ready
```

and there it is. You can **RUN** it to verify that it still works.

Modify it to count backwards by changing line 10 and adding the **STEP** keyword:

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

You can **LIST** and **RUN** it. Then let's **SAVE** this as **FORNEXTSTEP**. Type:

```
SAVE FORNEXTSTEP
```

Now let's ask for a directory again:

```
DIR
FORNEXT.BAS                                43 A      02-26-2012 04:57:02 PM
FORNEXTSTEP.BAS                            51 A      02-26-2012 05:12:56 PM
------------------------------------------------
                                            2 files
                                            0 directories
Ready
```

Both programs (files) are there. BASIC provides a shortcut for loading and running a program file – just type **RUN** followed by the file name. BASIC will do the **NEW**, **LOAD** and **RUN** all in a single step:

```
RUN FORNEXT
1
2
3
4
5
6
7
8
9
10
Ready
RUN FORNEXTSTEP
10
9
8
7
6
5
4
3
2
1
Ready
```

Saving Rules

1. Programs can be filed onto a SD card installed in the CFSound-IV.
2. A directory of what files are on the card can be obtained by the **DIR** command.
3. Programs are saved to the card with the **SAVE** command.
4. Programs are retrieved from the card with the **LOAD** command.
5. Programs may be retrieved and executed with the **RUN** command.
6. BASIC remembers the last program file name used with **LOAD**, **RUN** or **SAVE** so a modified program can be saved into the same file using a **SAVE** command without a name.

Back to Table of Contents

Things to do with Numbers

So far we've seen that the CFSound-IV can add and multiply numbers – using numbers directly or numbers stored in variables. The CFSound can also subtract and divide numbers. In Computer Terminology these actions are referred to as ***operators***. Let's try a few:

```
PRINT 4 - 2
2
Ready
PRINT 9 / 3
3
Ready
```

Here's a new operator that you probably haven't heard of or used before – remainder of division. In Computer Terminology this is referred to as a ***modulo operation***. The first number is divided by the second number and the remainder is returned. So if two numbers divide evenly, the modulo is zero – no remainder:

```
PRINT 10 % 5
0
Ready
```

However if the two numbers don't divide evenly, the modulo operator returns the remainder after the division that would be performed. So if we divide 5 by 4, the remainder or modulo would be 1:

```
PRINT 5 % 4
1
Ready
```

Operators usually go between two numbers or numeric variables. However there are some operators that go in front of a single number or variable. In Computer Terminology these are referred to as ***unary operators***. There are a couple of these, but the most common is a leading minus sign – referred to as negate:

```
A = 10
Ready
PRINT -A
-10
Ready
```

This is just a shorthand way of telling the CFSound-IV to subtract the number from zero.

Only Whole Numbers Please

Now let's try dividing two numbers that don't divide evenly – say 10 divided by 3:

```
PRINT 10 / 3
3
Ready
```

What happened? Shouldn't 10 divided by 3 equal 3 and 1/3? It should but this shows a limitation of BASIC – it only knows how to work with whole numbers – not decimals or fractions. So when it divides 10 by 3 it returns the whole number part and discards the fractional part. In Computer Terminology this is referred to as ***integer arithmetic***. While this can be a limitation for solving some problems most applications of the CFSound-IV can be performed only using integers – and there are some workarounds that we'll examine later. Remember, the modulo operator can return the remainder of the same division.

The Size of Numbers

So if BASIC can only work with whole numbers, how big can they be? What's the limit? This BASIC works with what programmers refer to as integers – they are 32 computer bits wide. A computer bit can only be on or off – when you have 32 of them across, with each one able to be on or off, the biggest number that they can represent is +4,294,967,295. which is the number 2 raised to the 32^{nd} power (4,294,967,296) less one reserved for the zero = 4,294,967,295.

Back to Table of Contents

Actually the computer in the CFSound-IV reserves 1 bit for a plus or minus indicator – a sign bit. So there are really only 31 bits available for the actual number. So the maximum positive number is +2,147,483,647 and the maximum negative number is –2,147,483,648.

Comparing Numbers

There are also operators for comparing two numbers. Numbers can be checked to see if they're the same or not or if one is larger or smaller than another. In Computer Terminology these checks are called ***comparison operators*** – two numbers are compared with each other. The results of a comparison are either true or false. In BASIC, a true comparison results in a 1 and a false comparison results in a zero.

To check for equality (two numbers being the same) we use the equal operator – which is the equals sign:

```
PRINT 1 = 0
0
Ready
PRINT 2 = 2
1
Ready
```

So how do you test if two numbers are not equal? BASIC uses a less than followed immediately by a greater than (no space in-between) as the not equal operator. Notice how this check gives the opposite result of the equal check.

```
PRINT 1 <> 0
1
Ready
PRINT 2 <> 2
0
Ready
```

Of course we can also check for numbers being less than, less than or equal, greater than and greater than or equal:

```
PRINT 2 < 2
0
Ready
PRINT 2 <= 2
1
Ready
PRINT 4 > 4
0
Ready
PRINT 4 >= 4
1
Ready
```

Combining Numbers in Order

You don't have to only do a single operation on numbers at a time. You can combine them into a series of operations to achieve the result you want. In Computer Terminology this is referred to as an ***expression***.

Usually, when you perform a bunch of operations on numbers you do them in a left-to-right order. So if you want to add two numbers, and now double the result you might write:

```
PRINT 2 + 3 * 2
8
Ready
```

You would expect the result to be 2 plus 3 equals 5 times 2 equals 10 – right? How did the CFSound-IV come up with the number 8?

It turns out that in BASIC, like most computer languages there is an order in which numeric operations are performed. In Computer Terminology this is referred to as ***operator precedence*** or ***operator priority***. If there wasn't a defined order then you could get different results each time or perhaps different results

Back to Table of Contents

between different machines. Negation is performed first, then Multiplication, Division and Modulo, then Addition and Subtraction, then Comparisons are performed last. So the CFSound does the 3 times 2 first, then it adds the 2 which gives 8 instead of 10.

What if you really want these operations done in the order that you wrote them down? Well it turns out that you can specify the order by grouping operations together using parenthesis. The CFSound-IV BASIC will execute operations inside parenthesis first, then use that result to perform the next operation in the expression. So to get the result that you wanted above, you could write:

```
PRINT (2 + 3) * 2
10
Ready
```

So when in doubt, or when you really want things done in a certain order, use parenthesis to group operations together. The CFSound will evaluate things from the inside out starting with the most nested set of parenthesis first.

Numeric Operator Rules

1. Numbers can be added, subtracted, multiplied, divided, remaindered, negated and compared.
2. Multiple operations can be performed sequentially to form a numeric expression.
3. Multiple operations are performed in a priority fashion from first to last: negation, multiplication and division, addition and subtraction, then comparison.
4. Multiple operations with the same priority are performed left to right.
5. Parenthesis can be used to change the order in which numeric expressions are evaluated.

Back to Table of Contents

Things to do with Strings

There are also operations that can be performed on strings. The most common of these are to combine two separate strings into a new single string by 'adding' one string onto the end of another. In Computer Terminology this is referred to as ***concatenation*** – the two strings are concatenated together:

```
PRINT "HELP" + "ME"
HELPME
Ready
Part1$ = "Help"
Ready
Part2$ = "Me"
Ready
PRINT Part1$ + " " + Part2$
Help Me
Ready
```

Comparing Strings

Like numbers you can also compare strings. The two strings are compared, a character at a time to determine if they are the same, less than or greater than each other:

```
PRINT "ONE" = "TWO"
0
Ready
PRINT "ONE" <> "TWO"
1
Ready
PRINT "ONE" > "TWO"
0
Ready
PRINT "ONE" < "TWO"
1
Ready
```


Back to Table of Contents

Now for Something at Random

By now we've learned a few different commands that the CFSound-IV BASIC understands. Let's try something new. Type this line:

```
10 PRINT RND(10) Enter
```

And now **RUN** it. The CFSound printed a random number between 0 and 9. **RUN** it a few more times:

```
10 PRINT RND(10)
Ready
RUN
8
Ready
RUN
7
Ready
RUN
5
Ready
RUN
1
Ready
```

Now make the program loop back to run continuously. Add line 20 to the program and **RUN** it. You will have to ESCape the program to stop it:

```
20 GOTO 10
LIST
10 PRINT RND(10)
20 GOTO 10
Ready
RUN
7
8
6
9
8
0
0
7
4
9
9
 ESC at line 10
Ready
```

What if we want to have random number from 0 to 100? Changed line 10 to this and **RUN** it:

```
10 PRINT "", RND(100);
LIST
10 PRINT "", RND(100);
20 GOTO 10
Ready
RUN
 59 51 31 73 82 62 38 92 78 58 71 84 28 94 46 43 2 59 26 80 32 41 44 43 12 31 30 84 18 58 4 80
11 3 56 39 14 60 0 42 33 95 95 9 56 80 91 16 54 7 86 37 12 25 45 37 82 67 29 95 26 9 23 62 74 57
80 6 16 83 85 68 64 39 22 45 37 81 50 56 93 56 58 4 70 17 0 28 81 99 61 22 36 74 27 9 19 12 9 38
32 87 41 18 35 12 70 28 40 60 22 38 83 23 58 23 17 30 75 95 11 46 12 67 7 26 52 7 39 1 70 67 69
39 53 0 63 28 25 94 39 27 86 94 7 20 25 52 94 88 45 38 99 25 92 72 25 45 15 67 5 48 19 66 71 20
51 74 12 36 89 24 6 4 60 80 29 55 65 46 48 30 29 14 86 80 7 84 50 26 74 53 78 44 3 9 19 94 8 24
42 99 92 58 31 73 27 46 23 55 49 29 82 27 75 49 97 54 75 64 88 62 49 8 65 52 73 36 9 74 60 35 62
37 46 63 1 88 46 50 64 5 96 21 43 68 19 77 51 37 71 57 25 55 65 21 84 89 77 72 72 76 68 84 49 6
93 82 42 93 57 75 39 87 94 70 94 61 97 32 45 79 39 92 24 10 49 20 8 7 39 7 16 78 2 16 15 90 53
19 10 11 76 82 99 70 27 83 97 94 29 34 78 35 69 22 70 26 29 84 90 97 97 11 16 3 42 83 77 52 86
39 48 91 80 37  ESC at line 10
Ready
```


Back to Table of Contents

It's a Function!

So the operation of **RND( )** is to produce random numbers. It takes a numeric value between the parentheses, computes a random number based upon the number that it's given and then takes on the value of that random number. It's kind of like a command and kind of like a variable at the same time. In Computer Terminology these combination command / variables are referred to as ***functions***. Because they are part of the BASIC language and don't have to be rewritten each time you want to use them they are referred to as ***built-in functions***.

Since functions act like a variable you can use them wherever you would read a variable – like in a **PRINT** command, an **IF / THEN** conditional check or in an expression. The difference between a function and a variable is that you can't assign a value to a function – you can only pass arguments to it and then use the value that it assumes as a result.

In Computer Terminology this is referred to as ***calling a function that takes an argument and returns a value***. It's like a variable, but you don't assign values to it – instead you pass it one or more arguments between the parenthesis and it acts like a variable whose value changes based upon the arguments.

There are other functions that are very useful when writing BASIC programs. Some take no arguments but return a value – either numeric or string. Some take one or more numeric arguments and return a numeric value. Some take string arguments and return a numeric value. Some take both string and numeric arguments and return a value.

So how do you know what type of value a built-in function will return? Just like a BASIC variable – it's all in the name. If a built-in function returns a string value, the function name ends with a dollar sign – otherwise it returns a numeric value. This makes sense because functions can be used in place of variables.

Function Rules

1. Functions behave like read-only variables that can supply different values or behavior in your programs depending upon what they are pre-defined to do and what arguments that they may require to do it.
2. When calling a function, the function name must be immediately followed by the opening parenthesis surrounding the function's arguments.
3. The type of a function is identified by the name, just like a variable.

A Casual Remark

The **REM** command allows you to put notes (REMarks) into your programs. The REM command actually does nothing - when BASIC 'sees' the **REM** command it skips over the rest of the line. In Computer Terminology the use of the REM command is referred to as ***commenting your code***. It serves a couple of purposes – reminding yourself about what you were trying to do with this code, and documenting what you hope that it does as an aid for others. While it doesn't seem important now with these tiny programs, it will become increasingly important when the programs get larger and you have many more of them to keep track of. It's considered good practice.

```
10 REM This is a comment line
```


Back to Table of Contents

Let's examine some of these functions. We will group them by what type of value they will return – numeric or string.

Functions Returning a Number

You've already seen one of these type of functions – **RND( )**. Remember it took a numeric argument and returned a numeric value. Here are some others:

The **ASC( )** function takes a string argument and returns the numeric value of the first character. The numeric value is the decimal representation of the character's ASCII value – an agreed upon world-wide standard. (see http://www.asciitable.com) . Let's try it:

```
PRINT ASC("A")
65
Ready
PRINT ASC("0")
48
Ready
```

The **ABS( )** function takes a number argument and returns it's ABSolute value – if the numeric argument is positive or zero the same number is returned. If the numeric argument is negative the positive value of the number is returned. Let's try it:

```
PRINT ABS(2)
2
Ready
PRINT ABS(0)
0
Ready
PRINT ABS(-3)
3
Ready
```

The **COS( )** function takes a number argument and returns it's COSine value interpreting the number as an angle in degrees. Since the CFSound-IV BASIC only supports whole, integer numbers the number that is returned is equal to the COSine of the angle times 1024. Let's try it. Remember that the COSine of 0 degrees should be 1.0, the COSine of 45 degrees should be 0.707, and the COSine of 90 degrees should be 0.0 – the results are multiplied by 1024 to allow integer manipulation:

```
PRINT COS(0)
1024
Ready
PRINT COS(45)
724
Ready
PRINT COS(90)
0
Ready
```

The **ERR( )** function takes no argument and returns the number of the last error that BASIC encountered. This will be useful later on when you want your programs to be able to handle some errors without stopping. There is a table of error number in the BASIC Reference at the end of this manual. Let's try it:

```
PRINT ERR()
0
Ready
PRINT 5 / 0
Divide by zero error
Ready
PRINT ERR()
6
Ready
```


Back to Table of Contents

The **FIND( )** function takes two arguments – both strings. It returns the zero-based index of the second string in the first, or -1 if the second string doesn't appear in the first string at all. This will be very useful later when you're trying to manipulate strings in your programs. Let's try it:

```
PRINT FIND("Now is the time", "is")
4
Ready
PRINT FIND("Now is the time", "Now")
0
Ready
PRINT FIND("Now is the time", "country")
-1
Ready
```

The **GETCH( )** function takes a number argument telling it how to behave, then returns a number based upon the next character that is entered via the attached communications device. If the argument is 0 the function returns -1 if no character is currently available, otherwise it returns the ASCII decimal value of the character. If the argument is non-zero the function waits for the next character to be received when it then returns the ASCII decimal value of the character.

The **HEX.VAL( )** function takes a string argument which it tries to interpret as a hexadecimal value (only characters 0-9 and A-F, a-f). If it successfully converts the hexadecimal string representation of a number to a value it returns that value – otherwise it causes a Syntax error. A few examples:

```
PRINT HEX.VAL("09AB")
2475
Ready
PRINT HEX.VAL("X91")
Syntax error - can't parse HEX.VAL(argument) to number
Ready
```

The **LEN( )** function takes a string argument and returns its length – how many characters does it contain. This will be very useful later when you're trying to manipulate strings in your programs. Let's try it:

```
PRINT LEN("Now is the time")
15
Ready
PRINT LEN("")
0
Ready
```

The **MULDIV( )** function takes three numeric arguments and returns the result of multiplying the first two together then dividing by the third. What's the value of that? Why can't you just do A * B / C ? The magic is that the three numeric arguments are converted from 32-bit signed integers to 64-bit signed integers first, then the multiply and divide are performed, then the result is converted back to a 32-bit signed integer and returned. Without this function, the only numbers that you could multiply together with a correct result would have to have a product of no more than +2,147,483,647.

The **MULMOD( )** function takes three numeric arguments and returns the result of multiplying the first two together and then taking the modulo with the third. Here's an example of using both MULDIV( ) and MULMOD( ) to calculate 55 percent of 999. Using 16-bit integer math would cause an overflow:

```
list
10 REM calculate 55 percent of 999 (999 * 55) / 100 = 549.45
20 PRINT MULDIV(999,55,100);".";MULMOD(999,55,100)
Ready
run
549.45
```


Back to Table of Contents

```
Ready
```

The **RND( )** function takes in a numeric argument and returns a random number that ranges from zero to the argument minus one.

The **SIN( )** function takes a number argument and returns it's SINe value interpreting the number as an angle in degrees. Since the CFSound-IV BASIC only supports whole, integer numbers the number that is returned is equal to the SINe of the angle times 1024. Let's try it. Remember that the SINe of 0 degrees should be 0.0, the SINe of 45 degrees should be 0.707, and the SINe of 90 degrees should be 1.0 – the results are multiplied by 1024 to allow integer manipulation:

```
PRINT SIN(0)
0
Ready
PRINT SIN(45)
724
Ready
PRINT SIN(90)
1024
Ready
```

The **VAL( )** function takes a string argument which it tries to interpret as a decimal value. If it successfully converts the decimal string representation of a number to a value it returns that value – otherwise it causes a Syntax error.

```
PRINT VAL("12345")
12345
Ready
PRINT VAL("-35")
-35
Ready
```

Functions Returning a String

These functions take one or more arguments and return a string value. You can tell that they return a string value because their function names end with a dollar sign.

The **CHR$( )** function takes a number argument and returns a single character string where the character is the ASCII character of the decimal number. (see http://www.asciitable.com). Let's try it:

```
PRINT CHR$(65)
A
Ready
PRINT CHR$(48)
0
Ready
```

The **ERR$( )** function takes no argument and returns the string of the last error message that BASIC encountered. This will be useful later on when you want your programs to be able to handle some errors without stopping. There is a table of error messages in the BASIC Reference at the end of this manual. Let's try it:

```
PRINT ERR$()
0 error
Ready
PRINT 5 / 0
Divide by zero error
Ready
PRINT ERR$()
Divide by zero error
Ready
```


Back to Table of Contents

The **FMT$( )** function takes two arguments; the first is a string containing information about how to format the second argument into the string value that it returns. The description of the format string argument can be found in the BASIC Reference section of this manual. Let's try it:

The **HEX.STR$( )** function takes a numeric argument and returns the hexadecimal string equivalent. Let's try it:

```
PRINT HEX.STR$(25)
19
Ready
PRINT HEX.STR$(256)
100
Ready
```

The **INSERT$( )** function takes three arguments – a source string to insert into, a zero-based offset of where to insert and a string to add into the source string. It returns the source string with the string to be added inserted at the numeric offset. The source string is not directly modified by the function. Let's try it:

```
10 REM test insert$
20 baseString$ ="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
30 insertString$ ="insert"
35 REM insert at beginning
40 PRINT INSERT$(baseString$,0,insertString$)
45 REM insert in middle
50 PRINT INSERT$(baseString$,13,insertString$)
55 REM insert past end
60 PRINT INSERT$(baseString$,30,insertString$)
Ready
run
insertABCDEFGHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMinsertNOPQRSTUVWXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZinsert
Ready
```

The **LEFT$( )** function takes two arguments – a source string and the number of characters from the beginning of the string to return. The source string is not directly modified by the function. Let's try it:

```
10 REM test left$
20 baseString$ ="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
35 REM take leftmost 5 characters
40 PRINT LEFT$(baseString$,5)
45 REM take leftmost 10 characters
50 PRINT LEFT$(baseString$,10)
Ready
run
ABCDE
ABCDEFGHIJ
Ready
```

The **MID$( )** function takes three arguments – a source string, a zero-based offset of where to start taking from and the number of characters to return. The source string is not directly modified by the function.

The **RIGHT$( )** function takes two arguments – a source string and the number of characters from the end of the string to return. The source string is not directly modified by the function.

The **REPLACE$( )** function takes three arguments.

Back to Table of Contents

The **STR$( )** function takes a single numeric argument and returns a string representation of that number.

Some Funny Characters

Your communications device – whether it is a PC or PS/2 keyboard or even the pop-up touch keypad has a limited set of characters. Your program may require some special characters or symbols be displayed – like a degree symbol after a temperature for example. The CFSound-IV can CFSound these characters that don't appear on the keyboard, but it takes a couple tricks.

The first trick is the understanding that each displayed character is actually based upon a number. Back in the 60's a standards committee got together and hashed out a definition of what number represents what character. It started with the old telegraphic codes for characters and evolved to support the newly designed tele-printer. In Computer Terminology this is called ***A.S.C.I.I. – which is the abbreviation for American Standard Code for Information Interchange***. It defines 128 characters and control codes, each of which can be uniquely represented by a seven bit number.

There's a chart of these numbers and their representative characters at http://www.asciitable.com . Now when we want to show a certain character we just have to the single 7-bit number to be sent to the CFSound. This means we need a function that takes in a decimal number and converts it to a single number representing the string character based upon this ASCII standard. BASIC has just such a function built-in.

The **CHR$( )** function takes a decimal number argument and returns a single character string where the character is the ASCII character of the decimal number. This character can then be sent to the console terminal by **PRINT**ing it. Let's try it:

```
PRINT CHR$(65)
A
Ready
PRINT CHR$(48)
0
Ready
```

So how do we CFSound characters that aren't in the ASCII standard? It turns out that there is another standard call Unicode which defines how to combine multiple non-ASCII codes into a single extended character. Remember that ASCII only defines 7-bit codes, 0 – 127. Since computers work with 8-bit codes, 0 – 255, that leaves codes 128 – 255 available. However the Unicode designers didn't want to limit themselves to just 128 more characters – multiple languages required many more. So they came up with an expandable scheme using multiple prefix characters that can be combined and decoded to represent many more extended characters.

Sounds complicated – and it can be – if you have to do the encoding and decoding. However the CFSound-IV only provides display support for what is known as the DOS – United States character set. There is a table of these supported characters at the end of this manual. To display them you just have to send the multiple character sequence underneath the character. The definition of what values represent what extended characters were also defined starting in the 60's and is loosely known as the ANSI code standard. ANSI stands for American National Standards Institute.

So let's try showing that degree character. From the table, it looks like the two codes are 194 and 176 – remember to separate the two function calls with a semicolon – if you use a comma instead it won't work because a space character (32) will get inserted between them:

```
print chr$(194);chr$(176)
°
Ready
```

And there it is. You can try some others from the table – some take two codes and some three. Using BASIC **PRINT** commands to display characters is referred to as ANSI operation.

Back to Table of Contents

Don't Get Boxed In

If you look at the Extended Characters table at the end of the manual, there are some that can be used for line drawing. Let's try and draw a simple box. We need the four corners, the lines across the top and bottom and the lines on the side. Try typing this short program in – don't forget to **NEW** first:

```
10000 REM Draw a box using ANSI characters
10005 PRINT CHR$(226); CHR$(148); CHR$(140);
10010 FOR I=1 TO 10
10015 PRINT CHR$(226); CHR$(148); CHR$(128);
10020 NEXT I
10025 PRINT CHR$(226); CHR$(148); CHR$(144)
10030 PRINT CHR$(226); CHR$(148); CHR$(130);
10035 FOR I = 1 TO 10
10040 PRINT " ";
10045 NEXT I
10050 PRINT CHR$(226); CHR$(148); CHR$(130)
10055 PRINT CHR$(226); CHR$(148); CHR$(148);
10060 FOR I=1 TO 10
10065 PRINT CHR$(226); CHR$(148); CHR$(128);
10070 NEXT I
10075 PRINT CHR$(226); CHR$(148); CHR$(152)
```

And now **RUN** it:

```
run
┌──────────┐
└──────────┘
Ready

save ansibox
Ready
```

If it works as shown then you can save it for later.

Now you're probably asking yourself why are the line numbers so large? The large line numbers allow us to come back later and put some lower-numbered lines before these. In this example they're going to let us save some typing.

Character Rules

1. Characters are actually a displayed version of a single number.
2. The first 128 characters (numbers 0 – 127) have been agreed upon and are referred to as the ASCII character set. Refer to http://www.asciitable.com
3. There is a built-in function, **CHR$( )** that can convert a numeric argument into a character.
4. Additional extended characters may be displayed using multiple character prefix codes. These codes are terminal emulator locale and codepage specific.
5. Using BASIC **PRINT** commands on the CFSound is known as ANSI operation.

Back to Table of Contents

Don't Repeat Yourself

Suppose that you want to draw two boxes, one above the other. You could type the same thing in again, duplicating all of these lines. Your programs would tend to get quite large – and, you might run out of line numbers. What you have now is a box-drawing routine. Suppose you could just call this routine whenever you wanted to draw a box.

Good News! You can. There is a routine calling command in most computer languages – including BASIC. In Computer Terminology these common routines are referred to as ***subroutines***. So how do you get to this part of your program from somewhere else? And how do you get back? The new command pair is **GOSUB / RETURN**. And they are a pair – you can't have one without the other. The Computer Terminology is ***calling into a subroutine*** and ***returning from a subroutine*** – call in and return back right after where you called from.

Let's add some additional lines to the program:

```
10 print "Here's a box:"

20 gosub 10000

30 print "And here's another:"

40 gosub 10000

50 end
```

And don't forget the last line that will turn the box drawing lines into a subroutine:

```
10080 return
```

Your program should now look like this:

```
list
10 PRINT "Here's a box:"
20 GOSUB 10000
30 PRINT "and here's another:"
40 GOSUB 10000
50 END
10000 REM Draw a box using ANSI characters
10005 PRINT CHR$(226); CHR$(148); CHR$(140);
10010 FOR I=1 TO 10
10015 PRINT CHR$(226); CHR$(148); CHR$(128);
10020 NEXT I
10025 PRINT CHR$(226); CHR$(148); CHR$(144)
10030 PRINT CHR$(226); CHR$(148); CHR$(130);
10035 FOR I = 1 TO 10
10040 PRINT " ";
10045 NEXT I
10050 PRINT CHR$(226); CHR$(148); CHR$(130)
10055 PRINT CHR$(226); CHR$(148); CHR$(148);
10060 FOR I=1 TO 10
10065 PRINT CHR$(226); CHR$(148); CHR$(128);
10070 NEXT I
10075 PRINT CHR$(226); CHR$(148); CHR$(152)
10080 RETURN
Ready
```

And when you **RUN** it:

```
run
Here's a box:
┌──────┐
└──────┘
and here's another:
┌──────┐
└──────┘
Ready
```


Back to Table of Contents

If you change the **GOSUB** to a **GOTO** you will get an error when your program executes the **RETURN** command – it doesn't know the way back that is remembered by the **GOSUB**.

Subroutines are used to replace common or duplicate code. If you look at our program there's another common piece of code that could be made into a subroutine – the PRINTing of the Unicode prefix for the box drawing extended characters: **PRINT CHR$(225); CHR$(148);**

Let's make this repetitive code into a separate subroutine – it could go above or below, we'll put it below:

```
10100 REM Print the Unicode prefix characters
10105 PRINT CHR$(226); CHR$(148);
10110 RETURN
```

```
list
10 PRINT "Here's a box:"
20 GOSUB 10000
30 PRINT "and here's another:"
40 GOSUB 10000
50 END
10000 REM Draw a box using ANSI characters
10005 PRINT CHR$(226); CHR$(148); CHR$(140);
10010 FOR I=1 TO 10
10015 PRINT CHR$(226); CHR$(148); CHR$(128);
10020 NEXT I
10025 PRINT CHR$(226); CHR$(148); CHR$(144)
10030 PRINT CHR$(226); CHR$(148); CHR$(130);
10035 FOR I = 1 TO 10
10040 PRINT " ";
10045 NEXT I
10050 PRINT CHR$(226); CHR$(148); CHR$(130)
10055 PRINT CHR$(226); CHR$(148); CHR$(148);
10060 FOR I=1 TO 10
10065 PRINT CHR$(226); CHR$(148); CHR$(128);
10070 NEXT I
10075 PRINT CHR$(226); CHR$(148); CHR$(152)
10080 RETURN
10100 REM Print the Unicode prefix characters
10105 PRINT CHR$(226); CHR$(148);
10110 RETURN
Ready
```

And now we have to change lines 10005, 10015, 10025, 10030, 10050 10055, 10065 and 10075. You could type these lines over, but as we'll see soon, BASIC has a shortcut – the **EDIT** command.

Rules for Subroutines

1. Any block of code can be turned into a subroutine by adding a **RETURN** command as the last line and then calling the code using the **GOSUB** command.
2. **GOSUB** commands must be paired with **RETURN** commands – you can't **GOTO** in or out of a subroutine. While it may appear to work, the stack that BASIC uses to remember where to **RETURN** to will eventually become confused and an error will occur. To leave a subroutine, **GOTO** the subroutine's **RETURN** statement.
3. Subroutines can be nested – code inside a subroutine can call another subroutine.

Back to Table of Contents

Making Changes

Type:

```
EDIT 10005
```

Notice how BASIC prints out the line, but stays at the end – it doesn't show the Ready prompt. The cursor is blinking at the end of the line. It's waiting for you to modify the line and tell it when you're done.

```
edit 10005

10005 PRINT CHR$(226); CHR$(148); CHR$(140); _
```

You can move your blinking cursor left and right with the keyboard left and right arrow keys ←. Some communication devices will also allow you to Home home to the beginning of the line or End end to the end of the line. You can delete characters left of the cursor using the backspace key backspace. If you want to quit editing without saving your changes use the double ESCape esc esc key. When you want to stop editing and save your changes press the enter key Enter↵.

One thing to remember about EDITing – you're always in 'insert' mode. Any printable character that you type will be inserted into the line wherever the cursor is blinking. If you make a mistake, you can use the backspace key and retype it, or use the arrow keys to move after it, backspace and retype.

Use the left arrow key to move the cursor from the end of the line to the C in the last **CHR$(140)** function. Now press the backspace key to remove characters until the space after the **PRINT**. Again use the left arrow key to move the cursor to the P in the **PRINT**. The type in **GOSUB 10100 :** - the colon is important, it allows you to group multiple commands on the same line. When you're done making these changes the line should look like:

```
10005 GOSUB 10100 : PRINT CHR$(140);
```

If it doesn't look like this use the arrow keys, backspace and typing to correct it. When it does look like this press Enter. The line should be accepted as if you had typed it in again instead of editing. You can look at it using the **LIST** command with the line number:

```
list 10005
10005 GOSUB 10100 : PRINT CHR$(140);
Ready
```

Now try running the program. It should work the same as it did before the change.

Edit the other lines to make the same change. Once you get the hang of using the **EDIT** command you'll probably find it much easier than typing the complete line over again, and you'll tend to make fewer mistakes.

```
list
10 PRINT "Here's a box:"
20 GOSUB 10000
30 PRINT "and here's another:"
40 GOSUB 10000
50 END
10000 REM Draw a box using ANSI characters
10005 GOSUB 10100 : PRINT CHR$(140);
10010 FOR I=1 TO 10
10015 GOSUB 10100 : PRINT CHR$(128);
10020 NEXT I
10025 GOSUB 10100 : PRINT CHR$(144)
10030 GOSUB 10100 : PRINT CHR$(130);
10035 FOR I = 1 TO 10
10040 PRINT " ";
10045 NEXT I
10050 GOSUB 10100 : PRINT CHR$(130)
10055 GOSUB 10100 : PRINT CHR$(148);
10060 FOR I=1 TO 10
10065 GOSUB 10100 : PRINT CHR$(128);
10070 NEXT I
10075 GOSUB 10100 : PRINT CHR$(152)
10080 RETURN
```


Back to Table of Contents

```
10100 REM Print the Unicode prefix characters
10105 PRINT CHR$(226); CHR$(148);
10110 RETURN
Ready
run
Here's a box:
┌──────────┐
└──────────┘
and here's another:
┌──────────┐
└──────────┘
Ready
```

And finally, now that you know you can have multiple statements on a line, you can shorten this up some more while still keeping it readable:

```
list
10 PRINT "Here's a box:" : GOSUB 10000
30 PRINT "and here's another:" : GOSUB 10000
50 END
10000 REM Draw a box using ANSI characters
10005 GOSUB 10100 : PRINT CHR$(140);
10010 FOR I = 1 TO 10 : GOSUB 10100 : PRINT CHR$(128); : NEXT I
10025 GOSUB 10100 : PRINT CHR$(144)
10030 GOSUB 10100 : PRINT CHR$(130);
10035 FOR I = 1 TO 10 : PRINT " "; : NEXT I
10050 GOSUB 10100 : PRINT CHR$(130)
10055 GOSUB 10100 : PRINT CHR$(148);
10060 FOR I=1 TO 10 : GOSUB 10100 : PRINT CHR$(128); : NEXT I
10075 GOSUB 10100 : PRINT CHR$(152)
10080 RETURN
10100 REM Print the Unicode prefix characters
10105 PRINT CHR$(226); CHR$(148); : RETURN
Ready
```

Rules for Editing

1. You can modify program lines in-place using the **EDIT** command.
2. While EDITing a program line, the arrow keys move the blinking cursor left and right and the backspace key deletes the character to the left.
3. While EDITing a program line any other character that you type will be entered into the line where the blinking cursor is – you are always in 'insert mode'.
4. You can discard your changes while editing by ESCaping – pressing the ESC key twice in a row.
5. You can accept your editing changes by pressing Enter – you don't have to be at the end of the line.
6. Changing the line number of an EDITed line will add or replace the newly numbered line in your program – the original line will remain in-place and intact.

Back to Table of Contents

There's a System

So far, we've only defined our own variables – numeric and string, simply by using them in our programs to remember values. We've used commands and operators to access, modify and compare their contents. And we've used built-in functions like read-only variables to extend our program operation with built-in functionality. What's next?

Just like the built-in functions, BASIC has some built-in variables to access and modify values used by the CFSound-IV 'system'. These are called System Variables and they use a special naming so that they don't interfere with your program variables. So what can these System Variables do for you?

How about timing? Let's try to make a simple metronome application. Type **NEW** and enter the following small program:

```
10 REM Metronome
15 t = 0 : INPUT "Enter tempo (1 - 32766) :",tempo
20 IF tempo < 1 OR tempo > 32766 THEN GOTO 15
25 FOR i = 1 TO tempo : d = d + 1 : NEXT i
30 IF t = 0 THEN PRINT "Tick" ELSE PRINT "Tock"
35 IF t = 0 THEN t = 1 ELSE t = 0
40 GOTO 25
Ready
```

Let's take a moment and see what this program is doing. Line 10 is a REMark – a comment – telling us what this program is.

Line 15 initializes the tick/tock variable – this isn't really necessary since variables are set to 0 the first time they're used, but it's good practice to start with a known value. The program user is then prompted to enter a number for the tempo – notice that the **INPUT** command can have an optional prompt string that it will display before waiting for the user to enter a value and press enter.

Line 20 checks the tempo that the user entered and, if it's not within range, requests that the user enter a different value by looping back to line 15. It's always a good idea to validate user input to your programs. Notice that it performs two comparisons on the tempo value and then combines them logically with the **OR** logical operator – **IF** the first condition **OR** the second condition is True **THEN** loop back to line 15.

Line 25 counts from one to the tempo value – it does nothing but waste time. The amount of time that must pass before the **FOR / NEXT** counting loop ends is dependent upon the tempo value entered. The d=d+1 statement is there solely to slow the loop down so reasonable tempo values have an effect.

Line 30 prints Tick or Tock depending upon the tick/tock variable t. Line 35 toggles the tick/tock variable t between zero and one.

Line 40 loops back to the delay loop.

Try running the program and entering different values for the tempo – a value of 4000 seems to be close to 60 beats per minute – once a second. You will have to ESCape the program to try another tempo:

```
run
Enter tempo (1 - 32766) :4000
Tick
Tock
Tick
Tock
Tick
Tock
Tick
Tock
Tick
 ESC at line 25
Ready
```

So how can we improve this program? It sure would be more useful to just enter the beats per minute instead of an arbitrary number. And this program kind of works backwards – a higher tempo value should result in a faster beats per minute – not slower.

Back to Table of Contents

We could try different equations to compute a delay loop count that corresponds to the requested tempo value but that would take a lot of trial and error to get the correct calculation that would relate the two values. A better idea would be to use some kind of timer that we could control to keep track of the elapsed time.

System Timers

The CFSound-IV system has such a timer – in fact, there are ten of them. How do you access them and how do they work?

In BASIC, a system variable is identified by preceding the name with the '@' character. To access the first timer you would write:

```
@TIMER[0] = value : REM sets timer 0

value = @TIMER[0] : REM gets timer 0
```

There is an array of ten timers: numbered 0 → 9. You might have guessed that the zero in brackets is telling BASIC that you want timer number 0. The brackets [ ] indicate that you are selecting a single timer – the one identified by the number in-between. In Computer Terminology this is referred to as ***indexing into an array***.

	@TIMER[10]	**values**
	0	
	1	
	2	
	3	
@TIMER[4] →	**4**	
	5	
	6	
	7	
	8	
	9	

So how do these timers work? Fifty times per second, the CFSound-IV checks these ten timers. If any of them are not equal to zero then the CFSound subtracts one from it. In Computer Terminology this is referred to as ***decrementing a non-zero timer***. So you start the timer by setting it to a non-zero value, then you can 'monitor' the timer by checking to see if it is at zero. Since it counts down by 50 times per second, to set the timer for one second you would set it to fifty. Cool.

So how do we set a timer count for beats per minute? We can calculate this given the counts per second and the seconds per minute to get the counts per beat:

$$\frac{\left(50\frac{counts}{second}\times 60\frac{seconds}{minute}\right)}{\frac{beats}{minute}} = 3000\frac{counts}{beats}$$

OK. Divide 3000 by the desired beats per minute and use that to set the timer. Then we can loop for the timer to be zero for our delay.

As for validating the user input, we can't divide by zero – it's an error so we have to be greater than that, and dividing 3000 by anything larger than 3000 will result in a zero – timer won't run at all. Reasonable numbers are probably greater than 30 bpm and less than 300 bpm. Let's change the program:

```
10 REM Metronome
15 t = 0 : INPUT "Enter tempo beats per minute (30 - 300) :",tempo
20 IF tempo < 30 OR tempo > 300 THEN GOTO 15
25 @TIMER[0] = 3000 / tempo
27 IF @TIMER[0] > 0 THEN GOTO 27
30 IF t = 0 THEN PRINT "Tick" ELSE PRINT "Tock"
35 IF t = 0 THEN t = 1 ELSE t = 0
40 GOTO 25
```


Back to Table of Contents

And you can run it and try different tempo values. Notice how line 27 keeps checking @TIMER[0] until it reaches zero – remember that the @TIMER system variable is being decremented 50 times a second outside of your program – in the background. In Computer Terminology this constant checking in a loop is referred to as **polling**.

Rules for System Variables

1. System variables access and control CFSound-IV system functions.
2. System variable names are prefixed with an '@' character.
3. System variables can have different meanings if they are read versus written.
4. Some system variables are read-only.
5. Some system variables are implemented as an array and require an index for selection.

Staging an Event

Suppose that we wanted to measure delays of seconds in our program. We could add a 1 second delay using a **@TIMER** system variable and increment the second variable from zero through fifty-nine every time that we read **@TIMER** is zero. There is another way that will allow our program to do other things instead of simply spinning in a loop waiting on a timer. Welcome to Events.

Events are changes in your program's execution that can happen in between each statement. The linear execution of your program's statements is interrupted to process the event and then execution resumes from where your program was interrupted. Events are associated with and triggered by some system variables and can occur as a result of hardware or software actions.

You control how system variable events are processed in your program by associating them with an subroutine to be called when the happens. In Computer Terminology this is known as an **event handler**. The event handler is a subroutine you write that will be called when the event is signaled by changes in the associated system variable.

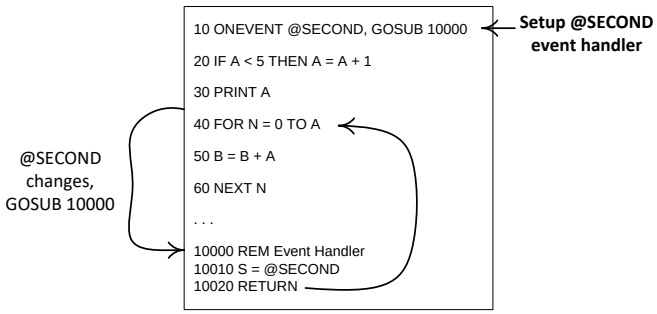
The event handler is configured or setup using the BASIC's **ONEVENT** command:

```
ONEVENT @systemvariable, GOSUB linenumber
```

This command records the handler subroutine line number in an event table for the specified system variable entry. Changes to system variables that cause events set a flag on their entry in the event table.

In between processing each program statement BASIC scans the event table looking for flagged entries. When a flagged entry is found BASIC performs the **GOSUB** to the recorded event handler line number. When the event handler subroutine returns, BASIC clears the flagged entry.

In this example line 10 establishes an event handler subroutine for the **@SECOND** system variable. Whenever the value of **@SECOND** changes the subroutine at line 10000 will be called.



Back to Table of Contents

Once the handler has been established by executing the **ONEVENT** statement the event can happen in between any two program statements. In this example the **@SECOND** changed between lines 30 and 40 but it could have occurred somewhere else. The only place that the event handler cannot be called is inside the event handler itself. This is because events are prioritized – an executing event handler can only be interrupted by a higher priority event than the event which is being handled. Notice that event handlers take time away from your program and they prevent lower priority event handlers from running when they are. It is important to quickly handle the event and **RETURN** so that your program can and other event handlers can continue to run.

Let's rewrite the metronome program to use events. We add line 12 to establish an event handler subroutine for **@TIMER[0]** – when it counts to zero the subroutine at line 1000 will be called. We remove line 27 that was 'polling' the **@TIMER** for zero. We move lines 30 and 35 to 1005 and 1010 and add line 1015 to reset the timer for the next tick/tock cycle.

Line 25 now 'primes the pump' so to speak – it starts the **@TIMER[0]** running. When it counts to zero then the subroutine at line 1000 is called, prints tick or tock and sets the timer for the next cycle.

Notice that once the timer has started the rest of the program in line 40 does nothing – it just loops back onto itself. In other applications there may be more work to do outside of the event handling.

```
10 REM Metronome using @TIMER events
12 ONEVENT @TIMER[0], GOSUB 1000
15 t = 0 : INPUT "Enter tempo beats per minute (30 - 300) :",tempo
20 IF tempo < 30 OR tempo > 300 THEN GOTO 15
25 @TIMER[0] = 3000 / tempo
40 GOTO 40
1000 REM @TIMER[0] Event Handler
1005 IF t = 0 THEN PRINT "Tick" ELSE PRINT "Tock"
1010 IF t = 0 THEN t = 1 ELSE t = 0
1015 @TIMER[0] = 3000 / tempo
1020 RETURN
Ready
run
Enter tempo beats per minute (30 - 300) :100
Tick
Tock
Tick
Tock
Tick
 <<< ESC at line 40 >>>
Ready
```

It is important to remember that all variables are shared between all parts of your program. If your program is performing some long calculation using a variable and you change that variable in an event handler your program may get confused or generate incorrect results since the event handler can potentially execute anywhere in your program after it is configured.

Rules for Events

1. Some system variables provide support for 'events' when their value changes.
2. A program can automatically call a subroutine when an event occurs using the **ONEVENT** statement to associate the system variable with the subroutine – to 'handle' the event.
3. While in an event handler subroutine identical or lower-priority events cannot be handled until after the **RETURN**.

Back to Table of Contents

BASIC Reference

BASIC is an integer, microcomputer basic designed for simple control applications.

BASIC executes programs consisting of one or more statements. Statements consist of an optional line number followed by reserved keyword commands specifying operations for BASIC to perform followed by required and / or optional arguments.

Program vs Direct Mode

Statements that begin with a line number are entered and held, sorted by line number, until BASIC is commanded to execute them. This is called the **Program** mode of operation. Statements entered without a line number are evaluated and executed immediately. This is called the **Direct** mode of operation.

Some keyword commands are **Direct** mode only and may not appear in a program. Some keyword commands are **Program** mode only and may not be evaluated and executed immediately after being typed in. These limitations are listed in the keyword command definitions below.

Programs

In BASIC a Program consists of one or more program lines. Each program line consists of a line number followed by one or more statements. Valid line numbers range from 1 to 2,147,483,647. Multiple statements in a program line must be separated by colons (":"). Program lines that are entered without a line number are executed directly. Only certain statements may be executed directly. When BASIC is awaiting statement or program line entry it issues a READY prompt via the serial port.

```
Ready
dir *.bas
TEVENT.BAS          250 A      11-09-2058 14:30:10
PROGRAM1.BAS         55 A      11-09-2058 15:52:44
SOUNDS.BAS          248 A      01-01-1980 00:00:00
TEST.BAS             63 A      01-01-1980 00:00:00
CEVENTS.BAS         144 A      01-01-1980 00:00:00
PROGRAM2.BAS         47 A      11-09-2058 15:58:14
ONGOTO.BAS          253 A      05-08-2052 14:35:54
ONGOSUB.BAS         272 A      11-09-2058 14:45:08
TIMER.BAS           185 A      11-15-2058 15:20:26
CHIMES.BAS          884 A      09-07-2021 16:55:10
DEMO.BAS           2143 A      11-13-2020 18:36:26
MSGTEST.BAS          78 A      11-11-2020 16:15:32
-----------------------
                   12 files
                    0 directories
Ready
```

Programs may be entered a line at a time by a stream of characters via the serial port, or by loading from a file off of an optional SD card. When entered via the serial port, a program line will replace any matching program line, and entering a line number only will delete the corresponding program line. Entered program lines are limited to 255 characters of length.

```
10 PRINT "This is a Test"
20 STOP
list
10 PRINT "This is a Test"
20 STOP
Ready
20
list
10 PRINT "This is a Test"
Ready
run
This is a Test
Ready
print "This is also a Test"
This is also a Test
Ready
```


Back to Table of Contents

Arbitrary Precision strongly recommends developing BASIC programs interactively via a connected terminal / computer so that error messages can be viewed and the program operation can be refined quickly – otherwise the program may silently stop running leaving no clue as to what has happened.

Program lines may be viewed with the **LIST** statement. All program lines may be cleared with the **NEW** statement. Program execution is started using the **RUN** statement. Upon power-up, BASIC clears the program memory and awaits statement or program line entry via the serial port.

Program lines may be edited via a connected ANSI terminal (or computer with ANSI terminal emulation) with the **EDIT** statement. (See the **EDIT** keyword command definition below for more information.) A command stack holding the last 10 commands is accessible using the ANSI terminal up and down arrow keys.

Entering an Escape character (0x1B) twice in succession via the serial port while a program is running will cause termination of the program and BASIC will output a message then await further statement or program line entry via the serial port. If the program is awaiting input by executing an **INPUT** statement a trailing carriage return may be necessary to terminate the **INPUT** before the Escape sequence is seen.

```
new
Ready
10 for i=1 to 10
20 print i
30 delay(10)
40 next i
list
10 FOR i=1 TO 10
20 PRINT i
30 DELAY(10)
40 NEXT i
Ready
run
 1
 2
 3
 4    <- Escape key pressed twice here
ESC at line 20
Ready
```

Line Numbers

Line numbers are 31 bit positive integers and can range in value from **1 ≤ line ≤ 2,147,483,647**. They are used by BASIC to keep the entered program lines in sequence and to be able to **EDIT**, **LIST** and replace or delete lines to develop programs interactively.

When entering program lines, line numbers should be advanced by 5, 10 or more for each line to allow room for the later addition of program statements between existing lines without having to renumber all of the lines. Subroutines and Functions should start at multiples of 1000 or more to allow each subroutine or function to grow in size without affecting other code.

There is a **RESQ** command to re-sequence all or portions of the line numbers if developing interactively.

Back to Table of Contents

Variables

BASIC has seven types of variables:

1. **Integer Numeric**
2. **Integer Numeric Arrays**
3. **Character Strings**
4. **Character String Arrays**
5. **Integer Numeric Functions**
6. **Character String Functions**
7. **Character string Labels.**

Variable names *are* **case sensitive**. The may contain letters, numbers and underscore but they must start with a letter. They can be up to 32 characters long. Names of character string variables must end with the dollar-sign character ('**$**').

Numeric variables are **signed 32-bit** and can assume the integer values **(–2,147,483,648 ≤ variable ≤ +2,147,483,647)**.

Character String variables are comprised of zero or more **unsigned 8-bit characters** are limited to **255 characters** in length.

Variable arrays are indexed with up to three array subscripts separated by commas and enclosed in square brackets **[ ]** and must be **DIM**ensioned before they are used.

Label variables are denoted by a leading accent grave character (``` – not apostrophe), may be up to 32-characters in length, are placed at the beginning of a program line after the line number and may be used in place of the line number in some program statements and commands. At program startup the entire program is scanned and the line numbers associated with any label variables are defined.

After a program has run and has stopped or been interrupted the currently defined variables and their current values may be listed with the **VARS** command.

The number of variables is limited only by the available memory. Shorter variable names execute slightly faster.

In the **BETA** software three additional variable types have been added:

8. **Real Numeric**
9. **Real Numeric Arrays**
10. **Real Numeric Functions**

Names of real variables must end with the percent character('%'). Real variables are signed 32-bit single precision floating point and can assume the values **(± ~10^{-38} ≤ variable% ≤ ~10^{38})**.

Constants

In BASIC, numeric constants are just the written version of a number that the program may not change during its execution. Constants may be expressed as decimal, binary, octal or hexadecimal values and have a **32-bit** integer range of permissible values.

Decimal constants are denoted by a sequence or 1 or more numeric digits 0 → 9. For example: 85 or 12459.

Binary constants are denoted with the two character prefix **0b** (zero, bee) followed by a string of one or more of the numeric binary digits 0 or 1. Each binary digit represents a single bit in the equivalent number. For example: **0b10100000** which is the binary representation for the decimal number 160.

Back to Table of Contents

Octal constants are denoted with the two character prefix **0o** (zero, oh) followed by a string of one or more or numeric octal digits 0→7. Each octal digit represents three bits in the equivalent number. For example: **0o377** which is the octal representation for the decimal number 255.

Hexadecimal constants are denoted with the **0x** (zero, ex) prefix followed by a string or one or more numeric hex digits 0→9 and A→F. Each hex digit represents four bits in the equivalent number. For example: **0x12AB** which is the hexadecimal representation for the decimal number 4779.

String constants are enclosed in double quotes **“ ”**. Whatever characters are enclosed within the double quotes comprise the string – without the enclosing quotes. ***Note the string cannot contain the null (zero) character as it is used internally to terminate the string value.***

In the **BETA** software real constants are denoted by a sequence of 1 or more numeric digits 0→9 followed by a decimal point (‘.’) and optionally followed by a sequence of zero or more numeric digits 0→9 providing the fractional portion of the number. In addition the number may be followed by an exponent comprised of the letter (‘E’ or ‘e’) followed by a sign (‘+’ or ‘-‘) and followed by a two numeric digit value. For example: 1E2 is the real value 100, 1E-02 is the real value 0.01.

Back to Table of Contents

System Variables

BASIC also has built-in system variables. System variables are denoted by a '**@**' character as the first character of the variable name. The system variable names are 'tokenized' when entered to save program memory and speed program execution: for example the system variable **@SECOND** would be tokenized to two bytes instead of seven bytes.

System variables **may not** be assigned a value by appearing in an **FOR, DIM, INPUT, READ, FINPUT #N** or **FREAD #N** statement. Some system variables are read-only and may not appear on the left hand side of a **LET** assignment statement.

Some system variables have ***Events*** associated with them and may be referenced in **ONEVENT, SIGNAL** and **WAIT** statements. See the description for the individual system variables and the ***Events*** section below for more information.

Timing

The following system variable provides the ability to determine time intervals:

@TIMER[x]

The **@TIMER[x]** system variables allow the BASIC program to measure or control time intervals. There are ten 16-bit timers; permissible values for [x] are 0 through 9.

Setting the variable to a non-zero value activates the timer. The non-zero value in the timer variable is decremented every 20mSEC (50 Hz) until it reaches zero. Upon reaching zero any associated event handler specified with the **ONEVENT** statement is activated.

The maximum timer value is 32767 providing a 655.34 second interval.

Input / Output

The following system variables provide access to input and output:

@PORT[x] / @PORT2[x]

The **@PORT[x]** and **@PORT2[x]** system variables allow the BASIC program to access I/O ports located on the CFSound expansion busses.

The **@PORT[x]** accesses the rear EXP1 module and the **@PORT2[x]** accesses the front EXP2 module.

There are 256 eight bit ports; permissible values for **[x]** are 0 through 255.

Setting the variable to a value writes the value to the I/O port **[x]**. Reading the variable returns the value from the I/O port **[x]**.

Note that ports 0, 1 and 2 are consumed by optional installed CFSound Contact I/O modules.

@CONTACT[x]

The **@CONTACT[x]** system variables allow the BASIC program to access the CFSound and Expansion Module contacts.

There are 56 contact inputs and 56 contact outputs; permissible values for **[x]** are 0 through 55.

Setting the variable to a '1' activates output contact **[x]**. Reading the variable returns the value from the input contact **[x]**.

Back to Table of Contents

@CLOSURE[x]

The **@CLOSURE[x]** system variables allow the BASIC program to access the CFSound and Expansion Module contact events.

There are up to 56 contact inputs; permissible values for [**x**] are 0 through 55.

Reading the variable returns a '1' if the input ***contact[x]*** has had a closure since last being read.

Closures are 'sticky' and the program must 'clear' the closure by assigning it a zero before it can be detected again.

Optionally an event handler specified with the **ONEVENT** statement may be activated upon an input closure, which automatically clears the closure.

```
10 ONEVENT @CLOSURE[0],GOSUB 100
20 ONEVENT @CLOSURE[1],GOSUB 200
30 GOTO 30
100 PRINT "contact 0 closed":RETURN
200 PRINT "contact 1 closed":RETURN
Ready
run
contact 0 closed
contact 1 closed
```

@OPENING[x]

The **@OPENING[x]** system variables allow the BASIC program to access the CFSound and Expansion Module contact events.

There are up to 56 contact inputs; permissible values for [**x**] are 0 through 55.

Reading the variable returns a '1' if the input ***contact[x]*** has had an opening since last being read.

Openings are 'sticky' and the program must 'clear' the opening by assigning it a zero before it can be detected again.

Optionally an event handler specified with the **ONEVENT** statement may be activated upon an input opening, which automatically clears the opening.

@PTT

Writing this system variable to a non-zero value activates the CFSound-IV PTT relay. Setting it to zero deactivates the PTT relay. Reading this system variable returns 1 if the PTT relay is active, else zero.

File Information

The following system variables provide information about previously opened files:

@FILE.SIZE[#N]

The **@FILE.SIZE[#N]** read-only system variable allows the BASIC program to determine the size in bytes of a previously opened file **#N**.

@FILE.POSITION[#N]

The **@FILE.POSITION[#N]** system variable allows the BASIC program to ascertain or set the position of the next file read or write operation of a previously opened file **#N**.

Back to Table of Contents

Files opened in read-only mode clip the set position to the size of the file. Files opened in write-only or update mode are extended if the position is set past the current size of the file.

@FEOF[#N]

The **@FEOF[#N]** system variable allows the BASIC program to determine when an end-of-file has occurred after an **FOPEN #N, INPUT #N, FREAD #N** or **FINPUT #N** statement.

Optionally an event handler specified with the **ONEVENT** statement may be activated upon an end-of-file occurring.

Socket Communications

The following system variables provide information about previously opened sockets:

@SOCKET.EVENT[#N]

The **@SOCKET.EVENT[#N]** system variable allows the BASIC program to determine the state of a previously opened streaming socket connection.

Optionally an event handler specified with the **ONEVENT** statement may be activated when an **SOCKET.EVENT** occurs. The **SOCKET.EVENT** values are:

SOCKET.EVENT[#N] value	Event	Description
0	None	No event
1	Disconnect / Done	Socket disconnected, connection done
2	Failed Open	SOCKET[#N] failed to open
3	Connection Timeout	No connection to socket within @SOCKET.TIMEOUT
4	Data Sent Timeout	No send data acknowledgment within @SOCKET.TIMEOUT
5	Data Received Timeout	No received data within @SOCKET.TIMEOUT

@SOCKET.TIMEOUT

This system variable allows the BASIC program to control the timeout period of socket connection, send data and receive data phases. This 16-bit signed value is copied into the timer variable and is decremented every 20mSEC (50 Hz) until it reaches zero.

Setting the value to zero is an immediate timeout.

A negative value is not decremented and results in an infinite timeout.

In the **BETA** software this system variable is now specified on a per socket connection:

@SOCKET.TIMEOUT[#N]

Back to Table of Contents

Real Time Clock

The following system variables provide access to the Real Time Clock / Calendar:

@SECOND / @MINUTE / @HOUR / @DOW / @DATE / @MONTH / @YEAR

Writing one of these variables except @SECOND stops the clock and updates the associated value.

Writing to the @SECOND variable updates the value and starts the clock running.

The values of these variables are updated once per second. Whenever one of the values of these variables changes, any associated event handler specified with the **ONEVENT** statement is activated.

See the ***Setting the Real Time Clock*** sample program in the Examples section for more information.

@SECOND	00 ≤ **seconds** ≤ 59
@MINUTE	00 ≤ **minutes** ≤ 59
@HOUR	00 ≤ **hour** ≤ 23
@DOW	0 ≤ **day of week** ≤ 6 (read-only, 0=Sunday)
@DATE	1 ≤ **date of month** ≤ 31
@MONTH	1 ≤ **month of year** ≤ 12
@YEAR	00 ≤ **year** ≤ 99

Sound Control

The following system variables provide control of the sound playout system:

@SOUND$

The **@SOUND$** system variable allows the BASIC program to queue sound files for playing.

A sound is queued by assigning the string value of the sound filename to the variable. The currently playing sound may be determined by reading the value of the variable. Queued sound files are played in the order that they were queued, being removed as they are played.

The queue may be flushed by assigning an empty string to the variable. When the queue becomes empty any associated event handler specified with the **ONEVENT** statement is activated.

Up to 128 sounds may be queued. ***Attempting to queue a sound when the queue is full results in an "Invalid .WAV file" error.***

Queued sounds play even if the BASIC program has stopped.

@VOL / @NSVOL

The **@VOL** and **@NSVOL** system variables allow the BASIC program to control the CFSound-IV volume.

The volume is set by assigning a numeric value to the variable. The current volume may be determined by reading the numeric value of the variable.

The range is 0 (mute) to 63 (max volume).

Note that the **@VOL** volume setting is saved in non-volatile memory and is restored every time the CFSound-IV powers up. ***The non-volatile memory has a limited number of write cycles (~100,000) and can be worn out by excessive writes so this function should not be used in a loop and with caution.*** The **@NSVOL** volume setting doesn't save the value in the non-volatile memory and doesn't have a use limit, however the volume will be restored to the last **@VOL** or pushbutton set value upon the next power-up or reset.

Back to Table of Contents

@MUTE

Writing this system variable to a non-zero value mutes the CFSound-IV speaker amplifier. Setting it to zero un-mutes the amplifier. Reading this system variable returns 1 if the amplifier is muted, else zero. The **RUN** command automatically un-mutes the speaker amplifier when the program is started.

@LINEIN

Writing this system variable to a non-zero value enables the CFSound-IV Line level Input. Setting it to zero disables the Line level input. Reading this system variable returns 1 if the line level input is enabled, else zero. The **RUN** command automatically disables the Line level input when the program is started. Audio on the Line level Input is amplified to the current volume level and is presented to the speakers and Line level Output when it is enabled and no other sound is playing.

@SOUNDFRAMEPRESCALER

This system variable sets the value of the number of 20mSEC (50Hz) ticks that elapse between **@SOUNDFRAMESYNC** events while a sound is playing.

@SOUNDFRAMEPRESCALER=1 yields 20 mSEC per frame

@SOUNDFRAMEPRESCALER=50 yields 1 SEC per frame

@SOUNDFRAMESYNC

This system variable returns the current frame number of the playing sound.

It starts at zero when a sound starts playing, and advances at the **@SOUNDFRAMEPRESCALER** rate. Due to implementation latency it can be off from 0 to 20mSEC from the actual start of the sound playing, but this offset should remain constant for the duration of the sound play out.

Optionally, an event handler specified with the **ONEVENT** statement may be activated whenever **@SOUNDFRAMESYNC** changes:

Back to Table of Contents

Serial Communications Control

The following system variables provide control over the serial communications:

@BAUD

The **@BAUD** system variables allow the BASIC program to control the CFSound-IV serial port baud rate.

The baud rate is set by assigning a numeric selector value to the variable. The current baud rate selector may be determined by reading the numeric value of the variable.

Note that the baud rate selector is saved in non-volatile memory and is restored every time the CFSound-IV powers up. ***The non-volatile memory has a limited number of write cycles (~100,000) and can be worn out by excessive writes so this system variable is only written to by BASIC it if it is not currently the desired value.***

@BAUD	Baud Rate
0	110
1	300
2	600
3	1200
4	1800
5	2400
6	3600
7	4800
8	7200
9	9600 (factory default)
10	14400
11	19200
12	28800
13	38400
14	57600
15	115200

@MSGENABLE

This system variable controls whether the serial data stream is parsed for messages as outlined in the **@MSG$** description below. It defaults to 1 (enabled).

The ability to disable **@MSG$** processing is required to support the **GETCH()** function on the serial port.

@MSG$

This system variable is updated by receipt of a serial data stream message that is framed with the **@SOM** and **@EOM** characters which are not included in the **@MSG$**.

@MSG$ retains the framed message ***until it is read*** at which point the search for the next received **@SOM** begins again.

Optionally an event handler specified with the **ONEVENT** statement may be activated upon receipt of the framed message.

@MSG$ may also be cleared by assigning it a string value, which is not saved. Note that this will reset any message accumulation that may be in process.

Back to Table of Contents

@SOM

This system variable determines the character used to delineate the Start of Message. It defaults to ASCII SOH (01). The Start of Message detection can be disabled by setting @SOM to zero, in which case the message accumulates until @EOM is detected only.

@EOM

This system variable determines the character used to delineate the End of Message. It defaults to ASCII ETX (03).

@EOT

This system variable returns 1 when any serial data sent by BASIC console operation, or **PRINT** statements has finished fully transmitting through the serial port.

It can be cleared by setting it to zero, but will immediately return 1 again unless serial data is actively being sent.

SMTP Control

The following system variables provide status of any Simple Mail Transfer Protocol operation in process:

@SMTP

These system variables are used to provide events and status when sending e-mail via the Simple Mail Transfer Protocol (SMTP). SMTP operation requires a properly configured network connection with a reachable mail server that supports either NON or AUTH LOGIN access.

@SMTP.EVENT

This system variable reflects the last SMTP event:

@SMTP.EVENT	Event
0	None
1	Status Update
2	Connection Aborted
3	Connection Failed
4	HELO / EHLO Failed
5	AUTH LOGIN Failed
6	FROM Failed
7	TO Failed
8	CC Failed
9	DATA Failed
10	QUEUE Failed
11	SUCCESS

@SMTP.MESSAGE$

This system variable holds any text message associated with the @SMTP.EVENT.

Back to Table of Contents

DMX Control

@DMX

These system variables are used to access the CFSound-IV's DMX512 functionality implemented via the Art-Net™ protocol.

@DMX.RESET

This system variable is provide for CFSound-III backwards compatibility but does nothing.

@DMX.MASTER

This system variable is provide for CFSound-III backwards compatibility but does nothing.

@DMX.FRAMEDELAY

This system variable is provide for CFSound-III backwards compatibility but does nothing.

@DMX.CHANNELS

This system variable is provide for CFSound-III backwards compatibility but does nothing.

@DMX.DATA[x]

Gets or sets the current value of channel x ($0 \le x \le 511$) if the optional DMX I/O module is present.

@DMXFRAMESYNC

This system variable is provide for CFSound-III backwards compatibility but does nothing.

Back to Table of Contents

Configuration Settings

@CONFIG

These system variables are used to access the CFSound configuration settings.

@CONFIG.ITEMS

This read-only system variable returns the total number of configuration setting items.

@CONFIG.TYPE[n]

This read-only system variable gets the type of the configuration item n:

@CONFIG.TYPE[n]	Item Type	Fields
1	Byte	0
2	Boolean	0
3	Unsigned short	0
4	Baudrate selector	0
5	Parity selector	0
6	Data Bits selector	0
7	Stop Bits selector	0
8	Keybeep selector	0
9	Firmware Version	0
10	Keypad style	0
11	Keypad scheme	0
12	Protocol selector	0
13	MAC address	6
14	IP address (only display if static)	4
15	IP address	4
16	Hex Byte	0
17	Hex Unsigned short	0
18	Hex Array	8
19	Short	0
20	RS485 Mode	0

@CONFIG.NAME$[n]

This read-only system variable gets the name of the configuration item n.

@CONFIG.VALUE$[n{, f}]

This read-only system variable gets the human readable value of the configuration item n. If the item has fields then the second argument specifies the zero-based field number.

@CONFIG.MIN[n]

This read-only system variable gets the allowed minimum value of the configuration item n.

@CONFIG.MAX[n]

This read-only system variable gets the allowed maximum value of the configuration item n.

Back to Table of Contents

@CONFIG.FIELDS[n]

This read-only system variable gets the number of fields of the configuration item n.

@CONFIG.FIELD$[n, f]

This read-only system variable gets the human readable value of the configuration item n's field f.

@CONFIG.SEPARATOR$[n, f]

This read-only system variable gets the human readable value of the configuration item n's field f separator.

@CONFIG.VALUE[n{, f}]

This system variable gets or sets the value of the configuration item n. If the item has fields then the second argument specifies the zero-based field number.

@CONFIG.DEFAULT[n{, f}]

This read-only system variable gets the default value of the configuration item n. If the item has fields then the second argument specifies the zero-based field number.

@CONFIG.WRITE[n]

Changes to the configuration settings affect the current value of RAM copies of the entries. Setting this write-only system variable to one forces the current value of item n to be written to the NVM backup store so that it will be remembered between restarts.

SD Card Control

@CARD.MOUNT

This system variable is used to mount and un-mount the SD card. If the SD card is present when BASIC is started it is automatically mounted. Writing a value of 0 un-mounts the card, 1 mounts it.

Graphics System Variables

Additional system variables supporting and controlling graphics are outlined in the **BASIC Graphics Programming** manual available online.

Back to Table of Contents

Operators

BASIC supports the following operators listed in priority from highest to lowest. Operators encountered during statement execution are evaluated in order of priority with higher priority operators executed before lower priority operators.

Operators work between a left and right operand – unary operators only work on a right, following operand.

Operator	Description	Priority
NOT	Logical NOT	7
-	Unary minus (negate, 2's complement)	7
~	Unary Bitwise NOT (1's complement)	7
*** / %**	Multiplication, division, modulus	6
+	Addition, string concatenation	5
-	Subtraction	5
<< >>	Left Shift, Right Shift	4
= <>	Assign / test equal, test NOT equal (numeric or string)	3
< <= > >=	Test Less Than, Less than or Equal, Greater Than, Greater than or Equal (numeric or string)	3
& | ^	Bitwise AND, OR, Exclusive OR	2
AND OR	Logical AND, OR	1

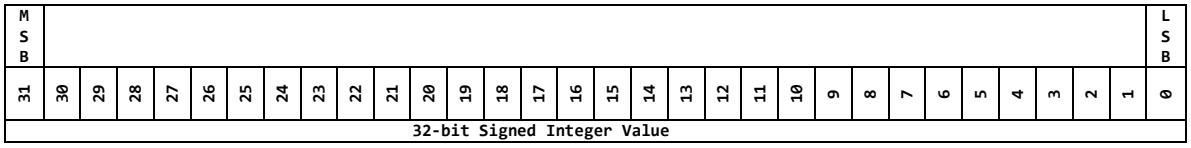
In the **BETA** software the modulus operator (%) is changed to MOD to make the percent sign available as a real variable name indicator.

Parenthesis may be used to change or enforce expression execution priority with the innermost grouped parenthesis expression evaluated first.

The six 'test' relational operators **(=, <>, <, <=, >, >=**) can be used in any expression, and evaluate to 1 if the tested condition is TRUE, and 0 if it is FALSE. The IF and LIF commands accept any non-zero value to indicate a TRUE condition.

Multiple 'test' operators can be combined with the logical NOT, AND, OR operators and suitable parenthesis.

There are six operators for bit manipulation (**~, &, |,** ^**,** <<**,** >>); these may only be applied to integer operands. The 32 'bit' positions in the integer are numbered from right to left starting with 0 (the **L**east **S**ignificant **B**it) up to 31 (the **M**ost **S**ignificant **B**it) or sign bit:



Thus the decimal value 1234 in binary 32-bit form is:

#	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1234	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**1**	**1**	**0**	**1**	**0**	**0**	**1**	**0**

And the decimal value -1234 in binary 32-bit form is:

#	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Back to Table of Contents

-1234	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	0	0	1	0	1	1	1	0

The bitwise ~ unary operator yields the one's complement of its following integer operand; that is, it converts each 1-bit into a 0-bit and vice versa. Thus the value ~1234 in binary 32-bit form is:

#	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
~1234	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	0	1	1	0	1	1	0	1

Note that each bit position in the ~1234 is inverted from their 1234 values.

The bitwise & (and) operator is often used to mask off or clear some set of bits. This can be used to determine which bits are set by and'ing a value with the mask of the bit to examine. So the value 1234 bitwise and'ed with 255 is 210:

#	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1234	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	1	0	1	0	0	1	0
& 255	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
= 210	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	1	0	0	1	0

```
print 1234 & 255
210
Ready
```

The bitwise | (or) operator is used to turn on or set some set of bits. So the value 1234 bitwise or'ed with 255 is 1279:

#	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1234	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	1	0	1	0	0	1	0
| 255	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
= 1279	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	1	1	1	1	1	1	1

```
print 1234 | 255
1279
Ready
```

The bitwise ^ (exclusive or) operator sets a one in each bit position where its operands have different bits, and zero where they are the same. This can be used to toggle specific bits by xor'ing a value with the bits to toggle. So the value 1234 xor'ed with 255 is 1069:

#	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1234	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	1	0	1	0	0	1	0
^ 255	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
= 1069	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	1	1	0	1

```
print 1234 ^ 255
1069
Ready
```

The bitwise << and >> perform left and right shifts of their left operand by the number of bit positions given by their right operand, which must be positive. Vacated bits on the right are filled by zeroes, vacated bits on the left are filled with the value of the sign bit.

Back to Table of Contents

The bitwise << (left shift) shifts the bits towards the left from LSB towards MSB, filling in the vacated LSB positions with zero bits. This is the same as multiplying by 4. Thus 1234 << 2 = 4936:

#	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1234	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	1	0	1	0	0	1	0
<< 2	←	←	←	←	←	←	←	←	←	←	←	←	←	←	←	←	←	←	←	←	←	←	←	←	←	←	←	←	←	←	←	←
= 4936	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	1	0	1	0	0	1	0	0	0

```
print 1234 << 2
4936
Ready
```

The bitwise >> (right shift) shifts the bits towards the right from MSB towards LSB, filling in the vacated MSB positions with copies of the sign bit 31. This is the same as dividing by 4. Thus1234 >> 2 = 308:

#	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1234	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	1	0	1	0	0	1	0
>> 2	→	→	→	→	→	→	→	→	→	→	→	→	→	→	→	→	→	→	→	→	→	→	→	→	→	→	→	→	→	→	→	→
= 308	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	1	0	1	0	0

```
print 1234 >> 2
308
Ready
```

Since the bits filling into the vacated MSB positions are copies of the sign bit, bit 31, then -1234 >> 2 = -309:

#	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
-1234	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	0	0	1	0	1	1	1	0
>> 2	→	→	→	→	→	→	→	→	→	→	→	→	→	→	→	→	→	→	→	→	→	→	→	→	→	→	→	→	→	→	→	→
= -309	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	0	0	1	0	1	1

```
print -1234 >> 2
-309
Ready
```


Back to Table of Contents

Expressions

In BASIC expressions consist of one or more variables, constants, functions or system variables that may optionally be joined together by Operators.

The evaluation order may be controlled by the judicious use of parenthesis.

Expressions may be nested up to 10 levels. Some examples:

```
a=10
Ready
print a*30
300
Ready
print fmt$("%02X", a)
0A
Ready
print a<<2
40
Ready
print (a<<2)=0
0
Ready
print (a<<2)<>0
1
Ready
print a^4
14
Ready
```

Functions

BASIC provides several built-in functions that may be used in expressions.

There must not be a space between the function name and the opening parenthesis.

Functions must be used in a statement such as a **LET** or **PRINT** – they cannot be executed standalone in immediate mode.

ASC(char)

Return the numeric ASCII value of the character argument.

```
PRINT ASC("A")
65
```

ABS(expr)

Return the absolute value of the numeric argument.

```
PRINT ABS(10), ABS(-10)
10 10
```

CHR$(expr)

Return an ASCII string containing the character equivalent of the expression argument.

```
PRINT CHR$(65)
A
```


Back to Table of Contents

COS(degrees)

Return a scaled sine value of the degree argument where -1024 ≤ **COS( )** ≤ 1024. The degree argument ranges from 0 → 360 and arguments larger than 360 degrees are converted modulo 360.

COS(0) = 1024, COS(90) = 0, COS(180) = -1024, COS(270) = 0, etc..

```
10 FOR d = 0 TO 360 STEP 15
20  integer = MULDIV(1000,COS(d),1024)
25  IF integer < 0 THEN sign$ = "-" ELSE sign$ = ""
27  integer = ABS(integer / 1000)
30  fraction = ABS(MULDIV(1000,COS(d),1024) % 1000)
40  PRINT "COS(";d;") = ";sign$;integer;".";fraction
50 NEXT d
Ready
run
COS(0) = 1.0
COS(15) = 0.965
COS(30) = 0.865
COS(45) = 0.707
COS(60) = 0.500
COS(75) = 0.258
COS(90) = 0.0
COS(105) = -0.258
COS(120) = -0.500
COS(135) = -0.707
COS(150) = -0.865
COS(165) = -0.965
COS(180) = -1.0
COS(195) = -0.965
COS(210) = -0.865
COS(225) = -0.707
COS(240) = -0.500
COS(255) = -0.258
COS(270) = 0.0
COS(285) = 0.258
COS(300) = 0.500
COS(315) = 0.707
COS(330) = 0.865
COS(345) = 0.965
COS(360) = 1.0
Ready
```

ERR( )

Return the last error number.

ERR$( )

Return the string representation of the last error number.

FILE.EXISTS(“path”)

Return ‘1’ if the file specified by the “path” argument exists, otherwise ‘0’.

FIND(expr$, searchexpr$ {, startpos})

Return the zero based position of string **search expr**ession in string **expr**ession starting at zero (or optional **startpos)** or -1 if the **search expr**ession was not found.

```
TEST$="012345" : PRINT FIND(TEST$, "3")
3
TEST$="012345" : PRINT FIND(TEST$, "3", 4)
-1
```


Back to Table of Contents

FMT$(fmt$ {, expr{$}, expr{$} … , expr{$}}})

Return a formatted ASCII string of zero or more numeric or string **expr**essions using the string format specification **fmt$**.

A format specification string expression **fmt$** consists of zero or more **{**optional**}** and required fields and has the following form:

%{Flags}{Width}{.Precision}Type

Each field of a format specification is a single character or a number signifying a particular format option. The simplest format specification contains only the percent sign and a **type** character (for example, %d). If a percent sign is followed by a character that has no meaning as a format field, the character is copied to the return value. For example, to produce a percent sign in the return value, use %%.

The optional fields, which appear before the required **type** character, control other aspects of the formatting, as follows:

<table>
<tr><td>Type</td><td>Required character that determines whether the associated argument is interpreted as a character, a string, or a number:
c character — s string
d signed decimal integer — o unsigned octal integer
i signed decimal integer — x unsigned hexadecimal integer
u unsigned decimal integer — X unsigned HEXADECIMAL integer
In the BETA software:
f real number {-}ddd.ddd — e real number {-}d.ddde±dd
E real number {-}d.dddE±dd — G,g real number {-}ddd.ddd or {-}d.ddd±edd</td></tr>
<tr><td>Flags</td><td>Optional character or characters that control justification of output and printing of signs, blanks, and octal and hexadecimal prefixes. More than one flag can appear in a format specification.
<table>
<tr><td>-</td><td>left align the result in the given field width</td></tr>
<tr><td>+</td><td>prefix the output with a sign (+/-) if the type is signed</td></tr>
<tr><td>0</td><td>if Width is prefixed with 0, zeroes are added until the minimum width is reached. If 0 and – appear, the 0 is ignored. If 0 is specified with an integer format, the 0 is ignored.</td></tr>
<tr><td>blank(‘ ’)</td><td>prefix the output with a blank if the result is signed and positive; the blank is ignored if both the blank and + flags appear</td></tr>
<tr><td>#</td><td>when used with o, x or X format, prefix any nonzero output value with 0, 0x or 0X respectively, otherwise ignored, when used with e, E, f, g and G format a decimal point is output even if the output has no fractional part</td></tr>
</table></td></tr>
<tr><td>Width</td><td>Nonnegative decimal integer controlling the minimum number of characters printed. If the number of characters in the output value is less than the specified width, blanks are added to the left or the right of the values — depending on whether the – flag (for left alignment) is specified — until the minimum width is reached. If Width is prefixed with 0, zeroes are added until the minimum width is reached (not useful for left-aligned numbers). The Width specification never causes a value to be truncated. If the number of characters in the output value is greater than the specified width, or if Width is not given, all characters of the value are printed (subject to the Precision specification).</td></tr>
<tr><td>Precision</td><td>Specifies a nonnegative decimal integer, preceded by a period (.), which specifies the number of characters to be printed, the number of decimal places, or the number of significant digits. Unlike the Width specification, the precision specification can cause truncation of the output value. If Precision is specified as 0 and the value to be converted is 0, the result is no characters output.
<table>
<tr><td>c</td><td>Precision has no effect</td></tr>
<tr><td>d,i,u,o, x,X</td><td>Precision specifies the minimum number of digits to be output. If the number of digits is less than Precision, the output is padded on the left with zeroes. The value is not truncated when the number of digits exceeds Precision</td></tr>
<tr><td>s</td><td>Precision specifies the maximum number of characters to be output. Characters in excess of Precision are not output</td></tr>
<tr><td colspan="2">In the BETA software:</td></tr>
<tr><td>e, E, f</td><td>Precision specifies the number of digits after the decimal point</td></tr>
<tr><td>g, G</td><td>Precision specifies the number of significant digits</td></tr>
</table></td></tr>
</table>

Back to Table of Contents

```
10 REM show time
15 ONEVENT @SECOND,GOSUB 100
20 GOTO 20
100 PRINT FMT$("%c%2d:%02d:%02d",13,@HOUR,@MINUTE,@SECOND);
105 RETURN
Ready
run
13:30:13 <<< ESC at line 20 >>>
Ready
```

GETCH(expr)

If ***expr*** evaluates to zero, **GETCH(0)** returns the numeric value of the next available serial character or it returns a -1 if no character is currently available from either enabled source.

If ***expr*** evaluates to non-zero, **GETCH(1)** waits for the next available serial character and then returns its numeric value.

This function requires that **@MSGENABLE = 0** to operate correctly.

HEX.STR$(expr{, digits})

Return a string containing the hexadecimal representation of the ***expr***ession. The optional digits parameter specifies the number of digits.

```
PRINT HEX.STR$(43690)
AAAA
Ready
PRINT HEX.STR$(43690,6)
00AAAA
Ready
```

HEX.VAL(expr$)

Return the numeric value of the hexadecimal string **expr**ession.

```
PRINT HEX.VAL("AAAA")
43690
Ready
```

INSERT$(expr$, start, expr2$)

Return a string with the contents of **expr**ession2 inserted in string **expr**ession at zero based position ***start***.

```
10 REM test insert$
20 s$ ="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
30 i$ ="insert"
35 REM insert at beginning
40 PRINT INSERT$(s$,0,i$)
45 REM insert in middle
50 PRINT INSERT$(s$,13,i$)
55 REM insert past end
60 PRINT INSERT$(s$,30,i$)
Ready
run
insertABCDEFGHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMinsertNOPQRSTUVWXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZinsert
Ready
```

LEFT$(expr$, len)

Return a string containing the leftmost **len**gth characters of string **expr**ession.

Back to Table of Contents

```
TEST$="This is a string" : PRINT LEFT$(TEST$,5)
This
```

LEN(expr$)

Return the length (number of characters) of string **expr**ession.

```
TEST$="This is a string" : PRINT LEN(TEST$)
16
```

MATH.ABS%(expr%) BETA software

Return the absolute value of a real **expr**ession.

MATH.ACOS%(expr%) BETA software

Return the arc cosine of a real expression in radians as 0 ≤ radians ≤ Π. A Run Time Library domain error occurs if expression is not between -1.0 and +1.0.

MATH.ASIN%(expr%) BETA software

Return the arc sine of a real expression in radians as -Π /2 ≤ radians ≤ Π/2. A Run Time Library domain error occurs if expression is not between -1.0 and +1.0.

MATH.ATAN%(expr%) BETA software

Return the arc tangent of a real expression in radians as -Π /2 ≤ radians ≤ Π/2.

MATH.COS%(expr%) BETA software

Return the cosine of a real expression in radians.

MATH.SIN%(expr%) BETA software

Return the sine of a real expression in radians.

MATH.TAN%(expr%) BETA software

Return the tangent of a real expression in radians.

MATH.EXP%(expr%) BETA software

Return the exponential of a real expression.

MATH.LOG%(expr%) BETA software

Return the natural logarithm of a real expression. A Run Time Library domain error occurs if the expression is negative. A Run Time Library range error occurs if the expression is 0.0.

MATH.LOG10%(expr%) BETA software

Return the base 10 logarithm of a real expression. A Run Time Library domain error occurs if the expression is negative. A Run Time Library range error occurs if the expression is 0.0.

MATH.POW%(exprX%, exprY%) BETA software

Return the power of real expression X raised to the power expression Y. A Run Time Library domain error occurs if X is 0.0 and Y is 0.0 or less, or if X is negative.

MATH.SQRT%(expr%) BETA software

Return the positive square root of a real expression.

MATH.CEIL%(expr%) BETA software

Return the smallest integral value not less than or equal to the real expression.

Back to Table of Contents

MATH.FLOOR%(expr%) BETA software

Return the largest integral value not greater than or equal to the real expression.

MATH.MIN(expr1, expr2) BETA software

Return the smallest of integer expression1 and expression2.

MATH.MIN%(expr1%, expr2%) BETA software

Return the smallest of real expression1 and expression2.

MATH.MAX(expr1, expr2) BETA software

Return the largest of integer expression1 and expression2.

MATH.MAX%(expr1%, expr2%) BETA software

Return the largest of real expression1 and expression2.

MATH.FLOAT%(expr{%}) BETA software

Return the real value of an integer or real expression.

MATH.INT(expr{%}) BETA software

Return the integer value of an integer or real expression.

MATH.PI% BETA software

Return the value of Π as a real expression = 3.14159265359.

Back to Table of Contents

MID$(expr$, start, len)

Return a string consisting of **len**gth number of characters of string **expr**ession from zero based ***start*** character position.

```
TEST$="This is a string" : PRINT MID$(TEST$,5,4)
is a
```

MULDIV(number, multiplier, divisor)

Return a 32 bit result of ((number * multiplier) / divisor) where number, multiplier and divisor are 64-bit internally. Useful for calculating percentages, etc., where the normal multiply would overflow a signed 32-bit number.

```
10 REM calculate 55 percent of 999
20 PRINT MULDIV(999,55,100);".";MULMOD(999,55,100)
Ready
run
549.45
```

MULMOD(number, multiplier, divisor)

Return a 32 bit result of ((number * multiplier) % divisor) where number, multiplier and divisor are 64-bit internally. Useful for calculating remainders of percentages, etc., where the normal multiply would overflow a signed 32-bit number.

RIGHT$(expr$, len)

Return a string containing the rightmost **len**gth characters of string **expr**ession.

```
TEST$="This is a string" : PRINT RIGHT$(TEST$,6)
string
```

REPLACE$(expr$, start, expr2$)

Return a string with the contents of **expr**ession**2** overwritten on string **expr**ession at zero based position ***start***.

```
10 REM test replace$
20 s$ ="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
30 r$ ="replace"
35 REM replace at beginning
40 PRINT REPLACE$(s$,0,r$)
45 REM replace in middle
50 PRINT REPLACE$(s$,13,r$)
55 REM replace past end
60 PRINT REPLACE$(s$,30,r$)Ready
run
replaceHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMreplaceUVWXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZreplace
Ready
```


Back to Table of Contents

RND(expr)

Return a pseudo random number that ranges from 0 to (**expr**ession - 1).

```
10 FOR i= 0 TO 10 : PRINT RND(10);" "; : NEXT i
Ready
run
3 0 4 3 8 3 1 1 3 4 3 Ready
```

SIN(degrees)

Return a scaled sine value of the degree argument where -1024 ≤ **SIN( )** ≤ 1024. The degree argument ranges from 0 → 360 and arguments larger than 360 degrees are converted modulo 360.

SIN(0) = 0, SIN(90) = 1024, SIN(180) = 0, SIN(270) = -1024, etc..

```
10 FOR d = 0 TO 360 STEP 15
20  integer = MULDIV(1000,SIN(d),1024)
25  IF integer < 0 THEN sign$ = "-" ELSE sign$ = ""
27  integer = ABS(integer / 1000)
30  fraction = ABS(MULDIV(1000,SIN(d),1024) % 1000)
40  PRINT "SIN(";d;") = ";sign$;integer;".";fraction
50 NEXT d
Ready
run
SIN(0) = 0.0
SIN(15) = 0.258
SIN(30) = 0.500
SIN(45) = 0.707
SIN(60) = 0.865
SIN(75) = 0.965
SIN(90) = 1.0
SIN(105) = 0.965
SIN(120) = 0.865
SIN(135) = 0.707
SIN(150) = 0.500
SIN(165) = 0.258
SIN(180) = 0.0
SIN(195) = -0.258
SIN(210) = -0.500
SIN(225) = -0.707
SIN(240) = -0.865
SIN(255) = -0.965
SIN(270) = -1.0
SIN(285) = -0.965
SIN(300) = -0.865
SIN(315) = -0.707
SIN(330) = -0.500
SIN(345) = -0.258
SIN(360) = 0.0
Ready
```

STR$(expr)

Return a string representation of the numeric argument.

```
TEST$ = STR$(1234) : PRINT TEST$
1234
```


Back to Table of Contents

SOCKET.SYNC.CONNECT(#N, "ip:port", connect( ), send( ), recv( ))

Initiate an outgoing synchronous network socket connection as file **#N** using the string representation of the IPv4 IP address and port number. The status of the connection, send, receive and disconnect process is returned as the value of this numeric function which can return the following values:

SOCKET.SYNC.CONNECT( ) returns	Description
0	Unknown / No Status
1	Disconnect / Done / No Error
2	Open Error – requested socket failed to open
3	Connection Timeout – no connection within the **@SOCKET.TIMEOUT** interval
4	Data Send Timeout – no send data acknowledgment within the **@SOCKET.TIMEOUT** interval
5	Receive Data Timeout – no received data within the **@SOCKET.TIMEOUT interval**

- The **connect( )** user function is called when a connection is established.
- The **send( )** user function is called to send data to the connected device using **PRINT #N** or **FPRINT #N** statement(s). Return zero to terminate the connection, one to proceed to the **recv( )** function or two to be called again to send more data.
- The **recv( )** user function is then called to receive data from the connected device using **INPUT #N** or **FINPUT #N** statements(s). Return zero to terminate the connection, one to return to the **send( )** function and two to be called again to receive more data.

See the Socket Programming section below for more information and sample programs.

SOCKET.SYNC.LISTEN(#N, ":port", connect( ), recv( ), send( ))

Initiate an incoming synchronous network socket reception as file **#N** using the string representation of the IPv4 IP port number. The status of the connection, send, receive and disconnect process is returned as the value of this numeric function which can return the following values:

SOCKET.SYNC.CONNECT( ) returns	Description
0	Unknown / No Status
1	Disconnect / Done / No Error
2	Open Error – requested socket failed to open
3	Connection Timeout – no connection within the **@SOCKET.TIMEOUT** interval
4	Data Send Timeout – no send data acknowledgment within the @**SOCKET.TIMEOUT** interval
5	Receive Data Timeout – no received data within the **@SOCKET.TIMEOUT** interval

- The **connect( )** user function is called when a connection is established.
- The **recv( )** user function is then called to receive data from the connected device using **INPUT #N** or **FINPUT #N** statements(s). Return zero to terminate the connection, one to return to the **send( )** function and two to be called again to receive more data.
- The **send( )** user function is called to send data to the connected device using **PRINT #N** or **FPRINT #N** statement(s). Return zero to terminate the connection, one to proceed to the **recv( )** function and two to be called again to send more data.

See the Socket Programming section below for more information and sample programs.

Back to Table of Contents

UBOUND(dimVariable{[dimNumber]})

Return the size of the **dimVariable** as it was declared in the **DIM** statement. The optional **dimNumber** in square brackets defaults to 0 and can be 0, 1 or 2 to obtain the size of the corresponding dimension.

```
10 REM test multidimensional arrays
15 DIM test[3,4,5]
20 FOR x = 0 TO UBOUND(test[0])-1
25  FOR y = 0 TO UBOUND(test[1])-1
27   FOR z = 0 TO UBOUND(test[2])-1
30    test[x,y,z] = x * y + z
32   NEXT z
35  NEXT y
40 NEXT x
45 PRINT "test[";UBOUND(test[0]);",";UBOUND(test[1]);",";UBOUND(test[2]);"] ="
50 FOR x = 0 TO UBOUND(test[0])-1
55  FOR y = 0 TO UBOUND(test[1])-1
57   FOR z = 0 TO UBOUND(test[2])-1
60    PRINT test[x,y,z];",";
62   NEXT z
63   PRINT ""
65  NEXT y
67  PRINT ""
70 NEXT x
Ready
run
test[3,4,5] =
0,1,2,3,4,
0,1,2,3,4,
0,1,2,3,4,
0,1,2,3,4,

0,1,2,3,4,
1,2,3,4,5,
2,3,4,5,6,
3,4,5,6,7,

0,1,2,3,4,
2,3,4,5,6,
4,5,6,7,8,
6,7,8,9,10,

Ready
```

VAL(expr$)

Return the numeric value of the string argument representation of a number. A leading zero forces the string argument to be interpreted as octal and a leading 0x or 0X forces the string argument to be interpreted as hexadecimal.

```
TEST = VAL("1234") : PRINT TEST
1234
Ready
PRINT VAL("08")
0
Ready
PRINT VAL("0377")
255
Ready
PRINT VAL("0xA0")
160
Ready
```

Graphics Support Functions

Additional commands supporting graphics are outlined in the **BASIC Graphics Programming** manual available online.

Back to Table of Contents

Statements

BASIC program lines consist of an optional integer line number followed by one or more statements. Multiple statements on a line are allowed, separated by a colon ('**:**'). Only the first statement on a line may have a line number or label. Here are some sample program statements:

```
10 REM This is a comment
20 FOR I=0 TO 10 : PRINT I : NEXT I
30 `WaitHere : IF @CLOSURE[24] = 0 THEN `WaitHere
. . .
```

Some statements are annotated as "Direct mode only" in their description and can only be executed immediately when entered without a line number. These are also referred to as commands.

In a similar fashion, some statements can only be executed from within an executing program and are annotated as "Program mode only" in their description.

Statements not marked as Direct or Program mode only may be used in either fashion.

The statement keywords are 'tokenized' when entered to save program memory and speed program execution: *ie:* the keyword **GOSUB** would be tokenized to a single byte instead of five bytes. In addition, each statement line number is converted to a four-byte unsigned integer form to save space and facilitate program execution. Saved programs are expanded (un-tokenized) on the SD card to allow program storage, viewing and editing with an external text editor if required.

The following section identifies what statement keywords are supported, describes how they may be used and whether they can be used in Program mode, Direct mode or both.

Back to Table of Contents

BREAK {line} / BREAK {`label}

Program mode only. Exit from within the innermost enclosing **FOR / NEXT** or **WHILE / WEND** loop – execution continues after the closest **NEXT** or **WEND** statement.

If the optional **line** or **`label** is present execution continues there.

```
10 REM for/break/next test
15 FOR i = 0 TO 10
20  IF i > 5 THEN BREAK `out
25  PRINT i
30 NEXT i
35 END
40 `out : PRINT "Done"
Ready
run
0
1
2
3
4
5
Done
Ready
```

```
10 REM while/break/wend test
15 WHILE a < 10
20  IF a > 5 THEN BREAK
25  PRINT a
30  a = a + 1
35 WEND
Ready
run
0
1
2
3
4
5
Ready
```

A **BREAK** statement can be used to exit a **FOR** / **NEXT** or **WHILE** / **WEND** loop from within a block **IF** statement.

```
10 REM for/break/next test
15 FOR i = 0 TO 10
20  IF i > 5 THEN
25   BREAK
30  ENDIF
35  PRINT i
40 NEXT i
Ready
run
0
1
2
3
4
5
Ready
```

```
10 REM while/break/wend test
15 WHILE a < 10
20  IF a > 5 THEN
25   BREAK
30  ENDIF
35  PRINT a
40  a = a + 1
45 WEND
Ready
run
0
1
2
3
4
5
Ready
```

You can only **BREAK** to a line or label that is outside of the innermost enclosing loop – jumping beyond an outer enclosing loop will cause a **NEXT** without matching **FOR**, **WEND** without matching **WHILE** or stack error when those statements are reached.

```
10 REM illegal for/break/next test
15 FOR x = 0 TO 2
20  FOR y = 0 TO 2
25   FOR z = 0 TO 2
30    IF z > 1 THEN BREAK 50
35    PRINT x,y,z
40   NEXT z
45  NEXT y
50 NEXT x
55 END
60 PRINT "break"
Ready
run
0 0 0
0 0 1
Nesting error in line 50 - NEXT var doesn't match FOR var
Ready
```


Back to Table of Contents

CHANGE string, replacement

Direct mode only. Search the entire program for a case sensitive match of **string** and then prompts Yes/No/All/eXit for **replacement**.

If either string or replacement contains a space enclose both in double-quotes and separate them with a comma:

```
change send_200 send_200_header
15 CONST send_200=0, send_content=1, send_file=2, send_404=3, send_disconnect=4
         ^ = send_200_header (Yes/No/All/eXit) ?y
15 CONST send_200_header=0, send_content=1, send_file=2, send_404=3, send_disconnect=4
40 recvdata$="" : senddata$="" : sendstate=send_200 : line=0
                                        ^ = send_200_header (Yes/No/All/eXit) ?a
40 recvdata$="" : senddata$="" : sendstate=send_200_header : line=0
1150 line=0:ONERROR GOTO 1152 : sendstate=send_404 : OPEN #1, filename$,"r" : sendstate=send_200_header : ONERROR GOTO 0
Ready
```

CLEAR

Erase all variables and close all open files.

CLOSE #N

Close file or internet streaming handle #***N*** (0 → 24) opened with **OPEN #N** statement.

CONST var{$}=value {, var{$}=value … }

Create one or more constant variables that can't be assigned to after they are created.

This statement is useful to replace numeric or string constants in a program with a named value that may make the program easier to understand – and configurable in one place instead of throughout the program.

```
10 REM define CONSTants
20 CONST Limit = 10, Abort$ = "Abort!"
30 Limit = 20
Ready
run
Read Only error in line 30
Ready
vars
Limit  -> r/o Int          = 10
Abort$ -> r/o Str$         = "Abort!"
Ready
```


Back to Table of Contents

CONTINUE

Program mode only. Causes execution to pass to the end of the innermost enclosing **FOR / NEXT** or **WHILE / WEND** loop - execution continues at the closest **NEXT** or **WHILE** statement.

```
10 REM for/continue/next test
15 FOR i = 1 TO 10
20  IF (i & 1) THEN CONTINUE
25  PRINT i
30 NEXT i
Ready
run
2
4
6
8
10
Ready
```

```
10 REM while/continue/wend test
15 WHILE a < 10
20  a = a + 1
25  IF (a & 1) THEN CONTINUE
30  PRINT a
35 WEND
Ready
run
2
4
6
8
10
Ready
```

A **CONTINUE** statement can be used to continue a **FOR** / **NEXT** or **WHILE** / **WEND** loop from within a block **IF** statement:

```
10 REM for/continue/next test
15 FOR i = 1 TO 10
20  IF (i & 1) THEN
25   CONTINUE
30  ENDIF
35  PRINT i
40 NEXT i
Ready
run
2
4
6
8
10
Ready
```

```
10 REM while/continue/wend test
15 WHILE a < 10
20  a = a + 1
25  IF (a & 1) THEN
30   CONTINUE
35  ENDIF
40  PRINT a
45 WEND
Ready
run
2
4
6
8
10
Ready
```


Back to Table of Contents

DATA {“}value{“}{, {“}value{“}, … {“}value{“}}

Program mode only. Enter "inline" **DATA** statements holding numeric or string values that can be accessed by **READ** and **ORDER** statements. All related **DATA** statements should be in a group of sequential lines. String values should be enclosed in double quotes.

When BASIC comes to a **READ** statement containing a list of variables it looks for a **DATA** statement that was previously located by the use of an **ORDER** statement. It then assigns the variables in **READ** statement the values from the **DATA** statement, one at a time in the order listed. Items in the **READ** and **DATA** statements are separated by commas.

```
10 REM data/read/order
15 ORDER 20
20 DATA 2, "test", 3, "next"
30 READ x, y$
40 PRINT x, y$
50 READ x, y$
60 PRINT x, y$
70 READ x, y$
80 PRINT x, y$
Ready
run
2 test
3 next
Out of Data error in line 70
Ready
```

DEL path

Delete files and/or directories on the SD card. The full ***path*** must be specified. Directories must be empty to be deleted.

In program mode ***Path*** may be a constant string or you can use a string variable as the ***path*** by concatenating it with an empty string.

```
10 REM del
15 TYPE test.txt
20 DEL test.txt
25 TYPE test.txt
Ready
run
Lorem ipsum dolor sit amet, consectetur adipiscing
elit.

File not open error in line 25 - test.txt: No File
Ready
```

```
10 REM del
15 File$ = "test.txt" : TYPE ""+File$
20 DEL ""+File$
25 TYPE ""+File$
Ready
run
Lorem ipsum dolor sit amet, consectetur adipiscing
elit.

File not open error in line 25 - test.txt: No File
Ready
```

In direct mode the quotation marks are not required.

DELAY value

Pause program execution for value * 20mSEC. The delay timer is 16-bits.

While the delay is in process, events can occur but any defined **ONEVENT** handlers will not be executed until the delay has expired. The maximum **DELAY** value is 32767 providing a 655.34 second interval.

```
10 REM delay for one second
20 DELAY 50
```


Back to Table of Contents

DIM var{$} [size{, size1{, size2}}]

Define a numeric or character string array **var**iable to hold **size** integers or character strings. Arrays may be defined with up to three dimensions shown as **size**, **size1** and **size2**. A minimum single dimension is required.

Array variable elements may then be accessed using numeric indices separated by commas and enclosed in square brackets that range from the first element of zero to the last element of each dimension: A[0], A[1], … , A[size - 1].

If an attempt is made to access a variable as an array before it has been dimensioned a "Dimension Error" will result.

If an attempt is made to access an array element with a negative index or an index beyond the currently defined array dimensions an "Index Out of Range Error" will result.

A variable may be re-dimensioned, however the current contents of the variable will be lost.

```
10 REM multidimensional arrays
15 DIM test[3,4,5]
20 FOR x = 0 TO 2
25  FOR y = 0 TO 3
27   FOR z = 0 TO 4
30    test[x,y,z] = x * y + z
32   NEXT z
35  NEXT y
40 NEXT x
Ready
```

DIR {path}

Show a directory of files present on the SD card. An optional ***path*** may be specified. Wildcard characters '?' and '*' may be used in the ***path*** with a '?' matching any single character and/or a '*' matching multiple characters to show multiple files matching the pattern.

```
dir s*.wav
S7DB.WAV   323,028 A      08-08-2013 12:36:48 PM
S7EB.WAV   306,060 A      05-04-2006 12:39:00 PM
S7FB.WAV 1,164,232 A      05-04-2006 12:42:00 PM
------------------
                3 file(s)      1,793,320 bytes
                0 dir(s)   7,929,528,320 bytes free
Ready
```

DIR #N, {path}

Write a directory of files on the SD card to a previously opened file #***N*** (0 → 24). An optional ***path*** may be specified. Wildcard characters '?' and '*' may be used in the ***path*** with a '?' matching any single character and/or a '*' matching multiple characters to show multiple files matching the pattern.

Back to Table of Contents

EDIT line

Direct mode only. Using an ANSI terminal allows editing a line by displaying the statement, moving the cursor with the Home, Left arrow, Right arrow and End keys.

The Backspace key is used to delete characters to the left of the cursor. Typed characters are entered at the cursor. The Enter key accepts the changes, a double ESC key aborts the edit.

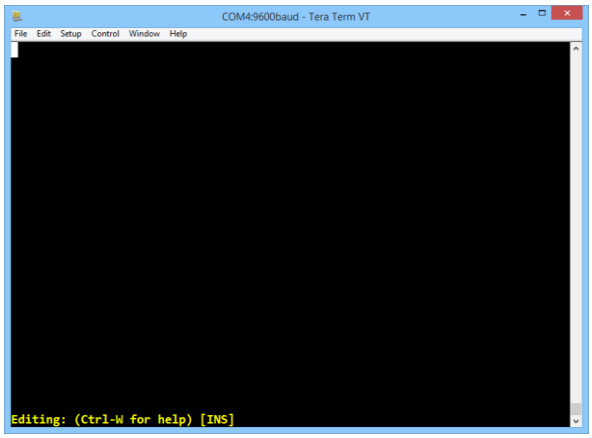
Experimental Full Screen Editor

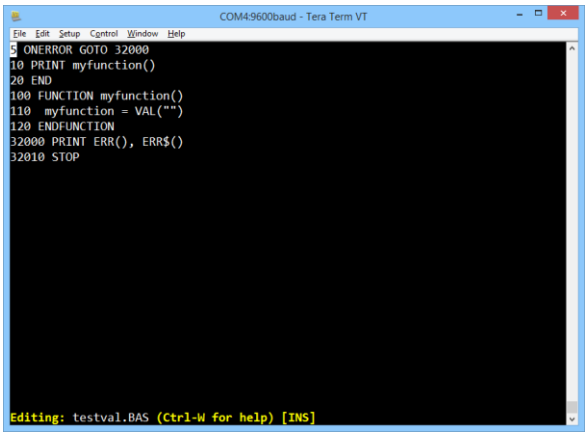
Available as a compilation option in the **BETA** software the **EDIT** command is enhanced:

EDIT / EDIT {filename} / EDIT "filename"

Requires use of a terminal emulator that supports VT100 cursor movement command sequences.

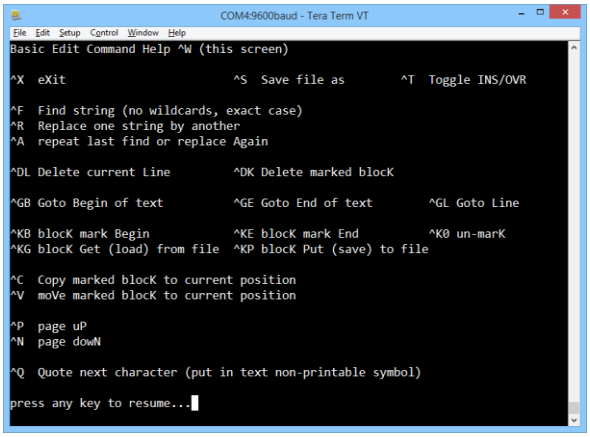
When the **EDIT** command is entered without a line number or with a filename a full screen editor is started that uses the terminal emulator window as the editor screen. Without a filename the editor starts with a blank screen, with a filename the editor loads the file (without an extension the .BAS is assumed and added):





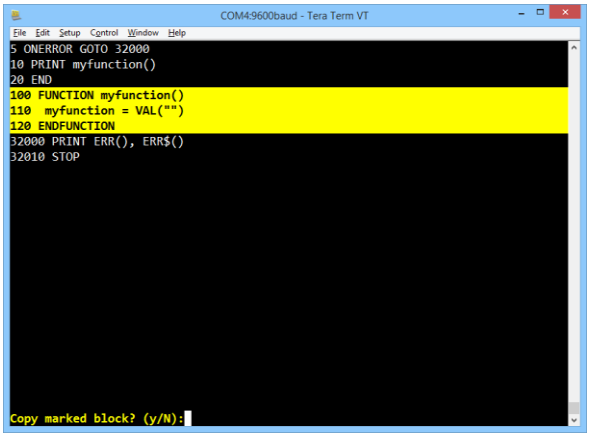
Text that is typed is inserted at the cursor location. The cursor may be moved using the **Home**, **Left**, **Up**, **Down**, **Right** and **End** keys. The **Backspace** key is used to delete characters to the left of the cursor. The **Enter** key starts a new line.

Editor commands are comprised of control key sequences (Ctrl + key) and may be listed within the editor using the Ctrl-W (**^W**) command as prompted on the bottom line when the editor is started:



Back to Table of Contents

Commands that require additional key selections to complete or require other user input also prompt on this bottom line:



It is important to note that the file that is being written or modified in the editor does not affect any program that is currently **LOAD**ed in BASIC – saving a new program file or modifying an existing program file does not **LOAD** it when the editor is exited – it must be reloaded to **RUN** it with the changes.

Editor Commands

Ctrl keys	Description
^X	eXit the editor – prompts to save if file modified
^S	Save the file – prompts to verify / change the filename to save to
^T	Toggle INSert / OVeRwrite mode
^F	Find string – no wildcards, exact case, prompts for search string
^R	Replace string – no wildcards, exact case, prompts for search string then replacement string
^A	Again – repeat last Find / Replace again
^DL	Delete current Line
^DK	Delete marked blocK
^GB	Goto Beginning of text
^GE	Goto End of text
^GL	Goto Line – prompts for line number
^KB	blocK mark Begin
^KE	blocK mark End – to include line end mark block end at beginning of following line
^K0	blocK un-mark (zero out begin / end marks)
^KG	blocK Get – prompts for filename to insert into marked block
^KP	blocK Put – prompts for filename to write marked block to
^C	Copy marked block to current cursor position
^V	moVe marked block to current cursor position
^P	page uP
^N	page down
^Q	Quote next character – prompt for next character to put in text, may be non-printable

The size of the terminal emulator window in lines and characters should match the value of the Basic Editor Lines and Basic Editor Cols configuration values. These values default to 25 lines and 80 columns (characters) and may be changed to utilize larger terminal windows.

Speed of operation may be improved by increasing both the terminal emulator and configured Baud Rate to a matching higher value than the default of 9600.

Back to Table of Contents

END

Program mode only. Terminate program with no message. Closes all open files.

ERROR value

Force an error. Program execution stops and an error message is displayed. There is a table of pre-defined Error values. To avoid confusion between a built-in error condition and a custom error condition the use of an error number that is either negative or outside of the range of defined errors is recommended.

```
10 ERROR 250
Ready
run
250 error in line 10
Ready
```

EVENT.DISABLE @systemvar BETA software

Program mode only. Disables event dispatching for **@systemvar** events without clearing any pending events. This is useful to protect program code sequences from interruption and shared variables from unexpected changes if an event were to occur.

EVENT.ENABLE @systemvar BETA software

Program mode only. Enables event dispatching for **@systemvar** allowing any pending or subsequent events to now occur.

Back to Table of Contents

FOR var = init TO limit {STEP increment} : statements : NEXT var

Program mode only. Perform a counted loop; incrementing ***var*** from the ***init*** expression value to the ***limit*** expression value by the optional ***increment*** expression value, executing statements up until the matching **NEXT** statement. The default value for the optional **STEP increment** is one. When the **NEXT** statement is reached execution resumes with the matching **FOR** statement if the **STEP increment** of the control **var**iable has not reached the **limit**.

- The **init**, **limit** and optional **STEP increment** expressions are only evaluated once when the **FOR / NEXT** loop is started – changing values of expressions within the loop will not affect the loop.
- **FOR / NEXT** loops can be nested and don't have to be the only statements on the line.
- **FOR / NEXT** loops can be exited from within the loop without the ***var***iable reaching the ***limit*** using the **BREAK** statement.
- **FOR / NEXT** loops can be continued from within the loop without executing all of the loop statements using the **CONTINUE** statement.
- As the **FOR / NEXT** counted loop uses the control stack to execute you should not jump out of or into the encompassing code block of statements. To leave the counted loop either force the ***var***iable to or beyond the ***limit*** value or use the **BREAK** statement.
- The number of nested **FOR / NEXT, WHILE / WEND** loops, block **IF / THEN / ELSE / ENDIF,** user defined **FUNCTION / ENDFUNCTION** and **GOSUB / RETURN** subroutines is limited.
- Execution of a **FOR** statement without a subsequent **NEXT** causes a "Nesting Error".
- Execution of a **NEXT** statement without a preceding **FOR** causes a "Nesting Error".

Back to Table of Contents

FINPUT #N, var{$}, … , var{$}

Get the value(s) for one or more **var**iables from a single line from a previously opened file *#N* (0 → 24).

Note that when an end of file occurs, the **var**iables will have their last value. Test the **@FEOF[#N]** system variable to detect this condition.

The data items in the file are separated by commas, and string values must be surrounded by double quotes. See the **FPRINT #N** statement below that can be used to produce a file in the correct format.

If the data in the file ends before all of the variables have been assigned values an "Out of Data Error" occurs. Incorrect data formatting in the file can cause a "Syntax Error" to occur.

```
10 OPEN #0, "test.csv", "r"
20 LIF @FILE.SIZE[#0] = 0 THEN PRINT "empty file" : END
30 FINPUT #0, Number, Number$
40 LIF @FEOF[#0] = 0 THEN PRINT Number, Number$ : GOTO 30
Ready
type test.csv
0,"zero"
1,"one"
2,"two"
Ready
run
0 zero
1 one
2 two
Ready
```

FPRINT #N, expr{, expr…}

Print one or more expression(s) to the previously opened file ***#N*** (0 → 24) that is **OPEN**ed for writing as a single line.

The data items on the line written to the file are separated by commas, with string values surrounded by double quotes. The produced file is compatible with the **FINPUT #N** statement.

```
10 OPEN #0, "test.csv", "w+"
20 FPRINT #0, 0, "zero" : FPRINT #0, 1, "one" : FPRINT #0, 2, "two"
30 CLOSE #0
run
Ready
Type test.csv
0,"zero"
1,"one"
2,"two"
Ready
```


Back to Table of Contents

FOPEN #N, recordlength, ”path”

Open the file named ***path*** as a fixed record length file #***N*** (0 → 24) for subsequent sequential / random access via **FREAD#** / **FWRITE#** statements.

- If ***recordlength*** is negative or greater than 255 it is forced to 255.
- The ***recordlength*** includes the trailing CR/LF character pair that terminates each record.
- If the file is empty, **@FEOF[#N]** will be set.

FREAD #N, recordnumber, var{$}, var{$}, … var{$}

Read ASCII data from fixed length records on file ***#N*** (0 → 24) previously opened by **FOPEN #N** into the list of variables.

Before the data is read, the file is positioned to the desired ***recordnumber*** by positioning the file to (***recordnumber*** x ***recordlength***). ***(***0 ≤ ***recordnumber*** x ***recordlength*** ≤ 2,147,483,648***)***. A negative ***recordnumber*** seeks to the end of the file.

- Reading at the current end of the file sets the **@FEOF[#N]** system variable and signals the associated event. Note that when an end of file occurs, the **var**iables will have their last value from a prior successful **FREAD**.
- Reading past the current end of the file generates a “FREAD record # Out of Range error”.
- The data items in the file are separated by commas, with string values surrounded by double quotes. If the data in the file ends before all of the variables have been assigned values an “Out of Data Error” occurs. Incorrect data formatting in the file can cause a “Syntax Error” to occur.

```
10 LIF FILE.EXISTS("test.dat") = 0 THEN PRINT "no test.dat" : END
20 FOPEN #1,20,"test.dat"
30 r = 0 : WHILE @FEOF[#1] = 0
40  FREAD #1,r,Number,String$
50  LIF @FEOF[#1] = 0 THEN PRINT Number, String$ : r = r + 1
60 WEND
70 CLOSE #1
Ready
type test.dat
0,"str0"
1,"str1"
2,"str2"
3,"str3"
4,"str4"
5,"str5"
6,"str6"
7,"str7"
8,"str8"
9,"str9"
Ready
run
0 str0
1 str1
2 str2
3 str3
4 str4
5 str5
6 str6
7 str7
8 str8
9 str9
Ready
```


Back to Table of Contents

FWRITE #N, recordnumber, var{$}, var{$}, ... var{$}

Write ASCII data into fixed length records on file ***#N*** (0 → 24) previously opened by ***FOPEN #N*** from the list of variables.

Before the data is written, the file is positioned to the desired ***recordnumber*** by positioning the file to (***recordnumber*** x ***recordlength***). ***(***0 ≤ ***recordnumber*** x ***recordlength*** ≤ 2,147,483,648***)***.

- A negative ***recordnumber*** seeks to the current end of the file.
- Writing at the current end of file extends the file by the record size. Writing past the current of file generates a "FWRITE record # Out of Range error".
- The data items written to the file are separated by commas, with string values surrounded by double quotes.
- The record is padded with spaces to ***recordlength*** including the trailing CR/LF character pair which terminates each record.
- The generated file may be viewed using the **TYPE** command.

```
10 IF FILE.EXISTS("test.dat") THEN DEL "test.dat"
15 FOPEN #1,20,"test.dat"
20 FOR r = 0 TO 9 : FWRITE #1, r, r, "str"+STR$(r) : NEXT r
25 CLOSE #1
Ready
run
Ready
type test.dat
0,"str0"
1,"str1"
2,"str2"
3,"str3"
4,"str4"
5,"str5"
6,"str6"
7,"str7"
8,"str8"
9,"str9"
Ready
```


Back to Table of Contents

FINSERT #N, recordnumber, var{$}, var{$}, ... var{$}

Insert ASCII data into fixed length records on file ***#N*** (0 → 24) previously opened by ***FOPEN #N*** from the list of variables using a temporary file FINSERT.TMP.

Before the data is inserted, the file is positioned to the desired ***recordnumber*** by positioning the file to (***recordnumber*** x ***recordlength***). ***(***0 ≤ ***recordnumber*** x ***recordlength*** ≤ 2,147,483,648***)***, and records in the file after ***recordnumber*** are shifted down.

- A negative ***recordnumber*** seeks to the end of the file before inserting.
- The data items inserted into the file are separated by commas, with string values surrounded by double quotes.
- The record is padded with spaces to ***recordlength*** including the trailing CR/LF character pair which terminates each record.
- The modified file may be viewed using the **TYPE** command.

```
type test.dat
0,"str0"
1,"str1"
2,"str2"
3,"str3"
4,"str4"
5,"str5"
6,"str6"
7,"str7"
8,"str8"
9,"str9"
Ready
list
10 LIF FILE.EXISTS("test.dat") = 0 THEN PRINT "no test.dat" : END
20 FOPEN #1,20,"test.dat"
25 FINSERT #1,0,-1,"str"+STR$(-1) : FINSERT #1,-1,10,"str"+STR$(10)
30 r = 0 : WHILE @FEOF[#1] = 0
40  FREAD #1,r,Number,String$
50  LIF @FEOF[#1] = 0 THEN PRINT Number, String$ : r = r + 1
60 WEND
70 CLOSE #1
Ready
run
-1 str-1
0 str0
1 str1
2 str2
3 str3
4 str4
5 str5
6 str6
7 str7
8 str8
9 str9
10 str10
Ready
```


Back to Table of Contents

FDELETE #N, recordnumber

Remove a fixed length record ***recordnumber (***0 ≤ ***recordnumber*** x ***recordlength*** ≤ 2,147,483,648***)*** on file ***#N*** (0 → 24) opened by **FOPEN #N** using a temporary file FDELETE.TMP.

- The modified file may be viewed using the **TYPE** command.

```
type test.dat
0,"str0"
1,"str1"
2,"str2"
3,"str3"
4,"str4"
5,"str5"
6,"str6"
7,"str7"
8,"str8"
9,"str9"
Ready
list
10 LIF FILE.EXISTS("test.dat") = 0 THEN PRINT "no test.dat" : END
20 FOPEN #1,20,"test.dat"
25 FDELETE #1,0
30 r = 0 : WHILE @FEOF[#1] = 0
40  FREAD #1,r,Number,String$
50  LIF @FEOF[#1] = 0 THEN PRINT Number, String$ : r = r + 1
60 WEND
70 CLOSE #1
Ready
run
1 str1
2 str2
3 str3
4 str4
5 str5
6 str6
7 str7
8 str8
9 str9
Ready
```


Back to Table of Contents

FUNCTION name{$}({parm1{$}{,parm2{$}, … parmN{$}}})

Program mode only. Define a user function of **name{$}** which takes zero or more parameter values.

Functions may be either integer or string using the variable naming convention of a trailing dollar sign for strings.

Functions require zero or more integer or string parameters enclosed in a parenthesized parameter list.

The name of the function becomes a defined variable global to the entire program. The parameter variables will be created and assigned values when the function is subsequently called. Any additional statements following on the same line are ignored.

- Functions are defined by program statements surrounded with a **FUNCTION** statment and ending with an **ENDFUNCTION** statement.
- Statements following the **FUNCTION** statement on the same line are not executed.
- Functions may be redefined using the same function **name{$}** as long as the integer or string function type is not changed.
- The function **name{$}** behaves like a global integer or string variable that has a zero or empty string value when the function is defined and can be used to provide a value to or return a value from the function.
- Since a function behaves like a global integer or string variable, it is executed by using the function's name in a command or expression as if accessing the variable – providing any requisite parameter values that the function requires as arguments.

```
10 PRINT myfunction()
20 END
30 FUNCTION myfunction()
40  myfunction = 5
50 ENDFUNCTION
Ready
run
5
Ready
```

```
10 PRINT myfunction$()
20 END
30 FUNCTION myfunction$()
40  myfunction$ = "result"
50 ENDFUNCTION
Ready
run
result
Ready
```

See the User Defined Functions section below for more information.

ENDFUNCTION

Program mode only. End a user defined function. Statements following the **ENDFUNCTION** statement on the same line are not executed.

When initially defining a **FUNCTION** the **ENDFUNCTION** statement terminates the definition.

When executing a previously defined **FUNCTION** the **ENDFUNCTION** causes program execution to return to the statements following the function call.

See the User Defined Functions section below for more information.

Back to Table of Contents

GOSUB line / GOSUB `label

Program mode only. Call a subroutine that starts at ***line*** or ***`label*** and ends with a **RETURN** statement. A subroutine consists of a group of program statements that start at a certain ***line*** number or ***`label*** and end in a line with a **RETURN** statement.

- To call the subroutine from your program use the **GOSUB** statement which transfers program execution to the specified line number and executes those program statements until it executes a **RETURN** statement.
- There can be multiple **RETURN** statements as long as every path of execution through the subroutine eventually executes one.
- Upon execution of the **RETURN** statement, program execution continues at the statement after the **GOSUB**.
- The number of nested **FOR / NEXT, WHILE / WEND** loops, block **IF / THEN / ELSE / ENDIF,** user defined **FUNCTION / ENDFUNCTION** and **GOSUB / RETURN** subroutines is limited.
- Repeatedly executing a **GOSUB** without a subsequent **RETURN** will result in a "Stack Overflow".
- The execution of a **RETURN** without a preceding **GOSUB** will result in a "Nesting Error".

GOTO line / GOTO `label

Program mode only. Program execution continues by jumping to ***line*** or ***`label***.

- Caution should be used to not jump into the middle of **FOR / NEXT, WHILE / WEND** loops, block **IF / THEN / ELSE / ENDIF,** user defined **FUNCTION / ENDFUNCTION** and **GOSUB / RETURN** subroutines.

Back to Table of Contents

HTTP.CGI #N, "/myuri.cgi", connect( ), request( ), response( ) BETA software

Program mode only. Initiates a HTTP request/response listener on file #N (0 → 24) using the IP Address and HTTP port as specified in the CFSound-IV Configuration.

Incoming HTTP request methods are handled as follows:

1) For GET or POST request methods if the URI matches the **"/myuri.cgi"** as specified in the statement the **connect( )** user function is called.

   The slash prefix and trailing **".cgi"** suffix are required, the **"myuri"** is application specific.

   The URI value of **"/config.cgi"** is used to access the CFSound-IV configuration settings and may not be overridden.

2) For GET request methods with no form data following the URI the **HTTP/1.0 200 OK** response header is sent, then the **response( )** user function is called to generate the response to the GET using **PRINT #N** statements.

   The **response( )** user function should return zero to terminate the response or non-zero to be called again to continue building the response.

   When a zero is returned the request/response sequence is complete and nothing further happens until the next incoming HTTP request.

3) For GET or POST request methods with form data following the URI the **HTTP/1.0 303 See other** response header is sent, then the **request( )** user function is called to process the form data which is retrieved using an **INPUT #N** statement.

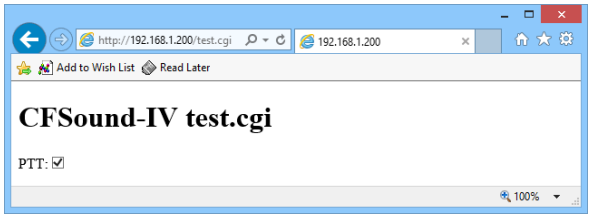
   Next the **response( )** user function is called to generate the response to the request with form data using **PRINT #N** statements.

   The **response( )** user function should return zero to terminate the response or non-zero to be called again to continue building the response.

   When a zero is returned the request/response sequence is complete and nothing further happens until the next incoming HTTP request.

Here is a short example that provides access to the CFSound-IV PTT relay using a checkbox. When the following short program is run, the HTTP.CGI statement at the beginning forms an accessible URI of **http://192.168.1.200/test.cgi**. The IP address and default HTTP port for the URI are set using the CFSound-IV configuration settings.

Using a browser, an access to the URI produces a very simple web page that is entirely generated by this program:



Accessing the URI with the browser, the **connect( )** user function is called first, then the **response( )** user function is called repeatedly to generate the webpage.

The value of the **response( )** user function starts at zero and its incrementing value is used with an **ON response, GOSUB** statement to dispatch multiple subroutines that build the webpage. Helper string functions are called to provide some of the HTML which is emitted using **PRINT #0** statements.

Back to Table of Contents

A hidden form field with the same name as the checkbox is used to always provide a PTT name=value pair in the form data since an unchecked checkbox generates no name/value pair.

Clicking the PTT checkbox auto submits the form and the **connect( )** user function is called first, then the **request( )** user function is called to receive the form data using an **INPUT #0** statement.

The form data is parsed and the CFSound-IV PTT relay is controlled as required. Then the **response( )** user function is again called repeatedly to build the webpage.

```
10 REM http cgi PTT control
20 HTTP.CGI #0, "/test.cgi", connect(), request(), response()
30 GOTO 30
40 FUNCTION connect()
60 ENDFUNCTION
70 FUNCTION request()
80  INPUT #0, req$
82  IF req$ = "PTT=0" THEN @PTT=0
84  IF req$ = "PTT=0&PTT=1" THEN @PTT=1
90 ENDFUNCTION
100 FUNCTION response()
110  ON response, GOSUB 200,300,400,500,600,700
125  response = response + 1
130 ENDFUNCTION
200 PRINT #0,HtmlBegin$();HeadBegin$();"<h1>CFSound-IV test.cgi</h1>";HeadEnd$() : RETURN
300 PRINT #0,"<body><p>";FormBegin$() : RETURN
400 PRINT #0,Hidden$("PTT", "0");"PTT:";Checkbox$("PTT", "1", @PTT, 1) : RETURN
500 PRINT #0,FormEnd$();"</body>" : RETURN
600 PRINT #0,HtmlEnd$() : RETURN
700 response = -1 : RETURN
100000 REM HTML Helpers
100005 FUNCTION HtmlBegin$()
100010  HtmlBegin$ = "<!DOCTYPE HTML><html>"
100015 ENDFUNCTION
100020 FUNCTION HtmlEnd$()
100025  HtmlEnd$ = "</html>"
100030 ENDFUNCTION
100035 FUNCTION HeadBegin$()
100040  HeadBegin$ = "<head>"
100045 ENDFUNCTION
100050 FUNCTION HeadEnd$()
100055  HeadEnd$ = "</head>"
100060 ENDFUNCTION
100065 FUNCTION FormBegin$()
100070  FormBegin$ = "<form method="+CHR$(34)+"post"+CHR$(34)+">"
100075 ENDFUNCTION
100080 FUNCTION FormEnd$()
100085  FormEnd$ = "</form>"
100090 ENDFUNCTION
100125 FUNCTION Hidden$(name$, value$)
100130  Hidden$ = "<input type="+CHR$(34)+"hidden"+CHR$(34)
100135  IF name$ <> "" THEN Hidden$ = Hidden$ + " name="+CHR$(34)+name$+CHR$(34)
100140  IF value$ <> "" THEN Hidden$ = Hidden$ + " value="+CHR$(34)+value$+CHR$(34)
100145  Hidden$ = Hidden$ + " />"
100150 ENDFUNCTION
100230 FUNCTION Checkbox$(name$, value$, checked, submit)
100235  Checkbox$ = "<input type="+CHR$(34)+"checkbox"+CHR$(34)
100240  IF name$ <> "" THEN Checkbox$ = Checkbox$ + " name="+CHR$(34)+name$+CHR$(34)
100245  IF value$ <> "" THEN Checkbox$ = Checkbox$ + " value="+CHR$(34)+value$+CHR$(34)
100250  IF checked > 0 THEN Checkbox$ = Checkbox$ + " checked"
100255  IF submit > 0 THEN Checkbox$ = Checkbox$ + " onClick="+CHR$(34)+"submit();"+CHR$(34)
100260  Checkbox$ = Checkbox$ + " />"
100265 ENDFUNCTION
Ready
```

HTML that accesses other files and content types such as .HTML, .CSS, .PNG, .GIF, .JPG, .BMP and .TXT on the SD card are supported as they are accessed with other, non-cgi GET requests and are handled by the CFSound-IV HTTP protocol outside of the scope of the HTTP.CGI statement.

Back to Table of Contents

IF test THEN line/`label/statement {ELSE line2/`label2/statement2}

Program mode only. Perform a conditional execution jump. The expression ***test*** is evaluated, and if non-zero, program execution continues at ***line*** or ***`label*** or the single ***statement*** is executed. If the optional **ELSE** clause is present and the ***test*** expression evaluates to zero program execution continues at ***line2*** or `label2 or the single ***statement2*** is executed.

Some **IF** statement examples:

```
10 IF A=0 THEN 100
20 IF A=1 THEN GOTO 200
30 IF A=0 THEN PRINT “A was zero” ELSE 100
40 IF A=1 THEN PRINT “A was zero” ELSE PRINT “A non-zero”
```

Multiple conditions can be tested at the same time by combining two or more ***test*** expressions with the logical **AND**, **OR** operators:

```
20 IF (A=1) AND (B=2) THEN PRINT "Both A and B are correct"
30 IF (A=1) OR (B=2) THEN PRINT "Either A or B is correct" ELSE PRINT “Neither A or B”
```

IF test THEN

statements

{ELSE

statements}

ENDIF

Program mode only. Perform a conditional execution block jump. There must be no statements on the same line following the **THEN**, **ELSE** and **ENDIF** keywords. The expression ***test*** is evaluated, and if non-zero, program execution continues with the following ***statements***. If the test expression evaluates to zero, program execution continues at the statements following the **ENDIF**. If the optional **ELSE** clause is present and the ***test*** expression evaluates to zero program execution continues at the ***statements*** following the **ELSE** up until the **ENDIF** is executed.

- As the conditional block **IF** / **THEN** / **ENDIF** uses the control stack to execute you should not jump out of or into either code block of statements. To leave the block **IF** statement **GOTO** the **ENDIF** statement.
- A **BREAK** statement can be used to exit a **FOR** / **NEXT** or **WHILE** / **WEND** loop from within a block **IF** statement.
- A **CONTINUE** statement can be used to continue a **FOR** / **NEXT** or **WHILE** / **WEND** loop from within a block **IF** statement.
- A **RETURN** statement can be used to return from a subroutine from within a block **IF** statement.

An **IF** / **THEN** / **ENDIF** statement example – notice the use of extra spaces to indent the statement blocks to improve readability:

```
10 REM block if
20 a=0
30 IF a=1 THEN
40  PRINT "if condition line 1"
42  PRINT "if condition line 2"
44  PRINT "if condition line 3"
50 ELSE
60  PRINT "else condition line 1"
62  PRINT "else condition line 2"
64  PRINT "else condition line 3"
70 ENDIF
```


Back to Table of Contents

INCLUDE path

Program or Direct mode. Include BASIC statements from a SD card file specified by ***path***. The full ***path*** to the file must be specified and must not start with a leading backslash.

- Statements in the file without line numbers are immediately executed. This is useful to define commonly used **CONST**ants for example.
- Statements with line numbers are entered as if they were typed in – adding new or replacing existing numbered lines.
- Note that **INCLUDE** must be the ONLY statement on a line.
- If a file extension is not present, the .BAS file extension on the filename at the end of the path is assumed and added.
- Program mode **INCLUDE**s cannot contain user defined **FUNCTION**s / **ENDFUNCTION**s or **`labels** as these are required to be already present in the program when execution begins.

In the **BETA** software **INCLUDE** statement operation has been enhanced – program mode **INCLUDE**s can now contain user defined **FUNCTIONS** and **`labels**.

- If the line numbers of the included statements are preceded by a plus sign '**+**' then those numbers are added to the highest line number of the program being included into; essentially offsetting the included statements to reside at the end of the current program.
- If the included statements comprise a **FUNCTION** the last line of the program should be an **END**, **STOP** or **GOTO** statement to prevent execution from continuing into the included **FUNCTION**.
- When using the offsetting line number feature line numbers within the included code are not adjusted and should be avoided by using **`labels**. Note that these **`labels** are global to the entire program and so must be unique to avoid duplicate **`label** errors.
- Line numbered statements that are included can be **LIST**ed after the program has been run at least once, but will not **SAVE** with the program being included into.

```
load TestFunctionInclude
Ready
list
10 REM functions
20 INCLUDE TestFunction
50 test(1,2,3):PRINT "= ";test
60 test(4,5,6):PRINT "= ";test
10000 STOP
Ready
type TestFunction.bas
+10 FUNCTION test(one,two,three)
+15  PRINT "test(";one;",";two;",";three;") ";
+20  test=one+two+three
+25 ENDFUNCTION
Ready
run
test(1,2,3) = 6
test(4,5,6) = 15
STOP in line 10000
Ready
list
10 REM functions
20 INCLUDE TestFunction
50 test(1,2,3):PRINT "= ";test
60 test(4,5,6):PRINT "= ";test
10000 STOP
10010 FUNCTION test(one,two,three)
10025  PRINT "test(";one;",";two;",";three;") ";
10045  test=one+two+three
10070 ENDFUNCTION
```


Back to Table of Contents

INPUT var{$}

Get a value for variable from the serial port.

INPUT "prompt", var

Get a value for **var**iable from the serial port with prompt. Prompt may be a constant string or you can use a string variable in the prompt by concatenating it to a string: **INPUT ""+A$, B$**

INPUT #N, var

Get a value for **var**iable from a previously opened file or internet socket handle #***N*** (0 → 24). Note that when an end of file occurs, the **var**iable will have its last value. Test the **@FEOF[#N]** system variable to detect this condition when inputting from a file.

{LET} var{$}=expr{$} (default statement)

Program or Direct mode. Set **var**iable = **expr**ession (This is the default statement, so the **LET** keyword is not required). An attempt to assign a string value to a numeric variable or a numeric value to a string variable will generate a "Type Error". Some examples:

```
LET a0 = 240
100 Z9$ = "Test"
@TIMER[0] = 240
```

LIF test THEN statement{: statement …}

Program mode only. **L**ong **IF** (perform a conditional execution of all statements to end of line). The expression ***test*** is evaluated, and if non-zero, all statements to the end of the current program line are executed.

```
20 LIF @CLOSURE[24]=1 THEN PRINT "25 closed" : GOSUB 100 : @CLOSURE[24]=0
30 GOTO 20
```

Multiple conditions can be tested at the same time by combining two or more ***test*** expressions with the logical **AND**, **OR** operators:

```
20 LIF (A=0) AND (@CLOSURE[24]=1) THEN PRINT "25 closed" : GOSUB 100 : @CLOSURE[24]=0
30 GOTO 20
```

LIST {start{, end}} … LIST {start{-end}}

Direct mode only. List one or more program lines to the serial port. The command may also specify a starting and ending line number to limit the range of lines that are displayed. A double escape sequence will stop the portion of the file that the CFSound has not already queued for output.

LIST #N {start{, end}} … LIST #N {start{-end}}

Direct mode only. List one or more program lines to a previously opened file #***N*** (0 → 24). The statement may also specify a starting and ending line number to limit the range of lines that are displayed. A double escape sequence will stop the portion of the file that the CFSound not already written.

Back to Table of Contents

LOAD path

Program or Direct mode. Load an BASIC program from a SD card file specified by ***path***. The full ***path*** to the program file must be specified and must not start with a leading backslash.

- When **LOAD** is used within a program, execution continues with the first line of the newly loaded program. In this case, the user variables are not cleared. This provides a means of chaining to a new program, and passing information to it.
- When used in a program note that **LOAD** must be the last statement on a line. If not present, the .BAS file extension on the filename at the end of the path is assumed and added.

```
load program1
Ready
list
10 PRINT "Program 1 A=",a
20 a=a+1
30 LOAD program2
Ready
load program2
Ready
list
10 PRINT "Program 2 A=",a:a=a+1:LOAD program1
Ready
run
Program 2 A= 0
Program 1 A= 1
Program 2 A= 2
Program 1 A= 3
 ESC at line 30
Ready
```

MD path

Direct mode only, requires a SD card. Make a new directory on the SD card. ***Path*** must be a complete path for the new directory, and it must not already exist. ***Path*** may be a constant string or you can use a string variable as the ***path*** by concatenating it to such a string: ***MD*** ""***+PATH$***.

MEMORY

Display the currently available program memory, resource memory and SD card memory if a SD card is present.

```
memory
SD card bytes free:     7,929,724,928
Resource bytes free:       23,938,413
Program bytes free:         8,385,960
Ready
```

NEW

Direct mode only. Erase all program statements, clear all variable values and close all open files.

NUM {start {, increment}}

Direct mode only. Displays and/or changes automatic line numbers. When manually entering program statements typing a '+' at the beginning of the line will automatically generate and enter the next line number. The current values for **start** and **increment** can be displayed by **NUM** with no arguments, the default for **start** is 10 and the default for **increment** is 5.

Back to Table of Contents

ON expr, GOSUB line0, line1, line2, … , lineN

Program mode only. Conditional program execution for multiple expression values using subroutines. The value of ***expr*** is evaluated, and a subroutine call is performed to the ***line0*** statement if it evaluates to zero, ***line1*** if one, etc.

- If the value of ***expr*** is negative or greater than the number of line numbers present, execution continues with the next statement. This would be considered the default case, but is also where the dispatched subroutines **RETURN** to.
- A line number of zero in the list also causes execution to continue with the next statement if the expression evaluates to that position.
- Upon return from the **GOSUB** execution continues with the next statement.
- **`Labels** can also be used in place of non-zero line numbers.

```
5 REM ONGOSUB Demo
10 a=0
20 ON a,GOSUB 100,200,300,400
30 GOTO 20
100 PRINT "1",
105 a=a+1
110 RETURN
200 PRINT "2",
205 a=a+1
210 RETURN
300 PRINT "3",
305 a=a+1
310 RETURN
400 PRINT "4",
405 a=a+1
410 RETURN
Ready
run
1234
1234
1234
1234
1234
1 ESC at line 105
Ready
```


Back to Table of Contents

ON expr, GOTO line0, line1, line2, … , lineN

Program mode only. Conditional program execution for multiple expression values using jumps. The value of ***expr*** is evaluated, and a jump is performed to the ***line0*** statement if zero, ***line1*** if one, etc.

- If the value of ***expr*** is negative or greater than the number of line numbers present, execution continues with the next statement. This is the default case.
- A line number of zero in the list also causes execution to continue with the next statement if the expression evaluates to that position.
- **`Labels** can also be used in place of non-zero line numbers.

```
5 REM ON GOTO DEMO
10 a=0
20 ON a,GOTO 100,200,300,400
30 GOTO 10
100 PRINT "1",
105 a=a+1
110 GOTO 20
200 PRINT "2",
205 a=a+1
210 GOTO 20
300 PRINT "3",
305 a=a+1
310 GOTO 20
400 PRINT "4",
405 a=a+1
410 GOTO 20
Ready
run
1234
1234
1234
1234 ESC at line 20
Ready
```


Back to Table of Contents

ONERROR GOTO line

Program mode only. Provide the ability for the program to perform one-shot error handling. After executing this command, upon any error, statement execution starts at ***line***, the **ERR( )** function has the value of the error number and the **ERR$( )** function has the string version of the error number. The **ONERROR** condition is then cleared so that subsequent errors result in program termination.

- The **ONERROR** can be disabled by specifying a ***line*** number of zero.

```
10 ONERROR GOTO 100
20 REM error follows
30 a=10/0
40 STOP
100 PRINT "Error #",ERR()," - ",ERR$()
Ready
run
Error # 6 - Divide by zero error in line 30
Ready
```

A common use of **ONERROR** statement is to allow execution of a command that might fail without causing the program to stop execution. For example if you want to delete a file with the **DEL** command, if the file didn't exist the **DEL** command would produce an error and the program would stop. By setting up an **ONERROR** handler to surround the **DEL** command the program will continue execution if the file to be deleted did or did not exist:

```
170 ONERROR GOTO 180 : DEL "WAVLIST.TXT" : ONERROR GOTO 0
180 REM execution continues here even if WAVLIST.TXT didn't exist
```

The **ONERROR** statement can also be used to perform error logging. There is an example of error logging in the **BASIC Examples** section below.

Back to Table of Contents

ONEVENT @systemvar, GOSUB line

Program mode only. Provide semi-asynchronous event handling via subroutines. Certain BASIC system variables can trigger events. The **ONEVENT** statement allows the event to force the execution of a subroutine when it occurs.

When the event occurs, after execution of any current statement that does not transfer control, control is transferred to the subroutine starting at ***line***.

*While in the event subroutine, only higher priority events will be recognized until after the **RETURN** statement is executed.*

An event handler can be disabled by specifying a ***line*** number of zero. Executing the **ONEVENT** statement clears the associated event in preparation for the subsequent event handling.

The following system variables can cause events and are listed in order of ***decreasing*** priority:

System Variable	Event Occurs
@TIMER[x]	One time whenever the timer counts down to zero. System variable **@TIMER[0]** is the highest priority, followed by **@TIMER[1]**, … then **@TIMER[9]**. $0 \le x \le 9$
@CLOSURE[x]	Whenever a contact I/O input contact has closed. $0 \le x \le 56$
@OPENING[x]	Whenever a contact I/O input contact has opened. $0 \le x \le 56$
@FEOF[#N]	After **INPUT #N**, **FINPUT #N** or **FREAD #N** reaches the end of file #N ($0 \rightarrow 9$)
@SOCKET.EVENT[#N]	After a SOCKET operation.
@SECOND	Once per second.
@MINUTE	Once per minute.
@HOUR	Once per hour.
@DOW	Once per day at midnight.
@DATE	Once per day at midnight.
@MONTH	Once per month at midnight of day 1.
@YEAR	Once per year at midnight of day 1 of the first month.
@SOUND$	After the last queued **@SOUND$** sound has finished playing.
@MSG$	After receipt of a serial character stream delineated by the **@SOM** and **@EOM** characters.
@EOT	Upon complete transmission of a serial character stream of one or more characters when both the output buffer and UART are empty.

Each BASIC implementation may have additional system variables that may cause events. Please consult each product's manual for details. There is more information about **Events** in the section below.

Here is a short program that outputs the current time, once per second, on the serial port. Note that the program's idle loop, which it executes while waiting for the second event to occur, consists of a single **GOTO** statement jumping to itself – it could be a large program:

```
5 REM print the time once per second
10 ONEVENT @SECOND,GOSUB 100
20 GOTO 20
100 PRINT CHR$(13);
105 PRINT FMT$("%2d",@HOUR);
110 PRINT ":";
115 PRINT FMT$("%02d",@MINUTE);
120 PRINT ":";
125 PRINT FMT$("%02d",@SECOND);
130 RETURN
Ready
run
14:47:15 ESC at line 30
Ready
```


Back to Table of Contents

OPEN #N, "path", "options"

Open file name ***path*** as file #***N*** (0 → 24) for subsequent access via **DIR #N**, **INPUT #N, FINPUT #N, PRINT #N** or **FPRINT #N** statements.

The ***options*** string characters determine the way that the opened file is handled and listed here:

"options"	**Description**
"r"	opens file for reading, if ***path*** does not exist an error is generated
"w"	opens file for writing, if ***path*** exists its contents are destroyed
"r+"	opens file for read and write, the ***path*** must exist
"w+"	opens an empty file for read and write, if ***path*** exists its contents are destroyed
"a+"	opens file for reading and appending (seek to end of file after open)
"b"	opens file in binary mode, no translations
"t"	opens file in text mode (default), CR/LF pairs are translated to LF on input and LF translated to CR/LF pairs on output.

ORDER line / `label

Program mode only. This statement positions the read data pointer to a **DATA** statement identified by ***line*** number or ***`label***.

The target statement at ***line*** or ***`label*** must be a series of one or more **DATA** statements.

PLAY file

Program or Direct mode. Play the sound ***file*** and wait until it completes. Program execution then continues with the next statement. If the ***file*** is not a valid .WAV file of the correct format, sample rate and sample size for the CFSound-IV an "Invalid .WAV File Error" is generated.

- ***File*** may be a constant string or you can use a string variable as the ***file*** by concatenating it to such a string: ***PLAY*** ""***+FILE$***. While the sound file is playing
- Events can occur during the **PLAY** statement, but any defined **ONEVENT** handlers will not be executed until the sound has finished playing.
- To play sounds while continuing program execution use the **@SOUND$** system variable.

Back to Table of Contents

PRINT expr{$}{, expr{$} ...}{,} … PRINT expr{$} {; expr{$} …}{;}

Print the value of one or more expression(s) to the serial port.

- If the expressions are separated by a comma (‘,’) then a space is output in between.
- If the expressions are separated by a semicolon (‘;’) then no space is output.
- If the statement ends in a comma or semicolon no Carriage Return / Line Feed pair is appended to the printed expressions allowing multiple print statements to display on the same line.

When **@ANSI.ENABLE=1**, the **PRINT** statement is also shown on the CFSound virtual display as drawn ANSI text. See the ANSI Mode for information on how the text is rendered on the CFSound.

PRINT #N, expr{$}{, expr{$} ...} … PRINT #N, expr{$}{; expr{$} …}

Print the value of one or more expressions to a previously opened file or internet streaming handle #**N** (0 → 24). The same formatting conditions as the **PRINT** statement above apply.

PRINT USING fmt$ {, expr{$} {, expr{$} … , expr{$}}}{;}

Print zero or more formatted numeric or string **expr**essions to the serial port.

- The individual expressions are formatted with the specifications in the **fmt$** string using the same format specification as the **FMT$( )** function above.
- If the statement ends in a semicolon no Carriage Return / Line Feed pair is appended to the printed expressions allowing multiple print statements to display on the same line.

When **@ANSI.ENABLE=1**, the **PRINT** statement is also shown on the CFSound virtual display as drawn ANSI text. See the ANSI Mode for information on how the text is rendered on the CFSound.

PRINT #N, USING fmt$ {, expr{$} {, expr{$} … , expr{$}}}{;}

Print zero or more formatted numeric or string **expr**essions to a previously opened file or internet streaming handle #**N**(0→ 24).

- The individual expressions are formatted with the specifications in the **fmt$** string using the same format specification as the **FMT$( )** function above.
- If the statement ends in a semicolon no Carriage Return / Line Feed pair is appended to the printed expressions allowing multiple print statements to display on the same line.

```
10 REM show time
15 ONEVENT @SECOND,GOSUB 100
20 GOTO 20
100 PRINT USING "%c%2d:%02d:%02d",13,@HOUR,@MINUTE,@SECOND;
105 RETURN
Ready
run
13:28:52 <<< ESC at line 20 >>>
Ready
```


Back to Table of Contents

READ var{$}{, var{$} ... , var{$}}

Program mode only. Read data from **DATA** program statements into **var**iables. You *MUST* issue an **ORDER** statement targeting a line containing a valid **DATA** statement before using **READ**.

RETURN

Program mode only. Return from a subroutine invoked via a **GOSUB** statement.

- You can **RETURN** from within a **FOR / NEXT**, **WHILE / WEND** or block **IF** statement inside of a subroutine.
- A return without a prior **GOSUB** will generate a "Nesting Error".

REM

Program or Direct mode. Remark (comment)... the remainder of line is ignored. Use to document the operation of the program and is highly recommended.

REN oldfile newfile / REN "oldfile","newfile"

Program or Direct mode. Rename oldfile to newfile. ***Oldfile*** and ***newfile*** may be constant strings or you can use string variables as the ***files*** by concatenating them to string constants: **REN ""+OLD$, ""+NEW$**. In Direct mode the quotes are not required. The newfile must not already exist.

Back to Table of Contents

RESQ {start{-end}{,new}{,incr}}

Direct mode only. Re-sequence the program line numbers from ***start*** through ***end*** beginning with the value of ***new*** advancing by ***incr***.

The default value of ***start*** is the first line of the program, the default for ***end*** is the last line of the program, the default for ***new*** is 10 and the default for ***incr*** is 5.

- The currently loaded program remains unchanged and should be saved before re-sequencing.
- The program is renumbered with all embedded references to the new line numbers corrected. It is displayed and written to a file with the same name as the original program with the extension **.RSQ**.
- If there are syntax errors in the current program, or references to non-existent line numbers, the **RESQ** will error and stop.
- No checks are made to avoid overlapping line numbers and the generated **.RSQ** file should be loaded, viewed and run before saving it over the original program file.
- After a successful re-sequencing the generated .RSQ file may be deleted.

```
list
10 ON N,GOTO 100,150,200
20 GOSUB 250
30 GOTO 30
100 REM
150 REM
200 STOP
250 RETURN
Ready
resq
Writing resequenced program to:test2.RSQ

10 ON N,GOTO 25,30,35
15 GOSUB 40
20 GOTO 20
25 REM
30 REM
35 STOP
40 RETURN
```

RUN {line} … RUN path

Direct mode only. Execute the program starting at the lowest or optional ***line*** number. If path is present, **LOADs** and **RUNs** a file directly at the lowest line number.

- If not present, the .BAS file extension on the filename at the end of the path is assumed and added.

SAVE {path}

Direct mode only. Save the current program to a disk file on the SD card with the filename specified in ***path***, or to the filename used in the previous **LOAD, SAVE** or **RUN** command if not specified.

- If not present, the .BAS file extension on the filename at the end of the path is assumed and added.
- If the provided name doesn't match the previous **LOAD**, **SAVE** or **RUN** filename, and the file already exists, an overwrite warning message is displayed requiring approval.

Back to Table of Contents

SEARCH string {filename]

Direct mode only. Perform a case insensitive search for the occurrence of the **string** displaying where it appears. The search **string** can be a program statement keyword or contain '*' and '?' wildcard characters to match multiple variations of the search string.

- If the optional **filename** is not present the currently loaded program is the search target for the occurrence of the search **string** displaying the lines where it appears.
- If the optional **filename** is present then that file becomes the search target and the file's lines are searched for the occurrence of the search string. The optional **filename** can contain '*' and '?' wildcard characters to search multiple files.

SIGNAL @systemvar

Signal an event associated with System variable.

In the **BETA** software the SIGNAL statement has been renamed:

EVENT.SIGNAL @systemvar

SORT var{$}

Sort an array of strings or integers in ascending order. Here's an example of sorting an integer array:

```
10 REM sorting integers
15 CONST size = 20
20 DIM a[size]
30 FOR n = 0 TO size-1 : a[n] = RND(1000) : NEXT n
40 PRINT "before sort:" : FOR n = 0 TO size-1 : PRINT a[n];" "; : NEXT n
50 SORT a : PRINT ""
60 PRINT "after soft:" : FOR n = 0 TO size-1 : PRINT a[n];" "; : NEXT n
70 PRINT ""
Ready
run
before sort:
540 808 884 101 604 388 201 802 712 350 948 793 615 305 477 455 563 526 136 481
after soft:
101 136 201 305 350 388 455 477 481 526 540 563 604 615 712 793 802 808 884 948
Ready
```

And here's an example of sorting a string array:

```
10 REM sorting strings
15 CONST size = 10
20 DIM a$[size]
30 FOR n = 0 TO size-1 : FOR i = 0 TO 5 : a$[n] = a$[n] + CHR$(RND(25)+65) : NEXT i : NEXT n
40 PRINT "before sort:" : FOR n = 0 TO size-1 : PRINT a$[n];" "; : NEXT n
50 SORT a$ : PRINT ""
60 PRINT "after soft:" : FOR n = 0 TO size-1 : PRINT a$[n];" "; : NEXT n
70 PRINT ""
Ready
run
before sort:
AWRLHC APIAER XMDYBJ UDEMUI EJODGW HWWYJC VFRHND NEVMPH UJIPJG TKGKRK
after soft:
APIAER AWRLHC EJODGW HWWYJC NEVMPH TKGKRK UDEMUI UJIPJG VFRHND XMDYBJ
Ready
```


Back to Table of Contents

SMTP.SERVER “name”, “ipaddress”{, port{, “usernameb64”, “passwordb64”}}

Program mode only. Prepare the SMTP network stack with parameters for a subsequent **SMTP.SEND** command. SMTP operation requires a properly configured network connection with a reachable mail server that supports either NON or AUTH LOGIN access.

- The first two parameters are required and specify the mail server name, and the mail server’s IPv4 address.
- The third parameter specifies the TCP/IP port which defaults to 25 if not specified.
- The last two parameters are optional and are used to perform an authenticated login in response to an AUTH LOGIN status from the mail server. The usernameb64 is the response sent to the Username: prompt encoded as a Base64 string and the passwordb64 is the response sent the Password: prompt encoded as a Base64 string. No other authentication mechanism is supported.

Specify a connection to an intranet mail relay at IPv4 address 192.168.1.100 on port 25:

```
20 SMTP.SERVER "192.168.1.100", “192.168.1.100”
```

Specify a connection to mail server “mail.domain.com” at IPv4 address 216.119.100.100 on port 2525 with **usernameb64** of “sender@domain.com” base64 encoded as " c2VuZGVyQGRvbWFpbi5jb20=" and **passwordb64** of “1234” base64 encoded as " MTIzNA==". An online tool such as http://www.base64encode.org used to encode the username and password:

```
20 SMTP.SERVER "mail.domain.com",”216.119.100.100”,2525," c2VuZGVyQGRvbWFpbi5jb20="," MTIzNA=="
```


Back to Table of Contents

SMTP.SEND “from”, “to”, “cc”, “subject”, “message”

Program mode only. Send a text “message” via the mail server previously specified by the **SMTP.SERVER** command “from”, “to”, “cc” with “subject”.

In this example a single string message is sent without login using an intranet mail relay:

```
10 REM smtp test
15 ONEVENT @SMTP.EVENT, GOSUB 100
20 SMTP.SERVER "192.168.1.100", “192.168.1.100”
25 SMTP.SEND "cfsound4@acscontrol.com", "support@acscontrol.com", "", "test", "this is a simple
text message"
30 GOTO 30
100 REM SMTP Event Handler
105 ON @SMTP.EVENT,GOTO 115,120,125,130,135,140,145,150,155,160,165
110 PRINT "unknown event" : RETURN
115 PRINT "SMTP OK: " + @SMTP.MESSAGE$ : RETURN
120 PRINT "SMTP ABORTED: " + @SMTP.MESSAGE$ : STOP
125 PRINT "SMTP CONNECT: " + @SMTP.MESSAGE$ : STOP
130 PRINT "SMTP HELO: " + @SMTP.MESSAGE$ : STOP
135 PRINT "SMTP AUTH: " + @SMTP.MESSAGE$ : STOP
140 PRINT "SMTP FROM: " + @SMTP.MESSAGE$ : STOP
145 PRINT "SMTP TO: " + @SMTP.MESSAGE$ : STOP
150 PRINT "SMTP CC: " + @SMTP.MESSAGE$ : STOP
155 PRINT "SMTP DATA: " + @SMTP.MESSAGE$ : STOP
160 PRINT "SMTP QUEUE: " + @SMTP.MESSAGE$ : STOP
165 PRINT "SMTP SUCCESSFUL: " + @SMTP.MESSAGE$ : STOP
Ready
run
SMTP OK: 250-acsservr Hello [192.168.1.200]
SMTP OK: 250-TURN
SMTP OK: 250-ATRN
SMTP OK: 250-SIZE 2097152
SMTP OK: 250-ETRN
SMTP OK: 250-PIPELINING
SMTP OK: 250-DSN
SMTP OK: 250-ENHANCEDSTATUSCODES
SMTP OK: 250-8bitmime
SMTP OK: 250-BINARYMIME
SMTP OK: 250-CHUNKING
SMTP OK: 250-VRFY
SMTP OK: 250 OK
SMTP SUCCESSFUL: 221 2.0.0 acsservr Service closing transmission channel

STOP in line 165
Ready
```

Here’s the received e-mail with headers:

Here’s the received e-mail with headers:

```
Return-Path: <cfsound4@acscontrol.com>
Received: from acsservr (pool-96-252-xxx-xxx.tampfl.fios.verizon.net [96.252.xxx.xxx]) by
maila20.webcontrolcenter.com with SMTP;
   Wed, 12 Mar 2014 08:24:37 -0700
Received: from 192.168.1.100 ([192.168.1.200]) by acsservr with Microsoft SMTPSVC(5.0.2195.7381);
               Wed, 12 Mar 2014 10:24:36 -0500
From: cfsound4@acscontrol.com
To: support@acscontrol.com
Subject: test
Return-Path: cfsound4@acscontrol.com
Message-ID: <ACSSERVRhzsDYlYhiIA000002cf@acsservr>
X-OriginalArrivalTime: 12 Mar 2014 15:24:36.0421 (UTC) FILETIME=[2D634F50:01CF3E07]
Date: 12 Mar 2014 10:24:36 -0500
X-SmarterMail-TotalSpamWeight: 0 (Authenticated)

this is a simple text message
```


Back to Table of Contents

SMTP.SEND #N, “from”, “to”, “cc”, “subject” {, “header”}

Program mode only. Send the contents of a previously opened file #N via the mail server previously specified by the **SMTP.SERVER** command “from”, “to”, “cc” with “subject” and optional “header” that can specify MIME content headers to allow HTML e-mails to be sent from a file.

In this example a HTML file is sent via mail server “mail.domain.com” at the specified IPv4 address and port using an authenticated login and the requisite MIME content header:

```
10 REM smtp test
15 ONEVENT @SMTP.EVENT, GOSUB 100
20 SMTP.SERVER "mail.domain.com",”216.119.100.100”,2525," c2VuZGVyQGRvbWFpbi5jb20=","  MTIzNA=="
25 OPEN #0,"SampleEmail.html","rb"
30 Header$ = "MIME-Version 1.0" + CHR$(13)+CHR$(10) + "Content-Type: text/html; charset=" +
CHR$(34) + "ISO-8859-1" + CHR$(34) + "; "
35 SMTP.SEND #0, "cfsound4@domain.com", "support@acscontrol.com", "", "test", Header$
40 GOTO 40
100 REM SMTP Event Handler
105 ON @SMTP.EVENT,GOTO 115,120,125,130,135,140,145,150,155,160,165
110 PRINT "unknown event" : RETURN
115 PRINT "SMTP OK: " + @SMTP.MESSAGE$ : RETURN
120 PRINT "SMTP ABORTED: " + @SMTP.MESSAGE$ : STOP
125 PRINT "SMTP CONNECT: " + @SMTP.MESSAGE$ : STOP
130 PRINT "SMTP HELO: " + @SMTP.MESSAGE$ : STOP
135 PRINT "SMTP AUTH: " + @SMTP.MESSAGE$ : STOP
140 PRINT "SMTP FROM: " + @SMTP.MESSAGE$ : STOP
145 PRINT "SMTP TO: " + @SMTP.MESSAGE$ : STOP
150 PRINT "SMTP CC: " + @SMTP.MESSAGE$ : STOP
155 PRINT "SMTP DATA: " + @SMTP.MESSAGE$ : STOP
160 PRINT "SMTP QUEUE: " + @SMTP.MESSAGE$ : STOP
165 PRINT "SMTP SUCCESSFUL: " + @SMTP.MESSAGE$ : STOP
Ready
run
SMTP OK: 250-maila20.webcontrolcenter.com Hello [96.252.xxx.xxx]
SMTP OK: 250-SIZE 31457280
SMTP OK: 250-AUTH LOGIN CRAM-MD5
SMTP OK: 250 OK
SMTP SUCCESSFUL: 221 Service closing transmission channel

STOP in line 165
Ready
```

Here is the contents of file SampleEmail.html – notice that the file is opened for reading in binary mode:

```
<html>
<body>
This is a test HTML e-mail message sent from a file
<p><p>
<a href="http://www.cfsound.com">http://www.cfsound.com</a>
</body>
</html>
```

And here's the received e-mail (with headers):

```
Return-Path: <cfsound4@acscontrol.com>
Received: from mail.domain.com (pool-xxx-xxx-xxx-xxx.tampfl.fios.verizon.net [xxx.xxx.xxx.xxx]) by
maila20.webcontrolcenter.com with SMTP;
   Tue, 11 Mar 2014 07:27:16 -0700
From: cfsound4@domain.com
To: support@acscontrol.com
Subject: test
MIME-Version: 1.0
Content-Type: text/html; charset="ISO-8859-1";
```


Back to Table of Contents

```
Content-Transfer-Encoding: 8bit;
Message-ID: <58e95feaed60455b987e168b4a279b33@com>
X-SmarterMail-TotalSpamWeight: 0 (Authenticated)


      This is a test HTML e-mail message sent from a file
      http://www.cfsound.com
```

SOCKET.ASYNC.CONNECT #N, "ip:port", connect( ), send( ), recv( )

Program mode only. Initiate an outgoing asynchronous network socket connection as file #N using the string representation of the IPv4 IP address and port number. If the socket opens without error execution continues with the following statement. The status of the connection, send, receive and disconnect process is returned via the **@SOCKET.EVENT[#N]** system variable.

- The **connect( )** user function is called when a connection is established.
- The **send( )** user function is called to send data to the connected device using PRINT #N or FPRINT #N statement(s). Return zero to terminate the connection, one to proceed to the **recv( )** function and two to be called again to send more data.
- The **recv( )** user function is then called to receive data from the connected device using INPUT #N or FINPUT #N statements(s). Return zero to terminate the connection, one to return to the **send( )** function and two to be called again to receive more data.

See the Socket Programming section below for more information and sample programs.

SOCKET.ASYNC.LISTEN #N, ":port", connect( ), recv( ), send( )

Program mode only. Initiate an incoming asynchronous network socket reception as file #N using the string representation of the IPv4 port number. If the socket opens without error execution continues with the following statement. The status of the connection, receive, send and disconnect process is returned via the **@SOCKET.EVENT[#N]** system variable.

- The **connect( )** user function is called when a connection is established.
- The **recv( )** user function is then called to receive data from the connected device using INPUT #N or FINPUT #N statements(s). Return zero to terminate the connection, one to return to the **send( )** function and two to be called again to receive more data.
- The **send( )** user function is called to send data to the connected device using PRINT #N or FPRINT #N statement(s). Return zero to terminate the connection, one to proceed to the **recv( )** function and two to be called again to send more data.

See the Socket Programming section below for more information and sample programs.

In the **BETA** software there is an optional 'continue( )' function at the end of the **SOCKET.ASYNC.LISTEN** statement which is called when the connection disconnects. Return non-zero to loop back and wait for another connection without exiting the statement. Note that the function cannot be called continue( ) as this would conflict with the **CONTINUE** statement. To continually remain in the **SOCKET.ASYNC.LISTEN** statement it is sufficient to code a '1' as the continue( ) function expression:

SOCKET.ASYNC.LISTEN #0, ":1000", connect( ), recv( ), send( ), relisten( )
or
SOCKET.ASYNC.LISTEN #0, ":1000", connect( ), recv( ), send( ), 1

Back to Table of Contents

STOP

Program mode only. Terminate the program and issue a **STOP** message. Closes all open files.

```
10 a=a+1
20 STOP
Ready
run
STOP in line 20
Ready
```

TYPE path

Program or Direct mode. Display the contents of a SD card filename named ***path*** as ASCII characters on the serial port. ***Path*** may be a constant string or you can use a string variable as the ***path*** by concatenating it to such a string: ***TYPE*** ""***+PATH$***. In Direct mode the quotes are not required.

A double escape sequence will stop the portion of the file that the CFSound not already queued.

VARS

Program or Direct mode. Display a table of the name, type and values of the variables that have been defined or created by use on the serial port.

Before a program has been RUN for the first time there are no variables defined.

```
vars
wait_here            -> r/o Label        = line 50
start_async_listen   -> r/o Label        = line 1010
socket_event_handler -> r/o Label        = line 1040
none                 -> r/o Label        = line 1070
ok                   -> r/o Label        = line 1080
no_open              -> r/o Label        = line 1090
no_connect           -> r/o Label        = line 1100
send_err             -> r/o Label        = line 1110
recv_err             -> r/o Label        = line 1120
network_err          -> r/o Label        = line 1125
connect              -> r/w IntFunction  = 0 @ line 1140
send                 -> r/w IntFunction  = 0 @ line 1180
exit_send            -> r/o Label        = line 1220
recv                 -> r/w IntFunction  = 0 @ line 1250
set_relays           -> r/o Label        = line 2010
send_status          -> r/o Label        = line 2120
cmd$                 -> r/w Str$         = ""
relays               -> r/w Int          = 0
Ready
```


Back to Table of Contents

WAIT @systemvar

Execution pauses at this statement until the associated system variable has been signaled.

In the **BETA** software the WAIT statement has been renamed:

EVENT.WAIT @systemvar

Note that all statements on the same line before the WAIT are executed continuously while waiting.

In this example, program execution would pause at line 110 until all of the queued sounds had finished playing:

```
10 @SOUND$="one.wav"
20 @SOUND$="two.wav"
30 @SOUND$="three.wav"
40 @SOUND$="four.wav"
50 @SOUND$="five.wav"
60 @SOUND$="six.wav"
70 @SOUND$="seven.wav"
80 @SOUND$="eight.wav"
90 @SOUND$="nine.wav"
100 @SOUND$="ten.wav"
110 WAIT @SOUND$
```

In this example, program execution would pause at line 40 until all of the queued serial data had finished sending:

```
10 REM test @EOT
20 FOR I=1 TO 10:PRINT "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ": NEXT I
40 WAIT @EOT
50 PRINT "EOT"
Ready
run
0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ
EOT
Ready
```

In this ***incorrect example***, program execution would lock forever on line 20 since all statements on the same line before the WAIT are executed continuously while waiting. Since these statements reload the timer that the WAIT is waiting on, the program will never execute past this line:

```
5 REM Wrong use of the WAIT statement
10 PRINT "start timer[]:wait timer[]"
20 @TIMER[0]=50:WAIT @TIMER[0]
30 PRINT "done"
Ready
run
start timer():wait timer()
 ESC at line 20
Ready
```


Back to Table of Contents

WHILE test : statements : WEND

Program mode only. Conditional execution code block loop. The expression ***test*** is evaluated, and if non-zero, program execution continues with the following ***statements***. If the test expression evaluates to zero, program execution continues at the statements following the **WEND**.

- **WHILE / WEND** blocks can be nested and don't have to be the only statements on the line.
- **WHILE / WEND** blocks can be exited from within without the test expression evaluating to zero using the **BREAK** statement.
- **WHILE / WEND** blocks can be continued from within without executing all of the statements using the **CONTINUE** statement.
- As the **WHILE / WEND** code block loop uses the control stack to execute you should not jump out of or into the code block loop of statements. To leave the loop, force the **test** expression to return zero or use the **BREAK** statement.
- Execution of a **WHILE** without a subsequent **WEND** will result in a "Nesting Error".
- Execution of a **WEND** without a preceding **WHILE** will result in a "Nesting Error".

Here are some examples:

```
10 REM while/wend test
15 WHILE a < 10 : PRINT a : a = a + 1 : WEND
Ready
run
0
1
2
3
4
5
6
7
8
9
Ready
```

```
10 REM while/wend test
15 WHILE a < 10
20  PRINT a
25  a = a + 1
30 WEND
Ready
run
0
1
2
3
4
5
6
7
8
9
Ready
```

```
10 REM while/wend test
15 WHILE a < 10
20  WHILE a < 5
22   a = a + 1
23  WEND
25  a = a + 1
30  PRINT a
35 WEND
Ready
run
6
7
8
9
10
Ready
```

Graphics Statements

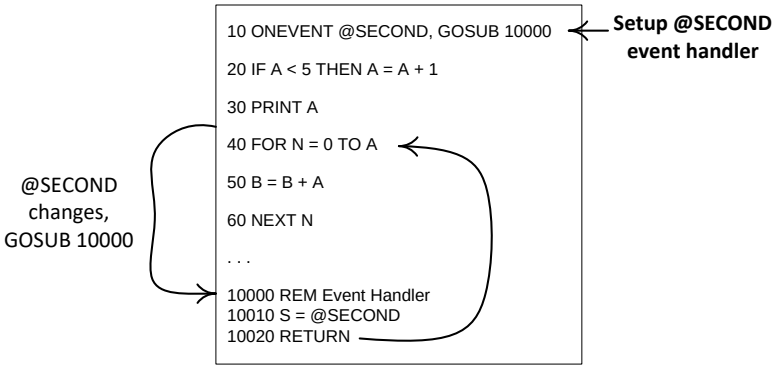
Additional statements and commands supporting graphics are outlined in the **BASIC Graphics Programming** manual available online.

Back to Table of Contents

Events

BASIC provides the concept of an ***Event***. Events occur outside of the normal program execution flow and are processed in between the execution of individual program statements. Some system variables have ***Events*** associated with them and may be referenced in **ONEVENT, SIGNAL** and **WAIT** statements. There are three ways to process an event: asynchronously with an **ONEVENT** handler, synchronously with a **WAIT** statement or by polling the system variable's value in the program to see when the event occurs.

In order to process an event asynchronously, BASIC has to be informed of what code to execute when a certain event occurs. This is done using the **ONEVENT** statement. After BASIC executes each program statement, it scans the table of events looking to see if any have been signaled. If an **ONEVENT** handler for a signaled event has been specified by the program, then BASIC will force a subroutine call to the event handler before the next program statement is executed. The event is 'cleared' when the subroutine **RETURN**s.



Events have an implicit priority with higher priority events being able to interrupt execution of lower priority event handlers. Here's an example of **@SECOND** event handling:

```
10 REM show time every second
15 ONEVENT @SECOND,GOSUB 100
20 GOTO 20
100 PRINT USING "%c%2d:%02d:%02d", 13, @HOUR, @MINUTE, @SECOND;
105 RETURN
Ready
run
15:54:18
```

This prints the current time, as it changes, once per second, on the serial port.

In order to handle an event synchronously a program may wait for an event to occur by using the **WAIT** statement. Program execution stalls at that statement until the specified event happens. Alternatively, the program may poll the associated system variable's value in a loop looking for the event to have been signaled. Here's an example of polling for the **@SECOND** system variable to change:

```
10 REM poll @SECOND
15 Seconds = @SECOND
20 LIF Seconds <> LastSeconds THEN PRINT Seconds : LastSeconds = Seconds
25 GOTO 15
Ready
run
54
55
56
 ESC at line 15
Ready
```

This would print the value of **@SECOND** system variable every time it changes – once per second.

Back to Table of Contents

The **SIGNAL** statement may be used in a program to force an event to happen.

It is very important to note that the **ONEVENT** handler subroutine executes in the context of the running program: it has access to all program variables. Since the event handler may be executed at any time in between any program statements care should be used when changing program variables from within an event handler as it may cause unexpected results in the execution of other program statements that may be using and depending upon the values of those same variables. Incorrect or unexpected program execution may result – code event handlers carefully.

See the **ONEVENT** statement definition above for a table showing what events may be processed and listing their relative priority.

Back to Table of Contents

User Defined Functions

BASIC also provides the ability for the user to write and call functions that are defined using statements in the program. Two statements provide this capability; **FUNCTION** and **ENDFUNCTION**.

The **FUNCTION** statement starts a function definition. It is followed by a variable name that also identifies its type as integer or string ($). In the **BETA** software a function returning a real value is identified with a trailing (%). All program statements between the **FUNCTION** and **ENDFUNCTION** statements comprise the function body.

The function variable name is then followed by a parenthesized list of zero or more parameter variables. Parameter variables exist only within the function body and receive the values of the function argument expressions when the function is called. The name of the function becomes a defined variable that is global to the entire program. Any additional statements following on the same line as the **FUNCTION** statement are ignored.

The **ENDFUNCTION** statement ends a function definition. Any additional statements following on the same line are ignored. When the defined function is called the **ENDFUNCTION** statement behaves like a **RETURN** – program execution continues after from where the **FUNCTION** was called.

User Functions may be located at the beginning of the program, or elsewhere. At program startup the entire program is scanned and all of the function definitions are recorded as global variables. After they have been defined the functions may be called by their name within the program to execute the statements that they encompass.

The function call actually invokes a defined function by creating the function's parameter list variables, which will only exist and be accessible from within the function body. The expressions in the argument list of the function call are then evaluated and the resulting values are assigned to the matching function parameters by position. Then the function body's statements are executed. The function parameter variables and any additional variables that are created inside the function exist and are valid throughout the execution of the function (and any nested function calls) and are discarded upon execution of the **ENDFUNCTION** command.

Here are a couple of examples of simple function definitions:

```
100 FUNCTION MyFunction(parm1, parm2$)
110  FOR I = 1 to parm1                  Defines an integer function named MyFunction
120   PRINT parm2$                       that takes two parameters; an integer named
130  NEXT I                              parm1 and a string named parm2$
150 ENDFUNCTION


. . .                                    Calls the integer function MyFunction passing
1000 MyFunction(10, "Test")              in two arguments; 10 for parameter parm1 and
. . .                                    "Test" for parameter parm2


100 FUNCTION Test$()
110  Test$ = "Result"                    Defines a string function named Test$ that
120 ENDFUNCTION                          takes no parameters


. . .
1000 PRINT Test$()                       Calls the string function Test$ with no
. . .                                    arguments and PRINTs its return value
```

Values may be passed to a function in three ways; through the function call parameter list, through the global function's variable name or through any other variable that is defined outside of the execution of the function.

Values may be returned from a function in two ways; through the global function's variable name or through any other variable defined outside of the execution of the function.

The operation of user functions is better shown with some examples. In this first example a simple integer function named **test( )** is defined that takes three arguments and adds them together with the sum

Back to Table of Contents

returned through the function name by assignment in line 110. Lines 100 through 115 define the function. In lines 15 and 20 the function **test( )** is called twice with different arguments for the function parameters:

```
10 REM integer function
15 test(1,2,3):PRINT "= ";test
20 test(4,5,6):PRINT "= ";test
25 END
100 FUNCTION test(arg1,arg2,arg3)
105  PRINT "test(";arg1;",";arg2;",";arg3;") ";
110  test=arg1+arg2+arg3
115 ENDFUNCTION
Ready
run
test(1,2,3) = 6
test(4,5,6) = 15
Ready
vars
test -> r/w IntFunction  = 15 @ line 100
Ready
```

The first thing to observe is if we **STOP** the program inside of the **FUNCTION** you can see the parameter variables that receive values from the function arguments using the **VARS** command. As these parameter variables exist only within the function body they are discarded when the function returns from the function call at the **ENDFUNCTION** statement:

```
112 stop
list
10 REM integer function
15 test(1,2,3):PRINT "= ";test
20 test(4,5,6):PRINT "= ";test
25 END
100 FUNCTION test(arg1, arg2, arg3)
105  PRINT "test(";arg1;","; arg2;","; arg3;") ";
110  test=arg1+arg2+arg3
112 STOP
115 ENDFUNCTION
Ready
run
test(1,2,3) STOP in line 112
Ready
vars
test   -> r/w IntFunction  = 6 @ line 100
 arg1  -> r/w Int          = 1
 arg2  -> r/w Int          = 2
 arg3  -> r/w Int          = 3
Ready
```

In the prior example, in lines 15 and 20 we are calling the **test( )** function and then **PRINT**ing its value. This can be combined into a single step by calling the function with arguments from the **PRINT** statement:

```
list
10 REM integer function
15 PRINT "= ";test(1,2,3)
20 PRINT "= ";test(4,5,6)
25 END
100 FUNCTION test(arg1,arg2,arg3)
105  PRINT "test(";arg1;",";arg2;",";arg3;") ";
110  test=arg1+arg2+arg3
115 ENDFUNCTION
Ready
run
= test(1,2,3) 6
= test(4,5,6) 15
Ready
```

Here's an example of a similar string function named **test$( )** that takes three string arguments and concatenates them together with the result returned by assignment to the function name:

Back to Table of Contents

```
10 REM string function
15 PRINT "= ";test$("one","two","three")
20 PRINT "= ";test$("four","five","six")
25 END
100 FUNCTION test$(arg1$,arg2$,arg3$)
105  PRINT "test$(";arg1$;",";arg2$;",";arg3$;") ";
110  test$=arg1$+arg2$+arg3$
115 ENDFUNCTION
Ready
run
= test$(one,two,three) onetwothree
= test$(four,five,six) fourfivesix
Ready
```

The following example is a string function named **test$( )** that takes a mixed string / integer parameter list – the first parameter named **in$** passes in the string value and the second parameter named **howmany** tells the function how many times to concatenate the input string with itself. Notice how the function variable is global and retains its value outside of the function – in this case causing an unintended side effect of the prior call's result being prefixed to the second function call's result:

```
10 REM functions
15 PRINT "= ";test$("A",10)
20 PRINT "= ";test$("abc",20)
25 END
100 FUNCTION test$(in$, howmany)
105  PRINT "test$(";in$;",";howmany;") ";
110  FOR count=1 TO howmany:test$=test$+in$:NEXT count
115 ENDFUNCTION
Ready
run
= test$(A,10) AAAAAAAAAA
= test$(abc,20) AAAAAAAAAAabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabc
Ready
```

Function calls may also be nested – you can call a function from within a function.

In the following example the function **cube$( )** calls function **cube( )** to obtain the integer value of its argument cubed, then converts that to a string that is **PRINT**ed. Note that the **FUNCTION** definitions can be in any order as they are identified and defined when the program starts. Also note that the functions appear as variables with assigned values that may be viewed using the **VARS** command:

```
10 REM functions
40 PRINT cube$(10)
50 END
100 REM define functions
105 FUNCTION cube(value)
110  cube=value*value*value
115 ENDFUNCTION
120 FUNCTION cube$(value)
125  cube$=STR$(cube(value))
130 ENDFUNCTION
Ready
run
1000
Ready
vars
cube  -> r/w IntFunction  = 1000 @ line 105
cube$ -> r/w Str$Function = "1000" @ line 120
Ready
```

Functions can be dynamically redefined – they are treated as variables – as long as the type of the function (integer / string) remains the same. In this example the **compute( )** function is redefined to calculate the square of its argument the first time, then the cube of its argument the next time by executing through the function definitions dynamically it using a **GOSUB / RETURN**:

Back to Table of Contents

```
10 REM function redefinition
20 GOSUB 1000 : REM define compute FUNCTION as square
30 PRINT compute(10)
40 GOSUB 1100 : REM define compute FUNCTION as cube
50 PRINT compute(10)
60 END
1000 REM compute square
1010 FUNCTION compute(value)
1020  compute = value * value
1030 ENDFUNCTION
1040 RETURN : REM required for GOSUB redefinition
1100 REM compute cube
1110 FUNCTION compute(value)
1120  compute = value * value * value
1130 ENDFUNCTION
1140 RETURN : REM required for GOSUB redefinition
Ready
run
100
1000
Ready
vars
compute -> r/w IntFunction  = 1000 @ line 1110
Ready
```

Another example of functions calling functions:

```
10 REM function nesting
30 PRINT ten(2)
999 END
1000 FUNCTION ten(value)
1010  nine(value) : ten = value * nine : PRINT "nine ",
1020 ENDFUNCTION
1030 FUNCTION nine(value)
1040  eight(value) : nine = value * eight : PRINT "eight ",
1050 ENDFUNCTION
1060 FUNCTION eight(value)
1070  seven(value) : eight = value * seven : PRINT "seven ",
1080 ENDFUNCTION
1090 FUNCTION seven(value)
1100  six(value) : seven = value * six : PRINT "six ",
1110 ENDFUNCTION
1120 FUNCTION six(value)
1130  five(value) : six = value * five : PRINT "five ",
1140 ENDFUNCTION
1150 FUNCTION five(value)
1160  four(value) : five = value * four : PRINT "four ",
1170 ENDFUNCTION
1180 FUNCTION four(value)
1190  three(value) : four = value * three : PRINT "three ",
1200 ENDFUNCTION
1210 FUNCTION three(value)
1220  two(value) : three = value * two : PRINT "two ",
1230 ENDFUNCTION
1240 FUNCTION two(value)
1250  one(value) : two = value * one : PRINT "one ",
1260 ENDFUNCTION
1270 FUNCTION one(value)
1280  one = value
1290 ENDFUNCTION
Ready
run
one two three four five six seven eight nine 1024
Ready
vars
ten   -> r/w IntFunction  = 1024 @ line 1000
nine  -> r/w IntFunction  = 512 @ line 1030
eight -> r/w IntFunction  = 256 @ line 1060
seven -> r/w IntFunction  = 128 @ line 1090
six   -> r/w IntFunction  = 64 @ line 1120
five  -> r/w IntFunction  = 32 @ line 1150
four  -> r/w IntFunction  = 16 @ line 1180
three -> r/w IntFunction  = 8 @ line 1210
```


Back to Table of Contents

```
two   -> r/w IntFunction  = 4 @ line 1240
one   -> r/w IntFunction  = 2 @ line 1270
Ready
```

And again – if we add a **STOP** statement in the program in the deepest nested function you can see the nested function arguments – notice that each function uses a parameter variable named **value** but these variables are local to each function. This is shown in the following **VARS** command output by the indentation:

```
1285 stop
list
10 REM function nesting
30 PRINT ten(2)
999 END
1000 FUNCTION ten(value)
1010  nine(value) : ten = value * nine : PRINT "nine ",
1020 ENDFUNCTION
1030 FUNCTION nine(value)
1040  eight(value) : nine = value * eight : PRINT "eight ",
1050 ENDFUNCTION
1060 FUNCTION eight(value)
1070  seven(value) : eight = value * seven : PRINT "seven ",
1080 ENDFUNCTION
1090 FUNCTION seven(value)
1100  six(value) : seven = value * six : PRINT "six ",
1110 ENDFUNCTION
1120 FUNCTION six(value)
1130  five(value) : six = value * five : PRINT "five ",
1140 ENDFUNCTION
1150 FUNCTION five(value)
1160  four(value) : five = value * four : PRINT "four ",
1170 ENDFUNCTION
1180 FUNCTION four(value)
1190  three(value) : four = value * three : PRINT "three ",
1200 ENDFUNCTION
1210 FUNCTION three(value)
1220  two(value) : three = value * two : PRINT "two ",
1230 ENDFUNCTION
1240 FUNCTION two(value)
1250  one(value) : two = value * one : PRINT "one ",
1260 ENDFUNCTION
1270 FUNCTION one(value)
1280  one = value
1285 STOP
1290 ENDFUNCTION
Ready
run
STOP in line 1285
Ready
vars
ten             -> r/w IntFunction  = 0 @ line 1000
nine            -> r/w IntFunction  = 0 @ line 1030
eight           -> r/w IntFunction  = 0 @ line 1060
seven           -> r/w IntFunction  = 0 @ line 1090
six             -> r/w IntFunction  = 0 @ line 1120
five            -> r/w IntFunction  = 0 @ line 1150
four            -> r/w IntFunction  = 0 @ line 1180
three           -> r/w IntFunction  = 0 @ line 1210
two             -> r/w IntFunction  = 0 @ line 1240
one             -> r/w IntFunction  = 2 @ line 1270
 value          -> r/w Int          = 2
  value         -> r/w Int          = 2
   value        -> r/w Int          = 2
    value       -> r/w Int          = 2
     value      -> r/w Int          = 2
      value     -> r/w Int          = 2
       value    -> r/w Int          = 2
        value   -> r/w Int          = 2
         value  -> r/w Int          = 2
          value -> r/w Int          = 2
Ready
```


Back to Table of Contents

Passing Arrays By Reference

In the **BETA** software the ability to pass arrays to a function has been added. Normally, when a user defined function is called the parameter variables in the function definition are created and then the expressions in the function call argument are evaluated and are assigned to the parameter variables. This is referred to as passing by value since a copy of the argument variable's contents is used to initialize the parameter variable.

The only way for a function to access an array was to **DIM**ension the array outside of the function, then refer to that global array by its name within the function body. This precluded the ability to write an array handling function that could be called to operate on multiple arrays as they could not be passed to the function as an argument.

Passing arrays to a function by value would require making copies of the entire array contents which could take a long time and require excessive memory. In addition any changes made to the array inside the function would be lost when the function returned unless the array was copied back out which also may not be desirable. So instead array arguments are passed to the function parameter variables by reference. The pass by reference operation is denoted by a leading backslash character '\' immediately in front of the array argument.

In this example two integer arrays of different sizes are defined; array1[ ] and array2[ ]. Two user defined functions; InitializeArray( ) and PrintArray( ) are defined that take a numeric parameter variable as an argument. Note the use of the **UBOUND( )** built-in function to allow the **FOR / NEXT** loops to adjust for the size of the referenced array parameter. These user defined functions are then called to initialize and print both DIMensiond arrays, passing them by reference:

```
10 REM Function Pass Array by Reference
20 DIM array1[10], array2[20]
30 InitializeArray(\array1) : InitializeArray(\array2)
40 PrintArray(\array1) : PrintArray(\array2)
50 END
100 FUNCTION InitializeArray(array)
110  FOR i = 0 TO UBOUND(array)-1 : array[i] = i : NEXT i
120 ENDFUNCTION
200 FUNCTION PrintArray(array)
210  FOR i = 0 TO UBOUND(array)-1 : PRINT array[i];", "; : NEXT i
220  PRINT ""
230 ENDFUNCTION
Ready
run
0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
Ready
```


Back to Table of Contents

Errors

The following errors can be produced. The placeholder ‘dd’ in the message is replaced with the line number where the error was detected if the error was encountered in a running program. Some Syntax Errors will provide additional information after the line number further identifying the error:

ERR( )	ERR$( )	Causes
1	"Syntax error in line dd"	Incorrect statement format
2	"Illegal program command error in line dd"	Direct mode only statement in program mode
3	"Illegal direct command error in line dd"	Program mode only statement in direct mode
4	"Line number error in line dd"	Target line number not in program
5	"Wrong expression type error in line dd"	Numeric value when String expected or vice versa
6	"Divide by zero error in line dd"	Division by zero
7	"Nesting error in line dd "	NEXT without preceding FOR, RETURN without preceding GOSUB, etc.
8	"File not open error in line dd "	CLOSE#, LIST#, PRINT# or INPUT# without successful OPEN statement
9	"File already open error in line dd "	OPEN# on already open file
10	“File # Out of Range in line dd”	File # out of range 0 - 23
11	"Input error in line dd "	Numeric value expected in INPUT # statement
12	"Dimension error in line dd "	Dimension error
13	“Index out of range in line dd”	Subscript out of range
14	"Data error in line dd "	ORDER line # not DATA statement, READ past DATA statements
15	"Out of memory error in line dd "	Insufficient memory
16	"No File System error in line dd "	BASIC running without SD card
17	“Unknown @var error in line dd “	Unknown system variable
18	"Timer # out of range error in line dd "	@TIMER(x) subscript out of range 0 - 9
19	"Port # out of range error in line dd "	@PORT(x) subscript out of range 0 - 255
20	"Contact # out of range error in line dd "	@CONTACT(x), @CLOSURE(x), @OPENING(x) subscript out of range
21	"Stack Overflow error in line dd "	Too many nested FOR and/or GOSUB and/or events
22	"No SD card error in line dd "	Statement requiring SD card with no card detected
23	"Invalid .WAV file error in line dd "	.WAV file format not 44.1KHz 16-bit mono or stereo or @SOUND$ queue full
24	“DRAW.x arguments Out of Range error in line dd”	One or more argument to a DRAW.x statement are out of range
25	“FWRITE record # Out of Range error in line dd”	Attempt to FWRITE to a record number that is past the immediate end of file
26	“FWRITE exceeds record length error in line dd”	Length of data in FWRITE variables list including commas and quotes exceeds the record length specified in the associated FOPEN
27	“FINSERT record # Out of Range error in line dd”	Attempt to FINSERT to a record number that is past the immediate end of file
28	“FINSERT exceeds record length error in line dd”	Length of data in FINSERT variables list including commas and quotes exceeds the record length specified in the associated FOPEN
29	“FDELETE past end of file error in line dd ”	FDELETE record number exceeds file length
30	“Can’t delete file error in line dd”	Can’t delete file
31	“Can’t make directory error in line dd”	Can’t create directory
32	“Can’t rename file error in line dd”	Can’t rename file
33	“No DMX module”	
34	“DMX Channel # Out of Range”	
35	“DMX Analog # Out of Range”	
36	“DMX Analog # Read Only”	
37	“Unknown Command error in line dd”	BASIC doesn’t recognize the command
38	“Can’t use @VAR in line dd”	Illegal use of system variable in FOR, DIM, INPUT, READ, FREAD or FINPUT statement
39	“Mis-matched quotes in line dd”	Missing one of a pair of double quotes delimiting a string
40	“RGB” argument error”	Problem with an argument to the RGB( ) function
41	“Unsupported bitmap file”	Problem with filename argument to the DRAW.BITMAP statement
42	“FREAD record # out of range	
43	“Resource not found”	
44	“Resource already exists”	
45	“Font # out of range”	
46	“.fonts file invalid”	
47	“Scheme # out of range”	
48	“.schemes file invalid”	
49	“Obj # out of range”	

Back to Table of Contents

50	“Screen # out of range”	
51	“.screens file invalid”	
52	“Config # out of range”	
53	“Config Item < min or > max”	
54	“DRAW.POLYGON”	
55	“SD Card”	
56	“File System”	
57	“Read Only”	Attempt to write to a CONST variable
58	“Option # Out of Range”	
59	“Data # Out of Range”	
60	“SMTP Connection Failed”	
61 – 74	Internal Usage for experimental features	
75	“Run Time Library”	**BETA** Problem with argument to MATH.x%( ) command.
76 - 99	Internal Usage for experimental features	
100 - 32767	“x error in line dd”	ERROR x statement

Back to Table of Contents

Debugging and Troubleshooting Programs

First, we cannot stress enough the importance of developing and debugging your BASIC programs interactively. While you can write a small program using a text editor on your PC, write the program to the SD card as CFSOUND.BAS, install it into the CFSound-IV and have it run, if there are any spelling or logic errors the program may silently stop running leaving you with no clue as to what happened. You will waste a lot of time blindly making changes and re-trying your program without success.

To interactively develop your program using BASIC you must establish a communications channel between yourself and the CFSound-IV that allows you to interact with it. There are a few ways to do this, see the Communicating with the CFSound-IV section of this manual above for more information.

Once you're connected and communicating you can try your program again. Instead of naming the program as CFSOUND.BAS, try giving it a different, more descriptive name on the SD card. This accomplishes two things; first the program will not automatically run so you can control how and when it executes, and second you can have multiple copies of your program with different names so you can try different debugging techniques to determine what it happening. When you finally have your program debugged you can then save it as CFSOUND.BAS to make its execution automatic again.

Now that you're connected to the CFSound-IV, if there is an error when your program runs you will now see the error message that is produced. The error message produced by a running program usually references a line number that the error was encountered in. For example:

```
10 FOR i = 0 TO 90 step 5 : PRINT 1024 / COS(i);","; : NEXT i
Ready
run
1,1,1,1,1,1,1,1,1,1,1,2,2,2,3,5,11,Divide by zero error in line 10
Ready
```

To dissect this error first note that the message is indicating a problem with division, the only command that has a division operation in line 10 is the **PRINT**. We can use the **VARS** command to see the current variables and their values and a direct mode **PRINT** command to show the value that we're attempting to divide by **COS(i)**:

```
vars
i -> r/w Int          = 90
Ready
print cos(i)
0
Ready
```

Sure enough **COS(90) = 0** and so **1024 / COS(90)** is an attempt to divide by zero = error.

The table of Errors above has some short descriptions of possible causes for each error number.

Stack Overflow Errors

One type of error that may be difficult to debug is stack overflow. Think of a stack as a pile of papers. You can stack papers onto the pile, but it will only hold so much before it overflows. To get at a paper that was stacked earlier you have to remove items, one at a time, until you reach the sheet that you want. Thus the stack is a last-in, first out. Putting items onto the top of the stack is called 'pushing' and removing items off the top of the stack is called 'popping'.

BASIC uses a stack to remember where and what it was doing so that it can return to it later. The first and perhaps most obvious use is to implement the **GOSUB / RETURN** statements. When BASIC encounters a **GOSUB** command it remembers where it was by 'pushing' the location in the code where it was running on the stack, then it jumps to the subroutine and starts executing code there. When it encounters the **RETURN** statement in the subroutine, it 'pops' the information where it was running off the stack and resumes running there.

You can see that while **GOSUBs** can be 'nested' – that is to say a **GOSUB** can call a subroutine which contains another **GOSUB** to another subroutine, et cetera, it is important that each **GOSUB** must be

Back to Table of Contents

matched with a corresponding **RETURN** – otherwise the stack will overflow. Event handling also involves **GOSUB**ing to an event handler subroutine that ends with a **RETURN** statement.

BASIC also uses the stack to 'remember' other things. When executing **FOR / NEXT** loops, the program locations of the beginning and end of the loop, the index variable, the step value and the limit value are all pushed onto the stack and remain there for the duration of the loop's execution – when the **NEXT** statement ends the loop then the control information is popped off. The same is true for **WHILE** / **WEND** loops, **FUNCTION** / **ENDFUNCTION** calls and block style **IF / ELSE / ENDIF** statements – they all use the control stack.

So the stack can be overflowed by too many nested statements that utilize it, or by repeatedly calling a subroutine without returning. Here's an admittedly contrived example of repeatedly calling a subroutine using **GOSUB** without ever executing a **RETURN**:

```
10 GOSUB 100
100 A = A + 1
110 GOTO 10
Ready
run
Stack Overflow error in line 10 - GOSUB with full stack
Ready
vars
A -> r/w Int          = 63
Ready
```

Here's a similar contrived example of repeatedly calling a **FOR** loop without ever executing the **NEXT** statement:

```
10 FOR I = 1 TO 10
20 GOTO 10
30 NEXT I
Ready
run
Stack Overflow error in line 10 - FOR with full stack
Ready
```

And another contrived example of repeatedly entering a block style **IF / ENDIF**:

```
10 IF A < 10 THEN
15  GOTO 10
20 ENDIF
Ready
run
Stack Overflow error in line 10 - Block IF with full stack
Ready
```

And finally another contrived example of repeatedly calling a **FUNCTION** without ever executing the **ENDFUNCTION**:

```
10 MyFunction()
20 GOTO 20
100 FUNCTION MyFunction()
120  GOTO 10
130 ENDFUNCTION
Ready
run
Stack Overflow error in line 10 - function() with full stack
Ready
```

While these examples are deliberately constructed to cause the stack overflow error an error in logic or program construction can cause this type of error to concur.

Nesting Errors

As discussed in Stack Overflow Errors above, there are several commands that utilize the control stack to save and restore execution information. All of these commands consist of pairs of statements – one that pushes the information to save on the stack, and a corresponding statement that pulls the information to

Back to Table of Contents

restore from the stack. A nesting error occurs when BASIC encounters one of the statements in the pair without finding or having encountered the other.

Here's a small example of a **FOR** statement that doesn't have a **NEXT** and another of a **NEXT** without a **FOR**:

```
10 FOR i = 0 TO 10
20  PRINT i
30 END
Ready
run
Nesting error in line 10 - FOR without matching NEXT
Ready

new
Ready
10 NEXT i
Ready
run
Nesting error in line 10 - NEXT without preceding FOR
Ready
```

Similarly a small example of a **WHILE** statement without a **WEND** and another of a **WEND** without a **WHILE**:

```
10 WHILE I < 10 : I = I + 1
Ready
run
Nesting error in line 10 - WHILE without matching WEND
Ready

new
Ready
10 I = I + 1 : WEND
run
Nesting error in line 10 - WEND doesn't match WHILE
Ready
```

One half of the pair doesn't have to be missing to cause a nesting error – if a program jumps into the middle of a pair then the two halves are encountered out of sequence and that will result in a nesting error:

```
10 GOTO 30
20 FOR ndx = 0 TO 9
30  a = ndx * 10
40 NEXT ndx
Ready
run
Nesting error in line 40 - NEXT without preceding FOR
Ready

new
Ready
10 GOTO 30
20 WHILE incr > 0
30  incr = incr - 1
40 WEND
Ready
run
Nesting error in line 40 - WEND doesn't match WHILE
Ready
```


Back to Table of Contents

Socket Programming

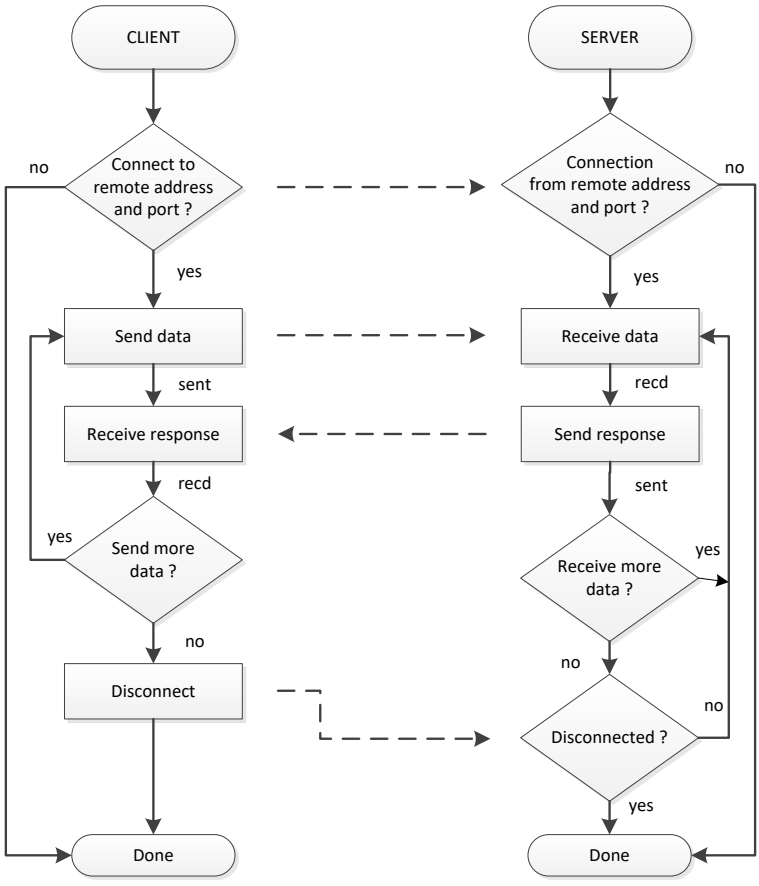
BASIC provides a set of primitives that allow networked devices to exchange messages using a pre-arranged protocol. One or more devices can communicate using TCP/IP over the Ethernet interface in a client/server fashion. The implementation of this network communication is referred to as "socket programming".

This network socket programming achieves communication between two machine applications using a client/server methodology – one device's application establishes itself as a network server that 'listens' for incoming TCP/IP connections on a specified port number. The other device's application implements a network client that 'connects' to a listening server with subsequent message exchange. The client device then 'disconnects' from the server and the process can repeat.

BASIC Sockets

The BASIC socket implementation is highly simplified and half-duplex – communication ping- pongs back and forth between the client and server devices. The Client device initiates a connection to a listening Server device and after the connection is established sends the first message then receives a response. The Server device listens for a connecting Client device and after connection is established receives the first message then sends a response. This is slightly different from conventional socket programming where each side can send and receive at the same time.

The following diagram is a simplified representation of the communication sequence:



There are two styles of socket programming – blocking or synchronous and non-blocking or asynchronous. BASIC has statements and functions to facilitate both styles. Synchronous sockets do not

Back to Table of Contents

allow any other program statements to be executed while the socket is connected, sending or receiving – program execution is totally consumed by the socket command. Asynchronous sockets initiate the socket operation and then return program execution to the program – the status of the socket command is returned via a system event variable.

Blocking Sockets

Synchronous or blocking socket operations are implemented using both built-in and user defined functions. The built-in function's return value provides the program with the status of the operation. The user defined functions are called by the built-in function to provide control points during the socket operation. There are two blocking functions to implement the client or server on a device.

Client Blocking Connection

The synchronous socket client connection takes the form:

```
SOCKET.SYNC.CONNECT(#N, "ip:port", connect( ), send( ), recv( ))
```

#N	File number
"ip:port"	String representation of IPv4 address and TCP/IP port number
connect( )	User function called when connection occurs
send( )	User function called to send data to 'file' #N
recv( )	User function called to receive data from 'file' #N

The **SOCKET.SYNC.CONNECT( )** function opens the TCP/IP network connection as an open file **#N** to the remote device using the **"ip.port"** string.

- When the connection is established the user provided **connect( )** function is called.
- When this function returns, the user provided **send( )** function is called to output data to the remote device using **PRINT #N** or **FPRINT #N** statements. The **send( )** function should return one to wait for received data, two to send more data or zero to disconnect.
- Next the user provided **recv( )** function is called to input data from the remote device using **INPUT #N** or **FINPUT #N** statements. The **recv( )** function should return one to send more data, two to receive more data or zero to disconnect.

The process repeats between **send( )** and **recv( )** calls until one of the functions returns zero or an error condition occurs. The **SOCKET.SYNC.CONNECT( )** function then returns a numeric value representing the "result" of the connection:

SOCKET.SYNC.CONNECT( ) returns	Description
0	Unknown / No Status
1	Disconnect / Done / No Error
2	Open Error – requested file **#N** failed to open
3	Connection Timeout – no connection within the **@SOCKET.TIMEOUT** interval
4	Data Send Timeout – no send data acknowledgment within the **@SOCKET.TIMEOUT** interval
5	Receive Data Timeout – no received data within the **@SOCKET.TIMEOUT interval**

Back to Table of Contents

Server Blocking Listen

The synchronous socket server listen takes the form:

```
SOCKET.SYNC.LISTEN(#N, ":port", connect( ), recv( ), send( ))
```

#N	File number
":port"	String representation of TCP/IP port number
connect( )	User function called when connection occurs
recv( )	User function called to receive data from 'file' #N
send( )	User function called to send data to 'file' #N

The **SOCKET.SYNC.LISTEN( )** function opens a TCP/IP port as file **#N** using the **":port"** string.

- When a remote client establishes a connection to the listening port the user provided **connect( )** function is called.
- When this function returns, the user provided **recv( )** function is called to input data from the remote device using **INPUT #N** or **FINPUT #N** statements. The **recv( )** function should return one to send more data, two to receive more data or zero to disconnect.
- Next the user provided **send( )** function is called to output data to the remote device using **PRINT #N** or **FPRINT #N** statements. The **send( )** function should return one to wait for received data, two to send more data or zero to disconnect.

The process repeats between **recv( )** and **send( )** calls until one of the functions returns zero or an error condition occurs. The **SOCKET.SYNC.LISTEN( )** function then returns a numeric value representing the "result" of the connection:

SOCKET.SYNC.LISTEN( ) returns	Description
0	Unknown / No Status
1	Disconnect / Done / No Error
2	Open Error – requested file **#N** failed to open
3	Connection Timeout – no connection within the **@SOCKET.TIMEOUT** interval
4	Data Send Timeout – no send data acknowledgment within the **@SOCKET.TIMEOUT** interval
5	Receive Data Timeout – no received data within the **@SOCKET.TIMEOUT** interval

Non-blocking Sockets

Asynchronous or non-blocking sockets are implemented using a statement to 'start' the communication, user functions to control the order and flow of data and a system variable to return the "result" of the connection. The user defined functions are called in-between other program statements that are executing to provide control points during the socket operation. There are two statements to 'start' the asynchronous socket communication.

Client Non-blocking Connection

The asynchronous client connection is started using the following program statement:

```
SOCKET.ASYNC.CONNECT #N, "ip:port", connect( ), send( ), recv( )
```

#N	File number
"ip:port"	String representation of IPv4 address and TCP/IP port number
connect( )	User function called when connection occurs
send( )	User function called to send data to 'file' #N
recv( )	User function called to receive data from 'file' #N

Back to Table of Contents

The **SOCKET.ASYNC.CONNECT** statement opens the TCP/IP network connection as an open file **#N** to the remote device using the **"ip.port"** string. If this is successful execution continues with the following program statements.

- When the connection is established the user provided **connect( )** function is called.
- When this function returns, the user provided **send( )** function is called to output data to the remote device using **PRINT #N** or **FPRINT #N** statements. The **send( )** function should return one to wait for received data, two to send more data or zero to disconnect.
- Next the user provided **recv( )** function is called to input data from the remote device using **INPUT #N** or **FINPUT #N** statements. The **recv( )** function should return one to send more data, two to receive more data or zero to disconnect.

The process repeats between **send( )** and **recv( )** calls until one of the functions returns zero or an error condition occurs. The @**SOCKET.EVENT[#N}** system variable then fires and event and returns a numeric value representing the "result" of the connection:

@SOCKET.EVENT[#N]	Description
0	Unknown / No Status
1	Disconnect / Done / No Error
2	n/a – the 'open' was done initially
3	Connection Timeout – no connection within the **@SOCKET.TIMEOUT** interval
4	Data Send Timeout – no send data acknowledgment within the **@SOCKET.TIMEOUT** interval
5	Receive Data Timeout – no received data within the **@SOCKET.TIMEOUT** interval

Server Non-blocking Listen

The asynchronous socket server listen takes the form:

```
SOCKET.ASYNC.LISTEN #N, ":port", connect( ), recv( ), send( )
```

#N	File number
":port"	String representation of TCP/IP port number
connect( )	User function called when connection occurs
recv( )	User function called to receive data from 'file' #N
send( )	User function called to send data to 'file' #N

The **SOCKET.ASYNC.LISTEN** statement opens a TCP/IP port as an open file **#N** using the **":port"** string. . If this is successful execution continues with the following program statements.

- When a remote client establishes a connection to the listening port the user provided **connect( )** function is called.
- When this function returns, the user provided **recv( )** function is called to input data from the remote device using **INPUT #N** or **FINPUT #N** statements. The **recv( )** function should return one to send more data, two to receive more data or zero to disconnect.
- Next the user provided **send( )** function is called to output data to the remote device using **PRINT #N** or **FPRINT #N** statements. The **send( )** function should return one to wait for received data, two to send more data or zero to disconnect.

The process repeats between **recv( )** and **send( )** calls until one of the functions returns zero or an error condition occurs. The **SOCKET.SYNC.LISTEN( )** function then returns a numeric value representing the "result" of the connection:

Back to Table of Contents

@SOCKET.EVENT[#N]	Description
0	Unknown / No Status
1	Disconnect / Done / No Error
2	n/a – the file 'open' was done initially
3	Connection Timeout – no connection within the **@SOCKET.TIMEOUT** interval
4	Data Send Timeout – no send data acknowledgment within the **@SOCKET.TIMEOUT** interval
5	Receive Data Timeout – no received data within the **@SOCKET.TIMEOUT** interval

Communication Protocol

In order for meaningful communication to take place both the Client and Server devices have to agree on the exchanged message sequence and format. In the BASIC implementation messages are sent using the **PRINT #N** or **FPRINT #N** statements and messages are received using the **INPUT #N** or **FINPUT #N** statements. A message is considered to be "received" when the trailing carriage return from the sending **PRINT** statement is seen by the corresponding **INPUT** statement.

Messages can be constructed of multiple variables or constants, but there must be an agreement between the client and server. The **PRINT #N / INPUT #N** combination allows the value of a single variable to be communicated. The type of the variable must match between the **PRINT #N** on one device and the **INPUT #N** on the other device.

The **FPRINT #N / FINPUT #N** combination allows the values of multiple variables to be communicated. The order and type of the variables must match between the **FPRINT #N** on one device and the **FINPUT #N** on the other device. While an output message can be constructed using multiple **FPRINT #N** statements with trailing semi-colons except for the last one the received input message must be handled by a single **FINPUT #N** statement. The maximum message size is limited to the maximum size of a string variable.

The sequence of messages must also be defined although the simplified half-duplex implementation mandates that the connecting client device sends data first then receives and the listening server device receives data first then sends. Multiple messages can be sent in each direction at a time as long as there is an agreed upon message value such as an empty message that can be used to switch direction.

Either end can disconnect first but if the client sends data that isn't acknowledged or is waiting to receive data, a timeout will occur. If the server sends data that isn't acknowledged a timeout will occur; however if the server is waiting to receive data and the client disconnects before the timeout this is not considered to be an error as it is the result of a normal client disconnection.

Back to Table of Contents

Socket Examples

Here are some simple examples showing client and server, blocking and non-blocking. Note that because the underlying communications protocol of using **FINPUT #N var, var2$** and **FPRINT #N var, var2$** are the same that the sample blocking client can call the sample blocking or non-blocking server and vice versa.

Blocking Client

The following program is an example of a simple synchronous or blocking client. It is connecting to a server program on another device that essentially echoes back what it has received until the connection is terminated.

Line 1005 uses an **ON GOTO** statement to print the result of the **SOCKET.SYNC.CONNECT( )** function.

The **send( )** user defined function in lines 1065-1075 outputs an incrementing number n and a random numeric string as long as n < 10 and returns the non-zero value one. When n >= 10 the **send( )** function returns the zero value which causes the **SOCKET.SYNC.CONNECT( )** function to disconnect and exit.

The **recv( )** user defined function receives the echoed values and compares the returned string with what was sent. If they are identical the **recv( )** function returns the non-zero value one and the **SOCKET.SYNC.CONNECT( )** function continues. If there isn't a match the **recv( )** function returns the zero value and the **SOCKET.SYNC.CONNECT( )** function exits.

```
10 REM test synchronous ip.connect()
15 n = 0 : senddata$ = "" : recvdata$ = "" : GOSUB 1000 : GOTO 15
1000 REM connection subroutine
1002 PRINT "connecting... ";
1005 ON SOCKET.SYNC.CONNECT(#0, "192.168.1.205:1000", connect(), send(), recv()), GOTO `unknown,  `ok, `no_open, `no_connect,
`send_err, `recv_err
1010 RETURN
1012 `unknown : PRINT " unknown" : RETURN
1015 `ok : PRINT " success" : RETURN
1020 `no_open : PRINT "can't open" : RETURN
1025 `no_connect : PRINT "no connection" : RETURN
1030 `send_err : PRINT " send error" : RETURN
1035 `recv_err : PRINT " recv error" : RETURN
1040 REM connect function
1045 FUNCTION connect()
1050  PRINT "connect ";
1055 ENDFUNCTION
1060 REM send function
1065 FUNCTION send()
1070  send = 0 : LIF n < 10 THEN senddata$ = STR$(RND(32767)) : n = n + 1 : FPRINT #0, n, senddata$ : PRINT ">";n; : send = 1
1075 ENDFUNCTION
1080 REM recv function
1085 FUNCTION recv()
1090  recv = 0 : FINPUT #0, r, recvdata$ : LIF recvdata$ = senddata$ THEN PRINT "<"; : recv = 1
1095 ENDFUNCTION
Ready
run
connecting... connect >1<>2<>3<>4<>5<>6<>7<>8<>9<>10< success
connecting... connect >1<>2<>3<>4<>5<>6<>7<>8<>9<>10< success
connecting... connect >1<>2<>3<>4<>5<>6<>7<>8<>9<>10< success
connecting... connect >1<>2<>3<>4<>5<>6<>7<>8<>9<>10< success
connecting... connect >1<>2<>3<>4<>5<>6<>7<>8<>9<>10< success
connecting... connect >1<>2<>3<>4<>5<>6<>7<>8<>9<>10< success
connecting... connect >1<>2<>3<>4 <<< ESC at line 1090 >>>
Ready
```


Back to Table of Contents

Blocking Server

This is an example of a simple synchronous or blocking server. It listens for connections of clients then receives data in the agreed upon format and echoes back what it has received.

Line 1005 uses **an ON GOTO** statement to print the results of the **SOCKET.SYNC.LISTEN( )** function.

The **recv( )** user defined function in lines 1060-1075 receives the value for variables **n** and **recvdata$** from the connected client and returns a non-zero value of one.

The **send( )** user defined function in lines 1080-1095 copies **recvdata$** to **senddata$** and outputs the value for variables **n** and **senddata$** and returns a non-zero value of one.

The **recv( ) / send( )** repeats until the client disconnects then the **SOCKET.SYNC.LISTEN( )** function returns.

Notice how the **SOCKET.SYNC.LISTEN( )** periodically times out and the program prints "no connection".

```
10 REM test socket.sync.listen()
15 n = 0 : recvdata$ = "" : senddata$ = "" : GOSUB 1000 : GOTO 15
1000 REM listen subroutine
1002 PRINT "listening... ";
1005 ON SOCKET.SYNC.LISTEN(#0, ":1000", connect(), recv(), send()), GOTO `unknown, `ok, `no_open, `no_connect, `recv_err,
`send_err
1010 RETURN
1012 `unknown : PRINT "unknown" : RETURN
1015 `ok : PRINT " success" : RETURN
1020 `no_open : PRINT "can't open" : RETURN
1025 `no_connect : PRINT " no connection" : RETURN
1030 `recv_err : PRINT " recv error" : RETURN
1035 `send_err : PRINT " send error" : RETURN
1040 REM connect function
1045 FUNCTION connect()
1050  PRINT "connect ";
1055 ENDFUNCTION
1060 REM recv function
1065 FUNCTION recv()
1070  recv = 1 : FINPUT #0, n, recvdata$ : PRINT "<";n;
1075 ENDFUNCTION
1080 REM send function
1085 FUNCTION send()
1090  send = 1 : senddata$ = recvdata$ : FPRINT #0, n, senddata$ : PRINT ">";
1095 ENDFUNCTION
Ready
run
listening...  no connection
listening... connect <1><2><3><4><5><6><7><8><9><10> success
listening... connect <1><2><3><4><5><6><7><8><9><10> success
listening... connect <1><2><3><4><5><6><7><8><9><10> success
listening... connect <1><2><3><4><5><6><7><8><9><10> success
listening... connect <1><2><3><4><5><6><7><8><9><10> success
listening...  no connection
listening...  <<< ESC at line 1025 >>>
Ready
```


Back to Table of Contents

Non-blocking Client

This is an example of a simple asynchronous or non-blocking client. It is connecting to a server on another device that essentially echoes back what it has received until the connection is terminated.

Line 20 establishes an event handler subroutine starting at line 1020 for **@SOCKET.EVENT[#0]** events.

Line 25 initializes the send data variables as well as a **done** flag variable and then calls the connection subroutine.

Lines 1000-1015 starts the non-blocking connection and then returns.

Lines 30-35 execute a simple program loop that increments variable **a** while checking for the **done** flag variable to be set. When **done** is set the program prints the current value of the incrementing variable **a** and then starts the connection again.

The **@SOCKET.EVENT[#0]** handler subroutine in lines 1020-1055 print the "result" of the **SOCKET.ASYNC.CONNECT** statement and set the **done** flag variable.

The **RUN** shows the connections each interspersed with the incrementing variable **a** current value showing that program execution continues outside of the socket connection.

```
10 REM test socket.async.connect
20 ONEVENT @SOCKET.EVENT[#0],GOSUB 1020
25 n = 0 : senddata$ = "" : recvdata$ = "" : done = 0 : GOSUB 1000
30 a = a + 1 : LIF done = 1 THEN PRINT "a = ";a : GOTO 25
35 GOTO 30
1000 REM connection subroutine
1005 PRINT "connecting... ";
1010 SOCKET.ASYNC.CONNECT #0, "192.168.1.205:1000", connect(), send(), recv()
1015 RETURN
1020 ON @SOCKET.EVENT[#0], GOTO `none,`ok,`no_open,`no_connect,`send_err,`recv_err
1025 RETURN
1030 `none : PRINT "unknown" : done = 1 : RETURN
1035 `ok : PRINT " success" : done = 1 : RETURN
1040 `no_open : PRINT "can't open" : done = 1 : RETURN
1045 `no_connect : PRINT "no connection" : done = 1 : RETURN
1050 `send_err : PRINT " send error" : done = 1 : RETURN
1055 `recv_err : PRINT " recv error" : done = 1 : RETURN
1060 REM connect function
1065 FUNCTION connect()
1070  PRINT "connect ";
1075 ENDFUNCTION
1080 REM send function
1085 FUNCTION send()
1090  send = 0 : LIF n < 10 THEN senddata$ = STR$(RND(32767)) : n = n + 1 : FPRINT #0, n, senddata$ : PRINT ">";n; : send = 1
1095 ENDFUNCTION
1100 REM recv function
1105 FUNCTION recv()
1110  recv = 0 : FINPUT #0, r, recvdata$ : LIF recvdata$ = senddata$ THEN PRINT "<"; : recv = 1
1115 ENDFUNCTION
Ready
run
connecting... connect >1<>2<>3<>4<>5<>6<>7<>8<>9<>10< success
a = 3091
connecting... connect >1<>2<>3<>4<>5<>6<>7<>8<>9<>10< success
a = 6186
connecting... connect >1<>2<>3<>4<>5<>6<>7<>8<>9<>10< success
a = 9280
connecting... connect >1<>2<>3<>4<>5<>6<>7<>8<>9<>10< success
a = 12376
connecting... connect >1<>2<>3<>4<>5<>6<>7<>8<>9<>10< success
a = 15471
connecting... connect >1<>2<>3<>4<>5<>6<>7<>8<>9<>10 <<< ESC at line 30 >>>
Ready
```


Back to Table of Contents

Non-blocking Server

This is an example of a simple asynchronous non-blocking server. It listens for connections from client devices and essentially echoes back what it has received until the connection is terminated.

Line 20 establishes an event handler subroutine starting at line 1020 **for @SOCKET.EVENT[#0]** events.

Line 25 initializes the send data variables as well as a **done** flag variable and then calls the connection subroutine.

Lines 1000-1015 starts the non-blocking listening connection and then returns.

Lines 30-35 execute a simple program loop that increments variable **a** while checking for the **done** flag variable to be set. When **done** is set the program prints the current value of the incrementing variable **a** and then starts the connection again.

The **@SOCKET.EVENT[#0]** handler subroutine in lines 1020-1055 print the "result" of the **SOCKET.ASYNC.LISTEN** statement and set the **done** flag variable.

The **RUN** shows the connections each interspersed with the incrementing variable's current value showing that program execution continues outside of the socket connection.

```
10 REM test socket.async.listen
20 ONEVENT @SOCKET.EVENT[#0],GOSUB 1020
25 n = 0 : senddata$ = "" : recvdata$ = "" : done = 0 : GOSUB 1000
30 a = a + 1 : LIF done = 1 THEN PRINT "a = ";a : GOTO 25
35 GOTO 30
1000 REM connection subroutine
1005 PRINT "listening... ";
1010 SOCKET.ASYNC.LISTEN #0, ":1000", connect(), recv(), send()
1015 RETURN
1020 ON @SOCKET.EVENT[#0], GOTO `none,`ok,`no_open,`no_connect,`send_err,`recv_err
1025 RETURN
1030 `none : PRINT "unknown" : done = 1 : RETURN
1035 `ok : PRINT " disconnect" : done = 1 : RETURN
1040 `no_open : PRINT "can't open" : done = 1 : RETURN
1045 `no_connect : PRINT "no connection" : done = 1 : RETURN
1050 `send_err : PRINT " send error" : done = 1 : RETURN
1055 `recv_err : PRINT " recv error" : done = 1 : RETURN
1060 REM connect function
1065 FUNCTION connect()
1070  PRINT "connect ";
1075 ENDFUNCTION
1080 REM send function
1085 FUNCTION send()
1090  send = 1 : senddata$ = recvdata$ : FPRINT #0, n, senddata$ : PRINT ">";
1095 ENDFUNCTION
1100 REM recv function
1105 FUNCTION recv()
1110  recv = 1 : FINPUT #0, n, recvdata$ : PRINT "<";n;
1115 ENDFUNCTION
Ready
run
listening... no connection
a = 33407
listening... connect <1><2><3><4><5><6><7><8><9><10> disconnect
a = 41115
listening... connect <1><2><3><4><5><6><7><8><9><10> disconnect
a = 44172
listening... connect <1><2><3><4><5><6><7><8><9><10> disconnect
a = 47228
listening... connect <1><2><3><4><5><6><7><8><9><10> disconnect
a = 50284
listening... connect <1><2><3><4><5><6><7><8><9><10> disconnect
a = 53340
listening... connect  disconnect
a = 53832
listening...  <<< ESC at line 30 >>>
Ready
```


Back to Table of Contents

BASIC Examples

Here are a few sample programs that illustrate the various BASIC language features and what can be done with a few lines of code.

Setting the Real-Time Clock

Set the CFSound-IV's Real-Time Clock with this short program. The program prompts for the values of the Month, Date, Year, Hour, Minute and Second while range checking the values, then displays the formatted time on the connected ANSI terminal once a second.

```
10 REM set the CFSound-IV real-time clock
15 INPUT "set the RTC first (y/n):", s$
20 IF s$="y" THEN 35
25 IF s$="Y" THEN 35
30 GOTO 155
35 INPUT "month (1-12):", m
40 IF m <1 THEN 35
45 IF m >12 THEN 35
50 @MONTH=m
55 INPUT "date (1-31):", d
60 IF d <1 THEN 55
65 IF d >31 THEN 55
70 @DATE=d
75 INPUT "year (0000-9999):", y
80 IF y <0 THEN 75
85 IF y >9999 THEN 75
90 @YEAR=y
95 INPUT "hour (00-23):", h
100 IF h <0 THEN 95
105 IF h >23 THEN 95
110 @HOUR=h
115 INPUT "minute (00-59):", m
120 IF m <0 THEN 115
125 IF m >59 THEN 115
130 @MINUTE=m
135 INPUT "second (00-59):", s
140 IF s <0 THEN 135
145 IF s >59 THEN 135
150 @SECOND=s
155 ONEVENT @SECOND,GOSUB 170
160 a=0
165 GOTO 160
170 PRINT CHR$(13);
175 ON @DOW,GOSUB 265,270,275,280,285,290,295
180 ON @MONTH,GOSUB 200,205,210,215,220,225,230,235,240,245,250,255,260
185 PRINT d$+" "+m$+FMT$(" %2d",@DATE)+FMT$(", %02d",@YEAR);
190 PRINT FMT$(" %2d", @HOUR)+":"+FMT$("%02d",@MINUTE)+":"+FMT$("%02d",@SECOND);
195 RETURN
200 m$="???":RETURN
205 m$="JAN":RETURN
210 m$="FEB":RETURN
215 m$="MAR":RETURN
220 m$="APR":RETURN
225 m$="MAY":RETURN
230 m$="JUN":RETURN
235 m$="JUL":RETURN
240 m$="AUG":RETURN
245 m$="SEP":RETURN
250 m$="OCT":RETURN
255 m$="NOV":RETURN
260 m$="DEC":RETURN
265 d$="SUN":RETURN
270 d$="MON":RETURN
275 d$="TUE":RETURN
280 d$="WED":RETURN
285 d$="THU":RETURN
290 d$="FRI":RETURN
295 d$="SAT":RETURN
```


Back to Table of Contents

Two Sound Sequences

The CFSound-IV can play a single sequence of sounds in CFSound Mode using a CFSOUND.INI file to configure the sequence contact number and sound range. Here's a simple BASIC program that will allow two different sequences each controlled by a built-in contact.

Remember that the @CLOSURE[x] system variable index argument x is zero based, so for Contact #25 the x value would be 24, etc. .

Contact #25 activations cycle through sounds ONE.WAV, TWO.WAV, THREE.WAV and FOUR.WAV, and contact #26 activations cycle through sounds FIVE.WAV, SIX.WAV, SEVEN.WAV and EIGHT.WAV.

Here's how it works. The program lines 10 and 20 setup event handlers for contact closures on contacts #25 and #26. The subroutine at line 1000 is called whenever a closure is detected on contact #25, the subroutine at line 2000 is called whenever a closure is detected on contact #26. Line 30 clears the two sequence variables that keep track of what sound to play next. The variable S0 keeps track of what sound to play for contact #25, and S1 tracks the sounds for contact #26. When a closure is detected on contact #25, the subroutine at line 1000 stops any currently playing sound by clearing the @SOUND$ system variable. Line 1010 then starts playing the next sound in the sequence based upon the current value of S0, and advances the value of S0 for the next contact closure. When a closure is detected on contact #26, the subroutine at line 2000 stops any currently playing sound by clearing the @SOUND$ system variable. Line 2010 then starts playing the next sound in the sequence based upon the current value of S1, and advances the value of S1 for the next contact closure.

```
5 REM play two sequences off of the two built-in rear contacts
10 ONEVENT @CLOSURE[24], GOSUB 1000
20 ONEVENT @CLOSURE[25], GOSUB 2000
30 S0 = 0: S1 = 0
40 GOTO 40
1000 REM contact #25's sequence
1005 @SOUND$=""
1010 ON S0,GOSUB 1100,1105,1110,1115
1015 S0 = S0 + 1
1020 IF S0 > 3 THEN S0=0
1025 RETURN
1100 @SOUND$="ONE.WAV" : RETURN
1105 @SOUND$="TWO.WAV" : RETURN
1110 @SOUND$="THREE.WAV" : RETURN
1115 @SOUND$="FOUR.WAV" : RETURN
2000 REM contact #26's sequence
2005 @SOUND$=""
2010 ON S1,GOSUB 2100,2105,2110,2115
2015 S1 = S1 + 1
2020 IF S1 > 3 THEN S1=0
2025 RETURN
2100 @SOUND$="FIVE.WAV" : RETURN
2105 @SOUND$="SIX.WAV" : RETURN
2110 @SOUND$="SEVEN.WAV" : RETURN
2115 @SOUND$="EIGHT.WAV" : RETURN
```


Back to Table of Contents

Different Sounds for Contact Closure / Opening

The CFSound-IV can play a single sound in response to a contact closure or opening in CFSound Mode using the file naming / contact / attribute association. In order to play two different sounds for the contact closing or opening, a simple BASIC program is required.

Remember that the @CLOSURE[x] system variable index argument x is zero based, so for Contact #25 the x value would be 24, etc. .

In this sample, Contact #25 plays sound ONE.WAV when it closes, and sound TWO.WAV when it opens. Contact #26 plays sound THREE.WAV when it closes, and sound FOUR.WAV when it opens.

Here's how it works. The program lines 10 through 40 poll the contact #25 & #26 @CLOSURE and @OPENING system variables. When one is found active (non-zero) the desired sound file is played, then the triggering system variable is cleared by setting it to zero.

```
5 REM play sounds on contact open and close
10 LIF @CLOSURE[24] THEN PLAY "ONE.WAV":@CLOSURE[24]=0:GOTO 10
20 LIF @OPENING[24] THEN PLAY "TWO.WAV":@OPENING[24]=0:GOTO 10
30 LIF @CLOSURE[25] THEN PLAY "THREE.WAV":@CLOSURE[25]=0:GOTO 10
40 LIF @OPENING[25] THEN PLAY "FOUR.WAV":@OPENING[25]=0:GOTO 10
50 GOTO 10
Ready
```

Starting / Stopping a Sound with a Single Button

The CFSound-IV can play a single sound in response to a contact closure or opening in CFSound Mode using the file naming / contact / attribute association. In order to toggle between starting and stopping a sound with a contact closure, a simple BASIC program is required.

Remember that the @CLOSURE[x] system variable index argument x is zero based, so for Contact #25 the x value would be 24, etc. .

In this sample, a single momentary push button connected between the Contact #25 input and Ground on the Main connector starts and stops a sound. Contact #25 plays sound SOUND.WAV when it closes if no sound is currently playing, and stops playing the sound when it closes and a sound is playing.

Here's how it works. The program loops through lines 10 through 30 polling the contact #25 @CLOSURE system variable. In line 10, if there is a closure AND there is a sound currently playing, the sound is stopped, then the triggering system variable is cleared by setting it to zero. In line 20, if there is a closure AND there isn't a sound currently playing then the desired sound is started playing, and then the system variable is cleared by setting it to zero.

```
5 REM start/stop sound with a single push button on Contact #25 input
10 LIF (@CLOSURE[24]=1) AND (@SOUND$<>"") THEN @SOUND$="":@CLOSURE[24]=0:GOTO 10
20 LIF (@CLOSURE[24]=1) AND (@SOUND$="") THEN @SOUND$="SOUND.WAV":@CLOSURE[24]=0:GOTO 10
30 GOTO 10
Ready
```


Back to Table of Contents

Activating Multiple Output Contacts for a Sound

The CFSound-IV can activate a single output contact when a sound is played in CFSound mode. Here's a simple BASIC program that will allow multiple output contacts to be controlled when a sound plays.

Remember that the @CLOSURE[x] system variable index argument x is zero based, so for Contact #25 the x value would be 24, etc.. This example assumes that the CFSound-IV is equipped with a Contact I/O 8 module installed on the rear expansion connector to provide output contacts 0 – 7.

In this sample, a closure on contact #25 plays sound ONE.WAV and activates output contacts 1 and 2 while the sound is playing. A closure on contact #26 plays sound TWO.WAV and activates output contacts 1 and 3 while the sound is playing.

Here's how it works. The program runs a loop in lines 10 through 30 looking to see if an input closure was detected on contacts #25 and #26. A closure on contact #25 jumps to line 100. A closure on contact #26 jumps to line 200. This process is referred to as 'polling' the input contacts for closures. Starting at line 100 the desired output contacts are activated, then the sound is played, then the output contacts are deactivated. The contact closure is cleared, and the program starts polling again. The same process is programmed starting at line 200 for the other contact and desired output contact configuration.

```
5 REM Poll the two contact inputs for closures
10 IF @CLOSURE[24] THEN GOTO 100
20 IF @CLOSURE[25] THEN GOTO 200
30 GOTO 10
100 REM Input 25 had a closure
110 @CONTACT[0]=1:@CONTACT[1]=1
120 PLAY "ONE.WAV"
130 @CONTACT[0]=0:@CONTACT[1]=0
140 @CLOSURE[24]=0
150 GOTO 10
200 REM Input 26 had a closure
210 @CONTACT[0]=1:@CONTACT[2]=1
220 PLAY "TWO.WAV"
230 @CONTACT[1]=0:@CONTACT[2]=0
240 @CLOSURE[25]=0
250 GOTO 10
```


Back to Table of Contents

Play Sound Sequence with PTT Relay On Beginning of Each Sound

This example plays five tracks in sequence upon contact #25 closures and activates the PTT relay for the first 10 seconds of each track playing.

Here's how it works. The program initializes the sequence number variable and establishes an event handler for @TIMER[0] in line 20. It then loops in line 40 waiting for any currently playing sound to finish. If no sound is playing or when a track finishes playing then line 60 waits for a closure on contact #25. When a closure is seen the subroutine starting at line 1000 is called.

The subroutine starts a sound track playing depending upon the current value of the sequence variable by calling a sequence number specific subroutine in line 1010. It then advances the sequence variable for the next contact closure in line 1020.

Each track in the sequence is started playing by assigning the track sound file name to the @SOUND$ system variable, then calling another subroutine at line 2000 to start the PTT relay timeout before returning.

The subroutine at line 2000 turns on the PTT relay and starts the @TIMER[0] for 10 seconds. When @TIMER[0] counts to zero the handler that was configured in line 20 executes starting at line 3000 – which turns off the relay 10 seconds after the sound was started – whether it was playing or not.

```
10 REM Five track sequence with 10 second PTT
20 Seq = 0 : ONEVENT @TIMER[0], GOSUB 3000
30 REM wait here for sound to finish, clear any in-between button presses
40 IF @SOUND$ = "" THEN @CLOSURE[24] = 0 ELSE 40
50 REM wait here for contact #25 to close (zero-based) then play and advance sequence
60 IF @CLOSURE[24] THEN GOSUB 1000 ELSE 60
70 GOTO 40
1000 REM Play track, advance sequence number for next button push
1010 ON Seq, GOSUB 1100, 1200, 1300, 1400, 1500
1020 Seq = Seq + 1 : IF Seq > 4 THEN Seq = 0
1030 RETURN
1100 @SOUND$ = "Track1.wav" : GOSUB 2000 : RETURN
1200 @SOUND$ = "Track2.wav" : GOSUB 2000 : RETURN
1300 @SOUND$ = "Track3.wav" : GOSUB 2000 : RETURN
1400 @SOUND$ = "Track4.wav" : GOSUB 2000 : RETURN
1500 @SOUND$ = "Track5.wav" : GOSUB 2000 : RETURN
2000 REM Turn on PTT relay, start timer for 10 seconds
2010 @PTT = 1
2020 @TIMER[0] = 10 * 50
2030 RETURN
3000 REM Turn off PTT relay
3010 @PTT = 0
3020 RETURN
```


Back to Table of Contents

Play Three Sounds Each With Different Outputs at Specific Times

In this example three different buttons trigger the playing of three different sounds. While each track is playing at different times during the track the output relays are set to various combinations to illuminate different portions of the exhibit being presented by the audio program.

Here's how it works. Line 15 establishes that @SOUNDFRAMESYNC events will occur once a second for any playing sound. Lines 20 through 40 loop looking for a contact closure on inputs 1, 2, or 3 (zero-based). When a closure is detected then the code for that track is executed by going to that labeled program line.

Labeled lines 45, 50 and 55 that are executed each play a specific track for the contact closure that started them. First a @SOUNDFRAMESYNC event handler subroutine specific for that track is configured, then the sound is started playing by assigning the track filename to the @SOUND$ system variable. Control then transfers to labeled line 60 to wait for the end of the playing sound. When the track finishes any button presses that occurred during are cleared and control transfers back to labeled line 20 to wait for another button press.

While each track is playing, the configured @SOUNDFRAMESYNC handler specific to the track is called, once per second. Each handler checks the current value of the @SOUNDFRAMESYNC system variable which is the number of seconds that the track has been playing. At specific times in seconds for each track the output contacts are configured to illuminate different portions of the exhibit.

```
10 REM Track Specific Outputs
15 @SOUNDFRAMEPRESCALER = 50 : REM sound frame number event once per second
20 `WaitForButton : REM wait for a button to be pressed
25 IF @CLOSURE[0] THEN `PlayTrack1
30 IF @CLOSURE[1] THEN `PlayTrack2
35 IF @CLOSURE[2] THEN `PlayTrack3
40 GOTO `WaitForButton
45 `PlayTrack1 : ONEVENT @SOUNDFRAMESYNC, GOSUB `Track1Events : @SOUND$ = "Track1.WAV" : GOTO `WaitForEnd
50 `PlayTrack2 : ONEVENT @SOUNDFRAMESYNC, GOSUB `Track2Events : @SOUND$ = "Track2.WAV" : GOTO `WaitForEnd
55 `PlayTrack3 : ONEVENT @SOUNDFRAMESYNC, GOSUB `Track3Events : @SOUND$ = "Track3.WAV" : GOTO `WaitForEnd
60 `WaitForEnd : IF @SOUND$ <> "" THEN `WaitForEnd
65 ONEVENT @SOUNDFRAMESYNC, GOSUB 0 : @CLOSURE[0] = 0 : @CLOSURE[1] = 0 : @CLOSURE[2] = 0 : GOTO `WaitForButton
1000 `Track1Events : frame = @SOUNDFRAMESYNC
1005 LIF frame = 9 THEN @CONTACT[0] = 1 : RETURN
1010 LIF frame = (1 * 60) + 20 THEN @CONTACT[0] = 0 : RETURN
1015 RETURN
1100 `Track2Events : frame = @SOUNDFRAMESYNC
1105 LIF frame = 21 THEN @CONTACT[1] = 1 : RETURN
1110 LIF frame = 33 THEN @CONTACT[1] = 0 : RETURN
1115 LIF frame = 44 THEN @CONTACT[2] = 1 : @CONTACT[3] = 1 : RETURN
1120 LIF frame = (1 * 60) + 3 THEN @CONTACT[2] = 0 : @CONTACT[3] = 0 : RETURN
1125 RETURN
1200 `Track3Events : frame = @SOUNDFRAMESYNC
1205 LIF frame = 21 THEN @CONTACT[4] = 1 : RETURN
1210 LIF frame = 30 THEN @CONTACT[4] = 0 : RETURN
1215 LIF frame = 31 THEN @CONTACT[5] = 1 : RETURN
1220 LIF frame = 43 THEN @CONTACT[5] = 0 : RETURN
1225 LIF frame = 46 THEN @CONTACT[6] = 1 : RETURN
1230 LIF frame = 59 THEN @CONTACT[6] = 0 : RETURN
1235 LIF frame = (1 * 60) + 12 THEN @CONTACT[7] = 1 : RETURN
1240 LIF frame = (1 * 60) + 31 THEN @CONTACT[7] = 0 : RETURN
1245 RETURN
```


Back to Table of Contents

Autoplay Entire Sequence Only While Contact Closed

In this example, a sequence of six sounds is triggered by a contact closure on input #25. The sequence starts but will only continue to play as long as input #25 remains closed. If input #25 opens during playout, the sequence must be restarted by another closure of input #25.

Here's how it works. Lines 20 through 90 form an infinite **WHILE / WEND** loop – the **WHILE** condition in line 20 is always non-zero so the loop repeats forever.

Line 30 loops until a closure is seen in input #25 (zero-based).

Line 40 starts the sequence of six sounds with the **FOR / NEXT** loop. The SoundNumber variable is used as the loop counting variable. Each sound in the sequence is started by assigning the sound file name to the **@SOUND$** system variable. The sound file name is generated using the **FMT$** function and the current value of the SoundNumber variable. The sounds are named S01.WAV, S02,WAV, …, S06.WAV.

Lines 50 and 60 wait for either input #25 to open or the current sound to finish playing. In line 50 if the input #25 is no longer closed then the currently playing sound is stopped and the **FOR / NEXT** loop is exited via the **BREAK** statement. Line 60 loop back to line 50 if the sound is still playing.

Line 70 is the bottom of the **FOR / NEXT** loop – if SoundNumber hasn't reached 6 yet then the loop is continued at the FOR statement in line 40. If the entire sequence of all six sounds have finished playing then the closure on input #25 is cleared, and the outermost **WHILE / WEND** loop continues.

```
10 REM Autoplay sequence only while contact remains closed
20 WHILE 1
30  WHILE @CLOSURE[24] = 0 : WEND
40  FOR SoundNumber = 1 TO 6 : @SOUND$ = FMT$("S%02u.WAV", SoundNumber)
50   `wait_done : LIF @CONTACT[24] = 0 THEN @SOUND$ = "" : BREAK `done
60   IF @SOUND$ <> "" THEN `wait_done
70  NEXT SoundNumber
80  `done : @CLOSURE[24] = 0
90 WEND
```


Back to Table of Contents

Autoplay Random Sequence Only While Contact Closed

In this example, a random sequence of eight sounds is triggered by a contact closure on input #25. The sequence starts but will only continue to play as long as input #25 remains closed. If input #25 opens during playout, the sequence must be restarted by another closure of input #25.

Here's how it works. Lines 20 through 110 form an infinite **WHILE / WEND** loop – the **WHILE** condition in line 20 is always non-zero so the loop repeats forever.

Line 30 loops until a closure is seen in input #25 (zero-based).

Line 40 starts the sequence of eight sounds with the **FOR / NEXT** loop. The SoundNumber variable is used as the loop counting variable.

Line 50 generates a new random number from 1 to 8 while ensuring that the same number is not generated twice in row.

Each sound in the sequence is started by assigning the sound file name to the **@SOUND$** system variable. The sound file name is generated using the **FMT$**( ) function and the current value of the NewRandom variable. The sounds are named S01.WAV, S02,WAV, …, S08.WAV. The NewRandom number is then 'remembered' in the LastRandom variable so line 50 can avoid playing the same sound twice in a row.

Lines 70 and 80 wait for either input #25 to open or the current sound to finish playing. In line 70 if the input #25 is no longer closed then the currently playing sound is stopped and the **FOR / NEXT** loop is exited via the **BREAK** statement. Line 80 loop back to line 70 if the sound is still playing.

Line 90 is the bottom of the **FOR / NEXT** loop – if SoundNumber hasn't reached 8 yet then the loop is continued at the **FOR** statement in line 40. If the entire sequence of all eight random sounds have finished playing then the closure on input #25 is cleared, and the outermost **WHILE / WEND** loop continues.

```
10 REM Autoplay random sequence only while contact remains closed
20 WHILE 1
30  WHILE @CLOSURE[24] = 0 : WEND
40  FOR SoundNumber = 1 TO 8
50  WHILE LastRandom = NewRandom : NewRandom = RND(8)+1 : WEND
60  @SOUND$ = FMT$("S%02u.WAV", NewRandom) : LastRandom = NewRandom
70   `wait_done : LIF @CONTACT[24] = 0 THEN @SOUND$ = "" : BREAK `done
80   IF @SOUND$ <> "" THEN `wait_done
90  NEXT SoundNumber
100  `done : @CLOSURE[24] = 0
110 WEND
```


Back to Table of Contents

Autoplay Random Sequence No Repeats While Contact Closed

This is a slight variation on the previous example. This version ensures that there are no repeats of the same sound played in the random sequence, and that each new random sequence doesn't start with the same sound as the previous one.

In order to guarantee that each random sequence consists of all eight sounds with no repeats an array of eight numbers is initialized with the numbers 1 through 8 in lines 50 and 60. Then line 70 randomly shuffles the array contents – without changing any of the numbers in the array.

Line 80 checks to see of this newly shuffled sequence starts with the same sound number as the last time and, if so, regenerates and reshuffles a new sequence – otherwise it remembers the first sound in the sequence for the next time.

Line 40 determines the value for first sound number of the last sequence the first time the program is run.

Lines 90 through 150 work the same way as the previous example.

```
20 REM autoplay random sequence no repeats while contact remains closed
30 REM
40 LastFirst = RND(8)+1
50 DIM Sequence[8]
60 FOR i = 0 TO UBOUND(Sequence)-1 : Sequence[i] = i + 1 : NEXT i
70 FOR i=0 TO UBOUND(Sequence)-1:temp=Sequence[i]:k=RND(8):Sequence[i]=Sequence[k]:Sequence[k]=temp:NEXT i
80 IF LastFirst = Sequence[0] THEN GOTO 60 ELSE LastFirst = Sequence[0]
90 IF @CLOSURE[24] = 0 THEN 90
100 FOR SoundNumber = 0 TO 7
110  @SOUND$ = FMT$("S%02u.WAV", Sequence[SoundNumber])
120  LIF @CONTACT[24] = 0 THEN @SOUND$ = "" : @CLOSURE[24] = 0 : BREAK 50
130  IF @SOUND$ <> "" THEN 120
140 NEXT SoundNumber
150 @CLOSURE[24] = 0 : GOTO 50
```


Back to Table of Contents

Autoplay Random Sequence No Repeats While Contact Closed With Background Sound

This is a rewrite of the previous example that demonstrates some additional coding styles.

Notice the use of the **CONST** to define the sequence length in line 35. The current volume setting is also obtained from the **@VOL** system variable and remembered here.

Rather than use a **GOTO** at the bottom of the program the entire endless loop is enclosed in a **WHILE / WEND**.

`Labels have been used to eliminate the use of line numbers and to 'name' the program jumps.

Lines 90 through 96 play a background sound at reduced volume using the **@NSVOL** system variable to reduce the current volume by 8 while waiting for the sequence start contact closure, then stop the background and restore the volume before beginning to play the sequence.

```
20 REM autoplay random sequence no repeats while contact remains closed
30 REM and play background sound at reduced volume in-between sequences
35 CONST SeqLength = 8 : CurVol = @VOL
40 LastFirst = RND(SeqLength)+1
45 WHILE 1
50  DIM Sequence[SeqLength]
60  `ShuffleSeq : FOR i = 0 TO SeqLength-1 : Sequence[i] = i + 1 : NEXT i
70  FOR i = 0 TO SeqLength-1
72   temp = Sequence[i]
74   k = RND(SeqLength) : Sequence[i] = Sequence[k]
76   Sequence[k] = temp
78  NEXT i
80  IF LastFirst = Sequence[0] THEN `ShuffleSeq ELSE LastFirst = Sequence[0]
90  WHILE @CLOSURE[24] = 0
92   LIF @SOUND$ = "" THEN @NSVOL = CurVol - 8 : @SOUND$ = "SBACKGND.WAV"
94  WEND
96  @SOUND$ = "" : @NSVOL = CurVol
100  FOR SoundNumber = 0 TO SeqLength-1
110   @SOUND$ = FMT$("S%02u.WAV", Sequence[SoundNumber])
120   `WaitForSound : LIF @CONTACT[24] = 0 THEN @SOUND$ = "" : BREAK
130   IF @SOUND$ <> "" THEN `WaitForSound
140  NEXT SoundNumber
150  @CLOSURE[24] = 0
160 WEND
```


Back to Table of Contents

Control from a Serial Port

The CFSound-IV can be controlled by serial commands in CFSound mode. If your application requires custom functionality in addition to being controlled by serial commands use the **@MSG$** system variable to implement a serial protocol. This example shows a simple three character serial protocol that is used to play specific sounds and activate the push to talk relay while the sounds are playing.

The protocol consists of a single character sound number delimited by the default **@SOM** and **@EOM** characters. This yields a message structure of an ASCII Start of Header (SOH) character (CTRL-A), followed by the ASCII number of the sound to play ('1' – '4'), followed by a ASCII End of Text (ETX) character (CTRL-C). The files "ONE.WAV", "TWO.WAV", … , "FOUR.WAV" are on the SD card.

Here's how it works. An event handler is setup in line 20 – when a character string delimited by the **@SOM** and **@EOM** characters is received, control transfers to line 50 with the **@MSG$** variable holding the inner contents of the string. Line 60 copies the string and resets the **@MSG$** variable for receipt of the next message. The message number is converted from a string to a number in line 70, and is adjusted so that it is zero-based. Line 80 calls the subroutine matching the numeric value – the called subroutine activates the PTT relay, plays the sound, deactivates the PTT relay and returns. Line 90 then returns from the @MSG$ event handler.

```
10 REM setup @MSG$ event handler
20 ONEVENT @MSG$,GOSUB 50
30 GOTO 30
50 REM @MSG$ event handler
60 n$=@MSG$
70 n=VAL(n$)-1
80 ON n,GOSUB 100,200,300,400
90 RETURN
100 @PTT=1:PLAY "ONE.WAV":@PTT=0:RETURN
200 @PTT=1:PLAY "TWO.WAV":@PTT=0:RETURN
300 @PTT=1:PLAY "THREE.WAV":@PTT=0:RETURN
400 @PTT=1:PLAY "FOUR.WAV":@PTT=0:RETURN
```


Back to Table of Contents

Motion Triggered Sound with Indicator and Silence Toggle Button

In this example a long motion triggered sound lights an indicator connected to the PTT relay while it is playing, and a pushbutton connected to the second input allows the sound volume to be toggled during the playout.

Here's how it works. Line 20 remembers the current volume setting and initializes the volume toggle.

Line 40 checks for a closure on contact #25 (motion sensor, zero-based) and if it has been triggered temporarily sets the volume to the remembered value, resets the volume toggle, turns on the indicator, starts the long sound playing then clears the closure.

Line 47 checks to see if no sound was playing and loops back to line 40 if so. Line 60 checks to see if the sound was playing and stopped then turns off the indicator, clears both any motions sensor and volume input closures that have occurred and loops back to line 40 if so.

Lines 60 and 70 handle closures on the silence button input #26 (zero-based) and lower or raise the playing sound's volume as required by the toggle variable T.

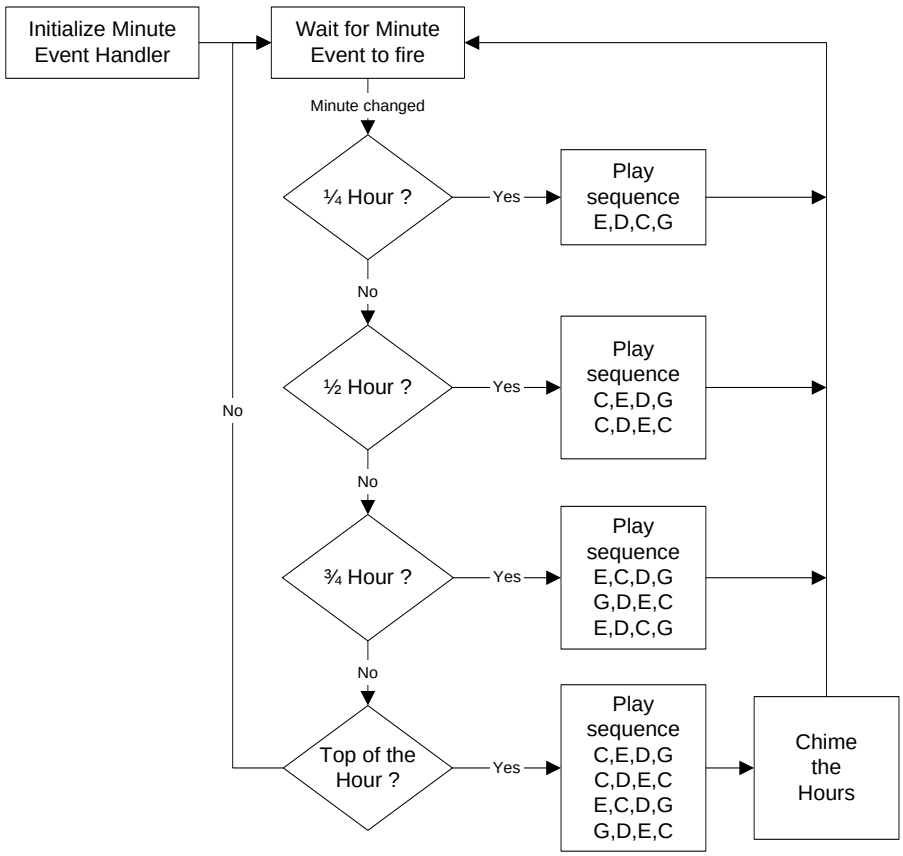
```
10 REM Motion Triggered Sound with Silence Toggle
20 V=@VOL : P=0 : T=0 : REM remember starting volume level, init volume toggle
35 REM if PIR closure restore volume to starting level, activate LED output, start sound, clear closure
40 LIF @CLOSURE[24] THEN @NSVOL=V : T=0 : @PTT=1 : @SOUND$="MOTION.WAV" : P=1 : @CLOSURE[24]=0
45 REM if sound not playing, deactivate LED output, wait for PIR closure
47 LIF (@SOUND$="") AND (P=0) THEN GOTO 40
50 LIF (@SOUND$="") AND (P=1) THEN P=0 : @PTT=0 : @CLOSURE[24]=0 : @CLOSURE[25]=0 : GOTO 40
55 REM sound playing - if SILENCE closure and volume not toggled, lower volume, clear closure, wait for sound to end
60 LIF (@CLOSURE[25]=1) AND (T=0) THEN @NSVOL=20 : T=1 : @CLOSURE[25]=0 : GOTO 50
70 LIF (@CLOSURE[25]=1) AND (T=1) THEN @NSVOL=V : T=0 : @CLOSURE[25]=0 : GOTO 50
80 GOTO 50
```


Back to Table of Contents

Westminster Chimes

Turn the CFSound-IV into a digital audible clock with this short program. The program plays a chime melody using pre-recorded waveforms to emulate the Big Ben clock in London. It plays a portion of the Westminster chimes on the quarter hour, and the entire melody at the top of the hour along with chiming the hour.

Here is a flowchart of the program's logic:



Looking at the diagram, you can see that you need five different note sequences, and the Hours chime. The note sequences can be generated using individual wave files for each note, or recorded or synthesized as short sequences. In this example, Cool Edit Pro was used to capture a bell sound, shorten its envelope, then generate the musical note sequences and the hours chime sound. The five sequence sound files and hours chime are named:

```
dir *.wav
SEQ_GDEC.WAV     581954 A      08-23-2006 16:45:44
SEQ_CDEC.WAV     581954 A      08-23-2006 16:43:50
SEQ_CEDG.WAV     581954 A      08-29-2006 10:17:18
SEQ_ECDG.WAV     581954 A      08-23-2006 16:44:54
SEQ_EDCG.WAV     581954 A      08-23-2006 16:42:58
HOURS.WAV        264434 A      08-23-2006 16:42:24
-----------------------
                   6 files
                   0 directories
```

Here's how it works. The Acs BASIC program initializes a line number of an event handler for the **@MINUTE** system variable that will be fired whenever the **@MINUTE** changes. It then falls into a loop waiting for the event to fire. Other statements can be executed while waiting, but to keep this example simple, it doesn't do anything else while waiting.

Back to Table of Contents

```
5 REM setup event handler
10 ONEVENT @MINUTE,GOSUB 100
15 REM wait here for event
20 GOTO 15
```

Whenever the **@MINUTE** changes, the program performs a **GOSUB** to the event handler program line. The event handler calculates the period of the hour by dividing the current minutes value by 15 minutes per period, and the minutes remaining in the period (remainder) by taking the modulo of the current minutes by 15. If the remainder is zero, then it is the start of a new period, and the event handler branches to the line number for the current period. If the remainder is not zero, the event handler returns. Note that the four decision diamonds above are collapsed into the single program line 110:

```
100 REM calculate period and remainder
102 p=(@MINUTE/15):r=(@MINUTE%15)
105 REM if remainder=0 then branch on period #
110 IF r=0 THEN ON p,GOTO 200,300,400,500
120 RETURN
```

For the quarter past, half past and three quarter past periods, the handler queues the appropriate note sequences to be played and returns. For the top of the hour, the handler queues the note sequences, and then queues the chime sound a number of times to match the hour. It then returns:

```
200 REM play whole sequence & chime hour
202 @SOUND$="SEQ_CEDG.WAV"
204 @SOUND$="SEQ_CDEC.WAV"
206 @SOUND$="SEQ_ECDG.WAV"
208 @SOUND$="SEQ_GDEC.WAV"
210 h=@HOUR:IF h>12 THEN h=h-12
211 IF h=0 THEN h=12
212 FOR c=h TO 1 STEP -1
215 @SOUND$="HOURS.WAV"
220 NEXT c
225 RETURN
300 REM play quarter past sequence
305 @SOUND$="SEQ_EDCG.WAV"
310 RETURN
400 REM play half past sequence
402 @SOUND$="SEQ_CEDG.WAV"
405 @SOUND$="SEQ_CDEC.WAV"
410 RETURN
500 REM play three quarters past sequence
502 @SOUND$="SEQ_ECDG.WAV"
504 @SOUND$="SEQ_GDEC.WAV"
506 @SOUND$="SEQ_EDCG.WAV"
510 RETURN
```

Renaming the program to CFSOUND.BAS and placing it along with the requisite sound files onto the SD card will turn your CFSound-IV into a Big Ben clock. Here's the entire program:

```
5 REM setup event handler
10 ONEVENT @MINUTE,GOSUB 100
15 REM wait here for event
20 a=0:GOTO 15
100 REM calculate period and remainder
102 p=(@MINUTE/15):r=(@MINUTE%15)
105 REM if remainder=0 then branch on period #
110 IF r=0 THEN ON p,GOTO 200,300,400,500
120 RETURN
200 REM play whole sequence & chime hour
202 @SOUND$="SEQ_CEDG.WAV"
204 @SOUND$="SEQ_CDEC.WAV"
206 @SOUND$="SEQ_ECDG.WAV"
208 @SOUND$="SEQ_GDEC.WAV"
210 h=@HOUR:IF h>12 THEN h=h-12
211 IF h=0 THEN h=12
212 FOR c=h TO 1 STEP -1
215 @SOUND$="HOURS.WAV"
220 NEXT c
225 RETURN
300 REM play quarter past sequence
```


Back to Table of Contents

```
305 @SOUND$="SEQ_EDCG.WAV"
310 RETURN
400 REM play half past sequence
402 @SOUND$="SEQ_CEDG.WAV"
405 @SOUND$="SEQ_CDEC.WAV"
410 RETURN
500 REM play three quarters past sequence
502 @SOUND$="SEQ_ECDG.WAV"
504 @SOUND$="SEQ_GDEC.WAV"
506 @SOUND$="SEQ_EDCG.WAV"
510 RETURN
```

Fixed Length Record File I/O

Here's a short demonstration of the FOPEN, FREAD and FWRITE commands:

```
5 DEL "test.dat"
10 FOPEN #1,20,"test.dat"
15 INPUT "how many records:",n
20 FOR r=0 TO n-1
30 FWRITE #1,r,r,"str"+STR$(r)
40 NEXT r
50 PRINT "reading records..."
60 r=0
70 FREAD #1,r,b,b$
75 IF @FEOF[#1] THEN 1000
80 PRINT "rec:",r,"=",b,",",b$
90 r=r+1:GOTO 70
1000 CLOSE #1
Ready
run
how many records:10
reading records...
rec: 0= 0,str0
rec: 1= 1,str1
rec: 2= 2,str2
rec: 3= 3,str3
rec: 4= 4,str4
rec: 5= 5,str5
rec: 6= 6,str6
rec: 7= 7,str7
rec: 8= 8,str8
rec: 9= 9,str9
Ready
type test.dat
0,"str0"
1,"str1"
2,"str2"
3,"str3"
4,"str4"
5,"str5"
6,"str6"
7,"str7"
8,"str8"
9,"str9"
Ready
```


Back to Table of Contents

Error Logging

While developing programs without a serial connection, or for stand alone program monitoring it may be advantageous to record any program errors that occur to the SD card. Then when the program stops running, the SD card can be inserted into a PC card reader and the error that caused the program to stop can be examined. The following code sets up **ONERROR** to transfer control to line 32000 where an ERRORS.TXT file is opened for appended writing and the causal error message is written at the end of the file:

```
10 REM Error Logging Example
20 ONERROR GOTO 32000
30 A=B/0
32000 OPEN #0,"ERRORS.TXT","a+w"
32005 PRINT #0,ERR$()
32010 CLOSE #0
32015 STOP
Ready
run
STOP in line 32015
Ready
type errors.txt
Divide by zero error in line 30
Ready
run
STOP in line 32015
Ready
type errors.txt
Divide by zero error in line 30
Divide by zero error in line 30
Ready
```


Back to Table of Contents

DMX Control Synchronized to Sound

This example plays an audio file for an exhibit with lights that are synchronized to the audio track. The CFSound-IV with external ArtNet™ Ethernet to DMX module synchronizes the fading up/down of the scene lights with the audio track.

Here's how it works. The show is started by pressing a button connected to the Contact #25 input. The show stops by pressing a button connected to Contact #26 or when the show's sound file SHOW.WAV ends.

Line 35 sets the **@SOUNDFRAMEPRESCALER** system variable to 50. This causes a **@SOUNDFRAMESYNC** event to fire every second while the sound is playing. The event handler is configured in line 70.

The program loops at lines beginning with the `WaitForStart label until the start button closes. The sound frame event handler is initialized and the show sound track is started in line 75.

While the sound is playing, the sound frame event handler beginning with the `FrameEvent label at line 1000 is called once per second. This handler checks the frame number to see when to fade up or fade down various DMX channels during the sound track.

```
10 REM show handler framework
15 CONST StartButton = 24, StopButton = 25
20 fadeup = 0 : fadeup_final = 0 : fadeup_channel = 0 : REM init fadeup globals
25 fadedown = 0 : fadedown_final = 0 : fadedown_channel = 0 : REM init fadedown globals
30 DIM dmxdata[4] : REM fade all globals
35 @SOUNDFRAMEPRESCALER = 50 : REM sound frame number event every second
40 `StopShow : REM stop show
45 @SOUND$ = "" : ONEVENT @SOUNDFRAMESYNC, GOSUB 0
50 `WaitForStart : REM check for show start button
55 IF @CLOSURE[StartButton] = 0 THEN `WaitForStart
60 REM show start pushed
65 @CLOSURE[StartButton] = 0
70 ONEVENT @SOUNDFRAMESYNC, GOSUB `FrameEvent
75 @SOUND$ = "show.wav"
80 `WaitForEnd : REM check for show end (sound or button)
85 IF (@CLOSURE[StopButton] = 0) AND (@SOUND$ <> "") THEN `WaitForEnd
90 REM show end (sound or button)
95 @CLOSURE[StopButton] = 0 : @SOUND$ = "" : FadeDownAll()
100 @CLOSURE[StartButton] = 0 : GOTO `StopShow
1000 `FrameEvent : REM Sound Frame Sync event handler
1005 frame = @SOUNDFRAMESYNC
1010 REM Channel 1 up at 1 second
1015 LIF frame = 1 THEN FadeUp(0, 0, 255) : RETURN
1020 REM Channel 1 down at 10 seconds
1025 LIF frame = 10 THEN FadeDown(0, 255, 0) : RETURN
1030 REM Channel 2 up at 20 seconds
1035 LIF frame = 20 THEN FadeUp(1, 0, 255) : RETURN
1040 REM Channel 2 down at 30 seconds
1045 LIF frame = 30 THEN FadeDown(1, 255, 0) : RETURN
1050 REM Channel 3 up at 40 seconds
1055 LIF frame = 40 THEN FadeUp(2, 0, 255) : RETURN
1060 REM Channel 3 down at 50 seconds
1065 LIF frame = 50 THEN FadeDown(2, 255, 0) : RETURN
1070 REM Channel 4 up at 60 seconds
1075 LIF frame = 60 THEN FadeUp(3, 0, 255) : RETURN
1080 REM Channel 4 down at 70 seconds
1085 LIF frame = 70 THEN FadeDown(3, 255, 0) : RETURN
1090 RETURN
10000 REM Fade Up DMX channel
10005 FUNCTION FadeUp(channel, initial, final)
10010  ONEVENT @TIMER[0], GOSUB `FadeUpTimerEvent
10015  fadeup_channel = channel : fadeup = initial : fadeup_final = final : @TIMER[0] = 2
10020 ENDFUNCTION
10025 `FadeUpTimerEvent : REM Fade Up timer event handler
10030  IF fadeup <= (fadeup_final - 2) THEN fadeup = fadeup + 2 ELSE fadeup = fadeup_final
10035  @DMX.DATA[fadeup_channel] = fadeup : dmxdata[fadeup_channel] = fadeup
10040  LIF fadeup <> fadeup_final THEN @TIMER[0] = 2 : RETURN
10045  ONEVENT @TIMER[0], GOSUB 0 : RETURN
```


Back to Table of Contents

```
10050 REM Fade Down DMX channel
10055 FUNCTION FadeDown(channel, initial, final)
10060  ONEVENT @TIMER[1], GOSUB `FadeDownTimerEvent
10065  fadedown_channel = channel : fadedown = initial : fadedown_final = final : @TIMER[1] = 2
10070 ENDFUNCTION
10075 `FadeDownTimerEvent : REM Fade Down timer event handler
10080   IF fadedown >= (fadedown_final + 2) THEN fadedown = fadedown - 2 ELSE fadedown =
fadedown_final
10085  @DMX.DATA[fadedown_channel] = fadedown : dmxdata[fadedown_channel] = fadedown
10090  LIF fadedown <> fadedown_final THEN @TIMER[1] = 2 : RETURN
10095  ONEVENT @TIMER[1], GOSUB 0 : RETURN
10100 REM Fade Down All DMX channels
10105 FUNCTION FadeDownAll()
10110  ONEVENT @TIMER[0], GOSUB 0 : ONEVENT @TIMER[1], GOSUB 0
10115  ONEVENT @TIMER[2], GOSUB `FadeAllTimerEvent : @TIMER[2] = 2
10120 ENDFUNCTION
10125 `FadeAllTimerEvent : REM Fade Down All timer event handler
10130  FOR n = 0 TO UBOUND(dmxdata) - 1
10135   IF dmxdata[n] >= 2 THEN dmxdata[n] = dmxdata[n] - 2 ELSE dmxdata[n] = 0
10140   @DMX.DATA[n] = dmxdata[n]
10145  NEXT n
10150  FOR n = 0 TO UBOUND(dmxdata) - 1
10155   LIF dmxdata[n] > 0 THEN @TIMER[2] = 2 : RETURN
10160  NEXT n
10165  ONEVENT @TIMER[2], GOSUB 0 : RETURN
```


Back to Table of Contents

Play Random Announcement Periodically

This example allows the CFSound-IV to periodically interrupt a music source playing through the line input and play a random pre-recorded announcement. The CFSound line input is connected to the music source, and the line output is connected back into the distribution amp if required or the built-in amplifier can be used to power the speakers.

Here is how it works. When the program is started, lines 40-60 capture a directory listing of .WAV files into a text file DIRLIST.TXT on the SD card. Lines 70-150 count the number of .WAV files that were found. Lines 170-230 create a fixed length record file of these .WAV filenames into a file WAVLIST.TXT that can be accessed randomly. Now the program begins normal operation. Lines 250-275 fades-down the volume, disables the line input, restores the volume to the current setting and then plays a random selected .WAV file. Lines 290-310 minimizes the volume, enables the line input, fades-up the volume to the current setting and waits for the inter-announcement time delay to expire before the process is repeated.

```
5 REM ********************************
10 REM Play random announcement periodically
20 REM ********************************
25 M=15 : REM minutes between announcements
30 REM ******************************
31 REM Capture directory of .WAV files
32 REM ******************************
35 REM
40 OPEN #0, "DIRLIST.TXT", "w"
50 DIR #0, "*.WAV"
60 CLOSE #0
65 REM ******************************
66 REM Count number of .WAV files found
67 REM ******************************
70 OPEN #0, "DIRLIST.TXT", "r"
80 N=0
100 INPUT #0, L$
110 IF @FEOF[#0] THEN 150
120 W=FIND(L$, ".WAV") : IF W <0 THEN 100
130 N=N+1 : GOTO 100
150 CLOSE #0 : OPEN #0, "DIRLIST.TXT", "r"
160 REM ****************************************************
161 REM Now create fixed recordlength file of filenames found
162 REM ****************************************************
170 ONERROR GOTO 180 : DEL "WAVLIST.TXT" : ONERROR GOTO 0
180 FOPEN #1, 16, "WAVLIST.TXT"
190 FOR F=0 TO N-1
200 INPUT #0, L$
210 W=FIND(L$, ".WAV") : F$=LEFT$(L$, W+4) : FWRITE #1, F, F$
220 NEXT F
230 CLOSE #0 : CLOSE #1 : FOPEN #1, 16, "WAVLIST.TXT"
240 REM ************************************************************************
241 REM Now fade-down, turn off line input, restore volume and play random sound
242 REM ************************************************************************
250 GOSUB 500 : @LINEIN=0 : @NSVOL=V
260 FREAD #1, RND(N), F$
270 ONERROR GOTO 280 : PLAY "" +F$ : ONERROR GOTO 0
280 REM ************************************************************************
281 REM Now minimize volume, turn on line input, fade-up and wait for time delay
282 REM ************************************************************************
290 @NSVOL=0 : @LINEIN=1 : GOSUB 550
300 FOR T=1 TO M : DELAY 3000 : NEXT T
310 GOTO 240
500 REM ************************************
501 REM Fade-down volume from current setting
502 REM ************************************
510 V=@VOL
520 FOR T=V TO 0 STEP -1 : @NSVOL=T : DELAY 2 : NEXT T
530 RETURN
550 REM **************************************
551 REM Fade-up volume back to current setting
552 REM **************************************
560 FOR T=0 TO V : @NSVOL=T : DELAY 2 : NEXT T
570 RETURN
```


Back to Table of Contents

Configuration Editor

This example allows a user with a connected terminal emulator to edit the CFSound-IV configuration settings. Here is how it works:

Line 15 disables the **@MSG$** function so that the **GETCH( )** function will operate as required.

Line 20 initializes the current item and items variables then displays the menu.

Lines 22 through 35 comprise an infinite **WHILE** loop that retrieves information about the current item, shows the item and processes any keys that are pressed.

Lines 8000 through 8615 are the **`HandleKeys** subroutine that waits for key input using the **GETCH(1)** function then processes the received key to navigate amongst the items, each item's fields, enter and exit editing mode, increment/decrement or default the current item, accept numeric/hexadecimal input for certain item types and allow the program to be exited. In order to accept the arrow cursor keys as input ANSI escape sequences are decoded. The Enter key is used to enter editing mode or exit and save the configuration value.

Lines 10000 through 10160 are the **`ShowItem** subroutine that displays the currently selected item and field as either being browsed or edited using ANSI escape sequences for inverting the text/background.

Lines 11000 through 11030 are the **`DefaultItem** subroutine that sets the currently selected item to its default value.

Lines 12000 through 12035 are the **`ShowMenu** subroutine that does just that.

Lines 13000 through 13070 are the **`UpdateField** subroutine that updates the currently selected configuration value with the numerically entered value.

Note that the entire program is written without a **GOTO** statement and **`label** variables are used to call subroutines. Nested block **IF/THEN/ELSE/ENDIF** statements facilitate this style of programming and are indented to show nesting levels – improving the program's readability.

```
10 REM Configuration Editor
15 @MSGENABLE = 0
20 item = 0 : items = @CONFIG.ITEMS : GOSUB `ShowMenu
22 WHILE 1
25  itemType=@CONFIG.TYPE[item]:fields=@CONFIG.FIELDS[item]:min=@CONFIG.MIN[item]:max=@CONFIG.MAX[item]
30  GOSUB `ShowItem : GOSUB `HandleKeys
35 WEND
8000 REM
8005 REM handle ansi terminal key input
8010 REM
8015 `HandleKeys : key = GETCH(1)
8020 REM
8025 REM Ansi escape sequences for arrow keys
8030 REM
8035 IF key = 27 THEN
8040  key = GETCH(1)
8045  IF CHR$(key) = "[" THEN
8050   key = GETCH(1)
8055   REM
8060   REM down arrow key
8065   REM
8070   IF CHR$(key) = "B" THEN
8075    IF editing = 0 THEN item = item + 1
8080    IF item >= items - 1   THEN item = 0
8085   ENDIF
8090   REM
8095   REM up arrow key
8100   REM
8105   IF CHR$(key) = "A" THEN
8110    IF editing = 0 THEN item = item - 1
8115    IF item < 0 THEN item = items - 2
8120   ENDIF
```


Back to Table of Contents

```
8125   REM
8130   REM right arrow key
8135   REM
8140   IF (CHR$(key) = "C") AND (editing = 1) THEN
8145    GOSUB `UpdateField
8150    IF fields AND field < fields - 1 THEN field = field + 1
8155   ENDIF
8160   REM
8165   REM left arrow key
8170   REM
8175   IF (CHR$(key) = "D") AND (editing = 1) THEN
8180    GOSUB `UpdateField
8185    IF fields AND field > 0 THEN field = field - 1
8190   ENDIF
8195  ENDIF
8200 ELSE
8205  REM
8210  REM enter key
8215  REM
8220  IF key = 13 THEN
8225   IF editing = 0 THEN
8230    field = 0 : editing = 1
8235   ELSE
8240    GOSUB `UpdateField : @CONFIG.WRITE[item] = 1 : editing = 0
8245   ENDIF
8250  ENDIF
8255  REM
8260  REM plus key
8265  REM
8270  IF (CHR$(key) = "+") AND (editing = 1) THEN
8275   IF fields THEN
8280    IF @CONFIG.VALUE[item,field] < max THEN @CONFIG.VALUE[item,field]=@CONFIG.VALUE[item,field]+1
8285   ELSE
8290    IF @CONFIG.VALUE[item] <= max THEN @CONFIG.VALUE[item] = @CONFIG.VALUE[item] + 1
8295   ENDIF
8300  ENDIF
8305  REM
8310  REM minus key
8315  REM
8320  IF (CHR$(key) = "-") AND (editing = 1) THEN
8325   IF fields THEN
8330    IF @CONFIG.VALUE[item,field] > min THEN @CONFIG.VALUE[item,field]=@CONFIG.VALUE[item,field]-1
8335   ELSE
8340    IF @CONFIG.VALUE[item] > min THEN @CONFIG.VALUE[item] = @CONFIG.VALUE[item] - 1
8345   ENDIF
8350  ENDIF
8355  REM
8360  REM X = exit
8365  REM
8370  IF CHR$(key & 223) = "X" THEN
8375   IF editing = 1 THEN
8380    editing = 0
8385   ELSE
8390    PRINT ""
8395    END
8400   ENDIF
8405  ENDIF
8410  REM
8415  REM R = reset (default)
8420  REM
8425  IF CHR$(key & 223) = "R" THEN
8430   IF editing = 1 THEN
8435    GOSUB `DefaultItem
8440   ELSE
8445    PRINT "  Default entire configuration ? (y/n):";
8450    IF CHR$(GETCH(1) & 223) = "Y" THEN
8455     FOR item = 0 TO items-2 : GOSUB `DefaultItem : NEXT item : item = 0
8460    ENDIF
8465   ENDIF
8470  ENDIF
8475  REM
8480  REM 0 - 9 keys
8485  REM
```


Back to Table of Contents

```
8490  IF (key >= 48) AND (key <= 57) AND (editing = 1) THEN
8495   IF (itemType=0) OR (itemType=2) OR (itemType=13) OR (itemType=14) THEN
8500    numberEdit = 1 : IF LEN(number$) < 5 THEN number$ = number$ + CHR$(key)
8505   ENDIF
8510   IF (itemType=12) THEN
8515    hexEdit = 1 : IF LEN(number$) < 2 THEN number$ = number$ + CHR$(key)
8520   ENDIF
8525   IF (itemType=17) THEN
8530    hexEdit = 1 : number$ = CHR$(key)
8535   ENDIF
8540  ENDIF
8545  REM A - F keys
8550  IF ((key & 223) >= 65) AND ((key & 223) <= 70) THEN
8555   IF (itemType=12) THEN
8560    hexEdit = 1 : IF LEN(number$) < 2 THEN number$ = number$ + CHR$(key & 223)
8565   ENDIF
8570   IF (itemType=17) THEN
8575    hexEdit = 1 : number$ = CHR$(key & 223)
8580   ENDIF
8585  ENDIF
8590  REM
8595  REM backspace key
8600  REM
8605  IF (key=8) AND (numberEdit=1) AND (LEN(number$) > 0) THEN number$=LEFT$(number$, LEN(number$)-1)
8610 ENDIF
8615 RETURN
10000 REM
10005 REM Show the current configuration item
10010 REM
10015 `ShowItem : PRINT CHR$(13) + "["; : PRINT item; : PRINT "] - ";
10020 PRINT @CONFIG.NAME$[item] + " = ";
10025 IF fields THEN
10030  IF editing THEN
10035   FOR f = 0 TO fields-1
10040    IF f = field THEN PRINT CHR$(27); "[7m";
10045    IF ((numberEdit = 1) OR (hexEdit = 1)) AND (f = field) THEN
10050     PRINT number$;
10055    ELSE
10060     PRINT @CONFIG.FIELD$[item,f];
10065    ENDIF
10070    IF f = field THEN PRINT CHR$(27); "[0m";
10075    PRINT @CONFIG.SEPARATOR$[item,f];
10080   NEXT f
10085  ELSE
10090   PRINT @CONFIG.VALUE$[item];
10095  ENDIF
10100 ELSE
10105  IF editing THEN
10110   PRINT CHR$(27); "[7m";
10115  ENDIF
10120  IF (numberEdit = 1) OR (hexEdit = 1) THEN
10125   PRINT number$;
10130  ELSE
10135   PRINT @CONFIG.VALUE$[item];
10140  ENDIF
10145  PRINT CHR$(27); "[0m";
10150 ENDIF
10155 PRINT CHR$(27); "[K";
10160 RETURN
11000 `DefaultItem : REM Default a single configuration item
11005 IF fields = 0 THEN
11010  @CONFIG.VALUE[item] = @CONFIG.DEFAULT[item]
11015 ELSE
11020  FOR field=0 TO fields-1:@CONFIG.VALUE[item,field]=@CONFIG.DEFAULT[item,field]:NEXT field:field=0
11025 ENDIF
11030 RETURN
12000 REM
12005 REM Show the menu
12010 REM
12015 `ShowMenu : PRINT "" : PRINT "Configuration Editor" : PRINT ""
12020 PRINT "up/down item |   R = default    | Enter = edit/save"
12025 PRINT "  +/- value  | left/right field |    X = exit"
12030 PRINT ""
```


Back to Table of Contents

```
12035 RETURN
13000 REM
13005 REM update field if number entered
13010 REM
13015 `UpdateField
13020 IF numberEdit THEN
13025  IF (LEN(number$)>0) THEN number=VAL(number$) ELSE number=0:IF number < min THEN number=min:IF
number > max THEN number=max
13030  IF fields THEN @CONFIG.VALUE[item,field] = number ELSE @CONFIG.VALUE[item] = number
13035  numberEdit = 0 : number$ = ""
13040 ENDIF
13045 IF hexEdit THEN
13050  IF (LEN(number$) > 0) THEN number=HEX.VAL(number$) ELSE number=0:IF number < min THEN
number=min:IF number > max THEN number=max
13055  IF fields THEN @CONFIG.VALUE[item,field] = number ELSE @CONFIG.VALUE[item] = number
13060  hexEdit = 0 : number$ = ""
13065 ENDIF
13070 RETURN
Ready
```


Back to Table of Contents

Simple text/html Web Server

This example implements a simple text/html file web server on port 8080 using BASIC Sockets. Here's how it works:

Line 15 defines the **send( )** function state machine states. Lines 20-45 dimension and initialize HTTP 1.0 response header arrays for the 200 OK response, content type and 404 not found response.

Lines 50-55 initializes some global variables then calls the HTTP web server subroutine in a loop.

In the web server subroutine the synchronous socket listen function is called to open a listening connection on port 8080. The standard http port 80 could be used, but then the built-in web server would have to be disabled in the configuration. Lines 1025-1050 process the function return results displaying on the console.

The **connect( )** function in lines 1060-1070 is called when a remote web browser connects to the CFSound-IV's IP address on port 8080.

After connection the **recv( )** function in lines 1080-1150 are called when the web browser sends the HTTP request headers. As each header line is received it is displayed on the console and then examined for the http GET file method request. The parse_get_filename( ) function in lines 1360-1400 is called for each request header line and returns a one when the GET filename has been seen and parsed. Lines 1095-1125 checks to see if the requested file exists and opens it and configures the send( ) function to send the 200 OK headers and file. If the file doesn't exist the **send( )** function is configured to send the 404 NOT FOUND headers and small HTML response. When an empty request header line is received the HTTP response process begins.

The **send( )** function in lines 1160-1340 sends the response and content headers then the requested file - displaying each line on the console. The connection is then closed and the **SOCKET.SYNC.LISTEN( )** function returns the connection's status which is also displayed on the console. The web server function returns and the process repeats.

```
10 REM text/html only http using socket.sync.listen()
15 CONST send_200_header = 0, send_content = 1, send_file = 2, send_404 = 3, send_disconnect = 4
20 DIM header200$[4] : header200$[0] = "HTTP/1.0 200 OK" : header200$[1] = "Cache-Control: no-cache"
25 header200$[2] = "Server: http://www.cfsound.com" : header200$[3] = "Connection: close"
30 DIM content$[2] : content$[0] = "Content-type: text/html" : content$[1] = ""
35 DIM header404$[6] : header404$[0] = "HTTP/1.0 404 Not found" : header404$[1] = "Server: http://www.cfsound.com"
40 header404$[2] = "Connection: close" : header404$[3] = "Content-type: text/html" : header404$[4] = ""
45 header404$[5] = "<!DOCTYPE HTML><html><body><h1>404 - file not found</h1></body></html>"
50 recvdata$ = "" : senddata$ = "" : sendstate = send_200_header : line = 0
55 filename$ = "" : GOSUB 1000 : GOTO 55
1000 REM web server subroutine
1005 PRINT "listening... ";
1010 @SOCKET.TIMEOUT = -1
1015 ON SOCKET.SYNC.LISTEN(#0,":8080", connect(), recv(), send()), GOTO `unknown, `ok, `no_open, `no_connect, `send_err, `recv_err
1020 RETURN
1025 `unknown : PRINT "unknown" : RETURN
1030 `ok : PRINT "OK - disconnect" : RETURN
1035 `no_open : PRINT "can't open port :8080" : RETURN
1040 `no_connect : PRINT "no connection timeout" : RETURN
1045 `recv_err : PRINT "data receive timeout - disconnect" : IF LEN(filename$) > 0 THEN CLOSE #1 : RETURN
1050 `send_err : PRINT "data sent timeout - disconnect" : RETURN
1055 REM connect function
1060 FUNCTION connect()
1065  @SOCKET.TIMEOUT = 50: PRINT "connect "
1070 ENDFUNCTION
1075 REM recv function
1080 FUNCTION recv()
1085  INPUT #0, recvdata$ : PRINT recvdata$
1090  IF LEN(recvdata$) > 0 THEN
1095   IF parse_get_filename(recvdata$) THEN
1100    IF FILE.EXISTS(filename$) THEN
1105     OPEN #1, filename$, "r" : sendstate = send_200_header
1110    ELSE
1115     sendstate = send_404
1120    ENDIF
1125   ENDIF
1130   recv = 2 : REM call recv() again
1135  ELSE
1140   recv = 1 : REM call send()
1145  ENDIF
1150 ENDFUNCTION
1155 REM send function
```


Back to Table of Contents

```
1160 FUNCTION send()
1165  ON sendstate, GOTO `Send200, `SendContent, `SendFile, `Send404, `SendDisconnect
1170  `SendDisconnect : send = 0 : GOTO `SendExit
1175  `Send200
1180   PRINT header200$[line]
1185   IF send_array_line(\header200$, line) THEN
1190    line = line + 1
1195   ELSE
1200    sendstate = send_content : line = 0
1205   ENDIF
1210   send = 2 : GOTO `SendExit
1215  `SendContent
1220   PRINT content$[line]
1225   IF send_array_line(\content$, line) THEN
1230    line = line + 1
1235   ELSE
1240    sendstate = send_file : line = 0
1245   ENDIF
1250   send = 2 : GOTO `SendExit
1255  `SendFile
1260   INPUT #1,senddata$ : PRINT senddata$
1265   IF @FEOF[#1] THEN
1270    CLOSE #1 : sendstate = send_disconnect : send = 0
1275   ELSE
1280    PRINT #0, senddata$ : send = 2
1285   ENDIF
1290   GOTO `SendExit
1295  `Send404
1300   PRINT header404$[line]
1305   IF send_array_line(\header404$, line) THEN
1310    line = line + 1
1315   ELSE
1320    sendstate = send_disconnect
1325   ENDIF
1330   send = 2 : GOTO `SendExit
1335  `SendExit
1340 ENDFUNCTION
1345 FUNCTION send_array_line(lines$, linenumber)
1350  send_array_line = 0 : PRINT #0, lines$[linenumber] : IF linenumber < UBOUND(lines$)-1 THEN send_array_line = 1
1355 ENDFUNCTION
1360 FUNCTION parse_get_filename(header$)
1365  parse_get_filename = 0 : ndx = FIND(header$, "GET /")
1370  IF ndx >= 0 THEN
1375   header$ = RIGHT$(header$, LEN(header$) - ndx - LEN("GET /")) : filename$ = ""
1380   ndx = FIND(header$, " ") : IF ndx >= 0 THEN filename$ = LEFT$(header$, ndx)
1385   IF LEN(filename$) = 0 THEN filename$ = "index.html"
1390   parse_get_filename = 1
1395  ENDIF
1400 ENDFUNCTION
```


Back to Table of Contents

Remote Control of CFSound-IV Contact I/O Relays

This example provides a Windows application running on a PC with the ability to control the Contact I/O 8 output relays on a remote CFSound-IV via Ethernet. The solution consists of two parts; a BASIC application that runs on the CFSound-IV and a Windows application running on a PC. The two programs interact via messages exchanged via an Ethernet connection using a client/server socket style of operation.

Here's how the BASIC application works. Lines 20-60 initialize some global variables (retain their value outside of FUNCTIONs), and configures a **@SOCKET.EVENT** handler subroutine. The code then starts an asynchronous socket listen and waits for the done flag to be set. When the done flag is set there is a short delay to allow the socket processing to unwind and then the process repeats.

Lines 1000-1280 comprise the socket connection subroutine and socket event handler. Line 1020 starts an asynchronous socket listen on port 1000 and specifies the functions to be called at different states of the process. Having started the socket listen process the subroutine returns. The code in lines 1040-1120 form the socket event handler subroutine which prints the socket's status, sets the done flag and then returns.

The individual socket state functions are called as the asynchronous socket operation progresses. After a remote connection to the listening socket is made the **connect( )** function is called first. Next the **recv( )** function is called to handle the data that was sent by the caller. It expects two values – a cmd$ string character and a number which it reads using the **FINPUT** statement in line 1260. If the cmd$ character is "=" then the number represents the caller's desired state of the output relays. If the cmd$ character is "?" then the number is irrelevant as the caller wants to obtain the current state of the relays. The **recv( )** function returns 1 which indicates that the **send( )** function should be called next to return some data to the caller.

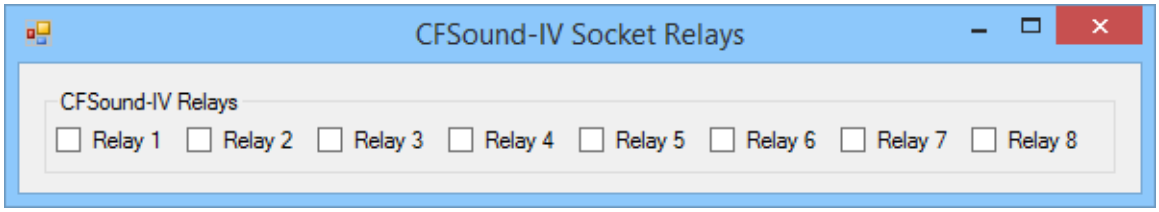
The **send( )** function decodes the received global cmd$ and performs the command if it is known, otherwise a "?" response is returned. The `set_relays subroutine updates the Contact I/O 8 output relays from the received data. The `send_status subroutine returns the current state of the relays.

```
10 REM Socket Relays - remotely controlled relays
20 @SOCKET.TIMEOUT = 200 : cmd$ = "" : relays = 0
30 ONEVENT @SOCKET.EVENT[#0],GOSUB `socket_event_handler
40 `listen_loop : done = 0 : GOSUB `start_async_listen
50 `wait_for_done : LIF done = 1 THEN DELAY(5) : GOTO `listen_loop
60 GOTO `wait_for_done
1000 REM connection subroutine
1010 `start_async_listen : PRINT "listening... ";
1020 SOCKET.ASYNC.LISTEN #0, ":1000", connect(), recv(), send()
1030 RETURN
1040 `socket_event_handler
1050 ON @SOCKET.EVENT[#0], GOTO `none,`ok,`no_open,`no_connect,`send_err,`recv_err
1060 RETURN
1070 `none : PRINT "unknown" : done = 1 : RETURN
1080 `ok : PRINT " disconnect" : done = 1 : RETURN
1090 `no_open : PRINT "can't open #0" : done = 1 : RETURN
1100 `no_connect : PRINT "no connection" : done = 1 : RETURN
1110 `send_err : PRINT " send error" : done = 1 : RETURN
1120 `recv_err : PRINT " recv error" : done = 1 : RETURN
1130 REM socket connect function
1140 FUNCTION connect()
1150  PRINT "connect ";
1160 ENDFUNCTION
1170 REM socket send function
1180 FUNCTION send()
1190  LIF cmd$ = "?" THEN GOSUB `send_status : send = 1 : GOTO `exit_send
1200  LIF cmd$ = "=" THEN GOSUB `set_relays : GOSUB `send_status : send = 1 : GOTO `exit_send
1210  FPRINT #0, "?", 0 : send = 1 : PRINT ">?";
1220  `exit_send
1230 ENDFUNCTION
1240 REM socket recv function
```


Back to Table of Contents

```
1250 FUNCTION recv()
1260  recv = 1 : FINPUT #0, cmd$, recvData : PRINT "<";cmd$;":";recvData;" ";
1270  IF cmd$ = "=" THEN relays = recvData
1280 ENDFUNCTION
2000 REM set relays subroutine
2010 `set_relays : FOR bit = 0 TO 7
2020   IF relays & (1 << bit) THEN @CONTACT[bit] = 1 ELSE @CONTACT[bit] = 0
2030  NEXT bit
2040 RETURN
2110 REM send status subroutine
2120 `send_status : FPRINT #0, "=", relays : PRINT ">";"=";relays; : RETURN
Ready
```

On the PC side a C# WinForms application uses a GUI and event driven synchronous socket programming to interact with the BASIC program on the CFSound-IV via the Ethernet. Here's the GUI for the single form application:



And here is the important code in MainForm.cs. When the program starts the MainForm( ) constructor initializes the form controls and calls the GetRelays( ) method to obtain the current state of the remote relays. GetRelays( ) initializes a CmdResponse object with the "?" command and calls the SocketCommandResponse( ) method with a reference to the object.

The bulk of the program operation takes place in the SocketCommandResponse( ) method. First an ipAddress of the remote CFSound-IV (192.168.1.200) is constructed followed by a remote endpoint of port 1000 on that address. Next a client IPv4, TCP/IP streaming socket is constructed.

The socket then attempts a connection to the remote endpoint. It then generates a message for the CFSound-IV by formatting a string of bytes using information obtained from the referenced CmdResponse method parameter and following the CFSound's expected FINPUT format. The message is then sent via the connected socket to the CFSound-IV and any received bytes are read from the socket.

The received bytes are converted back into a string which is trimmed and parsed into the referenced response string and data byte elements. The socket is then shut down and closed and the method returns.

The UpdateRelayCheckboxes( ) method then uses the returned data to update the state of the GUI checkboxes.

When a GUI checkbox is changed by the user the checkbox CheckChanged event handler fires and the handler method calls the SetRelays( ) method. The SetRelays( ) method uses the passed parameters to construct a CmdReponse object with the "=" command and the desired state of the remote relays then the SocketCommandResponse( ) method is called as before to connect to the CFSound-IV, issue the command and return the response. The response is then used as before to update the state of the GUI checkboxes.

As you can see by the BASIC and C# code, a "?" command string simply returns an "=" response with the data consisting of a single bit representing the current state of each relay. An "=" command string uses the supplied data to update the desired state of the relays and returns an "=" response with the updated relay state.

You can monitor the Ethernet communication using a PC program such as Wireshark to monitor TCP/IP traffic to port 1000, and the CFSound BASIC program's state using a PC program such as TeraTerm connected to the CFSound-IV serial port.

```
using System;
```


Back to Table of Contents

```
using System.Net;
using System.Net.Sockets;
using System.Text;
using System.Windows.Forms;

namespace SocketRelays
{
    public partial class MainForm : Form
    {
        private class CmdResponse
        {
            public string Cmd { get; private set; }
            public string Response { get; set; }
            public byte Data { get; set; }

            public CmdResponse(string cmd, string response, byte data)
            {
                Cmd = cmd;
                Response = response;
                Data = data;
            }
        }

        private byte _relays;

        private bool SocketCommandResponse(ref CmdResponse cmdResponse)
        {
            byte[] receivedBytes = new byte[10];

            try
            {
                IPAddress ipAddress = new IPAddress(new byte[]{192,168,1,200});
                IPEndPoint remotEndPoint = new IPEndPoint(ipAddress, 1000);

                Socket client = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);

                try
                {
                    client.Connect(remotEndPoint);

                    StringBuilder sb = new StringBuilder();
                    sb.AppendFormat("\"{0}\",{1}\r\n", cmdResponse.Cmd, cmdResponse.Data);
                    byte[] msg = Encoding.ASCII.GetBytes(sb.ToString());

                    client.Send(msg);

                    client.Receive(receivedBytes);
                    String stringRecd = Encoding.ASCII.GetString(receivedBytes);

                    stringRecd = stringRecd.TrimEnd(new[] {'\r', '\n', '\0'});
                    stringRecd = stringRecd.Replace("\"", "");
                    String[] elements = stringRecd.Split(new[] {'"', ','});

                    cmdResponse.Response = elements[0];
                    byte data;
                    if (byte.TryParse(elements[1], out data))
                    {
                        cmdResponse.Data = data;
                    }

                    client.Shutdown(SocketShutdown.Both);

                    client.Close();
                }
                catch (Exception ex)
                {
                    MessageBox.Show(ex.ToString(), @"Error", MessageBoxButtons.OK, MessageBoxIcon.Error);
                    return false;
                }
            }
            catch (Exception ex)
            {
                MessageBox.Show(ex.ToString(), @"Error", MessageBoxButtons.OK, MessageBoxIcon.Error);
                return false;
            }

            return true;
        }

        private bool IsBitSet(byte data, int pos)
        {
            return (data & (1 << pos)) != 0;
        }

        private void UpdateRelayCheckboxes()
        {
            checkBoxRelay1.CheckedChanged -= checkBoxRelay1_CheckedChanged;
            checkBoxRelay1.Checked = IsBitSet(_relays, 0);
            checkBoxRelay1.CheckedChanged += checkBoxRelay1_CheckedChanged;

            checkBoxRelay2.CheckedChanged -= checkBoxRelay2_CheckedChanged;
```


Back to Table of Contents

```
            checkBoxRelay2.Checked = IsBitSet(_relays, 1);
            checkBoxRelay2.CheckedChanged += checkBoxRelay2_CheckedChanged;

            checkBoxRelay3.CheckedChanged -= checkBoxRelay3_CheckedChanged;
            checkBoxRelay3.Checked = IsBitSet(_relays, 2);
            checkBoxRelay3.CheckedChanged += checkBoxRelay3_CheckedChanged;

            checkBoxRelay4.CheckedChanged -= checkBoxRelay4_CheckedChanged;
            checkBoxRelay4.Checked = IsBitSet(_relays, 3);
            checkBoxRelay4.CheckedChanged += checkBoxRelay4_CheckedChanged;

            checkBoxRelay5.CheckedChanged -= checkBoxRelay5_CheckedChanged;
            checkBoxRelay5.Checked = IsBitSet(_relays, 4);
            checkBoxRelay5.CheckedChanged += checkBoxRelay5_CheckedChanged;

            checkBoxRelay6.CheckedChanged -= checkBoxRelay6_CheckedChanged;
            checkBoxRelay6.Checked = IsBitSet(_relays, 5);
            checkBoxRelay6.CheckedChanged += checkBoxRelay6_CheckedChanged;

            checkBoxRelay7.CheckedChanged -= checkBoxRelay7_CheckedChanged;
            checkBoxRelay7.Checked = IsBitSet(_relays, 6);
            checkBoxRelay7.CheckedChanged += checkBoxRelay7_CheckedChanged;

            checkBoxRelay8.CheckedChanged -= checkBoxRelay8_CheckedChanged;
            checkBoxRelay8.Checked = IsBitSet(_relays, 7);
            checkBoxRelay8.CheckedChanged += checkBoxRelay8_CheckedChanged;
        }

        private void GetRelays()
        {
            CmdResponse relayStatus = new CmdResponse("?", "", 0);
            if (SocketCommandResponse(ref relayStatus))
            {
                if (relayStatus.Response == "=")
                {
                    _relays = relayStatus.Data;
                    UpdateRelayCheckboxes();
                }
            }
        }

        private void SetRelays(bool setOn, byte relay)
        {
            int temp;

            if (setOn)
            {
                temp = (1 << relay);
                _relays |= (byte)temp;
            }
            else
            {
                temp = ~(1 << relay);
                _relays &= (byte)temp;
            }

            CmdResponse relayUpdate = new CmdResponse("=", "", _relays);

            if (SocketCommandResponse(ref relayUpdate))
            {
                if (relayUpdate.Response == "=")
                {
                    _relays = relayUpdate.Data;
                    UpdateRelayCheckboxes();
                }
            }
        }

        public MainForm()
        {
            _relays = 0;
            InitializeComponent();
            GetRelays();
        }

        private void checkBoxRelay1_CheckedChanged(object sender, EventArgs e)
        {
            SetRelays(checkBoxRelay1.Checked, 0);
        }

        private void checkBoxRelay2_CheckedChanged(object sender, EventArgs e)
        {
            SetRelays(checkBoxRelay2.Checked, 1);
        }

        private void checkBoxRelay3_CheckedChanged(object sender, EventArgs e)
        {
            SetRelays(checkBoxRelay3.Checked, 2);
        }

        private void checkBoxRelay4_CheckedChanged(object sender, EventArgs e)
```


Back to Table of Contents

```
        {
            SetRelays(checkBoxRelay4.Checked, 3);
        }

        private void checkBoxRelay5_CheckedChanged(object sender, EventArgs e)
        {
            SetRelays(checkBoxRelay5.Checked, 4);
        }

        private void checkBoxRelay6_CheckedChanged(object sender, EventArgs e)
        {
            SetRelays(checkBoxRelay6.Checked, 5);
        }

        private void checkBoxRelay7_CheckedChanged(object sender, EventArgs e)
        {
            SetRelays(checkBoxRelay7.Checked, 6);
        }

        private void checkBoxRelay8_CheckedChanged(object sender, EventArgs e)
        {
            SetRelays(checkBoxRelay8.Checked, 7);
        }
    }
}
```


Back to Table of Contents

Breaking Changes from CFSound-III BASIC

There are many changes between the CFSound-III BASIC and the CFSound-IV BASIC. However most programs written for the CFSound-III can be used on the CFSound-IV with a few changes:

- Variable names now case sensitive and can be up to 32 characters in length.
- Numbers and numeric variables are now 32 bits instead of 16 bits.
- Dimensioned variables and subscripted system variables (system variables start with '@') now use square brackets [ ] surrounding the index instead of parenthesis ( ) and can have up to 3 subscripts.
- PRINT concatenation character (suppress carriage return) changed to semicolon ';' from comma',', comma now outputs a space between items.
- EXITFOR command replaced by BREAK command.
- DMX functionality changed from DMX module to ArtNet™ DMX over Ethernet and DMX system variables renamed.

Back to Table of Contents

CFSound-IV BASIC Beta Software Changes

Fixes and Corrections

- Corrected the inability to call a user defined function with no arguments.
- Corrected the parsing of expressions within array indices to not require surrounding parenthesis to force expression evaluation.
- Corrected the operation of **ONERROR** when an error occurs within a user defined function.
- Corrections and changes to the **SOCKET.ASYNC.LISTEN** operation to improve usability.
- Fixed superfluous error message when error'ing or **ESC**aping from inside a user function.
- Fixed bug in variable handling when leaving function scope that generated self-linked symbols causing variables defined inside of functions to sometimes appear outside.
- Fixed problem where code executing in user functions could interfere with the operation of the calling program code when the user function was called in an expression.

Improvements

- Doubled the BASIC control stack from 96 to 192 slots.
- Made the **@SOCKET.TIMEOUT** system variable stream specific: **@SOCKET.TIMEOUT[#N]**.
- Added the display of included compilation options to the BASIC sign-on message.

```
Basic v3.2.2 May 19 2015 13:16:06 { HTTP EDIT }
Ready
```

- Added support for the use of the TAB character in program lines.
- Added a scan for the first **DATA** statement to the label and function scan at program startup to preclude the necessity of an **ORDER** statement before a **READ** statement.
- Enhanced the stream I/O error messages by adding STDERR output to BASIC error messages and removing the extra output that was being sent to console.
- Added a missing error message for the attempted **LOAD** of a non-existent program file.
- Added a missing error message for the attempted **RUN** of a non-existent program file.
- Added ability to **INCLUDE** statement to allow inclusion of code containing user defined functions with an optional offsetting line number feature.
- Added support for double-quoted filenames to **EDIT** command to allow editing of files whose name begins with a number.

New Features

- Added the ability to **pass an array by reference** as a user defined function argument.
- Compilation Option to support the experimental **HTTP.CGI** statement and operation.
- Compilation Option to support the **experimental full screen editor** augmentation of the **EDIT** command.
- Changed the **SIGNAL** and **WAIT** statements to **EVENT.SIGNAL** and **EVENT.WAIT**.
- Added **EVENT.ENABLE** and **EVENT.DISABLE** statements to allow protection of variables shared between event handlers and the main program loop by disabling, then re-enabling the event dispatching without clearing any pending event.
- Added support for real (floating point) numbers and variables, indicated with a trailing '%' character. Changed the modulus operator from (**%**) to **MOD**.
- Added **MATH.xx%( )** functions for real trig and log functions.

Back to Table of Contents

BASIC Revisions

Version	Date	Notes
2.0	15-Nov-12	• First release for new 320x240 Color LCD hardware.
2.1	27-Nov-12	• Added support for zero line number fall through in **ON GOTO/GOSUB** statements.
2.2	17-Dec-12	• Added support for optional PS/2 I/O expansion module; added **@CONTACT, @CLOSURE** and **@OPENING** system variables and events. • Added support for internal programming application. • Changed **ERR$()** to return the full error message. • Added setting/clearing of **FEOF[#n]** after **OPEN** if file is empty.
2.3	3-Jan-13	• Added **@SOUND$** queued sound support. • Added user function capability with **FUNCTION/ENDFUNCTION** and **CALL** statements. • Corrected number of files shown in **DIR** listing. • Corrected **FONTS** and **SCHEMES** listings color value display.
2.4	Internal Release	• Corrected label screen object text alignment to agree with scheme font alignment. • Added ability to add text label to icon screen object. • Added **DRAW.ARC** and **DRAW.ARC.STYLED** commands. • Added **DRAW.LINE.DASHED** and **DRAW.BOX.DASHED** commands. • Added **SORT** command. • Added **INCLUDE** command. • Changed **TEXTWIDTH()** and **TEXTHEIGHT()** to **TEXT.WIDTH()** and **TEXT.HEIGHT()**. • Changed **HEXVAL()** and **HEXSTR$()** to **HEX.VAL()** and **HEX.STR$()**. • Added **BITMAP.WIDTH()** and **BITMAP.HEIGHT()**. • Added **@CAPTURE** and **@PWM** system variables and events. • Added streaming IP connection support for **OPEN #N, CLOSE #N, INPUT #N** and **PRINT #N**. Added **@IP.EVENT[]** system variable and events.
2.5	Internal Release	• Added **INCLUDE** command. • Added Knob control. • Added Textbox control. • Added support for `labels.
2.6	11-Sep-13	• Changes for use in CFSound-IV application – added CFSound specific system variables, changed contact and sound interfaces.
2.7	25-Nov-13	• Changes to allow compilation for either 16 or 32-bit integer values.
2.8	16-Dec-13	• Internal restructuring of variable representation.
2.9	12-May-14	• Added **NUM** command for auto-numbering support. • Added command history capability. • Initial **SMTP** send e-mail capability.
3.0	9-Aug-14	• Internal restructuring of program source files. • Added **@CONFIG.FIELD$[], @CONFIG.SEPARATOR$[]** and **@CONFIG.ITEMS** system variables to facilitate writing configuration editor sample. • Made user functions first class and callable in expressions. • Added **SOCKET** async and **@SOCKET** sync functionality. • Added **SEARCH, BREAK** and **CONTINUE** commands. • Removed **EXITFOR** command now superceded by **BREAK,** added dyadic **PRINT USING** and **FMT$** capability. • Added **CHANGE** command. • Added **FILE.EXISTS()** and **@FILE.SIZE[]** and **@FILE.POSITION[]**. • Added optional starting position argument to **FIND()**.
3.1	4-Sep-14	• Corrected handling of directory paths to support nested directories. • Added support for multidimensional arrays up to 3 dimensions. • Changed **HEX.STR$()** to output upper case hexadecimal digits. • Fixed bugs with **FINSERT** and **FDELETE** commands.
3.1.0	15-Oct-14	• Added support for binary, octal and hexadecimal constants.
3.1.1	28-Oct-14	• Internal changes to correct embeddable functionality. • Replaced random number generator.

Back to Table of Contents

3.1.2	22-Nov-14	• Correct internal processing of labels during startup to fix missing line numbers in startup error messages.
3.1.3	12-Dec-14	• Correct operation of **@SOUNDFRAMESYNC** system variable.
3.1.4	9-Jan-15	• Corrected non-operation of **@CLOSURE** and **@OPENING** events. • Fixed **RTC** wrong year calculation.
3.1.5	26-Jan-15	• Correct **@EOT** system variable operation. • Changed command history stack to not remember **EDIT** command result.
3.2.0	**BETA** 11-Apr-15	• Correct inability to call a function with no arguments. • Added ability to pass an array by reference as a function argument. • Doubled BASIC software stack from 96 to 192 slots.
3.2.1	**BETA** 20-Apr-15	• Corrections and changes to **SOCKET.ASYNC.LISTEN** operation to improve usability.
3.2.2	**BETA** 19-May-15	• Corrected parsing of expressions within indices to not require surrounding parenthesis. • Made **@SOCKET.TIMEOUT** stream specific: **@SOCKET.TIMEOUT[#N]**. • Experimental **HTTP.CGI** compilation option added. • Experimental full screen text editor augmentation of **EDIT** command compilation option added. • Added display of included compilation options to sign-on message. • Added support for TAB character in program lines. • Made screen capture functionality a compile option as new Editor control keys were causing screen captures to be taken. • Added scan for first **DATA** statement to label and function scan at program startup to preclude necessity of **ORDER** statement before **READ** statement. • Enhanced stream I/O error messages by adding STDERR output to BASIC error messages and removing extra output to console. • Fixed superfluous error message when error'ing or ESCaping from inside a user function. • Added missing error message for **LOAD** of a non-existent program file. • Changed **SIGNAL** and **WAIT** commands to **EVENT.SIGNAL** and **EVENT.WAIT**. • Added **EVENT.ENABLE** and **EVENT.DISABLE** commands to allow protection of variables shared between event handlers and main program loop by disabling, re-enabling the event dispatching without clearing any pending event.
3.2.3	**BETA** 21-May-15	• Correct **ONERROR** operation when error occurs within a user defined function. • Added missing error message for **RUN** of a non-existent program file.
3.2.4	**BETA** 3-Jun-15	• Correct operation of **@SOUND$** when executing tight single-line program loops.
3.2.5	**BETA** 4-Sep-15	• Experimental **I2C** compilation option added. • Fixed bug in variable handling when leaving function scope that caused self-linked symbols causing variables defined inside of functions to sometimes appear outside. • Added ability to quote **EDIT** command filename to allow editing of files whose names begin with a number. • Improvements to **INCLUDE** statement to allow inclusion of code containing user defined functions with an optional offsetting line number feature. • Adjustment to error numbers for conditional compilation options to always be defined to prevent subsequent error numbers from changing depending upon what options are defined.
3.3.0	**BETA** 1-Oct-15	• Support for real (floating point) numbers and variables indicated with a **'%'** character suffix. • Changed the integer modulo operator from **'%'** to **MOD**. • Added **MATH.xx%()** commands for real trig and log operations. • Added error **ERR_RTL_ERRNO** (75) to display run time library errors generated from the **MATH.xx%()** commands. • Fixed problem where code executing in user functions could interfere with the operation of the calling program code when the user function was called in an expression.
3.3.1	**BETA** 1-Dec-15	• Fixed handling of numeric types in some MATH.xx() functions.

Back to Table of Contents

3.3.2	**BETA** 4-Jan-16	• Corrected parsing of I2C.READ and I2C.READ.RESTART. • Added support to I2C.READ.x commands for Integer Array variables.
3.3.3	**BETA** 2-Feb-16	• Corrected operation of offsetting line number option in INCLUDEd files. • Corrected handling of array references to support references passed through nested function calls. • Changed VARS command to show array references equal to the referenced array instead of showing contents multiple times.
3.3.4	**BETA** 29-Feb-16	• Changed EDIT command to not allow editing of INCLUDEd program lines as they will be overwritten when the program is RUN. • Changed LIST to show INCLUDEd program lines with a '+' character in front of the resulting offset line number.
3.3.5	**BETA** 23-Aug-16	• Corrected bug in FPRINT, FINSERT and FWRITE statements where incorrect values for number variables are output caused by missed condition when support for real (floating point) numbers was added.
3.3.6	**BETA** 9-Sep-16	• Corrected function argument evaluation in SOCKET.SYNC.x() built-in functions to fix expression nesting and type errors that can occur during certain conditions.
3.3.7	**BETA** 21-Nov-16	• Added @I2C.x_TIMEOUT system variables to support I2C chips that require clock stretching to operate.
3.3.8	**BETA** 30-Nov-16	• Corrected operation of multibyte I2C.READ.x statements with respect to endianness.
3.3.9	**BETA** 6-Feb-17	• Added I2C.PING command. • Implemented hardware timeouts for I2C bus initialization, default the @I2C.x.TIMEOUT systems at the beginning of I2C.INIT instead of at end. • Internal code rewrite to break out groups of Basic commands into separate files to improve code readability and maintenance.
3.3.10	**BETA** 20-Jun-17	• Improved parsing of nested FOR/NEXT and WHILE/WEND loops to prevent the appearance of enclosed system variables and/or prefixed commands from generating incorrect nesting errors.

Back to Table of Contents

Index

Back to Table of Contents

Please Read Carefully:

Information in this document is provided solely in conjunction with Arbitrary Precision products. Arbitrary Precision reserves the right to make changes, corrections, modifications or improvements, to this document, and the products and services described herein at any time without notice.

All Arbitrary Precision products are sold pursuant to Arbitrary Precision's terms and conditions of sale.

Purchasers are solely responsible for the choice, selection and use of the Arbitrary Precision products and services described herein, and Arbitrary Precision assumes no liability whatsoever relating to the choice, selection or use of the Arbitrary Precision products and services described herein.

No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted under this document. If any part of this document refers to any third party products or services it shall not be deemed a license or grant by Arbitrary Precision for the use of such third party products and services, or any intellectual property contained therein or considered a warranty covering the use in any manner whatsoever of such third party products or services or any intellectual property contained therein.

UNLESS OTHERWISE SET FORTH IN ARBITRARY PRECISION'S TERMS AND CONDITIONS OF SALE ARBITRARY PRECISION DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY WITH RESPECT TO THE USE AND/OR SALE OF ARBITRARY PRECISION PRODUCTS INCLUDING WITHOUT LIMITATION IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE (AND THEIR EQUIVALENTS UNDER THE LAWS OF ANY JURISDICTION), OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

ARBITRARY PRECISION PRODUCTS ARE NOT AUTHORIZED FOR USE IN WEAPONS. NOR ARE ARBITRARY PRECISION PRODUCTS DESIGNED OR AUTHORIZED FOR USE IN: (A) SAFETY CRITICAL APPLICATIONS SUCH AS LIFE SUPPORTING OR SYSTEMS WITH PRODUCT FUNCTIONAL SAFETY REQUIREMENTS; (B) AERONAUTICAL APPLICATIONS; (C) AUTOMOTIVE APPLICATIONS OR ENVIRONMENTS, AND/OR (D) AEROSPACE APPLICATIONS. THE PURCHASER SHALL USE PRODUCTS AT PURCHASER'S SOLE RISK, EVEN IF ARBITRARY PRECISION HAS BEEN INFORMED IN WRITING OF SUCH USAGE.

Resale of Arbitrary Precision products with provisions different from the statements and/or technical features set forth in this document shall immediately void any warranty granted by Arbitrary Precision for the Arbitrary Precision product or service described herein and shall not create or extend in any manner whatsoever, any liability of Arbitrary Precision.

Arbitrary Precision and the Arbitrary Precision logo are trademarks of Arbitrary Precision.
Information in this document supersedes and replaces all information previously supplied.

www.arbitrary-precision.com