ACS Basic

User's Manual

v1.31

12/31/2010

ACS

Ackerman Computer Sciences

On The Cutting Edge of Technological Evolution

6233 E. Sawgrass Rd • Sarasota, FL. 34240 • (941)377-5775 FAX(941)378-4226
www.acscontrol.com

Notice

This Information or any portion thereof remains the property of ACS. The Information contained herein is believed to be accurate and ACS assumes no responsibility or liability for its use in any way and conveys no license or title under any patent or copyright and makes no representation or warranty that this Information is free from patent or copyright infringement.

ACS warrants that the specified product shall function in accordance with the features of the design for a period of one (1) year from the date of purchase. This warranty does not cover any ACS product which has been subjected to any abuse, misuse, accident, act of God, alteration or modification not authorized by ACS in writing. ACS offers no other warranty, either expressed or implied and specifically denies all other warranties, including any warranty for merchantability or fitness. ACS's sole obligation upon the discovery of any error in the specified product or breach of the warranty in this paragraph shall be to replace or repair the specified product or to correct the design of the specified product. Under no circumstances shall ACS, its owners, officers, employees or agents be held liable for any special, incidental, indirect, consequential or other damages (including lost profits, fees or revenues). Any purchase of our products or use of our services constitutes your complete agreement and binds you and your company to all the terms, policies, conditions and prices as is herein described.

ACS PRODUCTS ARE NOT AUTHORIZED FOR USE AS CRITICAL COMPONENTS IN LIFE SUPPORT DEVICES OR SYSTEMS WITHOUT THE EXPRESS WRITTEN APPROVAL OF ACS.

As used herein:

1. Life support devices or systems are devices or systems which, (a) are intended for surgical implant into the body, or (b) support or sustain life, and whose failure to perform, when properly used in accordance with instructions for use provided in the labeling, can be reasonably expected to result in a significant injury to the user.

2. A critical component is any component of a life support device or system whose failure to perform can be reasonably expected to cause the failure of the life support device or system, or to affect its safety or effectiveness.

Symbolic Abbreviations

In this manual, the following symbolic abbreviations apply:

#N	Represents a file number: #`0 <= `***`#N`***` <= #23`
var	Represents a numeric program variable
var$	Represents a string program variable
var()	Represents a numeric array program variable
@specialvar	Represents a special program variable
line	Represents a program line number: `0 <= `***`line`***` <= 65535`
[ ]	Delineates optional arguments or parameters
filename	Represents a DOS style 8.3 filename – up to 8 characters with an optional 3 character extension
path	Represents a complete path to a file including the ***filename*** without leading backslash. There is no concept of a current directory other than the root file system.
statement	Represents a program statement
expr	Represents a program expression
recordlength	Represents a Fixed Length File I/O record length: `0 <= `***`recordlength`***` <= 127` including the trailing CR/LF on the end of each record
recordnumber	Represents a Fixed Length File I/O record number: `0 <= `***`recordnumber`***` <= 32767`
color	Represents a 16-bit pixel color expressed as RGB565

Table of Contents

Features

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

ACS 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.

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 ACS Basic a Program consists of one or more program lines. Each program line consists of a line number followed by one or more statements. 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 ACS Basic is awaiting statement or program line entry it issues a READY prompt via the serial port.

```
ACS Basic v1.4 Sep 25 2006 11:44:00
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
LCDDEMO.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 Compact Flash 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
```

ACS strongly recommends developing Basic programs interactively via a connected terminal / computer or optional VGA / PS2 keyboard 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, ACS 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.)

Entering an Escape character (0x1B) twice in succession via the serial port while a program is running will cause termination of the program and ACS 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
```

Variables

ACS Basic has four types of variables: **16-bit Integer Numeric**, **16-bit Integer Numeric Arrays, unsigned 8-bit character Strings** and **unsigned 8-bit character String Arrays**.

Variable names *are not* case sensitive.

Numeric variables can assume the integer values **(–32768 ≤ variable ≤ +32767)**. Character Strings are limited to **255 characters** in length.

The 260 Numeric variables are named **A0 → A9 … Z0 → Z9**.

The 260 Numeric Array variables are named **A0( ) → A9( ) … Z0( ) → Z9( )**. Array variables must be **DIM**ensioned using the ***DIM*** statement before use.

The 260 Character variables are named **A0$ → A9$ … Z0$ → Z9$**.

The 260 Character Array variables are named **A0$( ) → A9$( ) … Z0$( ) → Z9$( )**. Array variables must be **DIM**ensioned using the ***DIM*** statement before use.

Note that the zero suffix variables may be referenced by their letter name only so that A is equivalent to A0, Z$ is equivalent to Z0$, etc.

Special Variables

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

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

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

@TIMER(x)

The **@TIMER(x)** special variables allow the ACS Basic program to measure or control time intervals. There are ten timers; permissible values for (x) are 0 through 9. Setting the variable to a non-zero value activates the timer. The 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.

@PORT(x), @PORT2(x)

The **@PORT(x)** and **@PORT2(x)** special variables allow the ACS Basic program to access I/O ports. 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-3 Contact I/O modules.

@CONTACT(x)

The **@CONTACT(x)** special variables allow the ACS Basic program to access CFSound-3 contacts. There are up to 56 contact inputs and up to 16 contact outputs depending upon what optional modules have been installed on the CFSound-3; 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).

@CLOSURE(x)

The **@CLOSURE(x)** special variables allow the ACS Basic program to access CFSound-3 contact events. . There are up to 56 contact inputs depending upon what optional modules have been installed on the CFSound-3; 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(24),GOSUB 100
20 ONEVENT @CLOSURE(25),GOSUB 200
30 GOTO 30
100 PRINT "contact 25 closed":RETURN
200 PRINT "contact 26 closed":RETURN
Ready
run
contact 25 closed
contact 26 closed
```

@OPENING(x)

The **@OPENING(x)** special variables allow the ACS Basic program to access CFSound-3 contact events. There are up to 56 contact inputs depending upon what optional modules have been installed on the CFSound-3; 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.

@FEOF(#N)

The **@FEOF(#N)** special variable allows the ACS 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.

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

These special variables allow the ACS Basic program to access the Real-Time Clock/Calendar. 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$

The **@SOUND$** special variable allows the ACS Basic program to queue sound files for playing. Queued sound files are played in the order that they were queued, being removed as they are played. 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. 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** special variables allow the ACS Basic program to control the CFSound-3 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-3 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.

@BAUD

The **@BAUD** special variables allow the ACS Basic program to control the CFSound-3 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. A selector is used to allow baud rates greater than 28800 which would result from the 16-bit integer limitation of the Basic language. Note that the baud rate selector is saved in non-volatile memory and is restored every time the CFSound-3 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 special variable should not be written in a loop or on every program execution. Exercise caution to avoid non-volatile memory failure. A good practice is to check the variable's value and only then write to it if it is not the desired value.***

@BAUD	Baud Rate
0	110
1	300
2	600
3	1200
4	1800
5	2400 (factory default)
6	3600
7	4800
8	7200
9	9600
10	14400
11	19200
12	28800
13	38400
14	57600
15	115200
16	230400

@MSG$

This special 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$**. It retains the framed message until it is read at which point the search for the next received **@SOM** begins again. It may also be cleared by assigning it a string value, which is not saved.

@MSGENABLE

This special variable controls whether the serial data stream is parsed for messages as outlined in the **@MSG$** description above. The ability to disable **@MSG$** processing is required to support the **GETCH()** function on the serial port. It defaults to 1 (enabled).

@EOT

This special variable returns 1 when any serial data sent by BASIC console operation, or PRINT or LCDx statements has finished transmitting. It can be cleared by setting it to zero, but will immediately return 1 again unless serial data is sending.

@SOM

This special variable determines the character used to delineate the Start of Message. It defaults to ASCII SOH (01).

@EOM

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

@PTT

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

@MUTE

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

@LINEIN

Writing this special variable to a non-zero value enables the CFSound-III Line level Input. Setting it to zero disables the Line level input. Reading this special variable returns 1 if the line level input is enabled, else zero. The **RUN** command automatically disables the Line level input. 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.

@DMXRESET

Writing this special variable to a non-zero value resets the optional DMX I/O module if present.

@DMXMASTER

Writing this special variable to a non-zero value enables the optional DMX I/O module as a master, controller if present. A value of zero enables sets slave, device mode.

@DMXFRAMEDELAY

Writing this special variable sets the inter-frame delay in multiples of 20mSEC when the optional DMX I/O module is present and configured as a master.

@DMXCHANNELS

Writing this special variable sets the number of channels transmitted times 2 if the optional DMX I/O module is present and configured as a master.

@DMXDATA(x)

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

@DMXANALOG(x)

Gets or sets the current value of analog input x ($0 \le x \le 7$) if the optional DMX I/O module is present.

@DMXFRAMESYNC

Returns a 1 if a DMX frame has been sent (DMX master mode) or received (DMX slave mode) since last checked else returns 0. Optionally an event handler specified with an **ONEVENT** statement may be activated when this event occurs.

@LCDADDRESS

This special variable sets the current value of the LCD address to be used with all of the LCDx commands. When a LCDx statement is processed, the value of @LCDADDRESS is tested.

If @LCDADDRESS is set to a value greater than or equal to zero, the generated LCDx commands include the LCD address prefix characters (0 ≤ @LCDADDRESS ≤ 255) inserted after the initial SOH and before the command character.

If @LCDADDRESS is set to a value less than zero, the generated LCDx commands do not include the LCD address prefix characters.

@LCDADDRESS defaults to a value of -1 when ACS Basic is started, the NEW statement is executed or a program is loaded.

See the ACS-LCD-128x64 and ACS-LCD-320x240 Display User Manuals for additional information about display addressing.

@LCDTYPE

This special variable sets the current value of the LCD type which controls the operation of the LCDx commands. The currently supported values are 0 = ACS LCD128x64 command formatting (the default), 1 = ACS LCD320x240 command formatting.

@LCDTYPE defaults to a value of 0 when ACS Basic is started, the NEW statement is executed or a program is loaded.

See the ACS-LCD-128x64 and ACS-LCD-320x240 Display User Manuals for additional information about display addressing.

@SOUNDFRAMEPRESCALER

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

@SOUNDFRAMESYNC

This special 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. This is a 16-bit signed integer that will wrap negative as it increments past 32767 requiring a judicious choice of @SOUNDFRAMEPRESCALER value to allow the range to accommodate the length of the sound being synchronized to:

@SOUNDFRAMEPRESCALER=1 yields 20 mSEC per frame → max 655 second sound

@SOUNDFRAMEPRESCALER=50 yields 1 SEC per frame → max 32768 second sound

@VGAMODE

This special variable gets or sets the current Video Graphics Adaptor resolution per the following table:

@VGAMODE	**Video Graphics Adaptor Resolution**
0	640 x 480 x 16 @ 72Hz
1	640 x 480 x 16 @ 75Hz
2	800 x 600 x 16 @ 72Hz
3	1024 x 768 x 16 @ 70Hz

Setting ***@VGAMODE*** sets ***@VGADRAWPAGE***=1, ***@VGAUPDATEPAGE***=0 and ***@VGASHOWPAGE***=0, fills the graphics page with black, restores the clipping rectangle to full screen and does a screen update to show the result.

@VGAENABLE

Gets or sets the state of the Video Graphics Adaptor blanking. Setting this non-zero (the default) will enable the display, setting this to zero will blank the display. The display contents are not affected by this command.

@VGADRAWPAGE

Gets or sets the current drawing page that will be used by the VGAx statements. Defaults to zero upon Reset or whenever the ***@VGAMODE*** is set. There are a total of five drawing pages numbered 0 → 4.

@VGAUPDATEPAGE

Gets or sets the current Video Graphics Adaptor display page that will be updated by the VGAx statements. Defaults to zero upon Reset or whenever the ***@VGAMODE*** is set. The number of available pages is a function of the ***@VGAMODE***:

@VGAMODE	**Resolution**	**Number of Pages**
0	640 x 480	109
1	640 x 480	109
2	800 x 600	69
3	1024 x 768	42

If ***@VGAAUTOUPDATE***=0 then setting ***@VGAUPDATEPAGE*** will cause the page to be updated.

@VGASHOWPAGE

Gets or sets the current Video Graphics Adaptor display page that will be displayed if ***@VGAENABLE***=1. Defaults to zero upon Reset or whenever the ***@VGAMODE*** is set. See ***@VGAUPDATEPAGE*** above for the number of available pages.

@VGAAUTOUPDATE

Gets or sets the state of the Video Graphics Adaptor update mechanism – VGAx commands will automatically cause a screen update if ***@VGAAUTOUPDATE***=1 (default). ***Setting @VGAAUTOUPDATE***=0 allows multiple VGAx commands to be issued without updating the screen resulting in faster drawing but requires setting ***@VGAUPDATEPAGE*** to cause the screen to update when drawing is done.

@VGAWIDTH

This read only special variable gets the width of the screen in pixels for the current Video Graphics Adaptor ***@VGAMODE*** setting.

@VGAHEIGHT

This read only special variable gets the height of the screen in pixels for the current Video Graphics Adaptor ***@VGAMODE*** setting.

@VGAPRINT

This special variable enables or disables whether PRINT statements are also sent to the optional Video Graphics Adaptor as ANSI text. The default is enabled (1). See the PRINT statement below for additional information.

@VGASHOWCURSOR

This special variable enables or disables the display of a flashing cursor showing the current PRINT position on the optional Video Graphics Adaptor. The default is enabled (1).

Events

ACS 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 special variables have ***Events*** associated with them and may be referenced in **ONEVENT, SIGNAL** and **WAIT** statements.

There are two ways to process an event: asynchronously with an **ONEVENT** handler or synchronously with a **WAIT** statement or by polling the special 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.

Events have an implicit priority with higher priority events being able to interrupt execution of lower priority event handlers. Here's an example of an event handling a closure on Contact 25 (contact numbers start at zero):

```
10 REM setup event subroutine for when contact 25 closes
15 ONEVENT @CONTACT(24),GOSUB 100
20 REM do whatever here
25 GOTO 20
100 REM contact 25 closed event
105 PRINT "CONTACT(25) closed"
110 RETURN
```

This would print "CONTACT(25) closed" whenever Contact 25 closes.

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 special variable's value in a loop looking for the event to have been signaled. Here's an example of polling for a closure on Contact 25:

```
10 REM poll contact(25) closures
15 IF @CONTACT(24) = 1 THEN 100
20 REM do whatever here
25 GOTO 15
100 REM contact 25 closed
105 PRINT "CONTACT(25) closed, clear it"
110 @CONTACT(24)=0
115 REM do whatever here
120 GOTO 15
```

This would print "CONTACT(25) closed, clear it" whenever Contact 25 closes. If you poll for events, you have to manually clear them in order to see the next one – ONEVENT handling does this clearing automatically.

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 below for a table showing what events may be processed and listing their relative priority.

Statements

ACS 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. A Direct mode of operation is available for some statements when they are entered without a line number and are executed immediately. Here are some sample program statements:

```
10 REM This is a comment
20 FOR I=0 TO 10:PRINT I:NEXT I
```

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, the statement line numbers are converted to a two-byte unsigned integer form to save space and facilitate program execution. Saved programs are expanded (un-tokenized) on the CF card to allow program storage, viewing and editing with an external text editor if required.

The following statement keywords are supported:

CLEAR

Erases all variables and closes all open files.

CLOSE #N

Close file #*N* (0 → 23) opened with **OPEN** statement.

DATA

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

DEL path

Delete files and directories on the Compact Flash card. The full ***path*** must be specified without a leading backslash. Directories must be empty to be deleted. ***Path*** may be a constant string or you can use a string variable as the ***path*** by concatenating it to such a string: **DEL ""+P$**.

DELAY value

Pause program execution for value * 20mSEC. While the delay is in process, Events can occur but any defined **ONEVENT** handlers will not be executed until the delay has expired.

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

DIM var[$](size)[, ...]

Dimension a numeric or character array **var**iable to hold **size** integers or character strings. Array variable elements may then be accessed using a numeric index in parenthesis that ranges from the first element of zero to the last element of size: A(0), A(1), … , A(size). 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 size an “Index Out of Range Error” will result. A variable may be re-dimensioned, however the current contents of the variable will be lost.

DIR [path]

Show files on the Compact Flash card. An optional ***path*** may be specified without a leading backslash. Wildcard characters ‘?’ and ‘*’ may be used to match multiple files.

DIR #N, [path]

Write a list of files on the Compact Flash card to an open file #***N*** (0 → 23). An optional ***path*** may be specified without a leading backslash. Wildcard characters ‘?’ and ‘*’ may be used to match multiple files.

EDIT line

Direct mode only. Using an ANSI terminal or the optional VGA module allows editing a line by displaying the statement, moving the cursor with the Home, Left arrow, Right arrow, End and Backspace keys. Typed characters are entered at the cursor. The Enter key accepts the changes, a double ESC key aborts the edit.

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.

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

EXITFOR line

Program mode only. Exit out of a **FOR/NEXT** loop by popping the **FOR** off of the control stack and jumping to ***line***.

FINPUT #N, var[$], … , var[$]

Gets value(s) for one or more **var**iables from a single line from file #***N*** (0 → 23). Note that when an end of file occurs, the **var**iables will have their last value. Test the **@FEOF(#N)** specialvar 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.

FOR var=init TO limit [STEP increment]

Program mode only. Perform a counted loop; incrementing ***var*** from the ***init*** value to the ***limit*** value by the optional ***increment*** value, executing statements up until the matching **NEXT** statement. The maximum number of nested **FOR/NEXT** loops and **GOSUB** subroutines is currently 50.

FOPEN #N, recordlength, "path"

Opens filename ***path*** as a fixed record length file #***N*** (0 → 23) 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.

FPRINT #N, expr[,expr…]

Prints one or more expression(s) to the file #N (0 → 23) that is **OPEN**ed for writing as a single line. The data items on the line in the file are separated by commas, with string values surrounded by double quotes. The produced file is compatible with the **FINPUT #N** statement.

FREAD #N, recordnumber, var[$], var[$], … var[$]

Reads ASCII data from fixed length records on file ***#N*** (0 → 23) opened by **FOPEN #N** into the list of variables. Before the data is read, the file is positioned to the desired ***recordnumber (0 ≤ recordnumber ≤ 32767)***. A negative ***recordnumber*** seeks to the end of the file.

Reading at the current end of the file sets the **@FEOF(#N)** specialvar 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.

FWRITE #N, recordnumber, var[$], var[$], ... var[$]

Writes ASCII data into fixed length records on file ***#N*** (0 → 23) opened by ***FOPEN #N*** from the list of variables. Before the data is written, the file is positioned to the desired ***recordnumber (0 ≤ recordnumber ≤ 32767)***. A negative ***recordnumber*** seeks to the 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 file may be viewed using the **TYPE** command.

FINSERT #N, recordnumber, var[$], var[$], ... var[$]

Inserts ASCII data into fixed length records on file ***#N*** (0 → 23) 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 (0 ≤ recordnumber ≤ 32767)***, 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 file may be viewed using the **TYPE** command.

FDELETE #N, recordnumber

Removes fixed length record ***recordnumber (0 ≤ recordnumber ≤ 32767)*** on file ***#N*** (0 → 23) opened by **FOPEN #N** using a temporary file FDELETE.TMP.

GOSUB line

Program mode only. Calls a subroutine that starts at ***line*** and ends with a **RETURN** statement. A subroutine consists of a group of program statements that start at a certain ***line*** number 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. Upon execution of the **RETURN** statement, program execution continues at the statement after the **GOSUB**. The maximum number of nested **FOR/NEXT** loops and **GOSUB**s is currently 50.

GOTO line

Program mode only. Program execution continues by jumping to ***line***.

IF test THEN line/statement [ELSE line2/statement2]

Program mode only. Conditional execution jump. The expression ***test*** is evaluated, and if non-zero, program execution continues at ***line*** 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 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”
```

INPUT var

Get value for variable from the serial port.

INPUT "prompt", var

Get value of **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 such a string: **INPUT** ""**+A$, B$**

INPUT #N, var

Get value for **var**iable from file #***N*** (0 → 23). Note that when an end of file occurs, the **var**iable will have its last value. Test the **@FEOF(#N)** specialvar to detect this condition.

[LET]var[$]=expr[$] (default statement)

Program or Direct mode. Sets **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** (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 program lines to the serial port. 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 display not already queued.

LIST #N [start[,end]] LIST #N [start[-end]]

Direct mode only. List program lines to open file #***N*** (0 → 9). 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 display not already written.

LOAD path

Program or Direct mode. Load an ACS Basic program from a Compact Flash 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 <u>not</u> 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 <u>last</u> statement on a line. If not present, the .BAS file extension on the filename at the end of the path is assumed.

```
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 CF card. Makes a new directory on the Compact Flash card. ***Path*** must be a complete path for the new directory without the leading backslash, 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*** ""+***P$***.

MEMORY

Displays the currently available program memory and CF card memory if a CF card is present.

NEW

Direct mode only. Erase all program statements, clear all variable values and closes all open files.

NEXT [var]

Program mode only. End of a counted loop. Statement execution resumes with the matching **FOR** statement if the step increment of the control variable has not reached the limit. Execution of a **NEXT** statement without a preceding **FOR** causes a “Nesting Error”.

ON expr, GOSUB line0, line1, line2, … ,lineN

Program mode only. Case statement dispatching via subroutines. The value of ***expr*** is evaluated, and a subroutine call 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. Upon return from the **GOSUB** execution continues with the next statement.

```
5 REM ONGOSUB Demo
10 a=0
20 ON a,GOSUB 100,200,300,400,500
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
500 PRINT "5"
505 a=0
510 RETURN
Ready
run
12345
12345
12345
12345
12345
1 ESC at line 105
Ready
```

ON expr, GOTO line0, line1, line2, … , lineN

Program mode only. Case statement dispatching via 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.

```
5 REM ON GOTO DEMO
10 a=0
20 ON a,GOTO 100,200,300,400,500
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
500 PRINT "5"
505 a=a+1
510 GOTO 20
Ready
run
12345
12345
12345
1234 ESC at line 20
Ready
```

ONERROR GOTO line

Program mode only. Provides one-shot error handling. Upon any error, statement execution starts at line, and 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 bracket 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
```

ONEVENT @specialvar, GOSUB line

Program mode only. Provides semi-asynchronous event handling via subroutines. Certain ACS Basic special variables can trigger events. The **ONEVENT** statement allows the event to be associated with the execution of a subroutine. 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 special variables can cause events and are listed in order of ***decreasing*** priority:

@SOUNDFRAMESYNC	The event occurs every @SOUNDFRAMEPRESCALER x 20mSEC while a sound is playing.
@DMXFRAMESYNC	The event occurs after a DMX frame is sent (master) or received (slave). (v1.24 or later)
@TIMER(x)	The event occurs one time whenever the timer counts down to zero. Special variable **@TIMER(0)** is the highest priority, followed by **@TIMER(1)**, … then **@TIMER(9)**. 0 ≤ x ≤ 9
@CLOSURE(x)	The event occurs whenever the associated CFSound-3 contact has closed. 0 ≤ x ≤ 55
@OPENING(x)	The event occurs whenever the associated CFSound-3 contact has opened. 0 ≤ x ≤ 55
@FEOF(#N)	The event occurs after **INPUT #N**, **FINPUT #N** or **FREAD #N** reaches the end of file #N (0 → 23)
@SECOND	The event occurs once per second.
@MINUTE	The event occurs once per minute.
@HOUR	The event occurs once per hour.
@DOW	The event occurs once per day at midnight.
@DATE	The event occurs once per day at midnight.
@MONTH	The event occurs once per month at midnight of day 1.
@YEAR	The event occurs once per year.
@SOUND$	The event occurs after the last queued **@SOUND$** sound has finished playing.
@MSG$	The event occurs after receipt of a serial character stream delineated by the **@SOM** and **@EOM** characters.
@EOT	The event occurs upon complete transmission of a serial character stream of one or more characters when both the output buffer and UART are empty.

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** self statement.:

```
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
```

OPEN #N, "path", "options"

Open filename ***path*** as file #***N*** (0 → 23) for subsequent access via **DIR #N**, **INPUT #N, FINPUT #N, PRINT #N** or **FPRINT #N** statements. The ***options*** string characters are:

"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

Program mode only. This statement positions the read data pointer to statement ***line*** number. The statement at ***line*** must be a series of one or more **DATA** statement.

PLAY file

Plays the sound ***file*** and waits 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-3 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*** ""***+P$***. 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.

In order to play sounds while continuing program execution use the **@SOUND$** special variable.

PRINT expr[, expr ...][,]

Prints one or more expression(s) to the serial port (and optional VGA display if **@VGAPRINT**=1). If the statement ends in a comma (",") no Carriage Return / Line Feed pair is appended to the printed expressions allowing multiple print statements to display on the same line.

If the optional Video Graphics Adaptor is installed, the **PRINT** statement is also shown on the attached display as ANSI text. The ANSI text is printed using a fixed-pitch font in the current @VGAMODE setting with the number of characters per line and lines per screen automatically adjusted to overlay the entire screen. The size of the fixed-pitch font is 5 x 7 pixels in a 6 x 12 box to improved screen readability, allow for lower-case descenders and accommodate a flashing underline cursor.

The location of the printed text starts at the upper left corner of the screen (0, 0) and ranges to the lower right corner. The location may be controlled by the use of embedded ANSI control sequences to position the 'cursor' before printing.

@VGAMODE	Upper Left Col, Row	Lower Right Col, Row
0 = 640 x 480	0, 0	106, 40
1 = 640 x 480	0, 0	106, 40
2 = 800 x 600	0, 0	133, 50
3 = 1024 x 768	0, 0	170, 64

The following ANSI cursor controls are supported by the Video Graphics Adaptor (or attached ANSI terminal) and may be invoked in **PRINT** commands by embedded the required ANSI character sequence in the PRINT statement's text:

Backspace (BS)

Value (ASCII 8 decimal / 08 hex) Receipt of this character causes the display to move the cursor one position to the left and over-write any displayed character with a blank (space). This ANSI control character may be issued using the **PRINT CHR$(8),** statement.

Horizontal Tab (HT)

Value (ASCII 9 decimal / 09 hex) Receipt of this character causes the display to move the cursor right to the next tab stop. Moving past the rightmost tab stop causes the cursor to move to the beginning of the following line with display scrolling up if the cursor was on the last line. There are 9 tab stops per line at positions 4, 8, 12, 16, 20, 24, 28, 32 and 36. This ANSI control character may be issued using the **PRINT CHR$(9),** statement.

Line Feed (LF)

Value (ASCII 10 decimal / 0A hex) Receipt of this character causes the display to move the cursor down to the next line in the same column. A carriage return (CR) character is automatically prepended. The display will scroll up if the cursor was on the last line. This ANSI control character may be issued using the **PRINT CHR$(10),** statement.

Vertical Tab (VT)

Value (ASCII 11 decimal / 0B hex) Receipt of this character causes the display to move the cursor down to the next line in the same column. The display will scroll up if the cursor was on the last line. This ANSI control character may be issued using the **PRINT CHR$(11),** statement.

Form Feed (FF)

Value (ASCII 12 decimal / 0C hex) Receipt of this character causes the display to move the cursor down to the next line in the same column. The display will scroll up if the cursor was on the last line. This ANSI control character may be issued using the **PRINT CHR$(12),** statement.

Carriage Return (CR)

Value (ASCII 13 decimal / 0D hex) Receipt of this character causes the display to move the cursor left to the first column on the current line. This ANSI control character may be issued using the **PRINT CHR$(13),** statement. Note that all **PRINT** statements without a trailing comma result in a trailing CR, LF sequence being sent to the VGA.

Cancel (CAN)

Value (ASCII 24 decimal / 18 hex) Receipt of this character causes the display to abort any escape sequence that may be in process. No other action is taken. This ANSI control character may be issued using the **PRINT CHR$(24),** statement.

Escape (ESC)

Value (ASCII 27 decimal / 1B hex) Receipt of this character causes the display to attempt to decode one or more of the following characters as a control or escape sequence that will affect the display. This ANSI control character may be issued using the **PRINT CHR$(27),** statement.

Displayed Characters

Values (ASCII 32 decimal / 20 hex through ASCII 127 decimal / 7F hex) Receipt of these characters cause the display to show the character on the screen at the current cursor location, and then move the cursor right to the next position. The cursor will automatically wrap to the beginning of the next line, if required, scrolling the screen contents up if the cursor was on the last line. The following characters are displayed:

	Upper Bits					
Lower Bits	0010	0011	0100	0101	0110	0111
0000	space	0	@	P	`	p
0001	!	1	A	Q	a	q
0010	“	2	B	R	b	r
0011	#	3	C	S	c	s
0100	$	4	D	T	d	t
0101	%	5	E	U	e	u
0110	&	6	F	V	f	v
0111	‘	7	G	W	g	w
1000	(	8	H	X	h	x
1001	)	9	I	Y	i	y
1010	*	:	J	Z	j	z
1011	+	;	K	[	k	{
1100	,	<	L	\	l	|
1101	-	=	M	]	m	}
1110	.	>	N	^	n	→
1111	/	?	O	_	o	←

Reset Display (ESC c)

Values (ASCII 27, 99 decimal / 1B, 63 hex) Receipt of this character sequence causes the display to clear, the cursor position to move to the upper left corner and the backlight to turn off. This ANSI control character sequence may be issued using the **PRINT CHR$(27),”c”,** statement.

Cursor Down (ESC D)

Values (ASCII 27, 68 decimal / 1B, 44 hex) Receipt of this character sequence causes the display to move the cursor down to the next line in the same column. The cursor will not move and the display will not scroll up if the cursor was on the last line. This ANSI control character sequence may be issued using the **PRINT CHR$(27),”D”,** statement.

Cursor Down to column 1 (ESC E)

Values (ASCII 27, 69 decimal / 1B, 45 hex) Receipt of this character sequence causes the display to move the cursor down to the next line and the first column. The cursor will not move and the display will not scroll up if the cursor was on the last line. This ANSI control character sequence may be issued using the **PRINT CHR$(27),”E”,** statement.

Cursor Up (ESC M)

Values (ASCII 27, 77 decimal / 1B, 4D hex) Receipt of this character sequence causes the display to move the cursor up to the previous line in the same column. The cursor will not move if the cursor was on the first line. This ANSI control character sequence may be issued using the **PRINT CHR$(27),”M”,** statement.

ANSI Escape Sequences (ESC [ )

Values (ASCII 27, 91 decimal / 1B, 5B hex) Receipt of this character sequence causes the display to attempt to decode one or more of the following characters as an ANSI control sequence. These sequences can have 1 or more parameters that are expressed as decimal numbers separated by a semicolon.

The absence of a parameter in a control sequence that accepts a single parameters causes it to assume a default parameter value of one:

ESC [D ***is the same as*** **ESC [1 D**

The absence of a parameter in a control sequence that accepts two or more parameters causes it to assume a default parameter value of zero.

ESC [m ***is the same as*** **ESC [0 m**

The ANSI escape character sequence may be issued using the **PRINT CHR$(27),”[”,** statement as a prelude to the optional parameters and requisite command character.

Cursor Up n lines (ESC [n A)

Values (ASCII 27, 91, 48-57, 65 decimal / 1B, 5B, 30-39, 41 hex) Receipt of this character sequence causes the display to move the cursor up ‘n’ lines in the same column. The cursor will not move up past the first line in the display. For example, the cursor may be moved up one line by using the **PRINT CHR$(27),”[1A”,** statement.

Cursor Up n lines to first column (ESC [n F)

Values (ASCII 27, 91, 48-57, 70 decimal / 1B, 5B, 30-39, 46 hex) Receipt of this character sequence causes the display to move the cursor up ‘n’ lines and to the first column. The cursor will not move up past the first line in the display. For example, the cursor may be moved up two lines to the first column by using the **PRINT CHR$(27),"[2F",** statement.

Cursor Down n lines (ESC [n B)

Values (ASCII 27, 91, 48-57, 66 decimal / 1B, 5B, 30-39, 42 hex) Receipt of this character sequence causes the display to move the cursor down 'n' lines in the same column. The cursor will not move past the bottom line in the display and the display will not scroll up. For example, the cursor may be moved down three lines by using the **PRINT CHR$(27),"[3B",** statement.

Cursor Down n lines to first column (ESC [n E)

Values (ASCII 27, 91, 48-57, 69 decimal / 1B, 5B, 30-39, 45 hex) Receipt of this character sequence causes the display to move the cursor down 'n' lines and to the first column. The cursor will not move past the bottom line in the display and the display will not scroll up. For example, the cursor may be moved down one line to the first column by using the **PRINT CHR$(27),"[1E",** statement.

Cursor Right n characters (ESC [n C)

Values (ASCII 27, 91, 48-57, 67 decimal / 1B, 5B, 30-39, 43 hex) Receipt of this character sequence causes the display to move the cursor right 'n' characters on the same line. The cursor will not move past the end of the current line. For example, the cursor may be moved right four characters by using the **PRINT CHR$(27),"[4C",** statement.

Cursor Left n characters (ESC [n D)

Values (ASCII 27, 91, 48-57, 68 decimal / 1B, 5B, 30-39, 44 hex) Receipt of this character sequence causes the display to move the cursor left 'n' characters on the same line. The cursor will not move past the beginning of the current line. For example, the cursor may be moved left three characters by using the **PRINT CHR$(27),"[3D",** statement.

Move cursor to n (ESC [n G)

Values (ASCII 27, 91, 48-57, 71 decimal / 1B, 5B, 30-39, 47 hex) Receipt of this character sequence causes the display to move the cursor to column 'n' on the current line. The cursor will not move past the beginning or end of the current line. For example, the cursor may be moved to the beginning of the current line by using the **PRINT CHR$(27),"[G",** statement.

Move cursor to r, c (ESC [r ; c H)

Values (ASCII 27, 91, [[48-57], 59, [48-57]], 72 decimal / 1B, 5B, [[30-39], 3B, [30-39]], 48 hex) Receipt of this character sequence causes the display to move the cursor to row 'r', column 'c'. The value for 'r' ranges from 0 – bottom row, the value for 'c' ranges from 0 – rightmost column. The values for 'r' or 'c' will be limited to the selected screen resolution if they exceed it. For example, the cursor may be moved to the home position (0, 0) by using the **PRINT CHR$(27),"[H",** statement.

Erase all or part of display (ESC [n J)

Values (ASCII 27, 91, 48-50, 74 decimal / 1B, 5B, 30-32, 4A hex) Receipt of this character sequence causes part or all of the display to clear. If 'n' = 0, the display is cleared from the cursor position to the end. If 'n' = 1, the display is cleared from the beginning to the cursor position. If 'n' = 2 the entire display is cleared, and the cursor is moved to the upper left (0, 0). For example, the screen may be cleared by using the **PRINT CHR$(27),"[2J",** statement.

Erase all or part of line (ESC [n K)

Values (ASCII 27, 91, 48-50, 75 decimal / 1B, 5B, 30-32, 4B hex) Receipt of this character sequence causes part or all of the line that the cursor is on to clear. If 'n' = 0, the line is cleared from the cursor position to the end of the line. If 'n' = 1, the line is cleared from the beginning to the cursor position. If 'n' = 2 the entire line is cleared. The position of the cursor is not affected by this command.

Save cursor position (ESC [n s)

Values (ASCII 27, 91, 114 decimal / 1B, 5B, 73 hex) Receipt of this character sequence causes the display to save the current cursor position.

Restore cursor position (ESC [n u)

Values (ASCII 27, 91, 116 decimal / 1B, 5B, 75 hex) Receipt of this character sequence causes the display to restore the previously saved cursor position.

Select Graphic Rendition (ESC [a ; b ; … f m)

Values (ASCII 27, 91, … , 109 decimal / 1B, 5B, … , 6D hex) Receipt of this character sequence causes the display to select how subsequent text is rendered. Up to 10 parameters may be specified, separated by semicolons from the following table of attributes.

Parameter value	Attribute
0	Reset / normalize all attributes
7	Negative – reverse on/off colors
8	Conceal – no off color drawn
27	Positive – normal on/off colors (default)
30	On color = BLACK
31	On color = RED
32	On color = GREEN
33	On color = YELLOW
34	On color = BLUE
35	On color = MAGENTA
36	On color = CYAN
37	On color = WHITE (default)
40	Off color = BLACK (default)
41	Off color = RED
42	Off color = GREEN
43	Off color = YELLOW
44	Off color = BLUE
45	Off color = MAGENTA
46	Off color = CYAN
47	Off color = WHITE

PRINT#N, expr[, expr ...]

Prints one or more expressions to a previously opened file #***N*** (0 → 23).

READ var[,var ...]

Program mode only. Reads data from 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. A return without a prior **GOSUB** will generate a "Stack Error".

REM

Comment... the remainder of line is ignored. Used to document the operation of the program.

REN oldfile newfile

Renames oldfile to newfile. ***Oldfile*** and ***newfile*** may be constant strings or you can use string variables as the ***files*** by concatenating them to empty strings: **REN ""+O$, ""+N$**. In Direct mode the quotes are not required.

RESQ [start[-end][,new][,incr]]

Direct mode only. Resequences 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 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 program, or references to non-existent line numbers, the **RESQ** will error and stop. The original program should be **SAVE**d before attempting to resequence it.

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.

```
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] or RUN path

Direct mode only. Executes the program starting at the lowest or optional ***line*** number. Basic version v1.22 added the ability to **LOAD** and **RUN** a file directly at the lowest line number by typing **RUN** filename. If not present, the .BAS file extension on the filename at the end of the path is added.

SAVE [path]

Direct mode only. Saves the current program to a disk file on the Compact Flash card with the filename specified in ***path***, or to the filename in the previous **LOAD** statement or **RUN** command if not specified. If not present, the .BAS file extension on the filename at the end of the path is added.

SIGNAL @specialvar

Signal an event associated with Special variable.

STOP

Program mode only. Terminates the program and issues a **STOP** message. Closes all open files.

```
10 a=a+1
20 STOP
Ready
run
STOP in line 20
Ready
```

TYPE path

Displays the contents of a CF 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*** ""***+P$***. In Direct mode the quotes are not required.

A double escape sequence will stop the portion of the file display not already queued.

WAIT @specialvar

Execution pauses at this statement until the associated special variable has been signaled.

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
```

LCDx Statements

The following LCD commands operate on an ACS LCD display connected to the CFSound-3 serial port that is configured for SOH/ETX protocol. The commands generate and send formatted strings of ASCII characters that the connected LCD display interprets to perform the operation. The proper command formatting for the attached display is controlled by the current value of the **@LCDTYPE** special variable. The **@LCDADDRESS** special variable may also optionally be used to selectively address multiple displays by inserting the display address into the generated commands. See the ACS-LCD-128x64 or ACS-LCD-320x240 Display User Manuals for additional information about these command's arguments.

LCDPRINT row[s], col, font, type, justify, expr (@LCDTYPE=0)

LCDPRINT rowstart, col, font, type, justify, expr (@LCDTYPE=1)

Displays an **expr** on an ACS LCD display connected to the CFSound-3 serial port.

LCDUNPRINT row[s], col, font, type, justify, expr (@LCDTYPE=0)

LCDUNPRINT rowstart, col, font, type, justify, expr (@LCDTYPE=1)

Un-displays an **expr** on an ACS LCD display connected to the CFSound-3 serial port.

LCDCLEAR row[s], colstart, colend (@LCDTYPE=0)

LCDCLEAR rowstart, rowend, colstart, colend (@LCDTYPE=1)

Clears an area of the screen on an an ACS LCD display connected to the CFSound-3 serial port.

LCDGRAPHIC row[s], col, data (@LCDTYPE=0 only)

Displays a byte of data on an ACS LCD display connected to the CFSound-3 serial port. This command is not supported on the ACS LCD-320x240 display.

LCDLINE startx, starty, endx, endy, color

Displays a line on an ACS LCD display connected to the CFSound-3 serial port.

LCDBOX corner1x, corner1y, corner2x, corner2y, color

Displays a box on an an ACS LCD display connected to the CFSound-3 serial port.

LCDPIXEL x, y, color

Displays a pixel on an ACS LCD display connected to the CFSound-3 serial port..

LCDCIRCLE x, y, radius, color

Draws a circle on an ACS LCD display connected to the CFSound-3 serial port.

LCDTONE frequency, duration

Produces a tone on an ACS LCD display connected to the CFSound-3 serial port.

LCDSAVE page

Saves a screen on an ACS LCD display connected to the CFSound-3 serial port.

LCDRESTORE page

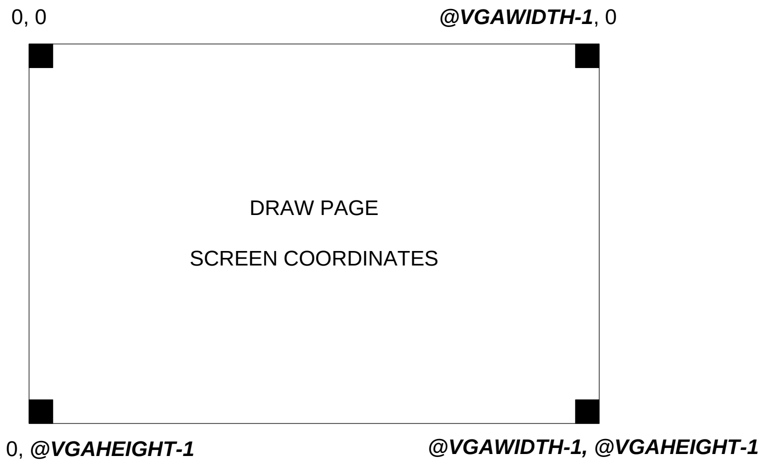
Restores a screen on an ACS LCD display connected to the CFSound-3 serial port.

LCDBITMAP startrow, col, ”path”

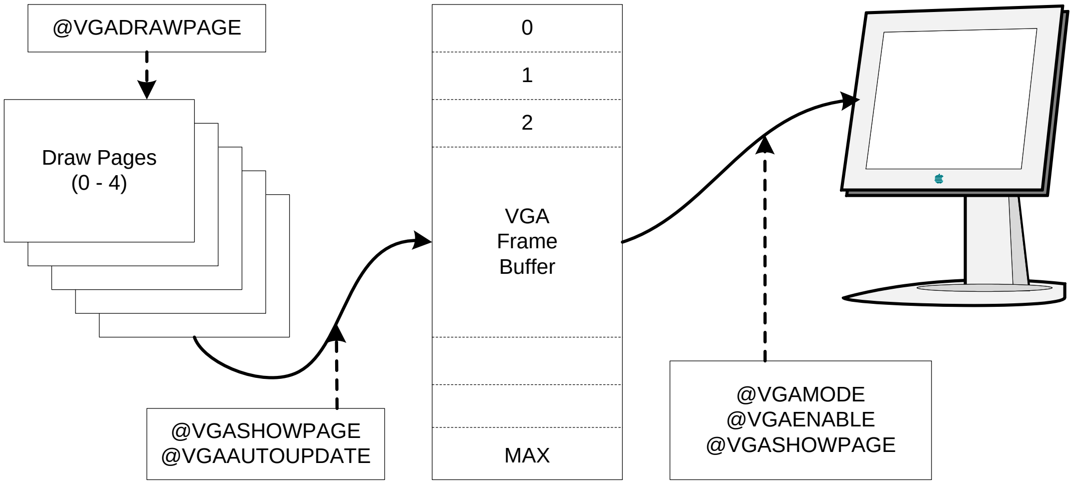
Displays a Windows .BMP bitmap file named ***path*** starting at ***startrow***(0 → 7) and ***col***umn on an ACS LCD display connected to the CFSound-3 serial port. Only mono, 16-color and 256 color bitmaps are supported. Any pixel whose color is not R=255, G=255, B=255 (white) will be displayed as an on pixel (black). Issues multiple LCD display Horizontal Load commands to image the bitmap on the display.

VGAx Statements

The following VGA statements operate on the optional Video Graphics Adaptor Adaptor installed in the CFSound-3. Attempting to execute these VGA statements without a VGA module installed results in a "**No VGA module** error". The VGAx statements affect the current ***@VGADRAWPAGE*** and utilize screen coordinates that start from x=0, y=0 in the upper left corner to x=***@VGAWIDTH-1***, y=***@VGAHEIGHT-1*** in the lower right corner:



There are 5 drawing pages selected via ***@VGADRAWPAGE*** that can be updated to the VGA frame buffer page via the ***@VGAUPDATEPAGE / @VGAAUTOUPDATE*** mechanism. The attached LCD / video monitor displays the VGA frame buffer contents via ***@VGASHOWPAGE***. See the ***@VGAx*** specialvars description above for more information.



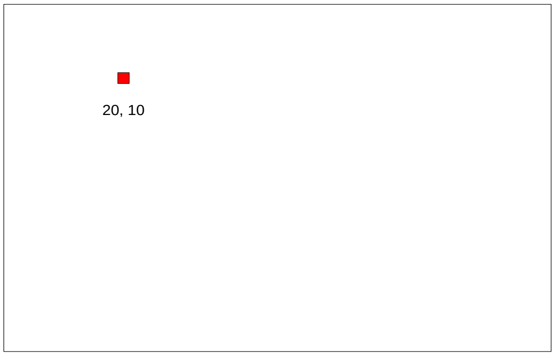
VGACLIPRECT topLeftX, topLeftY, bottomRightX, bottomRightY

Sets the clipping rectangle to the arguments provided. The arguments are internally sorted left to right, top to bottom, and force limited to the current screen resolution. The screen operations of VGAx statements are 'clipped' to the rectangular region specified – if the affected pixel coordinates are outside of the rectangle they are unchanged. Setting ***@VGAMODE*** resets the clipping rectangle to the entire screen area.

VGAPIXEL x, y, color

Sets the VGA pixel at coordinate ***x***, ***y*** to ***color*** if the ***x***, ***y*** location is within the current clipping rectangle.

```
5 REM Draw a red pixel
10 VGAPIXEL 20, 10, RGB(255, 0, 0)
```



VGAFILL color

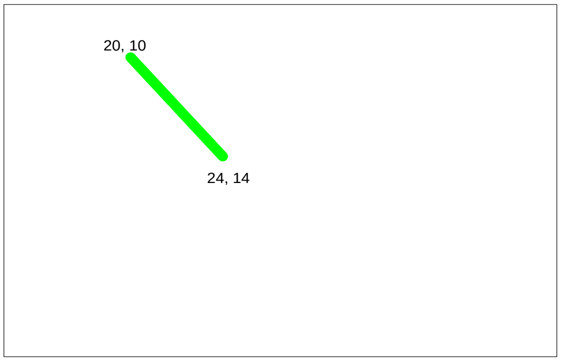
Fills the VGA screen within the clipping rectangle with ***color***.

```
5 REM Clear VGA screen to black
10 VGAFILL 0
```

VGALINE startX, startY, endX, endY, color

Draws a line consisting of ***color*** pixels from ***startX***, ***startY*** to ***endX***, ***endY*** coordinates – clipped to within the current clipping rectangle.

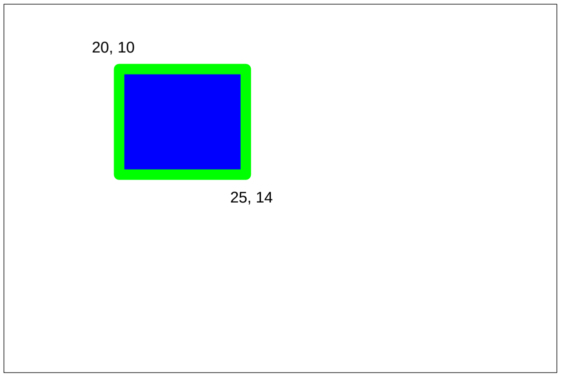
```
10 VGALINE 20, 10, 24, 14, RGB(0, 255, 0)
```



VGABOX corner1X, corner1Y, corner2X, corner2Y, color [, fillcolor]

Draws a rectangular box from ***corner1X***, ***corner1Y*** to ***corner2X***, ***corner2Y*** consisting of 4 lines of ***color*** pixels, optionally filled with ***fillcolor*** pixels – clipped to within the current clipping rectangle.

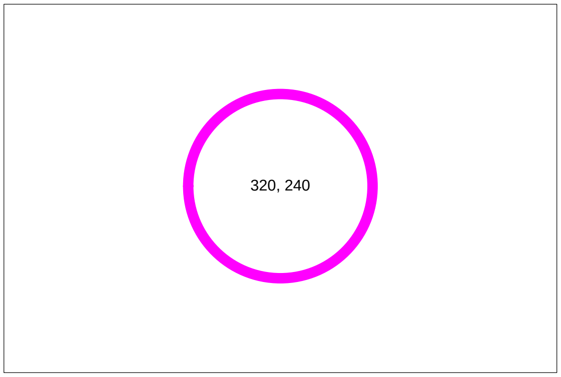
```
5 REM Draw green box filled with blue
10 VGABOX 20, 10, 25, 14, RGB(0, 255, 0), RGB(0, 0, 255)
```



VGACIRCLE centerX, centerY, radius, color

Draws a circle using ***color*** pixels centered at coordinates ***centerX***, ***centerY*** of ***radius*** – clipped to within the current clipping rectangle.

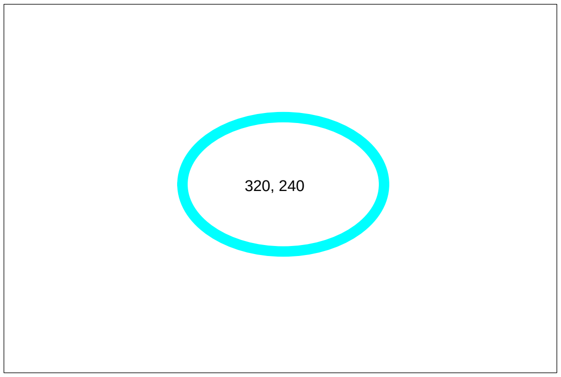
```
5 REM Draw magenta circle
10 VGACIRCLE 320, 240, 100, RGB(255, 0, 255)
```



VGAELLIPSE centerX, centerY, width, height, color [, fillcolor]

Draws an ellipse using ***color*** pixels of ***width*** and ***height*** centered at coordinates ***centerX***, ***centerY***, optionally filled with ***fillcolor*** pixels – clipped to within the current clipping rectangle.

```
5 REM Draw cyan ellipse
10 VGAELLIPSE 320, 240, 150, 75, RGB(0, 255, 255)
```

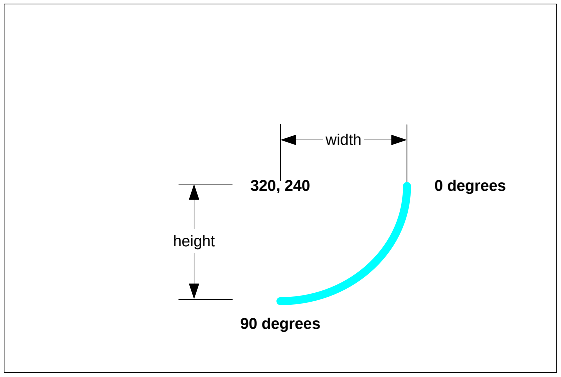


VGAARC centerX, centerY, width, height, startDegrees, endDegrees, color [, style]

Draws an arc using ***color*** pixels of ***width*** and ***height***, centered at coordinates ***centerX***, ***centerY***, starting at ***startDegrees*** through ***endDegrees***, optionally styled with one or more ***style*** bits – clipped to within the current clipping rectangle.

The starting and ending degree values should be between 0 and 359 degrees.

```
5 REM Draw cyan arc
10 VGAARC 320,240, 200, 200, 0, 90, RGB(0,255,255)
```



Style	**Name**	**Description**
0	Arc	Draws filled arc (pie segment)
1	Chord	Draws straight line between start and end angles
2	No Fill	Don’t fill the arc (empty pie segment)
4	Edged	Draw arc edges (outlined pie segment with No Fill)

Style *bits may be combined*

VGATEXT x, y, font, style, justify, onColor, offColor, expr

Draws the value of ***expr*** as characters using the ***font***, ***style***, ***justify***, ***onColor*** and ***offColor*** arguments – clipped to within the current clipping rectangle.

Font	**Description**
0	Small – 5 x 7 proportional
1	Medium – 9 x 16 proportional
2	Micro – 4 x 5 nominal uppercase only
3	Giant Numbers – 30 x 56 numbers only
4	Fixed – 5 x 7 fixed
5	Large – 18 x 32 proportional, doubled version of Medium

Style	**Description**
0	Normal
1	Inverted
2	No offColor pixels drawn

Style *bits may be combined*

Justify	**Description**
0	Left – text aligned to x=0, y
1	Centered – text aligned to @VGAWIDTH/2, y
2	Right – text aligned to @VGAWIDTH, y
3	Absolute – text left aligned to x, y
4	Right Absolute – text right aligned from x, y
5	Center Absolute – text centered on x, y

The x and y coordinates specified for the justified text refer to the top edge and left or right corners of the generated text display.

VGAPOLYGON coordsX, coordsY, color [, fillcolor]

Draws a polygon using ***color*** lines whose vertex coordinates are passed as numeric arrays ***coordsX***, ***coordsY***, optionally filled with ***fillcolor*** pixels. The coordinate arrays must be identically ***DIM***ensioned to be greater than or equal to 3 points (DIM x(2),y(2) = triangle coordinates x(0) y(0), x(1) y(1), x(2) y(2).

```
5 REM Draw random triangles
7 VGAFILL 0
10 DIM x(2),y(2)
20 x(0)=RND(@VGAWIDTH):x(1)=RND(@VGAWIDTH):x(2)=RND(@VGAWIDTH)
30 y(0)=RND(@VGAHEIGHT):y(1)=RND(@VGAHEIGHT):y(2)=RND(@VGAHEIGHT)
40 VGAPOLYGON x, y, RGB(RND(256), RND(256), RND(256)),RGB(RND(256),RND(256),RND(256))
50 GOTO 20
```

VGABITMAP upperX, upperY, “path”

Draws the Windows image .BMP or .JPG format file at ***path*** to the screen coordinate ***upperX***, ***upperY*** – clipped to within the current clipping rectangle. Windows bitmap files of 1BPP, 4BPP, 8BPP or 24BPP are supported. Windows JPEG files that are sequential, sRGB YUV420 encoded are supported. The entire file has to be read into the CFSound-III memory for processing and rendering so the file size is limited to approximately 2MB. Larger files load and display slower.

```
dir shuttle.bmp
SHUTTLE.BMP       2,085,942 A      11-12-2009 15:27:40
---------------------------
                          1 files
                          0 directories
Ready
@vgamode=3
Ready
vgabitmap 0,0,"shuttle.bmp"
Ready
```

VGABLIT destPage, destUpperX, destUpperY, width, height, srcPage, srcUpperX, srcUpperY, opcode

Transfers pixels from a source drawing page and rectangle to a destination drawing page and rectangle, altering the pixels during the transfer according to the opcode value.

The pixel destination is specified by the destination drawing page ***destPage*** and destination rectangle ***destUpperX***, ***destUpperY***, ***width*** and ***height***. The pixel source is specified by the source drawing page ***srcPage*** and source rectangle ***srcUpperX***, ***srcUppery***, ***width*** and ***height***.

During the transfer the pixels are altered according to the specified ***opcode***:

opcode	Name	Description
0	BLACKNESS	Fills the destination rectangle with 0x0000 value pixels (BLACK)
1	DEST_INVERT	Inverts the destination rectangle
2	NOT_SRC_COPY	Copies the inverted source rectangle to the destination
3	NOT_SRC_ERASE	Combines the colors of the source and destination rectangles using the Boolean OR operator and then inverts the resultant color
4	SRC_AND	Combines the colors of the source and destination rectangles by using the Boolean AND operator
5	SRC_COPY	Copies the source rectangle directly to the destination rectangle
6	SRC_ERASE	Combines the inverted colors of the destination rectangle with the colors of the source rectangle by using the Boolean AND operator
7	SRC_INVERT	Combines the colors of the source and destination rectangles by using the Boolean XOR operator
8	SRC_PAINT	Combines the colors of the source and destination rectangles by using the Boolean OR operator
9	WHITENESS	Fills the destination rectangle with 0xFFFF value pixels (WHITE)
10	SRC_BLACK_MASK	Copies non-black (≠ 0x0000) pixels from the source rectangle to the destination.

Operators

ACS 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
< <= > >=	LT, LE, GT, GE (numeric or string)	3
& | ^	Bitwise AND, OR, Exclusive OR	2
AND OR	Logical AND, OR	1

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 16 ‘bit’ positions in the integer are numbered from right to left starting with 0 (the **L**east **S**ignificant **B**it) up to 15 (the **M**ost **S**ignificant **B**it) or sign bit:

MSB															LSB
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
16-bit Integer value															

Thus the value 1234 in binary bit form is:

Decimal	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1234	0	0	0	0	0	1	0	0	1	1	0	1	0	0	1	0

And the value -1234 in binary bit form is:

Decimal	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
-1234	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 bit form is:

Decimal	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
~1234	1	1	1	1	1	0	1	1	0	0	1	0	1	1	0	1

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

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

Decimal	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1234	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	1	1	1	1	1	1	1	1
= 210	0	0	0	0	0	0	0	0	1	1	0	1	0	0	1	0

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

Decimal	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1234	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	1	1	1	1	1	1	1	1
= 1279	0	0	0	0	0	1	0	0	1	1	1	1	1	1	1	1

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 ^’ing a value with the bits to toggle. So the value 1234 ^ with 255 is 1069:

Decimal	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1234	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	1	1	1	1	1	1	1	1
= 1069	0	0	0	0	0	1	0	0	0	0	1	0	1	1	0	1

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.

The bitwise << shifts the bits towards the left from LSB towards MSB, filling in the vacated LSB positions with zero bits. Thus 1234 << 2 = 4936:

Decimal	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1234	0	0	0	0	0	1	0	0	1	1	0	1	0	0	1	0	
<< 2	←	←	←	←	←	←	←	←	←	←	←	←	←	←	←	←	← 0, 0
= 4936	0	0	0	1	0	0	1	1	0	1	0	0	1	0	0	0	

The bitwise >> shifts the bits towards the right from MSB towards LSB, filling in the vacated MSB positions with copies of the sign bit 15. Thus1234 >> 2 = 308:

Decimal		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1234		0	0	0	0	0	1	0	0	1	1	0	1	0	0	1	0
>> 2	0, 0 →	→	→	→	→	→	→	→	→	→	→	→	→	→	→	→	→
= 308		0	0	0	0	0	0	0	1	0	0	1	1	0	1	0	0

Since the bits filling into the vacated MSB positions are copies of the sign bit, bit 15 then -1234 >> 2 = -308:

Decimal		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
-1234		1	1	1	1	1	0	1	1	0	0	1	0	1	1	1	0
>> 2	1, 1 →	→	→	→	→	→	→	→	→	→	→	→	→	→	→	→	→
= -308		1	1	1	1	1	1	1	0	1	1	0	0	1	0	1	1

Expressions

In ACS Basic expressions consist of one or more variables, constants, functions or special 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

ACS Basic provides several 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)

Returns the numeric ASCII value of the character argument.

ABS(expr)

Returns the absolute value of the numeric argument.

CHR$(expr)

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

COS(degrees)

Returns 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..

ERR( )

Returns the last error number.

ERR$( )

Returns the string representation of the last error number.

FIND(var$, searchvar$)

Returns the zero based position of string **searchvar**iable in string **var**iable or -1 if the searchvariable was not found.

FMT$(fmt$, expr[$])

Returns a formatted ASCII string of **expr**ession using format specification **fmt$**. A format specification consists of [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 *type* character, control other aspects of the formatting, as follows:

Type	Required character that determines whether the associated *argument* is interpreted as a character, a string, or a number:	
	c	character
	d	signed decimal integer
	i	signed decimal integer
	u	unsigned decimal integer
	s	string
	o	unsigned octal integer
	x	unsigned hexadecimal integer
	X	unsigned HEXADECIMAL integer
Flags	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.	
	-	left align the result in the given field width
	+	prefix the output with a sign (+/-) if the type is signed
	0	if Width is prefixed with 0, zeros 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.
	blank(‘ ’)	prefix the output with a blank if the result is signed and positive; the blank is ignored if both the blank and + flags appear
	#	when used with o, x or X format, prefix any nonzero output value with 0, 0x or 0X respectively, otherwise ignored
Width	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, zeros 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).	
Precision	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.	
	c	**Precision** has no effect
	d,i,u,o, x,X	**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**
	s	**Precision** specifies the maximum number of characters to be output. Characters in excess of **Precision** are not output

GETCH(expr)

If ***expr*** evaluates to zero, **GETCH(0)** returns the numeric value of: the next available serial character (if **@MSGENABLE**=0) or the next PS/2 ASCII key character (if VGA module installed), or it returns a zero 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 (if **@MSGENABLE**=1) or PS/2 ASCII key character (if VGA module installed) and then returns its numeric value.

INSERT$(var$, start, var2$)

Returns a string ***var***iable with the contents of ***var***iable***2*** inserted 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$(var$, len)

Returns a string containing the leftmost **len**gth characters of string **var**iable.

LEN(var$)

Returns the length (number of characters) of string **var**iable.

MID$(var$, start, len)

Returns a string consisting of **len**gth number of characters of string **var**iable from zero based ***start*** character position.

MULDIV(number, multiplier, divisor)

Returns a 16 bit result of ((number * multiplier) / divisor) where number, multiplier and divisor are 32-bit internally. Useful for calculating percentages, etc., where the normal multiply would overflow a signed 16-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)

Returns a 16 bit result of ((number * multiplier) % divisor) where number, multiplier and divisor are 32-bit internally. Useful for calculating remainders of percentages, etc., where the normal multiply would overflow a signed 16-bit number.

RGB(red, green, blue)

Returns a 16 bit color value for use with the VGAx statements where red, green and blue are packed into a RGB565 format for the VGA – 5 bits of red, 6 bits of green and 5 bits of blue – 65536 colors. The red, green and blue arguments are limited to a range of 0 → 255, low-order bits are truncated.

RIGHT$(var$, len)

Returns a string containing the rightmost **len**gth characters of string **var**iable.

REPLACE$(var$, start, var2$)

Returns a string ***var***iable with the contents of ***var***iable***2*** overwritten 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
```

RND(expr)

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

SIN(degrees)

Returns 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..

STR$(expr)

Returns a string representation of the numeric argument.

VAL(expr$)

Returns the numeric value of the string argument representation of a number.

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:

Error #	Error Message	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
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 "	ACS Basic running without CF card
17	“Unknown @var error in line dd “	Unknown special 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 CF card error in line dd "	Statement requiring Compact Flash 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	“LCDx arguments Out of Range error in line dd”	One or more argument to a LCDx 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 recordlength 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 recordlength 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 error in line dd”	@DMX--- specialvar access attempted with no DMX I/O module present
34	“DMX Channel # Out of Range error in line dd”	@DMXDATA(x) access where x >= 511
35	“DMX Analog # Out of Range error in line dd”	@DMXANALOG(x) access where x >= 7
36	“DMX Analog # Read Only error in line dd”	Attempt to set @DMXANALOG(x)
37	“Unknown Command error in line dd”	ACS Basic doesn’t recognize the command
38	“Can’t use @VAR in line dd”	Illegal use of specialvar 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	“No VGA module error in line dd”	@VGA specialvar access or VGAx statement attempted with no VGA module present
41	“VGAMODE Out of Range error in line dd”	Attempt to set @VGAMODE to unsupported value
42	“VGADRAWPAGE Out of Range error in line dd”	Attempt to set @VGADRAWPAGE to unsupported value
43	“VGAUPDATEPAGE Out of Range error in line dd”	Attempt to set @VGAUPDATEPAGE to unsupported value
44	“VGASHOWPAGE Out of Range error in line dd”	Attempt to set @VGASHOWPAGE to unsupported value
45	“VGAPOLYGON argument error”	Problem with an argument to the VGAPOLYGON statement
46	“VGABLIT argument error”	Problem with an argument to the VGABLIT statement
47	“RGB” argument error”	Problem with an argument to the RGB( ) function

48	“Unsupported bitmap file”	Problem with filename argument to the VGABITMAP statement
49	Reserved	reserved
50	“FREAD record # Out of Range error in line dd”	Attempt to FREAD
49 - 65535	“x error in line dd”	ERROR x statement

Examples

Here are a few sample programs that illustrate the various language features and what can be done with some simple lines of code.

Setting the Real Time Clock

Set the CFSound-3'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.

```
5 REM set the cfsound rtc
7 INPUT "set the RTC first (y/n):", s$
8 IF s$="y" THEN 20
9 IF s$="Y" THEN 20
10 GOTO 110
20 INPUT "month (1-12):",m
25 IF m <1 THEN 20
27 IF m >12 THEN 20
30 @MONTH=m
35 INPUT "date (1-31):",d
40 IF d <1 THEN 35
42 IF d >31 THEN 35
45 @DATE=d
50 INPUT "year (00-99):", y
52 IF y <0 THEN 50
53 IF y >99 THEN 50
60 @YEAR=y
65 INPUT "hour (00-23):",h
70 IF h <0 THEN 65
72 IF h >23 THEN 65
75 @HOUR=h
80 INPUT "minute (00-59):",m
85 IF m <0 THEN 80
87 IF m >59 THEN 80
90 @MINUTE=m
95 INPUT "second (00-59):",s
100 IF s <0 THEN 95
102 IF s >59 THEN 95
105 @SECOND=s
110 ONEVENT @SECOND,GOSUB 1000
120 GOTO 120
1000 PRINT CHR$(13),
1002 ON @DOW,GOSUB 2000,2001,2002,2003,2004,2005,2006
1005 ON @MONTH,GOSUB 1200,1201,1202,1203,1204,1205,1206,1207,1208,1209,1210,1211,1212
1010 PRINT d$+" "+m$+FMT$(" %2d",@DATE)+FMT$(", %02d",@YEAR),
1015 PRINT FMT$(" %2d", @HOUR)+":"+FMT$("%02d",@MINUTE)+":"+FMT$("%02d",@SECOND),
1020 RETURN
1200 m$="???":RETURN
1201 m$="JAN":RETURN
1202 m$="FEB":RETURN
1203 m$="MAR":RETURN
1204 m$="APR":RETURN
1205 m$="MAY":RETURN
1206 m$="JUN":RETURN
1207 m$="JUL":RETURN
1208 m$="AUG":RETURN
1209 m$="SEP":RETURN
1210 m$="OCT":RETURN
1211 m$="NOV":RETURN
1212 m$="DEC":RETURN
2000 d$="SUN":RETURN
2001 d$="MON":RETURN
2002 d$="TUE":RETURN
2003 d$="WED":RETURN
2004 d$="THU":RETURN
2005 d$="FRI":RETURN
2006 d$="SAT":RETURN
```

Two Sound Sequences

The CFSound-3 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 ACS Basic program that will allow two different sequences each controlled by a built-in contact.

Remember that the @CLOSURE(x) special 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$ special 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$ special 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
```

Different Sounds for Contact Closure / Opening

The CFSound-3 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 ACS Basic program is required.

Remember that the @CLOSURE(x) special 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 specialvars. When one is found active (non-zero) the desired sound file is played, then the triggering specialvar 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-3 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 ACS Basic program is required.

Remember that the @CLOSURE(x) special 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 specialvar. In line 10, if there is a closure AND there is a sound currently playing, the sound is stopped, then the triggering specialvar 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 specialvar 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
```

Activating Multiple Output Contacts for a Sound

The CFSound-3 can activate a single output contact when a sound is played in CFSound mode. Here's a simple ACS Basic program that will allow multiple output contacts to be controlled when a sound plays.

Remember that the @CLOSURE(x) special variable index argument x is zero based, so for Contact #25 the x value would be 24, etc.. This example assumes that the CFSound-III 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
```

Control from a Serial Port

The CFSound-3 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$ special 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 CF card.

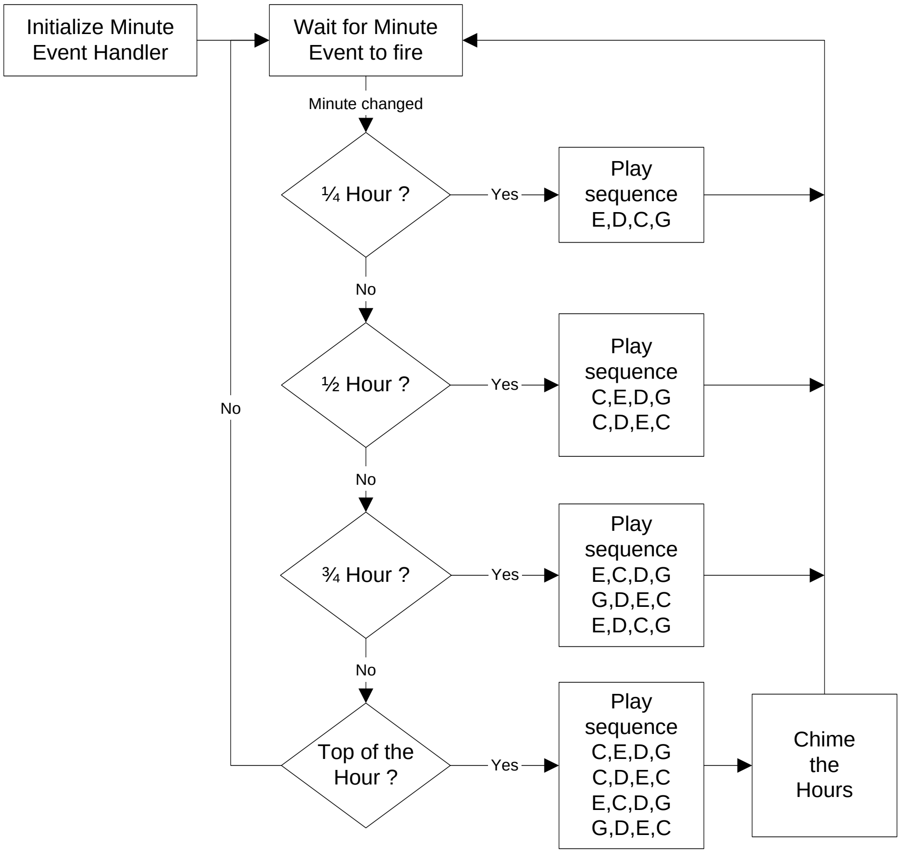
Here's how it works. An event handler is setup in line 20 – when a character string delimited by the @SOM and @EOM characters is received, control transfers to line 50 with the @MSG$ variable holding the inner contents of the string. Line 60 copies the string and resets the @MSG$ variable for receipt 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$=@MSG$:@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
```

Westminster Chimes

Turn the CFSound-3 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:


Initialize Minute Event Handler
Wait for Minute Event to fire
Minute changed
¼ Hour ?
Yes
Play sequence E,D,C,G
No
½ Hour ?
Yes
Play sequence C,E,D,G C,D,E,C
No
No
¾ Hour ?
Yes
Play sequence E,C,D,G G,D,E,C E,D,C,G
No
Top of the Hour ?
Yes
Play sequence C,E,D,G C,D,E,C E,C,D,G G,D,E,C
Chime the Hours

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 special 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.

```
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
```

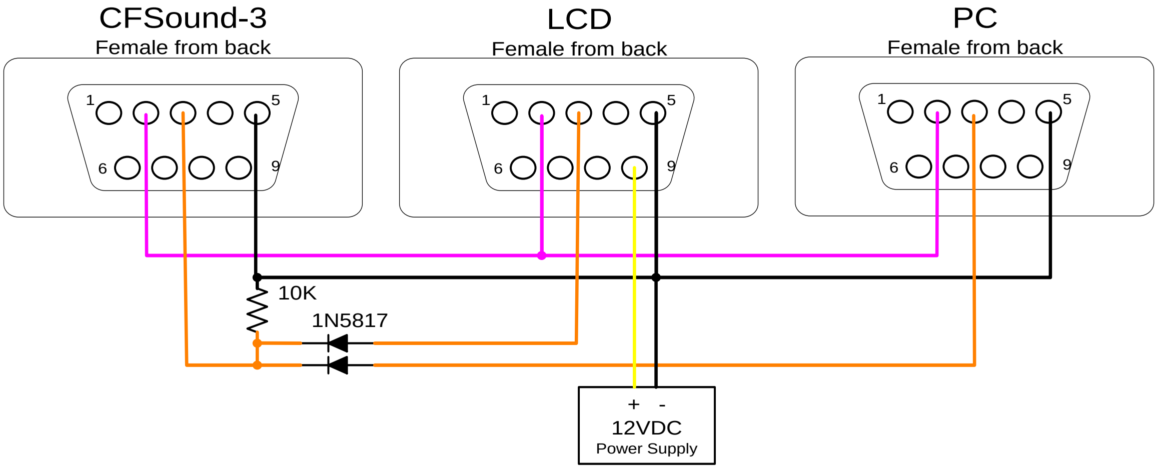
Renaming the program to CFSOUND.BAS and placing it along with the requisite sound files onto the CF card will turn your CFSound-3 into a Big Ben clock. Here's the entire program:

```
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$="HOURS.WAV"
220 NEXT c
225 RETURN
300 REM play quarter past sequence
305 @SOUND$="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
```

Jukebox with Display

Turn the CFSound-3 into a jukebox with display using this short program. The hardware consists of a CFSound-3 and the ACS-LCD-128x64 display with membrane switch, wired together with a serial cable. If a special cable is constructed, the PC can also be connected allowing for interactive software development. The following cable allows both the PC and the LCD to 'talk' to the CFSound-3 by using two diodes and a resistor for implement a wired-or of the LCD and PC TxD signals. This allows the PC to communicate with the CFSound-3 via Window's Hyperterminal accessory, and the LCD keystrokes to be sent to the CFSound-3 to interact with the Basic program:



The program captures a directory listing of the .WAV files present on the flash card and displays this listing on the LCD display. The Up and Down arrows on the membrane switch scroll the 'selection', shown in inverse font. Pressing the Enter key between the arrows plays the current selection. Several subroutines are used to simplify the main program logic.

Here's how it works. The program starts by clearing the LCD display and installing an event handler for the @MSG$ special variable:

```
1 REM
2 REM LCD Jukebox Demo
3 REM
10 REM clear display, install @msg$ handler
15 LCDCLEAR 255,0,127
25 ONEVENT @MSG$,GOSUB 8005
```

The ACS LCD Display frames its sent messages in a SOH / ETX character pair, which is the default value of the @SOM and @EOM special variables. When the program is running and not processing a Basic INPUT statement, characters received on the CFSound's serial port are processed looking for an @SOM / @EOM delimited message string. When such a message is detected, the @MSG$ variable receives the content of the message, and the @MSG$ event handler is signaled. This causes program execution to GOSUB to line 8005 after the current statement is finished.

Next the program generates a file that contains the directory of .WAV files present on the flash card, and then displays this list. The program then falls into an idle loop, waiting for LCD membrane switch key press messages to process:

```
30 REM generate and display wav file list
35 GOSUB 6005:GOSUB 7005
100 GOTO 100
```

The subroutine to generate the .WAV directory file opens a text file WAVES.TXT for destructive writing, directing the output of the DIR command into that file. The file is then re-opened for reading, and the number of lines in the file are counted, subtracting 3 lines for the summary lines at the bottom of the DIR command:

```
6000 REM
6001 REM Generate list of .WAV files
6002 REM
6005 OPEN #0,"WAVES.TXT","w"
6010 DIR #0,*.wav
6015 CLOSE #0
6020 n=0:OPEN #0,"WAVES.TXT","r"
6025 INPUT #0,l$
6030 LIF LEN(l$) >0 THEN n=n+1:GOTO 6025
6035 CLOSE #0
6040 IF n >3 THEN n=n-3
6045 RETURN
```

The subroutine to display the WAVES.TXT file on the LCD display clears the screen, then skips over lines in the file that have 'scrolled off' the top of the display contained in the b variable. It then 'prints' the next 8 lines which is all that the LCD can show at a time. The LCD print subroutine t$ variable receives the file name from each line, discarding the following file size information. The t variable receives the desired display print type, 1=normal, 2=inverse depending upon whether or not the screen row index variable i matches the current screen selection variable s, and the currently selected .WAV filename is saved in variable s$. Finally the LCDPRINT rows variable r receives the computed row bit number and the line is printed on the LCD display:

```
7000 REM
7001 REM Display file list on LCD
7002 REM
7005 LCDCLEAR 255,0,127:OPEN #0,"WAVES.TXT","r"
7010 FOR i=0 TO b:INPUT #0,l$:NEXT i
7015 FOR i=0 TO 7
7020 INPUT #0,l$:t$=LEFT$(l$,FIND(l$," "))
7025 t=1:LIF i=s THEN t=2:s$=t$
7030 r=1<<i:LCDPRINT r,0,4,t,0,t$
7035 NEXT i
7040 CLOSE #0
7045 RETURN
```

The @MSG$ event handler subroutine is called whenever a delimited message string has been received from the LCD display. The handler captures the received @MSG$ into the k$ variable, freeing the special variable to receive another message. The received message is then parsed to see if a LCD Reset message or Keypress message has been received. Display reset messages simply refresh the display. Keypress messages are further decoded to determine which key was pressed on the display and are dispatched to corresponding code fragments for processing.

Currently, only 3 keys are handled; the Up and Down arrows, and the Enter key between them. The Down arrow key advances the selection variable s to the bottom of the display, then advances the display skip lines variable b as required, redrawing the display. The Up arrow key decrements the selection variable s to the top of the display, then decrements the display skip lines variable b as required, redrawing the display. The Enter key stops any currently queued sound that is playing and starts the selected sound playing:

```
8000 REM
8001 REM LCD received message handler
8002 REM
8005 k$=@MSG$
8010 IF MID$(K$,0,1) ="K" THEN 8050
8015 IF MID$(K$,0,1) ="R" THEN 8025
8020 RETURN
8024 REM R command
8025 GOSUB 7005
8030 RETURN
8049 REM K commands
8050 k=ASC(MID$(K$,2,1)) -ASC("0")
8055 ON k,GOTO 8100,8200,8300,8400,8500,8600,8700
8060 RETURN
8099 REM K30 - left most key
8100 RETURN
```

```
8199 REM K31 - mid left key
8200 RETURN
8299 REM K32 - mid right key
8300 RETURN
8399 REM K33 - right most key
8400 RETURN
8499 REM K34 - down arrow key
8500 LIF ((s<n) &(s<7)) THEN s=s+1:GOTO 8510
8505 IF ((n>s)&((b+s)<(n-2))) THEN b=b+1
8510 GOSUB 7005
8515 RETURN
8599 REM K35 - up arrow key
8600 LIF S>0 THEN s=s-1:GOTO 8610
8605 IF b>0 THEN b=b-1
8610 GOSUB 7005
8615 RETURN
8699 REM K36 - enter key
8700 @SOUND$="":@SOUND$=s$:RETURN
```

Running the program while connected to the PC with Hyperterminal using the above cable produces the following text. Notice the ACS-LCD-128x64 commands delimited with the ASCII SOH (01) / ETX (03) characters:

```
run
␁CFF007F␃␁CFF007F␃␁P0100420TWO.WAV␃␁P0200410THREE.WAV␃␁P0400410ONE.WAV␃␁P0800410FIVE.WA
V␃␁P1000410SIX.WAV␃␁P2000410SEVEN.WAV␃␁P4000410EIGHT.WAV␃␁P8000410NINE.WAV␃
```

Renaming the program to CFSOUND.BAS and placing it along with the requisite sound files onto the CF card will turn your CFSound-3 into a Jukebox with LCD display. Here’s the entire program:

```
1 REM
2 REM LCD Jukebox Demo
3 REM
10 REM clear display, install @msg$ handler
15 LCDCLEAR 255,0,127
25 ONEVENT @MSG$,GOSUB 8005
30 REM generate and display wav file list
35 GOSUB 6005:GOSUB 7005
100 GOTO 100
6000 REM
6001 REM Generate list of .WAV files
6002 REM
6005 OPEN #0,"WAVES.TXT","w"
6010 DIR #0,*.wav
6015 CLOSE #0
6020 n=0:OPEN #0,"WAVES.TXT","r"
6025 INPUT #0,l$
6030 LIF LEN(l$) >0 THEN n=n+1:GOTO 6025
6035 CLOSE #0
6040 IF n >3 THEN n=n-3
6045 RETURN
7000 REM
7001 REM Display file list on LCD
7002 REM
7005 LCDCLEAR 255,0,127:OPEN #0,"WAVES.TXT","r"
7010 FOR i=0 TO b:INPUT #0,l$:NEXT i
7015 FOR i=0 TO 7
7020 INPUT #0,l$:t$=LEFT$(l$,FIND(l$," "))
7025 t=1:LIF i=s THEN t=2:s$=t$
7030 r=1<<i:LCDPRINT r,0,4,t,0,t$
7035 NEXT i
7040 CLOSE #0
7045 RETURN
8000 REM
8001 REM LCD received message handler
8002 REM
8005 k$=@MSG$
8010 IF MID$(K$,0,1) ="K" THEN 8050
8015 IF MID$(K$,0,1) ="R" THEN 8025
8020 RETURN
8024 REM R command
8025 GOSUB 7005
8030 RETURN
8049 REM K commands
```

```
8050 k=ASC(MID$(K$,2,1)) -ASC("0")
8055 ON k,GOTO 8100,8200,8300,8400,8500,8600,8700
8060 RETURN
8099 REM K30 - left most key
8100 RETURN
8199 REM K31 - mid left key
8200 RETURN
8299 REM K32 - mid right key
8300 RETURN
8399 REM K33 - right most key
8400 RETURN
8499 REM K34 - down arrow key
8500 LIF ((s<n) &(s<7)) THEN s=s+1:GOTO 8510
8505 IF ((n>s)&((b+s)<(n-2))) THEN b=b+1
8510 GOSUB 7005
8515 RETURN
8599 REM K35 - up arrow key
8600 LIF S>0 THEN s=s-1:GOTO 8610
8605 IF b>0 THEN b=b-1
8610 GOSUB 7005
8615 RETURN
8699 REM K36 - enter key
8700 @SOUND$="":@SOUND$=s$: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
```

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 CF card. Then when the program stops running, the CF 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
```

DMX Control Synchronized to Sound

This example plays an audio file for an exhibit at the Alamo Museum in San Antonio, Texas. The CFSound-III with DMX module synchronizes the fading up/down of the house lights and 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 ends.

The @SOUNDFRAMEPRESCALER specialvar is set to 50. This causes a @SOUNDFRAMESYNC event to fire every second while the sound is playing. The subroutine at line 1000 is executed every time this happens and uses the one second sound frame number to start DMX channels fading up/down to make the show happen.

```
10 REM Program to fade DMX controlled lamps up and down during the playout of audio file
15 REM Start DMX
20 @DMXMASTER=1:@SOUNDFRAMEPRESCALER=50
25 REM Stop Show
30 @SOUND$="":ONEVENT @SOUNDFRAMESYNC,GOSUB 0:GOSUB 9000
35 REM Check for show start button
40 IF @CLOSURE(24)=0 THEN 40
42 @CLOSURE(24)=0
45 REM Show start
50 ONEVENT @SOUNDFRAMESYNC,GOSUB 1000
55 @SOUND$="ALAMO.WAV"
60 REM Check for show end (sound or button)
65 IF (@CLOSURE(25)=0) AND (@SOUND$<>"") THEN 65
70 @CLOSURE(25)=0:@SOUND$=""
75 GOTO 25
1000 REM Sound Frame Sync handler
1005 S=@SOUNDFRAMESYNC
1010 REM Phil's Intro
1015 LIF S=1 THEN C1=0:I1=255:M1=127:GOSUB 10100:RETURN
1020 REM Charli
1025 LIF S=170 THEN C1=0:I1=127:M1=0:GOSUB 10100:C0=1:I0=0:M0=255:GOSUB 10000:RETURN
1030 REM Lunette
1035 LIF S=185 THEN C1=1:I1=255:M1=0:GOSUB 10100:C0=2:I0=0:M0=255:GOSUB 10000:RETURN
1040 REM Bowie's room
1045 LIF S=220 THEN C1=2:I1=255:M1=0:GOSUB 10100:C0=3:I0=0:M0=255:GOSUB 10000:RETURN
1050 REM Kitchen
1055 LIF S=243 THEN C1=3:I1=255:M1=0:GOSUB 10100:C0=4:I0=0:M0=255:GOSUB 10000:RETURN
1060 REM Ramp
1065 LIF S=249 THEN C1=4:I1=255:M1=0:GOSUB 10100:C0=5:I0=0:M0=255:GOSUB 10000:RETURN
1070 REM Gunade
1075 LIF S=281 THEN C1=5:I1=255:M1=0:GOSUB 10100:C0=6:I0=0:M0=255:GOSUB 10000:RETURN
1080 REM Trevino
1085 LIF S=295 THEN C1=6:I1=255:M1=0:GOSUB 10100:C0=7:I0=0:M0=255:GOSUB 10000:RETURN
1090 REM XCastenada
1095 LIF S=313 THEN C1=7:I1=255:M1=0:GOSUB 10100:C0=8:I0=0:M0=255:GOSUB 10000:RETURN
1100 REM norCasten
1105 LIF S=326 THEN C1=8:I1=255:M1=0:GOSUB 10100:C0=9:I0=0:M0=255:GOSUB 10000:RETURN
1110 REM Teran
1115 LIF S=350 THEN C1=9:I1=255:M1=0:GOSUB 10100:C0=10:I0=0:M0=255:GOSUB 10000:RETURN
1120 REM Long Barracks
1125 LIF S=371 THEN C1=10:I1=255:M1=0:GOSUB 10100:C0=11:I0=0:M0=255:GOSUB 10000:RETURN
1130 REM convent
1135 LIF S=402 THEN C1=11:I1=255:M1=0:GOSUB 10100:C0=12:I0=0:M0=255:GOSUB 10000:RETURN
1140 REM ConventCourt
1145 LIF S=429 THEN C1=12:I1=255:M1=0:GOSUB 10100:C0=13:I0=0:M0=255:GOSUB 10000:RETURN
1150 REM SouthCourt
1155 LIF S=438 THEN C1=13:I1=255:M1=0:GOSUB 10100:C0=14:I0=0:M0=255:GOSUB 10000:RETURN
1160 REM Fortin de Cos
1165 LIF S=482 THEN C1=14:I1=255:M1=0:GOSUB 10100:C0=15:I0=0:M0=255:GOSUB 10000:RETURN
1170 REM moonlight
1175 LIF S=503 THEN C1=15:I1=255:M1=0:GOSUB 10100:C0=16:I0=0:M0=255:GOSUB 10000:RETURN
1180 REM 4th column
1185 LIF S=540 THEN C1=16:I1=255:M1=0:GOSUB 10100:C0=17:I0=0:M0=255:GOSUB 10000:RETURN
1190 REM 1st 2nd columns
1195 LIF S=557 THEN C1=17:I1=255:M1=0:GOSUB 10100:C0=18:I0=0:M0=255:GOSUB 10000:RETURN
1200 REM 3rd column
1205 LIF S=565 THEN C1=18:I1=255:M1=0:GOSUB 10100:C0=19:I0=0:M0=255:GOSUB 10000:RETURN
```

```
1210 REM Foothold Nor
1215 LIF S=588 THEN C1=19:I1=255:M1=0:GOSUB 10100:C0=20:I0=0:M0=255:GOSUB 10000:RETURN
1220 REM Low Barrack
1225 LIF S=604 THEN C1=20:I1=255:M1=0:GOSUB 10100:C0=21:I0=0:M0=255:GOSUB 10000:RETURN
1230 REM Long Barracks
1235 LIF S=611 THEN C1=21:I1=255:M1=0:GOSUB 10100:C0=22:I0=0:M0=255:GOSUB 10000:RETURN
1240 REM Convent On
1245 LIF S=612 THEN C0=23:I0=0:M0=255:GOSUB 10000:RETURN
1250 REM Long Barracks Off
1255 LIF S=623 THEN C1=22:I1=255:M1=0:GOSUB 10100:RETURN
1260 REM Bowie's Room On
1265 LIF S=642 THEN C0=24:I0=0:M0=255:GOSUB 10000:RETURN
1270 REM Convent Off
1275 LIF S=676 THEN C1=23:I0=0:M0=255:GOSUB 10100:RETURN
1280 REM Bowie's Room Off
1285 LIF S=645 THEN C1=24:I1=255:M1=0:GOSUB 10100:RETURN
1290 REM Palisade On
1295 LIF S=681 THEN C0=25:I0=0:M0=255:GOSUB 10000:RETURN
1300 REM Palisade Off
1305 LIF S=686 THEN C1=25:I1=255:M1=0:GOSUB 10100:RETURN
1310 REM Church On
1315 LIF S=688 THEN C0=26:I0=0:M0=255:GOSUB 10000:RETURN
1320 REM Dawn
1325 LIF S=696 THEN GOSUB 11000:RETURN
1330 REM 1 thru 8 off
1335 LIF S=710 THEN S1=1:E1=7:GOSUB 11100:RETURN
1340 REM 9 thru 14 off
1345 LIF S=717 THEN S1=8:E1=13:GOSUB 11100:RETURN
1350 REM 15 thru 20 off
1355 LIF S=726 THEN S1=14:E1=19:GOSUB 11100:RETURN
1360 REM 21 thru 27 off
1365 LIF S=735 THEN S1=20:E1=26:GOSUB 11100:RETURN
1370 RETURN
9000 REM Fadeup house lights, others off
9005 C0=0:I0=@DMXDATA(0):M0=255:GOSUB 10000
9010 FOR C9=1 TO 31:@DMXDATA(C9)=0:NEXT C9
9015 RETURN
10000 REM Fadeup channel C0 from I0 to M0
10005 ONEVENT @TIMER(0),GOSUB 10050
10010 F0=I0:@TIMER(0)=2
10015 RETURN
10050 IF F0<=(M0-4) THEN F0=F0+4 ELSE F0=M0
10055 @DMXDATA(C0)=F0
10060 LIF F0<>M0 THEN @TIMER(0)=2:RETURN
10065 ONEVENT @TIMER(0),GOSUB 0:RETURN
10100 REM Fadedown channel C1 from I1 to M1
10105 ONEVENT @TIMER(1),GOSUB 10150
10110 F1=I1:@TIMER(1)=2
10115 RETURN
10150 IF F1>=(M1+4) THEN F1=F1-4 ELSE F1=M1
10155 @DMXDATA(C1)=F1
10160 LIF F1<>M1 THEN @TIMER(1)=2:RETURN
10165 ONEVENT @TIMER(1),GOSUB 0:RETURN
11000 REM Fadeup all channels except house
11005 ONEVENT @TIMER(0),GOSUB 11050
11010 F0=0:@TIMER(0)=2
11015 RETURN
11050 IF F0<=(255-4) THEN F0=F0+4 ELSE F0=255
11055 FOR C9=1 TO 31:@DMXDATA(C9)=F0:NEXT C9
11060 LIF F0<>255 THEN @TIMER(0)=2:RETURN
11065 ONEVENT @TIMER(0),GOSUB 0:RETURN
11100 REM Fadedown channels S1->E1
11105 ONEVENT @TIMER(1),GOSUB 11150
11110 F1=255:@TIMER(1)=2
11115 RETURN
11150 IF F1>(0+4) THEN F1=F1-4 ELSE F1=0
11155 FOR C9=S1 TO E1:@DMXDATA(C9)=F1:NEXT C9
11160 LIF F1<>0 THEN @TIMER(1)=2:RETURN
11165 ONEVENT @TIMER(1),GOSUB 0:RETURN
```

Play Random Announcement Periodically

This example allows the CFSound-III to periodically interrupt a music source playing through the line input and play a random pre-recorded announcement. The CFSound-III 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’s 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 CF 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
```

VGA Display of Random Colored Triangles

This example draws random colored triangles on the CFSound-III optional VGA display.

Here's how it works. The VGA screen is cleared to black in line 7. Line 10 declares x and y to be dimensioned numeric arrays of three coordinates (index of 0, 1 and 2). Lines 20 and 30 fill these coordinate arrays with random x and y values that are limited to the current display mode's screen width (@VGAWIDTH) and height (@VGAHEIGHT).

The randomly generated triangle is then rendered on the screen in line 40, with randomly generated outline and fill colors using the RGB function. A descriptive text label is applied in line 42 and then the screen is updated with the result in line 45.

```
5 REM draw random triangles
7 @VGAMODE=0:@VGASHOWCURSOR=0:@VGAAUTOUPDATE=0:VGAFILL 0
10 DIM x(2),y(2)
20 x(0)=RND(@VGAWIDTH):x(1)=RND(@VGAWIDTH):x(2)=RND(@VGAWIDTH)
30 y(0)=RND(@VGAHEIGHT):y(1)=RND(@VGAHEIGHT):y(2)=RND(@VGAHEIGHT)
40 VGAPOLYGON x, y, RGB(RND(256), RND(256), RND(256)),RGB(RND(256),RND(256),RND(256))
42 VGATEXT 0, @VGAHEIGHT-40, 1, 2, 1, -1, 0, "TRIANGLES"
45 @VGAUPDATEPAGE=0
50 GOTO 20
```

VGA Display of Seconds on top of a bitmap

This example draws the seconds on top of a background bitmap on the CFSound-III optional VGA display.

Here's how it works. The VGA auto update is turned off in line 10.

In line 40 a bitmap is loaded into the second drawing page. In lines 50 and 55 the bitmap is copied back to drawing page 0 using the VGABLIT command and the VGA graphics page is updated to display it.

An event handler for the @SECOND specialvar is defined in line 60. Whenever the @SECOND changes, once per second, the subroutine starting at line 1000 is called. The program then loops forever at line 70.

When the @SECOND event handler fires the portion of the bitmap that will be overwritten is copied from where it was loaded into drawing page 1 to drawing page 0. Then the current value of the @SECOND variable is printed out on top of it using a white color with a style of no offColor pixels drawn. The VGA graphics page is then updated from the current drawing page and the subroutine returns.

```
10 REM Blit demo
20 @VGAAUTOUPDATE=0
30 REM load background bitmap
40 @VGADRAWPAGE=1:VGABITMAP 0,0,"test4.bmp":@VGADRAWPAGE=0
50 VGABLIT 0,0,0,640,400,1,0,0,5
55 @VGAUPDATEPAGE=0
60 ONEVENT @SECOND,GOSUB 1000
70 GOTO 70
1000 REM @second event handler
1010 VGABLIT 0,100,100,100,16,1,100,100,5
1020 VGATEXT 100,100,1,2,3, RGB(255,255,255),0,@SECOND
1025 @VGAUPDATEPAGE=0
1030 RETURN
```

Firmware Revisions

Version	Date	Notes
1.0	5-17-02	First started development.
1.1	10-20-04	Changes to run on CFSound-III prototype.
1.2	8-11-06	Additions to allow sound playing.
1.3	8-29-06	Changed DisplayProgramListing() to add a preceding space to a secondary keyword if it's preceded by an unsubscripted specialvar.
1.4	9-20-06	Upgrade VDSP toolset from 3.5 to 4.5. Changed MEMORY specialvar to call new heap_space_unused(0) to show program memory left. Added @BAUD special var. Added MULDIV() function. Added support for string lexicographical relation checking with <,<=,>,>= operators. Added divide by zero checking on /, % and MULDIV function. Increased size of available program memory from 4095 bytes to 131068 bytes by moving the heap from L1 to L2 memory. Added FIND() function. Corrected @MSG$ variable events. Corrected MID$() index to be zero based. Added LCDx statements to support ACS-LCD-128x64 on serial port.
1.5	11-15-06	Added @PTT special var. Clear CFSound Red LED indicator flashes if RUN command issued.
1.6	11-29-06	Added LCDBITMAP command.
1.7	2-08-07	Added @MUTE special var. Un-mute amplifier if RUN command issued.
1.8	6-25-07	Added @PORT2 special var and support functions for new CFSound-3 revision 3.
1.9	7-31-07	Added @LINEIN special var. Disable line input if RUN command issued. Added @NSVOL special var that changes the current volume but doesn't save it to NVRAM. Corrected syntax error on attempts to access @PORT2 special var.
1.10	8-29-07	Corrected LCDBITMAP command memory free() calls to be in reverse matching order to calloc() calls to minimize memory fragmentation. Increased size of the available program memory from 131068 to 524284 bytes. Cleared any pending TIMER events upon TIMER assignment. Clear pending escapes when RUN command issued. Fix SYNTAX_ERROR on empty command line. Fixed problem with WAIT statement hanging up due to ONEVENT handling clearing events between statements - now only clear events between statements if there is an event handler defined and executed. Added EDIT line command. Corrected @YEAR to return two digit year. Corrected the ability to GOTO self AND still process events.
1.11	9-12-07	Corrected @CONTACT()= assignments to be active true (non-zero assignment turns output contact on). Corrected ABS() function to return correct value. Rewrote string handling to be to be similar to numeric expression handling allowing true nesting. Change @DOW to be read-only, 0=Sunday -> 6=Saturday computed from the epoch Thursday January 1, 1970. Added VAL() to return the numeric value of a string argument. Added STR$() to return the string representation of a numeric argument. Added optional ELSE clause to IF/THEN statement. Added FOPEN #N, FREAD #N and FWRITE #N commands. Added @FEOF(#N) specialvar. Removed useless EXIT command. Added EXITFOR command to allow exiting to a line from within a for/next loop without receiving a nesting error. Added AND / OR logical operators. Corrected operator priority so that statements like A=0 OR A=2 and A*2+3 evaluate correctly. Changed bitwise ! to ~. Added NOT operator. Added ERR$() function to return string representation of last error number. Added FINSERT #N and FDELETE #N commands. Corrected DEL, REN and MD commands to allow use in programs. Added INSERT$ and REPLACE$ commands. Corrected LIF to return syntax error if line # appears after THEN.
1.12	9-24-07	Conditionalized contact closure and opening processing in BasicTimer_Process() to only set the event and remove the closure if there is an event handler defined to allow the use of @CLOSURE and @OPENING in a program without an event handler.
1.13	10-30-07	Increased stream I/O buffer size from 512 to 32256 bytes to speed up program loading and file I/O statements. Added escape detection to terminate TYPE command output. Added MULMOD() function.
1.14	11-27-07	Corrected @PORT() and @PORT2() special variables to update the OutputContacts[] as well if the port number is <=2.
1.15	4-22-08	Added call to flush uart tx queue in Basic_Process if escape sequence is detected to interrupt long program & type command output. Added @DMXxxxx specialvars. Fixed bug in ORDER statement not finding the referenced DATA statement.
1.16	4-22-08	Added @DMXANALOG specialvars. Corrected a race condition that caused @SOUND$ events to be missed.

Version	Date	Notes
1.17	8-27-08	Corrected lockup bug with FIND() function. Corrected Syntax Error in INPUT #N statements when end of file #N is reached. Added FINPUT #N statement. Corrected FREAD and FINPUT statements to give Out of Data error if they run out of data in the file #N before all of the variables are assigned values. Added three new error codes. Added @LCDADDRESS specialvar. Added additional error descriptions following the line number for some Syntax errors. Added @LCDTYPE specialvar to affect operation of LCDx statements and provide support for ACS-LCD320x240 on serial port.
1.18	12-09-08	Internal development version for DMX testing.
1.19	2-10-09	Added @SOUNDFRAMEPRESCALER and @SOUNDFRAMESYNC specialvars. Fixed syntax error display of "Expected 'x'" to correct the display of anticipated keyword tokens.
1.20	2-25-09	Added delay after setting @DMXRESET to allow DMX CPU time to reset.
1.21	9-03-09	Added support for new Video Graphics Adaptor Adaptor - @VGAx specialvars and VGAx statements. Added SIN(), COS() and RGB() functions. Corrected ONERROR GOTO statement to allow it to be disabled by specifying a zero line number.
1.22	12-02-09	Changed RUN statement to support optional line number or filename to be LOADed and RUN. Fixed REName command failure when new filename contains a Basic keyword.
1.23	12-04-09	Updated RTC variables in RTC_Init()so that time specialvars are correct when program starts.
1.24	4-05-10	Enabled @DMXFRAMESYNC specialvar.
1.25	8-18-10	Added ability to protect integrator developed programs.
1.26	8-24-10	Added ability to disable Basic sign-on message.
1.27	11-09-10	Added @EOT specialvar.
1.28	11-16-10	Increased MAX_STRING_SIZE from 127 to 255 characters. Added GETCH(x) function to allow working with single serial or PS/2 characters. Added @MSGENABLE specialvar to allow GETCH(x) to work with serial characters by disabling the @MSG$ specialvar. Added support for DIMensioned string variables. Added ability to escape LIST and TYPE commands.
1.29	12-2-10	Added RESQ resequencing command. Added support for VGA PS/2 numeric keypad and function keys. Added FPRINT #N to complement FINPUT #N. Corrected erroneous repeating error message display when running incorrect integrator developed program.
1.30	12-29-10	Changed #N range from 0-9 to 0-23 so up to 24 files can be open. Corrected operation of LIST command when used with a single line #.
1.31	12-29-10	Fixed problem with undetected FREAD past valid data – added new error #50 – "FREAD record # Out of Range". Added setting of @FEOF(#N) on FOPEN #N if the file is empty.

ASCII Table

Dec	Hex	Octal	Character
0	00	000	NUL (null)
1	01	001	SOH (start of heading)
2	02	002	STX (start of text)
3	03	003	ETX (end of text)
4	04	004	EOT (end of transmission)
5	05	005	ENQ (enquiry)
6	06	006	ACK (acknowledge)
7	07	007	BEL (bell)
8	08	010	BS (backspace)
9	09	011	TAB (horizontal tab)
10	0A	012	LF (line feed, new line)
11	0B	013	VT (vertical tab)
12	0C	014	FF (form feed, new page)
13	0D	015	CR (carriage return)
14	0E	016	SO (shift out)
15	0F	017	SI (shift in)
16	10	020	DLE (data link escape)
17	11	021	DC1 (device control 1)
18	12	022	DC2 (device control 2)
19	13	023	DC3 (device control 3)
20	14	024	DC4 (device control 4)
21	15	025	NAK (negative acknowledge)
22	16	026	SYN (synchronous idle)
23	17	027	ETB (end trans. block)
24	18	030	CAN (cancel)
25	19	031	EM (end of medium)
26	1A	032	SUB (substitute)
27	1B	033	ESC (escape)
28	1C	034	FS (file separator)
29	1D	035	GS (group separator)
30	1E	036	RS (record separator)
31	1F	037	US (unit separator)
32	20	040	Space
33	21	041	!
34	22	042	“
35	23	043	#
36	24	044	$
37	25	045	%
38	26	046	&
39	27	047	‘
40	28	050	(
41	29	051	)
42	2A	052	*
43	2B	053	+
44	2C	054	,
45	2D	055	-
46	2E	056	.
47	2F	057	/
48	30	060	0
49	31	061	1
50	32	062	2
51	33	063	3
52	34	064	4
53	35	065	5
54	36	066	6
55	37	067	7

56	38	070	8
57	39	071	9
58	3A	072	:
59	3B	073	;
60	3C	074	<
61	3D	075	=
62	3E	076	>
63	3F	077	?
64	40	100	@
65	41	101	A
66	42	102	B
67	43	103	C
68	44	104	D
69	45	105	E
70	46	106	F
71	47	107	G
72	48	110	H
73	49	111	I
74	4A	112	J
75	4B	113	K
76	4C	114	L
77	4D	115	M
78	4E	116	N
79	4F	117	O
80	50	120	P
81	51	121	Q
82	52	122	R
83	53	123	S
84	54	124	T
85	55	125	U
86	56	126	V
87	57	127	W
88	58	130	X
89	59	131	Y
90	5A	132	Z
91	5B	133	[
92	5C	134	\
93	5D	135	]
94	5E	136	^
95	5F	137	_
96	60	140	`
97	61	141	a
98	62	142	b
99	63	143	c
100	64	144	d
101	65	145	e
102	66	146	f
103	67	147	g
104	68	150	h
105	69	151	i
106	6A	152	j
107	6B	153	k
108	6C	154	l
109	6D	155	m
110	6E	156	n
111	6F	157	o
112	70	160	p
113	71	161	q
114	72	162	r
115	73	163	s
116	74	164	t

117	75	165	u
118	76	166	v
119	77	167	w
120	78	170	x
121	79	171	y
122	7A	172	z
123	7B	173	{
124	7C	174	|
125	7D	175	}
126	7E	176	→
127	7F	177	←

PS/2 ANSI Character Sequences

If the optional CFSound-3 VGA module is installed, the IBM PS/2 keys are translated into the follow ANSI character sequences:

PS/2 Key	ANSI Function	Decimal Character Sequence
Enter	Carriage Return	13
End	Cursor End	27, 91, 75
←	Cursor Left	27, 91, 68
Home	Cursor Home	27, 91, 72
↓	Cursor Down	27, 91, 66
→	Cursor Right	27, 91, 67
↑	Cursor Up	27, 91, 65
F1	Function 1	27, 79, 80
F2	Function 2	27, 79, 81
F3	Function 3	27, 79, 82
F4	Function 4	27, 79, 83
F5	Function 5	27, 79, 84
F6	Function 6	27, 79, 85
F7	Function 7	27, 79, 86
F8	Function 8	27, 79, 87
F9	Function 9	27, 79, 88
F10	Function 10	27, 79, 89
F11	Function 11	27, 79, 90
F12	Function 12	27, 79, 65