GrafExpress 2.0

Contains Both GrafExpress 16 and GrafExpress 256

An Incredible 128/512K Graphics Programming Environment

Addendum for GrafExpress 2.0

This addendum details some important additions to GrafExpress 2.0

IMPORT.BAS

This program will allow the user to take graphics created using any CoCo graphics utility and save a 100x100 maximum area region as a GrafExpress picture file.

To use IMPORT.BAS, first load the graphic image onto the normal HSCREEN 2 graphics screen. Run the program IMPORT.BAS. There will be a rectangular area in which the graphics are inverted that represents the graphic image to be imported. To move this inverted box around the screen, use the arrow keys. Use them with the shift key to move faster.

Toggle between moving and resizing the box by pressing the space bar. After doing this, the arrow keys will change the position of the lower right hand corner. Again, the maximum picture size is exactly 100x100 pixels, the left edge must always be on an even column, and the right edge must be on an odd one.

When you have correctly positioned and sized the area, press ENTER to select it. Then, type the name of the destination picture file. It is a good idea to use the extension .PIC, so that all of your picture files are easily recognizable.

After the program saves the file, it will return to the graphics screen for any further imports. Pressing BREAK at any time will quit the program.

Saving an entire screen:

Save the entire screen by saving 100x100 sized sections of it, and then loading them one at a time into memory under a GrafExpress program.

Chaining for more memory:

After loading all the needed graphics with a BASIC program, you might want to run your actual game/utility/etc. from another. This allows you to free up the low memory needed to load in graphics files.

In order to free up about 16K for BASIC workspace, insert this as the last line of the initial BASIC program that loads the graphics files:

```
999 C$="SLOW":EXEC:POKE 25,14:POKE 3584,0:RUN"PROG2"
```

where PROG2.BAS is the program that uses the graphics files loaded in the initial BASIC program.

CONVERT.BAS

This program allows you to import other fonts into the GrafExpress system. It currently comes with the CoCo 3 standard system font, although we have included and used the BIG.FNT in some of the demos/utilities.

Basically, the program will take a font previously loaded into the CoCo 3 Super Extended BASIC font area, convert it, and save it out as a GrafExpress font file. This file, when loaded after loading GE16.BIN or GE256.BIN, will patch GrafExpress to use that font. CONVERT.BAS will also directly patch either system to use the font automatically. We recommend that instead you load the font in the driver program, though. This prevents permanent changes to the GrafExpress executable files. See any of the included utility or demo BASIC programs for an example.

The fonts that can be used should be bounded by the standard CoCo 3 font size. In the box allowed for one letter and its margins, you should leave the rightmost two pixels empty. Ideally, you should also leave the leftmost pixel blank as well. If the font that you are attempting to convert does not follow these rules, CONVERT.BAS will alert you that it may mangle some of the characters in the process of compressing them. Experiment to see what fonts work and which do not.

Extras

A few other support programs have been included in the package. The first is DEMO16.BAS. This is another demonstration program (other than INTRO.BAS) for the 16 color GrafExpress system. Also, HIRESPIC.MKR has been added to its counterparts, LORESPIC.MKR and PIC.MKR.

ADOS 3 users:

If you are using GrafExpress 2.0 with the ADOS operating system, you must add this line to your BASIC program to insure compatability.

```
45    POKE&H6BFC,1:POKE&H6BFD,&H68:POKE&H6C07,1:POKE&H6C08,&H67:
      POKE&H6C10,1:POKE&H6C11,&H68:POKE&H6C16,&H67:POKE&H6C46,1:
      POKE&H6C47,&H68:POKE&H6C51,1:POKE&H6C52,&H67:POKE&H6C57,1:
      POKE&H6C58,&H69
```

Rights of the Customer

GrafExpress was designed to promote the creation and proliferation of new and innovative programs. To help reach that goal, Sundog Systems wishes to make clear certain rights of the customer, some of which are guaranteed by law, and others which are granted herein.

Not only can individuals write programs for themselves, but they can distribute those programs for use by others in any way they see fit. However, the GrafExpress system itself is a copyrighted product and can not be distributed in any way without the written consent of Sundog Systems. This means that individuals are free to give to others, publish or sell their programs as long as they do not include the GrafExpress system. (The individuals buying, reading and entering, or in any other way receiving the programs would need their own copy of the GrafExpress system in order to make use of the programs.)

Anyone wishing to market a product which uses the GrafExpress system has two possible methods. The first is to sell the product with the knowledge that customers would need to purchase (or previously have purchased) their own copy of GrafExpress in order to use the product. (Sundog Systems has nothing against the words "Sundog Systems' GrafExpress 2.0" appearing in advertisements as long as it is only used in the context of informing prospective customers that they need their own copy of GrafExpress in order to use the advertised product.)

The second method is to contact Sundog Systems and request a limited licensing contract for a specific, completed application program. The main purpose of such contracts is to insure that a customer of a specific application does not also have direct access to the GrafExpress system included in the package.

Table of Contents

SECTION ONE: General Information

Introduction

Many people believe that a program is only as fast as the computer on which it is run. Those same people need to know that it is often the case that it is the computer which is only as fast as the program that governs its actions. There are programming techniques which can often boost a routine's speed many times. For a long time there have been assembly language programmers with a fair understanding of these techniques who have created and used some fairly fast routines for their own programs. However, it is an all too rare occurrence for one of the really good routines to be made available for the use of outsiders (especially to those who are not assembly language programmers). GrafExpress is not only one of those rare occasions, but several of them in one. Most of the routines in GrafExpress were written with speed in mind and use various techniques to improve it. Some of them (such as window copy and scroll, sprite restoration and collision, box draw and music creation) not only use these techniques to boost speed, but (for the given computer) are most likely the fastest routines of their kind in existence.

It is very common to see methods such as metaphors and abstraction used to make life easier and less complicated for the computer user. It seems just as common for everything to be made proportionately complex for the programmer. This manual tries to put these methods to good use and avoid whenever possible getting too bogged down with the technical details of how things work. This would not be possible except that the GrafExpress system itself was created with the concepts of abstraction in mind (for example, although they are not completely identical, what BASIC calls a get/put buffer GrafExpress simply calls a picture). There were even many occasions during the creation of this manual that the author found something which was difficult to explain, and shrewdly decided to go back and improve the GrafExpress system to make it easier to explain, understand, and use.

The presentation of this powerful combination of qualities, extreme speed and ease of use, along with GrafExpress' extensive capabilities is an event which marks a major departure from the type of limited, overpriced and overly complex programming tools which this computer community has unfortunately been subjected to so often in the past. It is our hope that GrafExpress will help stimulate sweeping beneficial changes to this, our computer community, which will help it continue to grow and mature into the future.

If you are just learning about the GrafExpress system, the best way to get started is to first go through the section called **Before Getting Started**, and then try out the programs in **Included Applications**. After that come back to the next section, **Conceptually Speaking**, and continue with the rest of **SECTION ONE** and **SECTION TWO**. Feel free to skip around when that is useful (such as trips to the **Glossary** and **Command Definitions** sections). After reading the first two sections try out some of the commands thinking small and simple until you get the hang of it. You may find it useful to reread parts or all of the manual *after* you have some experience actually using GrafExpress (some things may suddenly make a lot more sense!). Assembly language programmers are strongly advised to get used to using GrafExpress in BASIC before reading **SECTION THREE** and trying it in assembly language.

Conceptually Speaking

This section is designed to help you understand some of the ideas that are the framework from which GrafExpress was constructed. In addition to this section, you will probably also find the **Glossary** useful in understanding the concepts which may be new to you.

GrafExpress: Two systems under one name

GrafExpress 2.0 is composed of two very similar systems, GrafExpress 16 (GE16) and GrafExpress 256 (GE256). GE16 is a graphics/sound system based on 16 color graphics. And GE256, of course, is a graphics/sound system based on 256 color graphics. GE16 will work on any monitor, but because of the method by which 256 colors are produced, GE256 will only work on composite monitors or color televisions. In order to keep things simple for you, the two systems were constructed with identical command sets, and nearly identical command parameters. There are only two differences you will find between GE16 and GE256. First, because 256 colors requires twice as much memory as 16 colors, all maximum x-resolutions are halved in GE256 as compared to GE16. And the second is that GE16 rounds off some coordinates to even or odd pixels in order to deal with the graphics easier (this manual specifically details every case where this is done), but GE256 never needs to do any rounding. These differences are mentioned everywhere they apply, but the differences will be easier for you to remember if you remember these two general rules instead of trying to remember every specific case.

Windows

A window has two purposes. The first is a containment for graphics primitives such as pixels, lines, and boxes. These drawing tools work inside the window as if the window were just a little screen. This has the flexibility that any image drawn in a window can be drawn somewhere else just by moving the window (or changing which window is current) and then reissuing the same exact commands. The second use is the ability for a window to be a viewport into spriteland. A window only acts as a viewport when that feature is turned on. When used as a viewport, a window has two positions at the same time. It has a physical position on the screen, and a logical position in spriteland.

In addition to its qualities of position(s) and having the viewport on or off, each window has some other characteristics. Every window has its own pen position, the last place accessed in the window, which can be used with some drawing commands to save memory and time. Each window also has its own current foreground and background colors. The current foreground color is used by most drawing commands. The background color is used by the clear-the-window command and sometimes for text. Unlike some other systems, GrafExpress windows do not save their background when they are defined.

Sprites

A sprite is an object with a shape, an appearance and a position. Like real world objects, a sprite's shape can often be determined by its appearance. The physical edge matches the

visual edge. A sprite can also have a shape which is different from its appearance, like a picture of a bird on a square piece of clear glass: The appearance is that of a bird, the shape is that of a square (not a bird). This is the basis for the two different characteristics each pixel of a sprite can have: transparency (appearance), and solidity (shape). Changing a sprite's position is like moving an object. It moves, you see it move, and if there is an object (another sprite) where you moved it they can hit each other (collide). Sprites can be turned on and off which is like being able to make a real world object exist or not exist. If it does not exist, poof, you can not see it and it does not touch any other object. The last quality a sprite can have is stickiness. A nonsticky sprite is like most real world objects in that it can move from one place to another without leaving a visible trace. A sticky sprite is kind of like a real world object which has a rubber stamper attached to its bottom. It leaves an image of itself everywhere it goes. Sprites also have a priority system which controls what is displayed when two sprites overlap. You can think of a sprite's number as being its distance from the viewer so that higher number sprites appear behind lower number sprites and the normal graphics are behind all of the sprites. Animation is accomplished in a fashion similar to that used by cartoon animators. Any changes in the sprites do not change the currently displayed screen (which animators often call a frame) but instead are recorded and are not actually displayed until the next screen (frame) is shown.

Music

In music, each note has several qualities. One is called tone, or timbre, and is caused by (or some would argue is the same thing as) its *waveform*. You can tell a bagpipe from a trumpet even when they play the same note because of the differences in their waveforms. The overall loudness of a note is called *volume*. On the other hand, how hard (loud) the note is attacked (started) and what happens to the loudness as the note is sustained is called its *envelope*. This means that the loudness of a note at any given instant depends on both its volume and its envelope. A fourth characteristic is how high or low the note is, musically speaking its pitch, or scientifically speaking its *frequency*. The last quality dealt with here is the length of the note. The actual length of a played note depends on two things. First of all is the notes *duration*, or its length relative to the notes around it (a note with a duration of 12 is twice as long as a note with a duration of 6, everything else being equal). And second is the *tempo* of the music, or how fast the music is played.

In this system the waveform and envelope buffers are one and the same. (A buffer is an area of memory that can hold information about something. In this case, it is information about waveforms and envelopes.) So if you want a sine wave with a flat envelope (a note with a flat envelope has the same loudness from start to finish and is said to have 0 decay), you can not define waveform 1 as a sine wave and envelope 1 as flat because whether it's being used as a waveform or an envelope, there is only one buffer 1. You could put the sine wave in waveform 1 (buffer 1) and the flat envelope in envelope 2 (buffer 2). Although this may cause some confusion at first, it does serve a purpose. Being able to use a buffer as either a waveform or an envelope gives you some run-time flexibility. For example at one point in your program you may need 2 waveforms and 4 envelopes while at another you need 4 waveforms and 1 envelope. Instead of having to allocate memory for 4 waveforms and 4 envelopes (8 buffers) you only need to allocate memory for 6 buffers (the most you use at once).

If you take music theory, you will find that music can be broken down into movements, sections, periods, phrases, phrase members and motifs (and probably even others). All vary in size and function and are too much for the average synth-hack to remember, so only one subdivision is provided in this system, that is the *musical subsection*. Musical subsections can be nested, so you can go ahead and break your music into motifs and phrases if you like or any other musical hierarchy you can think of. Their main purposes in this system is to provide a means to remove the redundancy found in most music and to give a unique means of access to different melodies and sound effects.

Before Getting Started

Before you go any further, it is recommended that you make a backup copy of your original GrafExpress disk. If you have two drives, place the GrafExpress original in drive 0, and a blank formatted disk in drive 1, then type **BACKUP 0 TO 1** and press **ENTER**. If you only have one drive then place the GrafExpress original in drive 0, then type **BACKUP 0** and press **ENTER** (you will need to swap disks several times). You should then put and keep your GrafExpress original in a safe place.

SECTION TWO: GrafExpress and BASIC

Getting Started

Characteristics of GrafExpress

The GrafExpress system is not designed as a strict BASIC expansion. It does not add new commands to the language. Instead, it allows a programmer to execute commands contained in a command string (somewhat like draw strings in BASIC). This means that you put the commands into a string, and then call GrafExpress (with the BASIC command EXEC) to execute the commands.

This method has some advantages:

1) It allows a type of named subroutine call which is called substring execution.

2) The GrafExpress system is faster at both interpreting numbers and locating BASIC variables than the BASIC interpreter.

3) GrafExpress does not have to deal with BASIC's syntactic rules (for example in GrafExpress two consecutive periods are always used to indicate a range).

4) Command strings can be built up at run-time which gives some extra flexibility.

5) It maintains a degree of compatibility with BASIC which is not possible with strict BASIC expansions. It is also protected (by BASIC itself) from the reset button.

6) Errors are presented in an easy to remember format (not just two-letter abbreviations).

This method also has some disadvantages:

1) Commands are not tokenized (a compression technique used by most BASICs) so programs take up more memory. However, one-character abbreviations are available for some of the more commonly used commands which is just as efficient as tokenization.

2) Only integers or simple variables such as 15 or X can be used as command parameters. Expressions or arrays such as X+5 or A(9) can not be used as parameters. **This is an important thing to remember! You will probably find yourself at first trying to use expressions inside of GrafExpress commands and they will not work, you will just get errors -- only integers or simple variables will work.**

There is also a characteristic which may be considered an advantage or disadvantage according to personal preference. All variables must be used in BASIC before they can be used in a command string. This is because the GrafExpress system can not allocate variable space, only BASIC can. This may require you to have a BASIC command such as **X=0** to declare a variable. The main advantage of requiring variables to be used in BASIC before being used with GrafExpress is that the system will often catch variable spelling errors or program logic errors that could otherwise just cause mysterious effects that are hard to track down.

Calling GrafExpress from BASIC

GrafExpress uses a command string to pass commands from BASIC to the GrafExpress system. The command string normally used is **C$**. Suppose you want to execute the GrafExpress command **CLW** (CLear the current Window). You would put the command into the command string with the BASIC statement **C$="CLW"**. You would then tell GrafExpress to execute the command with the BASIC statement **EXEC**. (Sometimes EXEC needs the suffix GE -- for more information see the explanation for line 50 in the following section.)

The first time GrafExpress is called, it displays a copyright screen. Just hit a key or button to make it go away and continue with the program.

Setting up GrafExpress

There are some things that need to be included in any BASIC program that is going to use GrafExpress. Below is the way a GrafExpress 16 application program might start. These lines are included in the file START16.BAS so that you can simply load and use them whenever you start a new program that uses GrafExpress 16. (Since there are only a limited number of ways to set up the GrafExpress system, and since Sundog Systems wishes to promote the creation and proliferation of software written under GrafExpress, Sundog Systems hereby gives permission to the customer to use and distribute [as part of an application program] the following six lines of code [lines 10 through 60] or any variation of these six lines as they see fit. This in no way gives permission for any type of distribution of the included applications, only these six lines. See **Rights of the Customer** on the inside front cover of this manual for more information on copyright policy.)

```
10 PCLEAR6:WIDTH32:CLEAR200,19999:POKE65496,0:LOADM"GE16":GOTO40
20 C$="ECHOOFF:SLOW":EXECGE:PALETTE12,TF:PALETTE13,TB:END
30 DATA54,0,0,7,56,63,32,36,6,54,16,18,24,27,8,9,40,45,7
40  C$="":POKE32766,VARPTR(C$)/256:POKE32767,VARPTR(C$)AND255:GE=20000:READTF,TB:
PALETTE12,TF:PALETTE13,TB:ONBRKGOTO20
50 C$="INIT128,256,192,10,2,10,10,100,4":EXECGE
60 FORA=0TO15:READB:PALETTEA,B:NEXT:READGB
```

Line 10

The **PCLEAR 6** is used to allocate space for saving and loading pictures. If you are not doing either of those operations, you would then normally begin with a **PCLEAR 1** instead. The **WIDTH 32** command is used because GrafExpress uses the memory normally used by the high-resolution (40 or 80 column) text screen. If you wish to edit your programs in the 40 or 80 column modes you can go ahead and do so, but the programs must run in the 32 column mode. The **CLEAR 200,19999** sets aside 200 bytes for string space (the area of memory where strings are stored) and tells BASIC to reserve all memory after 19999 (which is where the GrafExpress system resides). The 200 can be changed according to how much string space your program needs. The 19999 can be lowered if you need to reserve more memory (for example for an assembly language program), but it should never be made greater than 19999. The **POKE 65496,0** puts the computer in single speed. Disk operations should only be done in single speed and the next command is a disk operation. **LOADM "GE16"** loads the GrafExpress 16 system. **GOTO 40** skips over the ON BReaK routine and the data statements.

Line 20

This line is the ON BReaK routine (what the program does if the user presses **BREAK**). The **C$="ECHOOFF:SLOW":EXECGE** is an example of the execution of two GrafExpress commands. First they are put into the command string (**C$**) and then the command string is executed by calling GrafExpress (**EXEC GE**). The **ECHO OFF** command shuts off the echo of text to the current graphics window and the **SLOW** command puts the computer in single speed. The two **PALETTE** commands set the foreground and background colors on the 32 column screen to **TF** and **TB**. **END** ends the program.

Line 30

This line contains the data for the text foreground and background colors for the 32 column text screen, the 16 palette colors for the graphics screen, and the graphics screen border color. It is not necessary to do this, but always putting the colors on the same line will let you change color sets (for example from an RGB color set to a CMP color set) by just merging a one line program (saved in ASCII format).

Line 40

The commands **C$="" : POKE 32766,VARPTR(C$)/256 : POKE 32767,VARPTR(C$) AND 255 : GE=20000** link the command string (**C$**) with the GrafExpress system and set up the GrafExpress start address (**GE**) and should never be changed. The **READ TF,TB** sets up the text foreground and background color variables for the 32 column text screen and the **PALETTE** commands set the text colors. This is done here because the INIT command (line 50) latches the text colors, and if GrafExpress detects an error it will restore the text colors before returning to BASIC. The **ON BRK GOTO 20** sets line 20 as the ON BReaK routine.

Line 50

This line is another example of using a GrafExpress command. The INIT command tells GrafExpress to get ready to use a certain screen size and a certain number of sprites and windows etc. and its parameters will vary from program to program (see INIT command definition). The important thing here is that INIT is (and must be) the first GrafExpress command to be called, and the first time that GrafExpress is called the **GE** (GrafExpress starting address) must appear with the **EXEC** command. This sets the EXEC start address. As long as EXEC is not used with another number and the BASIC commands SAVEM, LOADM, CSAVEM, CLOADM, and DEF USR are not used the EXEC start address will not change. As long as the EXEC start address is not changed, following EXECs do not need to include the variable GE.

Line 60

This line sets the graphics color set according to the data in line 30. It also reads the graphics border variable (**GB**) which would have to be used in conjunction with the GrafExpress command **BORDER** sometime later in the program. This line, as with all of the commands concerning only setting the colors, is just a suggestion.

Other Lines

There are some things which should not be done during the rest of the program. The variable GE should not be changed, it should be kept as a pointer to the GrafExpress system. The CLEAR and PCLEAR instructions should not be used any later than the first line because they both destroy all BASIC variables including GE and the command string (C$). If you are using TF, TB and/or GB for the suggested color purposes, then you should remember not to use those variable names for anything else.

When actually writing a program (especially if you are just a beginner at using GrafExpress) it is a good idea to save your program before running it. Certain mistakes you might make in following the above instructions can cause the computer to lock up and the reset button will not always get your program back. So protect your program (and your time) and remember to save before you run.

Using GrafExpress 256

The file "START256.BAS" is a similar starting program changed to work with GrafExpress 256. The first difference is that is loads GE256 instead of GE16. The second difference is that it only sets the colors for the first 4 palettes (0..3). These 4 palette settings can be changed to produce a different set of 256 colors (allowing you to theoretically choose color sets from a palette of over 16 million colors). However, this palette set was chosen because it provides the best possible balance of shades and hues.

Definition of Syntax

This section gives the rules governing the syntax of GrafExpress commands and explains how to interpret the format in which syntax is presented in this manual. Remember that GrafExpress commands are not a strict BASIC expansion, these commands can not be issued as if they were BASIC's native commands. In order to be executed they must be placed in the command string (C$) and then the GrafExpress system must be called with EXEC. A complete listing of the syntactical definitions appears in the section **Command Summary**.

1. All tokens (words in upper case such as SPRITE and ON) are part of the syntax and must be entered. All letters in lower case and italics represent some number or variable to be entered. (**X** and **GB** are variables.)

Syntax	Incorrect	Correct
PLAY *sub#*	PLAY sub#	PLAY 2 or PLAY X
BORDER *color*	BORDER	BORDER 45 or BORDER GB
LOAD PIC *pic#*	LOAD PIC 5+3	LOAD PIC 8

2. At least one space must appear before a token if it is preceded by a variable. Spaces may not appear inside of tokens, numbers, or variable names. All other spaces are optional. (**X** and **VAR** are variables.)

Syntax	Incorrect	Correct
SUB *wnd#* ON	SUBXON or SUB XON	SUBX ON or SUB X ON
INIT PLAY	IN IT PLAY	INIT PLAY or INITPLAY
BORDER *color*	BORDER 4 2	BORDER42 or BORDER 42
PLAY *sub#*	PLAY VA R	PLAY VAR or PLAYVAR

3. Anything inside brackets is optional. The brackets are not part of the commands.

Syntax	Incorrect	Correct
WORK [*screen#*]	WORK [1]	WORK or WORK 1
[FAST] SHOW [*screen#*]	[FAST] SHOW 2	FAST SHOW 2 or SHOW or SHOW 2 or FAST SHOW

4. Any slash indicates to use one of the items. The slashes are not part of the commands.

Syntax	Incorrect	Correct
ECHO ON/OFF	ECHO ON/OFF	ECHO ON or ECHO OFF
WINDOW/& *wnd#*	/& 2	WINDOW 2 or & 2
MIX ON/OFF	MIX ONOFF	MIX ON or MIX OFF

5. Any nonalpha-numeric charcters other than brackets and slashes are part of the syntax and must be entered. (X and Y are variables.)

Syntax	Incorrect	Correct
ENV *env#* = *decayrate*	ENV 1 300	ENV 1 = 300 or ENV1=300
SET *xcoord,ycoord*	SET X Y	SET X,Y or SETX,Y

6. Multiple commands issued at once must be separated by colons.

Syntax	Incorrect	Correct
FONT *width,height* ECHO ON/OFF	FONT 6,8 ECHO ON	FONT 6,8:ECHO ON

Command Definitions

This section gives an explanation of what each command does and what variations (if any) each command has. All coordinates start at 0,0 and go from left to right and from top to bottom.

Drawing Commands

The coordinates used by the drawing commands refer to positions in the current window.

> *xcoord,ycoord*
Changes the pen position for the current window to *xcoord,ycoord*.

BORDER *color*
Changes the border to *color* (0..63).

BOX/LINE/RECT [*xstart,ystart..*] *xend,yend*
Draws a box (filled in), a line, or a rectangle (not filled in) to *xend,yend* using the current foreground color. The box, line, or rectangle starts at *xstart,ystart* if included, otherwise it starts at the current pen position.

CIRCLE *radius* > *xcoord,ycoord*
Draws a circle of size *radius* with its center at *xcoord,ycoord* using the current foreground color. CIRCLE is different from all of the other drawing commands in that it does not need to stay inside the current window. Its center can be out of the window (even have negative coordinates). However, only parts of the circle located within the current window will actually be drawn.

COLOR/= *foregroundcolor* [,*backgroundcolor*]
Sets the foreground color in the current window to *foregroundcolor* (0..15). Also sets the background color in the current window to *backgroundcolor* (0..15) if it is included. The symbol "=" is an optional abbreviation of the token COLOR. For example the commands **COLOR 5** and **= 5** mean the same thing. Before any COLOR command is issued in any given window, foreground color is 1 and background color is 0.

variable = COLOR/= *xcoord,ycoord*
Sets *variable* equal to the color of the pixel at *xcoord,ycoord*. The symbol "=" is an optional abbreviation of the token COLOR. For example **X=COLOR 2,4** and **X==2,4** mean the same thing.

FILL *xcoord,ycoord*
Does an area fill starting at *xcoord,ycoord* using the current foreground color. An area fill (as opposed to a flood fill) only fills the area that starts out with the same initial color as the pixel at *xcoord,ycoord* -- it does not "spill out" over other colors. Also, FILL does not affect

any graphics outside of the current window.

SET *xcoord,ycoord*

Sets the pixel located at *xcoord,ycoord* to the current foreground color.

Miscellaneous commands

variable$

A string name placed by itself will cause the execution of that string as a subroutine. Subroutine calls can be nested (up to 20 levels deep), but a string should neither directly nor indirectly call itself. For example, the commands **A$="CLW":C$="A$":EXEC** are equivalent to the commands **C$="CLW":EXEC**.

FAST

Puts the computer in double speed. Disk operations should not be done in double speed. Doing so can cause the loss of all disk files, so use **SLOW** before all disk operations.

variable = HJOY *joystick#*

Sets variable to the position of joystick *joystick#*. This is for use with the standard high-resolution joystick adaptor. Top,left = 0,0 and bottom,right = 639,639. This will only function properly in double speed (see FAST).

joystick#	Joystick	Axis
0	right	left/right
1	right	up/down
2	left	left/right
3	left	up/down

INIT *mem,xres,yres,#pics,#scrns,#sprts,#wnds,musicmem,#waves*

Initializes all of the buffers (screens, sprites, pictures etc.) and sets their size. INIT must be called before any other commands are used, failing to do so could cause a system crash.

```
 mem = number of kilobytes the computer has (128 or 512)
xres = horizontal resolution (128,160,256 or 320)
         (GrafExpress 256 only allows 128 or 160)
yres = vertical resolution (192,200 or 225)
 #pics = number of pictures (0..255)
#scrns = number of screens (1..255, all xres by yres in size)
#sprts = number of sprites (0..255)
 #wnds = number of windows (1..255)
musicmem = bytes reserved for musical subsections (0..8000)
 #waves = number of waveforms/envelopes reserved (0..30)
```

The INIT command can be reissued to change resolution or the amount of memory allocated to pictures, screens etc. but it will also erase any old picture, music, sprite and window definitions. At least one screen and one window must be allocated. In order to use nonsticky sprites, at least two screens must be allocated. INIT also sets the computer to single speed and latches the colors for the 32 column text screen (palettes 12 and 13). Anytime that GrafExpress detects an error, it will restore the text color to the colors latched by INIT before returning to

BASIC.

NOTE: Not all CoCos are created equal. Many have defective graphics chips which make some vertical resolutions (namely the 200 and 225) have one line more or less than they should. Also, the 128 width 16 color screen is nonstandard which may indicate that it does not work properly on some machines.

SLOW

Puts the computer in single speed. This is the default speed. (See FAST for disk warning.) The computer is automatically put into single speed when GrafExpress detects an error. (However, GrafExpress does not change the speed if it is called by an assembly language program with its own error trapping routine turned on.)

Music commands

ENV *env#* = *decayrate*

Defines envelope *env#* as having a decay rate of *decayrate* (0..32000). What this means is that the envelope starts at maximum (255) and decreases exponentially with each byte. A note with a duration of 24 would only use the first 24 bytes of its envelope. If you would like that note to use more of the envelope, but want its length to stay the same, then you should double its duration (to 48) and decrease the tempo by half. The exact formula used by the system to create the envelope is: byte n = byte n-1 - (byte n-1 * *decayrate* / 32768) where the first byte is 255 1/2.

INIT PLAY

Erases all musical subsections and reclaims all of the buffer space used by the previously defined subsections.

PLAY *sub#*

Executes (plays) musical subsection number *sub#*. PLAY was designed to be executed with the computer in double speed (see FAST). If played in single speed, all music will play half as fast (as if the tempo value were doubled) and will be played an octave lower. PLAY automatically closes any open musical subsection before it begins execution. PLAY temporarily shuts off the serial port so that the port does not receive random data. But because of a CoCo design flaw, the process of shutting the port off actual sends a character out the port. So, if you have a device connected to this port, keep it off-line (or just off) when doing sound.

SUB *sub#* ON

Turns on musical subsection number *sub#*. All musical subsection definition commands (all commands in the next section) will be appended to this subsection until another SUB ON command is issued. Musical subsections can be redefined at any time in one of two ways. One is to reissue the SUB ON command for that subsection, which will restart its definition. However, this method does not reclaim the memory used by its old definition. The other method is to use **INIT PLAY** which reclaims the memory used by the subsection's old definition, but erases all musical subsections in the process. Turning on (opening) one

subsection automatically closes any previously open subsection.

WAVE *wave#* = *weight1,wt2,wt3,wt4,wt5,wt6,wt7,wt8*

Defines waveform *wave#* as having the first eight sine wave harmonics of the given weights (each 0..255). Every waveform entry is found by the following method: Each weight is used as a multiplier for its respective harmonic, then the eight products are summed and the sum is divided by 256 -- all fractions being rounded up. The entire waveform is then decreased by an amount equal to the lowest value. In most waveforms the largest weight is given to the first harmonic (called the fundamental).

WAVE *wave#* (*index*) = *value*

This command is used to change waveforms (or envelopes) byte by byte. Each waveform table is 256 bytes long, numbered 0..255. You refer to which byte (with *index*) of which waveform or envelope (with *wave#*) is going to be changed to *value* (0..255). This command makes it possible to create any possible waveform or envelope, not just those provided by the standard ENV and WAVE definition commands. You will find, though, that the standard definitions are easier to use and are all you need most of the time.

-- Hint: One of the main reasons this command was included is so that you can create gunshot or explosion sound effects. Use this command to fill all 256 bytes of a waveform with random numbers using the BASIC expression RND(256)-1 to make what is called a noise waveform. Using a noise waveform and an envelope with a large rate of decay will provide gunshot type sounds. A smaller rate of decay will yield explosion sounds.

Musical subsection definition commands

The commands in this section are used to define musical subsections, they are not executed immediately. For example, if you give the command **PLAY 52,56,59,64** (C5,E5,G5,C6) the computer will not immediately play a C Major chord. Instead, it will add the command onto whichever musical subsection is currently open (see SUB *sub#* ON). It is when the musical subsection is played (PLAY *sub#*) that the C Major chord will actually sound. All of the commands in this section function like that, they are executed when the musical subsection they are placed in is played. Although execution is delayed, the values of any variables in the commands are read and stored at interpretation time.

ENV *env1,env2,env3,env4*

Sets the current envelopes of voices 1,2,3 and 4 to envelope buffers *env1*, *env2*, *env3* and *env4* respectively.

PLAY *freq1 freq2 freq3 freq4* [,*dur*]

Sounds the pitches *freq1*, *freq2*, *freq3*, and *freq4*. The frequencies 1 through 96 make up eight octaves of equal tempered chromatic scales with the tuning note being A-440 (freq=61). Chromatic means that the notes differ by half steps, so freq=62 gives an A#, freq=63 gives a B, freq=64 gives a C and so forth (see table below). Notes with a frequency of zero are silent (rests). Notes preceded by a "-" (negative notes) are tied (or slurred -- the position within the

envelope is not reset back to the beginning). If *dur* is included then the default duration is set to *dur* (1..250) and *dur* is used as the duration of the current note. If it is not included then the default duration is used as the duration of the current note. The default duration is set statically according to the context of the definition, not dynamically when the PLAY *sub#* command is issued. For example, if the commands **PLAY 52,56,59,54,24 : SUB3 : PLAY 52,56,59,54** were issued, the second PLAY statement would use the default duration of 24 even if musical subsection 3 set its own, different default duration.

Frequency Table

Octave	A	A♯/B♭	B	C	C♯/D♭	D	D♯/E♭	E	F	F♯/G♭	G	G♯/A♭
1	1	2	3	4	5	6	7	8	9	10	11	12
2	13	14	15	16	17	18	19	20	21	22	23	24
3	25	26	27	28	29	30	31	32	33	34	35	36
4	37	38	39	40	41	42	43	44	45	46	47	48
5	49	50	51	<u>52</u>	53	54	55	56	57	58	59	60
6	<u>61</u>	62	63	64	65	66	67	68	69	70	71	72
7	73	74	75	76	77	78	79	80	81	82	83	84
8	85	86	87	88	89	90	91	92	93	94	95	96

(Middle C and A-440 are underlined.)

SUB *sub#*

Calls musical subsection *sub#*. For example, if subsection 2 is currently open then **SUB 1** would add a call to subsection 1 to the current definition of subsection 2. When subsection 2 is actually PLAYed and this part of its definition is reached, subsection 1 will then be PLAYed and then execution will return to the next part of subsection 2's definition. Subsection calls can be nested, but a subsection should neither directly nor indirectly call itself. The subsection called with this command does not need to be defined at the time that this command is added to the currently open subsection, but does need to be defined before the currently open subsection is PLAYed.

SYNC *flag1 flag2 flag3 flag4*

Synchronizes the beginning waveform position of the next note for each voice with a nonzero flag number. This is used to stop cancellation of pitches where two or more voices have the same frequency at the same time. For example if you had the command **PLAY 52,52,56,59** you should precede it with the command **SYNC 1,1,0,0** to avoid the possibility of the first and second voices being out of phase and canceling each other out. Once voices are synchronized, they do not need to be synchronized again as long as they continue to play the same frequency as each other.

TEMPO *tempo*

Sets the current tempo to *tempo* (1..255). The higher the number, the slower the tempo.

VOL *vol1,vol2,vol3,vol4*

Sets the volumes (0..255) of voices 1,2,3 and 4 to *vol1*, *vol2*, *vol3* and *vol4* respectively. The higher the number, the louder the voice (0=silent). Although there is usually some room for tweaking, the sum of the four volumes should generally not exceed 255, otherwise foldover distortion can occur.

WAVE *wave1,wave2,wave3,wave4*

Sets the current waveforms of voices 1,2,3 and 4 to waveform buffers *wave1*, *wave2*, *wave3* and *wave4* respectively.

Picture commands

LOAD PIC *pic#*

Loads picture *pic#* from the BASIC low-resolution graphics area. The low-resolution graphics area must be made large enough with the BASIC command **PCLEAR 6** which should be issued at the beginning of the program before any variables are used (PCLEAR destroys variables). This is very important. If you forget to do a PCLEAR 6 and you go ahead and load a picture file from disk, you could erase your BASIC program! Before this command is called, the SLOW command should be used to put the computer in single speed and the picture file must be loaded from disk using the BASIC command **LOADM "*filename*"**. The EXEC used to call this command should use the GrafExpress start address (**EXEC GE**) because loading a machine language file resets the EXEC start address. Example: **C$="SLOW" : EXEC : LOADM "TEST.PIC" : C$="FAST : LOAD PIC 1" : EXEC GE** where TEST.PIC is the name of a picture file. As in the example, you may want to use FAST to put the computer back into double speed after you have loaded the picture from disk.

OFF *pic# = xoffset,yoffset*

Sets the offset for picture *pic#* to *xoffset,yoffset* (-100,-100..100, 100). When a picture with an offset of 0,0 is drawn, its upper left corner is drawn at the given coordinates. For example **PIC 1 > 20,30** would place the upper left corner of picture 1 at 20,30. If a picture has a nonzero offset, then the offset is added to the given coordinates to find where the upper left corner will be positioned. For example **OFF 1 = -4,11 : PIC 1 > 20,30** would cause the upper left corner to be drawn at 16,41 (20-4,30+11). A picture's offset is reset to 0,0 any time the picture is defined or redefined.

PIC *pic# = xstart,ystart..xend,yend* [;*transcolor*]

Defines picture number *pic#* as being equal to the graphics located within the rectangle having one corner at *xstart,ystart* and the opposite corner at *xend,yend* inclusively.

If the left edge is on an odd pixel, it is rounded down to the next even pixel. If the right edge is on an even pixel, it is rounded up to the next odd pixel. For example **PIC 1 = 21,20..38,42** would define picture 1 as being equal to the graphics located between 20,20 and 39,42. (There are no even/odd restrictions in GE256.)

If *transcolor* is included then all of the pixels that were set to *transcolor* when the picture was defined are transparent and nonsolid. It is as if those pixels did not exist in the picture and you can see right through the holes.

A picture can be redefined, but only to a size (measured by number of pixels) smaller or equal to its original size. A picture defined with a transparency color takes up twice the amount of memory as a normal picture does and rectangle pictures (see below) do not use any memory. The first time each picture is defined the sprite buffers are emptied to make room which means any current screen background under a sprite will be destroyed. So define all needed pictures

before using any sprites. Pictures are limited to being 100 or less pixels tall and wide (50 wide and 100 tall in GE256). Additionally, pictures defined with a transparency color are limited to a total of 8000 pixels or less (4000 or less in GE256).

PIC *pic#* = RECT *width,height*

Defines picture number *pic#* as being a totally transparent picture having the dimensions *width* and *height*. If width is odd, it is rounded up. This command is different than using PIC *pic# = xstart, ystart..xend,yend ;transcolor* to define a totally transparent picture. Although a rectangle picture is completely transparent, it is still solid. Whereas defining a picture with a transparent color makes pixels act nonexistent, defining a picture with PIC *pic#* = RECT *width,height* makes the pixels act like glass. They are transparent, but also solid.

-- Hint: Rectangle pictures use almost no memory and sprites defined with rectangle pictures use almost no processing time. If you are using a normal sprite for a stationary background object (such as a wall in a game or a tile in a point and click program) you might want to consider just drawing the object directly on the screen and using a sprite with a rectangle picture for collision purposes.

PIC *pic#* > *xworldcoord,yworldcoord*

Draws picture *pic#* at the world position *xworldcoord,yworldcoord* in the current window. If the picture is not totally in the window, it will be clipped as needed. For example the commands **WINDOW 2 = 0,0..19,19 : WINDOW 2 > 0,0 : PIC 1=0,0..11,9 : PIC 1 > 14,7** would only draw the left half of picture 1 (the right half is outside of the window). However, if **WINDOW 2 > 10,0** was used instead of **WINDOW 2 > 0,0** then all of picture 1 would have appeared, but 10 pixels to the left (at 4,7 instead of 14,7). It is much easier and much more common to draw pictures in a window which has world coordinates of 0,0.

variable = SAVE PIC *pic#*

Saves picture *pic#* (including its current offset) into the low-resolution graphics area. The low-resolution graphics area must be made large enough with the BASIC command **PCLEAR 6** which should be issued at the beginning of the program before any variables are used (PCLEAR destroys variables). The SAVE PIC command automatically puts the computer in single speed (just in case you would possibly forget) to avoid any disk errors. After the command is called, you should use the BASIC command **SAVEM "*filename*", 3584, *variable*, GE** using the same variable in the SAVEM command as you used in the SAVE PIC command. After saving the file to disk you will probably want to use **FAST** to put the computer back into double speed. Example: **X=0 : C$="X=SAVE PIC 1 : SLOW" : EXEC : SAVEM "TEST.PIC", 3584, X, GE : C$="FAST" : EXEC**

Screen commands

[FAST] SHOW [*screen#*]

Displays screen number *screen#*. Normally SHOW synchronizes to the 60 hz interrupt in order to prevent flicker. It does not do this if FAST is included. If *screen#* is not included, the current work screen is displayed. Before any SHOW commands are issued, the screen with the

highest number is the displayed screen. If you get an OUT OF MEMORY ERROR while using the show command, it means that you have more nonsticky sprites appearing on the screen than there is free memory. (See the section **Miscellaneous Information** for more about nonsticky sprites.)

WORK [*screen#*]

Makes screen number *screen#* the current work screen (all graphics commands are performed on the work screen). If *screen#* is not included, WORK uses the screen numbered one higher than the current work screen unless the current work screen is also the screen with the highest number in which case it makes screen number 1 the current work screen. Before any WORK commands are issued, screen number 1 is the current work screen.

!

Calls the command SHOW followed by the command WORK. This is a good command to use for normal two screen animation. (This command is called "switch".)

Sprite commands

If you change a sprite's definition, picture, position or on/off state, the change takes place in two stages. The first stage happens immediately when you make any of the above changes, the change is recorded and is used for any following collision checking (see the HIT command). It is not until stage two, however, that any of the above changes are actually displayed. Stage two comes when you use the SHOW or ! commands to update the display screen. This is done so that you can set up a number of changes to occur all at the same time when you are ready to show the next screen (the next frame of animation).

variable = *sprite#* HIT *spritelow* [..*spritehigh*]

Checks to see if sprite number *sprite#* collides with sprite number *spritelow*. If *spritehigh* is included then *sprite#* is checked against the range of sprites *spritelow* through *spritehigh*. If there are no collisions, *variable* is set to 0. If there are any collisions, *variable* is set to the number of the first sprite (that is the sprite with the lowest number) that collided with *sprite#*. Sprites that are turned off, and nonsolid parts of sprites that are turned on do not take part in collisions.

SPRITE/* *spritelow* [..*spritehigh*] = *pic#*

Defines sprite *spritelow* (or sprites *spritelow* through *spritehigh* if *spritehigh* is included) as using picture number *pic#*. Sprites can be redefined (remember that the sprite's displayed picture does not actually change until the SHOW or the ! command is issued). The symbol "*" is an optional abbreviation of the token SPRITE. For example the commands **SPRITE 1=2** and *** 1=2** mean the same thing.

-- Hint: Use redefining sprites to create animation.

SPRITE/* *spritelow* [..*spritehigh*] > *xworldcoord,yworldcoord*

Moves sprite *spritelow* (or sprites *spritelow* through *spritehigh* if *spritehigh* is included) to *xworldcoord,yworldcoord*. The sprite's displayed location does not change until the SHOW or the ! command is issued. If the sprite's current picture has a nonzero offset then the upper left corner will be at some position relative to the sprite's current position (see the OFF *pic#* = *xoffset,yoffset* command). The left edge of a sprite is always rounded down to the next even pixel. (No rounding is done in GE256.)

SPRITE/* *spritelow* [..*spritehigh*] ON/OFF

Turns sprite *spritelow* (or sprites *spritelow* through *spritehigh* if *spritehigh* is included) on or off. Before any SPRITE ON/OFF commands are issued, all sprites are off. Only sprites which are ON are displayed or take place in collision checking.

SPRITE/* *spritelow* [..*spritehigh*] [NON] STICKY

Defines sprite *spritelow* (or sprites *spritelow* through *spritehigh* if *spritehigh* is included) as being sticky or nonsticky. A nonsticky sprite can move around without altering the background underneath of it. When it is turned off, there is no visible trace of it. A sticky sprite, however, leaves copies of itself stuck to the screen where ever it is moved.

Text commands

ECHO ON/OFF

Turns the text echo on or off. When text echo is on, anything that is printed on the standard 32 column text screen is also printed in the current graphics window at the current pen position (rounded down to the nearest even pixel in GE16) using the current font size, mix instructions and color(s). This applies to the backspace character (ASCII code 8), the return character (ASCII code 13), and the standard text characters (ASCII codes 32..127). Text running off the right end of a window will wrap around to the left edge of the next line. Text running off the bottom of a window will cause the window to scroll up. The minimum size that a window needs to be to accept text is the size of one character (which depends on the current font size). If you want the backspace character to wrap properly, then you need to make the window a whole number of characters wide (such as 6,12,18,24,... pixels wide for a font width of 6) and have the pen's x position start at a whole multiple of the current font width (such as 0,6,12,18,... for a font width of 6). Before any ECHO command is issued, ECHO is turned off. ECHO should be turned off before a program ends (as it slows down printing in the 32 column text screen). ECHO is automatically shut off when GrafExpress detects an error (but not when BASIC detects an error).

FONT *width,height*

Sets the size of the text characters. The *width* can be 6 or 8 pixels for GE16, 3 or 4 pixels for GE256. The *height* can be anything between 8 and 12 pixels inclusively.

MIX ON/OFF

Turns the ability of mixing text with graphics on or off. When text mixing is turned on, only

the foreground part of the characters (drawn in the current foreground color) changes what appears on the screen. When mix is turned off, the background part of each text character is also drawn (in the current background color) and overwrites anything underneath of it.

Window commands

CLW

Clears the current window to the current background color. The current pen position is set to 0,0.

COPY *sourcewnd* [[FROM *sourecscr*] TO *destwnd*]

Copies the contents of window number *sourcewnd* to window number *destwnd*. If *destwnd* is not included then it defaults to being the same as *sourcewnd* and the current display screen is used as the source screen. If FROM *sourcescr* is included then screen number *sourcescr* is the source screen of the copy. If *destwnd* is included, but without the FROM option, then the source screen is the current work screen. The destination screen is always the current work screen. The source and destination windows do not need to be the same size, but the amount copied will be clipped so that it does not extend outside of either window.

A couple of extra words from the author

The preceding paragraph may be a bit hard to digest, so let me provide some examples. There are only three variations on the copy command.

Example 1: **COPY 1**

This would copy from window 1 of the current display screen to window 1 of the current work screen. It is important to note that this copy (as does any copy with the display screen as its source) will copy any currently displayed sprites.

Example 2: **COPY 1 TO 3**

This would copy from window 1 of the current work screen to window 3 of the current work screen.

Example 3: **COPY 2 FROM 1 TO 3**

This would copy from window 2 of screen 1 to window 3 of the current work screen.

WINDOW/& *wnd#*

Selects window number *wnd#* as the current window. The symbol "&" is an optional abbreviation of the token WINDOW. For example the commands **WINDOW 5** and **& 5** mean the same thing.

[SUB] WINDOW/& *wnd#* = *xstart,ystart..xend,yend*

Selects window number *wnd#* as the current window and defines the window as being the rectangular region on the screen with one corner *xstart,ystart* and the opposite corner *xend,yend*. The left edge of a window, like the left edge of a sprite, is rounded down to the next even pixel

(51 becomes 50, 98 stays 98). The right edge of a window is rounded up to the next odd pixel (51 stays 51, 98 becomes 99). (No rounding is done in GE256.) Redefining a window sets its current pen position to 0,0 but does not change its current color, world position or on/off state. All windows are initially defined as being the full screen, foreground color 1, background color 0, pen and world position 0,0 and sprites off. Windows can be redefined at any time.

If the SUB option is included then instead of using screen coordinates, the command uses coordinates relative to the current window. For example if window 2 was previously defined as 100,50..200,80 and it is the current window then the command **SUBWINDOW 3 = -50,20..55,45** would set window 3 to be the rectangular area with screen coordinates 100-50,50+20..100+55,50+45 or 50,70..155,95. Notice that subwindows do not have to be contained by the parent windows, only by the screen.

WINDOW/& *wnd#* > *xworldcoord,yworldcoord*

Selects window number *wnd#* as the current window and links the upper left corner of the window with the position *xworldcoord,yworldcoord* in spriteland. The left edge is rounded down to the next even world pixel. (No rounding is done in GE256.) Each window has two sets of coordinates, one on the screen and one in spriteland. They can be in different places, but are the same size. For example, if window 2 had been defined by **WINDOW 2 = 80,0..129,40** and its position in spriteland was then changed by **WINDOW 2 > 14,9** then its upper left corner on the screen would be at 80,0 and its lower right corner on the screen would be at 129,40. That gives it a size of (129-80+1) by (40-0+1) or 40 by 41 pixels. The window move command set the window's spriteland position to 14,9. That is, its upper left corner is at 14,9 in spriteland (its position on the screen has not changed). Since the window is 40 by 41 pixels then its lower right corner in spriteland is (14+40-1),(9+41-1) or 53,49. Suppose a sprite has its upper left corner at 16,19 in spriteland, this corresponds with two pixels to the right of and ten pixels below window 2's upper left corner in spriteland. So if window 2 has sprites turned on (see WINDOW ON/OFF) then that sprite will be displayed on the screen with its upper left corner at 82,139 which is two pixels to the right of and ten pixels below window 2's upper left corner on the screen.

WINDOW/& *wnd#* ON/OFF

Turns the spriteland viewport function of window *wnd#* on or off. When this function is on, sprites that are located within the window's spriteland location are displayed on the screen within the window's screen location. Sprites that are on one or more edges of the window are clipped (the parts outside of the window are not displayed). Until any WINDOW ON/OFF commands are issued, all windows are turned off.

Miscellaneous Information

If you want to avoid reading anything technical, you might want to only read the part on chaining programs in this section and avoid everything else. But the rest is at least interesting reading and at best useful so you might want to consider reading it some time.

Programs can be chained.

If you find that your program is getting larger and you are running out of memory, you may want to use program chaining. To do this, you divide your program into two separate programs. The first program initializes GrafExpress and defines most of the pictures, sprites, musical subsections etc. that the second program will need. When it finishes doing that, it runs the second program. For example if the second program was called PROG2 the first program could have as its last line: **999 C$="SLOW" : EXEC : RUN "PROG2"**

Although chaining to a program will erase all of your BASIC variables, all GrafExpress memory remains intact, so it remembers all of the definitions made by the first program. The second program can then go ahead and use the defined objects. It is very important, though, to make the beginning of the second program different from the beginning of a normal (nonchained) program. First of all, it can not reload the GrafExpress system as this will destroy some of GrafExpress' internal variables. Secondly, you should not use an INIT instruction in the second program. GrafExpress was initialized in the first program and reinitializing it would erase all of the definitions created by the first program. One thing that should <u>not</u> be changed or omitted is the linking of the command string. When the second program is chained, the command string is destroyed. When it is recreated (by **C$=""**) it will most likely be created in a different location than it was the first time, so it needs to be linked again. Likewise, GE needs to be set to 20000 again.

Waveforms are shifted.

Because of the type of assembly language addressing used by the music routines, waveforms are played shifted 180 degrees from the definition. In order to compensate, waveforms defined with weights for the eight harmonics are defined with a 180 degree shift, so that when they are played the actual shift is 0 degrees. If you define waveforms byte by byte, you can go ahead and define them with 0 degrees shift and have them played with 180 degrees shift -- you really can't tell the difference when you hear it. However, if you want to use a waveform/envelope buffer that has been defined with the eight harmonic weights and use it as an envelope (without first redefining it) you should keep the 180 degree shift in mind. (Envelopes are not shifted at all when they are played.)

Why this even/odd business with sprites and windows?

Computers use binary code to remember information such as screen data. Just as our decimal system makes it easy to work with powers of ten (57x100 = 5700 is a very simple problem) the computer's binary system makes it easy to work with powers of two (1,2,4,8,16,32 etc.). Another consideration is how the binary digits are grouped, in the case of the CoCo it is groups of 8 (called bytes). 16 color pixels require exactly 4 binary digits (2 to the 4th power equals 16). This means exactly two pixels fit in every byte. The left pixel of every byte is even

and the right pixel is odd. It is much easier and faster to handle things by bytes than by half bytes so windows, pictures and sprites are handled by bytes which requires all left edges to be displayed on even pixels and all right edges to be displayed on odd pixels. With 256 colors, each pixel is exactly one byte which is why GE256 does not require left or right edges to be even or odd.

[FAST] option for SHOW

Color monitors work by shooting three electron beams at a phosphorous screen which gives off light when hit by a beam. The beams scan from left to right turned on, then back to the left side turned off. The next left to right scan is done one line lower and so forth until the whole screen is drawn. Then the beams are turned off and moved to the top left corner to start again. What the command SHOW does is to wait until the beams are returning to the top left corner before it allows the previously displayed screen to be changed. This prevents changes in progress from being displayed (changes displayed in progress appear as flicker). Of course, it does slow things down a little because everything comes to a halt while the computer waits for the beam to go to the right place. It is possible to have faster flickerless animation by using the FAST option, but only if you have enough memory to use three screens instead of two and limit yourself to sticky sprites. Nonsticky sprites make use of two buffers for restoration purposes, and only two. Three would be needed for nonflickering, nonsticky sprites.

How do nonsticky sprites work?

When you use the SHOW command (or the ! command), GrafExpress updates all of the sprites. First it does a save phase. Any areas of the screen where there will be a nonsticky sprite are saved in temporary buffers. Then all of the sprites are drawn. After the sprites are all drawn, then the screen is actually displayed (this is why the sprites don't flicker, you never see them actually being drawn). The SHOW command isn't finished though. It needs to restore the screen that used to be the display screen by getting rid of all of the pixels on it that were parts of nonsticky sprites. It does this by copying the areas back that it had already saved in the temporary buffers. (But these are not the same temporary buffers already mentioned, those are for the newly displayed screen. These are a second set of buffers for the previously displayed screen.) If you don't quite get exactly how this creates the effect of nonsticky sprites, that's quite all right. The main thing to notice is that nonsticky sprites are slower (they require time spent for saving and restoring screen areas) and they require more memory (for the buffers) so it is often better to use sticky sprites where that is possible.

How many bytes should I reserve for musical subsections?

The musical subsection buffer needs to be big enough to contain the definitions for all of the musical subsections. One easy way to set its size is to start with a small size (like 100 bytes) and increase it whenever you run out of room (when you get a music buffer full error). If you want to try to calculate exactly how big it needs to be then you need to use the size of the subsection definition operations. Musical subsection calls and tempo settings use two bytes each. Closing a subsection (done automatically) requires one byte each time. All other operations (setting envelopes, volumes, waveforms, synchronizing and play commands) use five bytes each.

Errors and Explanations

*BAD # ERROR:
A number was less than -32000 or greater than 32000.

*BAD PIC DEF ERROR:
An attempt to load a picture from the low resolution graphics area failed. Either a file was not first loaded from disk, the file is not a valid picture file, or there is a disk error.

*FUNCTION ERROR:
A number was too big or too small for the command used. If this happens while redefining a picture, it can mean that the new definition is bigger than the original definition.

*MUSIC BUFFER FULL ERROR:
The music buffer size as declared by INIT is not big enough to hold all of the musical subsection definition commands being used.

*MUSIC SUB N.D. ERROR:
An attempt was made to call with PLAY *sub#* or SUB *sub#* a musical subsection that was not defined with SUB *sub#* ON.

*OFF SCREEN ERROR:
An attempt was made to define a window partially or completely off of the screen.

*OUT OF MEMORY ERROR:
There is not enough memory to allocate all of the screens, sprites etc. This error can also occur if an attempt is made to save a picture without having previously issued the BASIC command **PCLEAR 6**.

*OUT OF WINDOW ERROR:
An x,y coordinate is outside of the current window.

*PIC N.D. ERROR:
An attempt was made to move a picture or define a sprite with a picture that was not defined.

*PIC N.I. ERROR:
An attempt was made to define or move a picture or define a sprite with a picture that was not declared by INIT.

*SCREEN N.I. ERROR:
An attempt was made to SHOW, COPY FROM or WORK on a screen that was not declared by INIT.

*SPRITE N.D. ERROR:

An attempt was made to turn on, move, or change the stickiness of a sprite that was never defined with a picture number.

*SPRITE N.I. ERROR:

An attempt was made to define or use a sprite not declared by INIT.

*STACK ERROR:

Substring execution or musical subsection calls are nested too deep. This error is often caused by a string or subsection directly or indirectly calling itself.

*SYNTAX ERROR:

An illegal combination of tokens, values and punctuation was issued. That is, the attempted command does not fit the syntax of any valid GrafExpress command (see **Command Summary** for a list of valid commands).

*VARIABLE N.D. ERROR:

A variable was used which was not previously defined. Variables must be defined (used in the BASIC program) before they can be used by GrafExpress. This error can also occur if you forget to leave a space between a variable name and a token or if you misspell a token.

*WAVE/ENV N.I. ERROR:

An attempt was made to use or define a waveform or envelope not declared by INIT.

*WINDOW N.I. ERROR:

An attempt was made to define, move, make current, or change the on/off state of a window that was not declared by INIT.

Abbreviations
N.D. = not defined
N.I. = not initialized

Command Summary

Drawing commands

> *xcoord,ycoord*
> BORDER *color*
> BOX/LINE/RECT [*xstart,ystart..*] *xend,yend*
> CIRCLE *radius* > *xcoord,ycoord*
> COLOR/= *foregroundcolor* [,*backgroundcolor*]
> *variable* = COLOR/= *xcoord,ycoord*
> FILL *xcoord,ycoord*
> SET *xcoord,ycoord*

Miscellaneous commands

> *variable$*
> FAST
> *variable* = HJOY *joystick#*
> INIT *mem,xres,yres,#pics,#scrns,#sprts,#wnds,musicmem,#waves*
> SLOW

Music commands

> ENV *env#* = *decayrate*
> INIT PLAY
> PLAY *sub#*
> SUB *sub#* ON
> WAVE *wave#* = *weight1,wt2,wt3,wt4,wt5,wt6,wt7,wt8*
> WAVE *wave#* (*index*) = *value*

Musical subsection definition commands

> ENV *env1,env2,env3,env4*
> PLAY *freq1 freq2 freq3 freq4* [,*dur*]
> SUB *sub#*
> SYNC *flag1 flag2 flag3 flag4*
> TEMPO *tempo*
> VOL *vol1,vol2,vol3,vol4*
> WAVE *wave1,wave2,wave3,wave4*

Picture commands

> LOAD PIC *pic#*
> OFF *pic#* = *xoffset,yoffset*
> PIC *pic#* = *xstart,ystart..xend,yend* [;*transcolor*]
> PIC *pic#* = RECT *width,height*
> PIC *pic#* > *xworldcoord,yworldcoord*
> *variable* = SAVE PIC *pic#*

Screen commands

[FAST] SHOW [*screen#*]
WORK [*screen#*]
!

Sprite commands

variable = *sprite#* HIT *spritelow* [..*spritehigh*]
SPRITE/* *spritelow* [..*spritehigh*] = *pic#*
SPRITE/* *spritelow* [..*spritehigh*] > *xworldcoord,yworldcoord*
SPRITE/* *spritelow* [..*spritehigh*] ON/OFF
SPRITE/* *spritelow* [..*spritehigh*] [NON] STICKY

Text commands

ECHO ON/OFF
FONT *width,height*
MIX ON/OFF

Window commands

CLW
COPY *sourcewnd* [[FROM *sourcescreen*] TO *destwnd*]
WINDOW/& *wnd#*
[SUB] WINDOW/& *wnd#* = *xstart,ystart..xend,yend*
WINDOW/& *wnd#* > *xworldcoord,yworldcoord*
WINDOW/& *wnd#* ON/OFF

Valid Ranges for Command Parameters

Circles
x/y coord: -32000..32000
radius: 1..32000

Colors
border: 0..63
screen: 0..15 (0..255*n)

Coordinates
screen: 0,0.. screen width-1, screen height-1
window: 0,0.. window width-1, window height-1
x/y worldcoord: -32000..32000

Font size
width: 6 or 8 (3 or 4*)
height: 8..12

Initialization

memory: 128 or 512
x-resolution: 128, 160, 256 or 320 (128 or 160 only*)
y-resolution: 192, 200 or 225
pictures: 0..255
screens: 1..255
sprites: 0..255
windows: 1..255
waveform buffers: 0..30
subsections size: 0..8000

Music

decay rate: 0..32000
duration: 1..250
frequency: -96..96
subsections: 1..30
tempo: 1..255
volume: 0..255
waveform index: 0..255
waveform weights: 0..255

Picture

x/y offset: -100..100
width and height: 1..100 (width: 1..50, height: 1..100*)
size (in pixels): 1..8000 if defined with a transparency color,
1..10000 otherwise
(1..4000 if defined with a transparency color,
1..5000 otherwise*)

*These parameters refer to GrafExpress 256, all other parameters are the same for both GrafExpress 16 and GrafExpress 256

Glossary

Border -- The area of your monitor which surrounds the image of the screen.

Buffer -- An area of memory that can hold information about something (such as an envelope or a sprite).

Byte -- A unit of computer memory. Each byte can take on one of 256 different values (usually 0..255).

Collision -- A collision is what happens when the solid parts of two or more sprites (which are turned on) occupy the same position in spriteland. Collisions can only be detected with the HIT command.

Decay -- A decrease in loudness while a note is sustained.

Decay Rate -- The speed at which a note decays (gets softer).

Display Screen -- The screen which is currently displayed on the monitor. The display screen can be set with either the SHOW or the ! command.

Double Speed -- Technically, it is when the computer is running at 1.79 MHz, twice as fast as the default single speed mode. The double speed mode (called by the command FAST) is preferred when graphics are used because it makes them faster and smoother. Disk operations, however, are not reliable in double speed and should be performed in single speed (called by the command SLOW).

Duration -- The length of a note relative to other notes.

Envelope -- The behavior of the loudness as a note is sustained. This system represents an envelope with a buffer of 256 bytes.

Frequency -- The pitch of a note, or how high or low it sounds.

Harmonic -- A component of a note's waveform. The first harmonic (the fundamental) sounds at the notes frequency. The second sounds at twice the frequency and the third at three times the frequency and so forth.

Musical Subsection -- A group of musical commands which can be defined or executed.

Offset -- The displacement added to a picture's given location to determine where it is drawn or displayed.

Pen -- A place holder for a graphics position. It remembers the last window location accessed.

It is never displayed.

Picture -- An object with a shape and appearance. Pictures can be drawn and/or assigned to sprites.

Pixel -- The smallest area of a screen or window which can have its own color. In the high-resolution modes (screen width=256 or 320) pixels are approximately square and on the low-resolution modes (screen width=128 or 160) the pixels are approximately twice as wide as they are tall.

Rectangle Picture -- A picture which is completely solid and transparent, and is defined with the command PIC *pic#* = RECT *width,height*.

Screen -- A rectangular group of pixels which can be altered and displayed. When this manual refers to a screen it does not mean the front of your monitor, but instead an area of memory which can be displayed on your monitor.

Single Speed -- Single speed (.89 MHz) is called by the SLOW command. See Double Speed for more information.

Solidity -- A characteristic a picture can have. Parts of a picture are called solid if, when used by a sprite, those parts can collide with another sprite's picture (be detected with the HIT command).

Sprite -- An object with a shape, an appearance and a position which exists in spriteland and can be seen in a window which has its viewport capability turned on.

Spriteland -- The place where all sprites exist. It is not a part of any screen, rather it is a large plane accessible only with a window whose viewport capability is on.

Stickiness -- A characteristic a sprite can have. A sprite which is sticky can be moved, but where ever it goes a copy of its image will be stuck to the screen.

Tempo -- Overall speed of music.

Token -- Any group of all capital letters (such as SPRITE and COLOR) found in any of the command definitions.

Transparency -- A characteristic a picture can have. Parts of a picture are called transparent if those parts do not appear when the picture is drawn (or displayed as a sprite's image).

Viewport -- The function a window serves when it allows sprites in spriteland to appear on the screen.

Volume -- Overall loudness of a note.

Waveform -- The shape of a sound's vibration pattern which gives the sound its tone quality. This system represents a waveform with a buffer of 256 bytes.

Window -- An area of the screen in which graphics primitives can be used. Also an area in which sprites can be displayed.

Work Screen -- The screen where all graphics commands are directed. The work screen can be set with either the WORK or the ! commands.

SECTION THREE: GrafExpress and Assembly Language

All of the commands in this system that can be called from BASIC can also be accessed from assembly language (with the exception of substring execution). In order to do this, your assembly language program needs to include four parts.

The first part is the definition. This part defines the system calls so that you can use them in your program. Although it can be placed anywhere in your program, it is best to put it at the beginning to decrease the number of forward references (too many of which can cause problems with one pass assemblers). The definition is included in the file called GE.DEF. (There are some differences in the way assembly language programs must be written between various assemblers: You may need to make some small changes to conform to the syntax of your own assembler.)

```
* part 1: Definition; location: anywhere...
INSTAL EQU 20033    * set the label INSTAL equal to 20033
MVPEN  EQU 12544
BORDER EQU 12545
BOX    EQU 12546
* (continues with BOXTO through GETWND)
```

The second part is the parameter list. When you call a command such as **SET 20,40** the first parameter is 20 and parameter number two is 40. For a command like **X=COLOR 42,73** the first parameter is X (the result is returned in the first parameter). The second and third parameters are 42 and 73 respectively. The parameters should be placed in a data storage area (not in the program where they might be executed). They need to be kept together and in order. The parameter list is also included in GE.DEF.

-- Hint: Putting the parameter list in the direct page will speed up your program and save memory. Just remember to set the DP register before trying to use the direct page!

```
* part 2: Parameter list; location: Data storage area...
PRM1   RMB 2        * reserve two (memory) bytes for parameter #1
PRM2   RMB 2
* (continues with PRM3 through PRM9)
```

The third part is installation. Before you try to use any system calls, you need to provide the system with some information. The first thing you need to give it is where you put the parameter list. This is so that when you use a system call, it knows where to find the values you put into the parameters. To do this the X register should be set equal to the location of parameter 1 when INSTAL is called. The second thing you need to set is how the system will handle errors (see **Notes on Debugging** for information on the **BAD # ERROR**). One way to handle errors is to have the system do the same thing it does when it's called from BASIC: Print the error message and line number and stop the program. However, this can only be done if your assembly language program is being called from BASIC and you are not altering BASIC with your program (there needs to be something for the system to return to). To choose this method, the U register should be set to 0 when INSTAL is called. The second method is to provide your own error routine. If your program is almost entirely in assembly language this

is probably the best method. To do this U should be set equal to the location of your error handling routine when INSTAL is called. Then, if (when) an error occurs, instead of trying to return to BASIC the system will go to your error handling routine with the one byte error code and a two byte error address on the stack (if **PULS A,X** is the first command of your error handling routine, it would move the error code to the A register and the location of the system call that caused the error to register X).

Although GrafExpress allows you to have an error handling routine, you should not simply continue giving commands after an error has occurred. Many errors are not caught until part way through a command's execution, so it is possible for system data structures to be left in invalid states when an error occurs. There are two common types of error handling routines. The first is usually used for a project in the works. It should save the address where the error occurred in a place where you can find it (to help with debugging) and should then either execute the ERROR system call with the error code in the A register if BASIC is still intact, or it should save the error code just as it did the error address and then abort execution somehow. For a program that is in a completed form and may be used by people other than the original programmer, you may want to just have the program print the message FATAL ERROR # along with the error code and simply go to an endless loop. If you want your program to try to recover from the error, then the first system call it must execute is INIT in order to get the system's data structures back in order (not doing so can result in a system crash!).

```
* part 3: Installation; location: Before executing any system calls...
        LDX #PRM1     * tell system where parameters are located
        LDU #ERR      * tell system where to go if an error occurs
        JSR INSTAL    * install the system
```

The fourth part is execution. The first part of execution is to set the parameters to the desired values. Parameters that are not used for the system call in question can be just left alone. To invoke the system call, the pseudo operation FDB is followed by the name of the system call. Execution of system calls does not change the values of any of the parameters (unless the system call is one that returns a value, see below) or registers (except the Program Counter will have increased by two so that execution continues with the command following the system call). For example, the command **SET 20,40** could be implemented in assembly language with:

```
* part 4: Execution; location: Anywhere after installation...
        LDD #20       * load register D with the x coordinate
        STD PRM1      * put it in parameter 1
        LDD #40       * do the same for the y coordinate
        STD PRM2
        FDB SET       * execute system call SET
        * (rest of program continues here)
```

System calls that return a value put it in parameter one and the input to the call starts with parameter two. For example, the command **register A = COLOR 40,27** could be implemented as:

```
        LDD #40
        STD PRM2      * put x coordinate in parameter 2
        LDD #27
```

```
        STD PRM3        * and y coordinate in parameter 3
        FDB GETCLR      * execute system call GETCLR
        LDA PRM1+1      * load register A with result
```

Notice that LDA PRM1+1 is used because the answer is a whole word and only the low order byte was wanted. If LDA PRM1 was used, register A would always be set to zero no matter what the color of pixel 40,27. In general, an offset of one should be used with one byte registers such as A and B and no offset should be used with two byte registers such as D, X and Y. Also, if storing a one byte register into (the second byte of) a parameter, it is up to you to make sure that the first byte of the parameter is zero.

Additional Commands

ALLOC

This system call allows you to allocate blocks of memory. GrafExpress uses all of the memory outside of the normal 64k (unless you have a 512k machine and tell GrafExpress that you only have 128k [with the INIT command] in which case the lower 384k is available for your use). On calling ALLOC the desired size (in bytes) of the memory block should be in the D register. Blocks can only be up to 8k (8192 bytes) in size (any larger request will generate a FUNCTION ERROR). After the system call is finished, the A register will contain the number of the first (physical) page in which the memory block resides, and the second page of a memory block is always the next physical page (some memory blocks will fit completely inside the first page, but even a request for just two bytes could possibly go over a page break). The X register will point to the beginning of the memory block within the first page. Any call to ALLOC clears the sprite buffers used for nonsticky sprites just the same way as defining a new picture does.

Suppose you wanted to allocate 1000 bytes of memory, and clear it to all zeros. Your code might look like this:

```
        LDD #1000       * Ask GrafExpress for 1000 bytes.
        FDB ALLOC
        STA PAGE        * Remember the page# and address for
        STX ADDR        *   future access of the memory block.
        STA $FFA0       * Map the first physical page of the block
                        *   into logical page 0.
        INCA            * Map the second physical page of the block
        STA $FFA1       *   into logical page 1.
        LDD #0          * Use D as the index and
LOOP    CLR D,X         *   X as the base address.
        ADDD #1
        CMPD #999       * Run the index from 0 to 999.
        BLS LOOP
        LDA #56         * Change the first two logical pages
        STA $FFA0       *   back to what they were before.
        INCA
        STA $FFA1
```

However, using the memory management unit (MMU) is not always so easy. First of all, you may need to mask the interrupts to prevent them from accessing those logical pages which no longer reference the same physical pages as the interrupts are expecting. Also, the above code will only work if the code itself is located outside of the first two logical pages (somewhere after 16384). If it were located within the first two pages it would map itself out of memory

and the CPU would instantaneously start executing the GrafExpress data area (CRASH)! If the code had to be located in the first two pages, then you would have to map the data somewhere else. In that case you need to change X by an appropriate value so that it points into the actual logical pages that you use. For example, if you decided to use logical pages 4 and 5 ($FFA4 and $FFA5) you could use an LEAX 32768,X before using X as the base register (logical page 4 starts at 32768). The need to have a good familiarity with your computer's service manual (especially the section on memory management) before trying to deal with the MMU can not be overemphasized.

ECHO

If BASIC is kept intact, then when you want to print text to the graphics screens you can use the system call **ECHON** and then output your characters by using the standard ROM subroutine CHROUT by putting the character's ASCII code into register A and doing a **JSR [$A002]**. However, not only must BASIC and its variables be intact, but it must be running in all RAM mode. If you do not do anything to change it then it is already doing so. If you are using the RAM normally occupied by BASIC for something else then do not call ECHON! It will attempt to modify the memory as if BASIC was there. Instead, just use the system call **ECHO** every time you would normally use **JSR [$A002]**. ECHO does not have to be turned on with ECHON and it does not use any of BASIC's memory or subroutines.

-- Hint: The system call ECHO works whether BASIC is intact or not, and a call to ECHO is faster and uses less memory than a JSR [$A002] so it is probably a good idea to just always use ECHO.

ERROR

Prints the error message (and the number of the line which called the machine language program, if any) corresponding to the error code in register A. This system call should only be used if BASIC is intact.

GETWND

Gets the starting position in memory of the current window. It returns the number of the first physical page of memory of the current screen in the A register. The rest of the screen is allocated in consecutive pages (the number of pages depends on the size of the screen). It returns the index to the actual window in the X register. Proper use of this system call requires a deal of experience with the MMU (for the same reasons as the ALLOC system call). Examples of this system call in use can be found in the file ART.ASM.

UNINST

Uninstalls the GrafExpress system. This should be called after your assembly language program is finished using the system. It restores the SWI vector.

Notes on Debugging

The real key to debugging in assembly language is to avoid having to do it in the first place. The assembly language programmer's best tool for accomplishing this is BASIC. Try to write your programs in BASIC first, and then switch them to assembly language in small manageable

sections.

If you are using an assembler that has a debugger that breaks on SWI, then you can debug your programs even though it is not possible to load and use the GrafExpress system with most debuggers. The first byte of every system call is the SWI instruction so most assemblers will just stop when ever they hit a system call. You can then examine the progress of your program and restart execution at two bytes after the location where it stopped. This can be very time consuming in a large program when you do not have any idea of where a program flaw is located, but if you can narrow down its location, then this method can be very useful. If you are going to do this, then you need to provide a dummy (do nothing) routine for INSTAL such as

```
INSTAL   RTS              * this is the whole dummy routine
```

You also need to remove (or turn into a comment) the definition of INSTAL provided in the GE.DEF file. After you are finished debugging, remove the dummy INSTAL routine and replace the real one before final assembly of your program.

Many debuggers also use SWI to implement single stepping. If yours does, do not try to single step through the JSR INSTAL instruction as that would enable GrafExpress' use of the SWI and it would try to interpret the second half of the following command as a system call. Most assemblers will not allow you to single step through a system call.

One advantage of having the GrafExpress system installed is that it often turns what would have been an all out machine language crash into a simple error. If something happens such as your assembly language program writes over itself, the stack overwrites your program, or your program jumps off into space (a data area instead of a program area), then there is a good chance that GrafExpress will stop it before too much damage is done. This happens when the CPU tries to execute a piece of data starting with a hex $3F which is the code for SWI. Now if that byte is not followed by a valid system call (which is more than half of the time) then the system will catch it and flag it as a syntax error. Unfortunately this does not do anything to prevent the infamous endless loop and other types of nondestructive errors, so as usual, just try to avoid making those errors.

The BAD # ERROR is only checked for at interpretation time (while a command string is being interpreted). This error condition is not checked for system calls since they are direct calls (not interpreted). This only affects commands where one or more of the parameters have a range with an upper limit of 32000. If you give a number with an absolute value greater than 32000, the system may react unpredictably. (The BAD # ERROR check was not added to the software interrupt routine because in the system's current form it would require a great deal of overhead in the form of eighteen compare instructions for every single system call.)

SysCall	Command Equivalent
Drawing commands	
MVPEN	> *xcoord,ycoord*
BORDER	BORDER *color*
BOX	BOX *xstart,ystart..xend,yend*
BOXTO	BOX *xend,yend*
CIRCLE	CIRCLE *radius* > *xcoord,ycoord*
COLOR	COLOR *foregroundcolor*
COLORS	COLOR *foregroundcolor,backgroundcolor*
FILL	FILL *xcoord,ycoord*
GETCLR	*variable* = COLOR *xcoord,ycoord*
LINE	LINE *xstart,ystart..xend,yend*
LINETO	LINE *xend,yend*
RECT	RECT *xstart,ystart..xend,yend*
RECTTO	RECT *xend,yend*
SET	SET *xcoord,ycoord*
Miscellaneous commands	
n.a.	*variable$*
FAST	FAST
HJOY	*variable* = HJOY *joystick#*
INIT	INIT *mem,xres,yres,#pics,#scrns,#sprts,#wnds,musicmem,#waves*
SLOW	SLOW
Music commands	
DFENV	ENV *env#* = *decayrate*
INPLAY	INIT PLAY
DOPLAY	PLAY *sub#*
SUBON	SUB *sub#* ON
DFWAVE	WAVE *wave#* = *wght1,wt2,wt3,wt4,wt5,wt6,wt7,wt8*
DFBYTE	WAVE *wave#* (*index*) = *value*
Musical subsection definition commands	
ENV	ENV *env1,env2,env3,env4*
PLAY	PLAY *freq1 freq2 freq3 freq4*
PLAYDR	PLAY *freq1 freq2 freq3 freq4,dur*
SUB	SUB *sub#*
SYNC	SYNC *flag1 flag2 flag3 flag4*
TEMPO	TEMPO *tempo*
VOL	VOL *vol1,vol2,vol3,vol4*
WAVE	WAVE *wave1,wave2,wave3,wave4*

Picture commands

LDPIC	LOAD PIC *pic#*
DFOFF	OFF *pic#* = *xoffset,yoffset*
DFPIC	PIC *pic#* = *xstart,ystart..xend,yend*
DFPICT	PIC *pic#* = *xstart,ystart..xend,yend;transcolor*
DFPICR	PIC *pic#* = RECT *width,height*
MVPIC	PIC *pic#* > *xworldcoord,yworldcoord*
SVPIC	*variable* = SAVE PIC *pic#*

Screen commands

SHOW	SHOW
FSHOW	FAST SHOW
SHOWS	SHOW *screen#*
FSHOWS	FAST SHOW *screen#*
WORK	WORK
WORKS	WORK *screen#*
SWITCH	!

Sprite commands

HIT	*variable* = *sprite#* HIT *spritelow*
HITR	*variable* = *sprite#* HIT *spritelow..spritehigh*
DFSP	SPRITE *spritelow* = *pic#*
DFSPR	SPRITE *spritelow..spritehigh* = *pic#*
MVSP	SPRITE *spritelow* > *xworldcoord,yworldcoord*
MVSPR	SPRITE *spritelow..spritehigh* > *xworldcoord,yworldcoord*
SPON	SPRITE *spritelow* ON
SPRON	SPRITE *spritelow..spritehigh* ON
SPOFF	SPRITE *spritelow* OFF
SPROFF	SPRITE *spritelow..spritehigh* OFF
SPST	SPRITE *spritelow* STICKY
SPRST	SPRITE *spritelow..spritehigh* STICKY
SPNST	SPRITE *spritelow* NONSTICKY
SPRNST	SPRITE *spritelow..spritehigh* NONSTICKY

Text commands

ECHON	ECHO ON
ECHOFF	ECHO OFF
FONT	FONT *width,height*
MIXON	MIX ON
MIXOFF	MIX OFF

Window commands

CLW	CLW
COPY	COPY *sourcewnd*

COPYTO COPY *sourcewnd* TO *destwnd*
COPYFR COPY *sourcewnd* FROM *sourcescreen* TO *destwnd*
WND WINDOW *wnd#*
DFWND WINDOW *wnd# = xstart,ystart..xend,yend*
DFSWND SUB WINDOW *wnd# = xstart,ystart..xend,yend*
MVWND WINDOW *wnd# > xworldcoord,yworldcoord*
WNDON WINDOW *wnd#* ON
WNDOFF WINDOW *wnd#* OFF

Additional commands

	Input	Output (registers)
ALLOC	D	A,X
ECHO	A	
ERROR	A	
GETWND		A,X
UNINST		

Commonly used abbreviations

DF = define
MV = move
PIC = picture
SP = sprite
SPR = sprite range
WND = window

Error codes

These are the codes returned on the stack when an error occurs during a system call and an error trapping routine is turned on. For explanations of each error, see the section called Errors and Explanations.

Code	Error
1	SYNTAX
2	OUT OF WINDOW
3	OFF SCREEN
4	FUNCTION
5	BAD #
6	OUT OF MEMORY
7	STACK
8	SCREEN N.I.
9	WINDOW N.I.
10	PIC N.I.
11	PIC N.D.
12	BAD PIC DEF
13	SPRITE N.I.
14	SPRITE N.D.
15	MUSIC SUB N.D.
16	WAVE/ENV N.I.
17	MUSIC BUFFER FULL
18	VARIABLE N.D.

SECTION FOUR: Included Applications

Before using any of the included applications, make sure you have followed the instructions in the section called **Before Getting Started.** The necessary files listed for each program should be on the disk in the default drive when you run that program.

If you are not using an RGB monitor for the 16 color programs (INTRO, PIC.MKR and WAVE.MKR), then you need to deviate from the instructions just a little. When the instructions tell you to RUN a program (such as type **RUN "INTRO"** and press **ENTER**) you should instead LOAD the program (for example type **LOAD "INTRO"** and press **ENTER**) and then type **MERGE "CMP"** and press **ENTER** and then type **RUN** and press **ENTER.** If you like, you can save the program after merging the file "CMP" so that in the future you can just run that program instead of going through the LOAD/MERGE/RUN sequence every time. (Do not try to merge "CMP" with ArtExpress. It is a 256 color program and is already set up to run on composite video.)

The first time GrafExpress is called, it displays a copyright screen. Just hit a key or button to make it go away and continue with the program.

Introducing GrafExpress

This program is a collection of short demonstrations of some of the capabilities of the GrafExpress system.

Using the program

Put your copy of the GrafExpress disk in the default disk drive, type **RUN "INTRO"** and press **ENTER.** For this program you need a standard joystick or mouse in the right joystick port. After the tone sounds and you have seen enough of a particular demonstration, press any key (except BREAK) to continue with the next demonstration. On the last demonstration (the sprite demo) the box changes color when you hit it with the animated face. Try placing the face below the box so that its antennae hit the box every seventh frame.

Necessary files: INTRO.BAS, GE16.BIN, FACE1.PIC, FACE2.PIC, FACE3.PIC

Inside the program

This program is a good source of ideas for how to accomplish certain effects. The text that is written to the screen at run-time can be used as documentation for the program. For example, if you wanted to see how the program managed to do stretchy lines then list through the program looking for the print statement that talks about stretchy lines. The code that does stretchy lines is located immediately following the text (although the necessary INIT statement and window definitions may be located above the print statement).

Pic-Maker 1.0

This program is a simple picture editor. It creates/saves/loads and edits pictures with tools such as pixels, straight lines, rectangles and boxes.

Using the program

Put your copy of the GrafExpress disk in the default disk drive, type RUN "PIC.MKR" and press ENTER to edit pictures with approximately square pixels (pictures intended to be used with the 256 or 320 width screens). If you want to edit pictures with elongated pixels (pictures intended to be used with the 128 or 160 width screens) then instead, type RUN "LORESPIC.MKR" and press ENTER. For these programs you need a standard joystick or mouse in the right joystick port.

To select a color, click (move the arrow with the joystick and press the button) on one of the color boxes at the top of the screen. The current color is displayed in the box to the right of the text "Current color =". To select a drawing tool, click on the tool's icon (the box with the picture of the tool on it). The currently selected tool is the one with a small white box located under its icon. In order to use a tool, move the cursor (arrow) to the zoom window (the window to the left with the grid lines) and press the joystick button. The window to the right is the correctly sized picture. If using lines, rectangles, or boxes the first click sets a point on the first end or corner and sounds a tone and a second click is used to mark the opposite end or corner. Lines can only be horizontal or vertical and the program will do a double beep if you try to make any other type of line.

To use any of the text icons (SAVE/LOAD/KILL/DIR/QUIT) just click on them. The SAVE/LOAD and KILL functions will request a file name. If you choose these by accident, just press ENTER without entering a file name. Picture files created with this program require one granule of disk space. The DIR function prints the number of free granules and gives a directory listing of the default drive. QUIT exits the program after a YES/NO prompt.

Necessary files: PIC.MKR (or LORESPIC.MKR), GE16.BIN

Inside the program

Pic-Maker uses abbreviations (= for COLOR, & for WINDOW and * for SPRITE) where ever possible, which reduces the program's size by about 30 percent. This application was the earliest written of the applications, being written before the author had a lot of experience with the GrafExpress system (as the system had not even been completed when this program was started). The result is that some things were done "the hard way" which makes the program a little difficult to follow. However, you will probably find the subroutines located at the end of the program fairly simple to follow and very educational:

Line 1030: Open dialogue box

Line 1050: Close dialogue box

These routines open and close the window used by the SAVE/LOAD/KILL and QUIT options.

Line 1070: Update cursor

This moves the arrow to where ever the joystick points.

Line 1090: Copy current screen

This routine is used when graphics drawn on one screen needs to be duplicated to another screen. Notice that the cursor is shut off and the ! (switch) command is used to make it stop being displayed so that it is not also copied from the source screen (which is the display screen).

Line 1110: Get name

Inputs a file name for the SAVE/LOAD and KILL options.

Line 1130: Do box

This is used for drawing the light colored left and top edges and the dark colored bottom and right edges which outline many rectangular areas on the screen.

The ambitious programmer might want to try to change the low-resolution version from using FONT 8,8 to using FONT 6,8 which would look a lot better on the low-resolution screen. This change could be as simple as just adding a font command, but could also include recentering text in windows and text icons or even resizing the windows and text icons. Other things that you might want to add would be error checking for the disk routines (so that the program doesn't crash every time there is a disk error) or an option to save the picture using a transparency color.

The assembly language programmer could change parts or all of either version of this program over to assembly language. A good place to start would be the box routine which for large boxes is terribly slow in BASIC. The routine (line 550) uses the dot draw routine (line 1020) and some of the variables it uses are TX,TY (temporary storage for X and Y which hold the end corner of the box), OX,OY (the start corner of the box) and CC (current color).

Note: In line 880 Pic-Maker uses the function FREE(PEEK(2394)) to return the number of free granules in the current drive. If you find that this is not compatible with your disk system, change it to FREE(0). That will work on all disk systems, but will always print the number of free granules in drive 0 (which is not necessarily the current drive).

Wave-Maker 1.0

This program is a waveform editor. It allows you to change the weights of the harmonics, see what the waveform looks like and hear what it sounds like. It also allows the adjustment of the frequency, decay, tempo and duration.

Using the program

Put your copy of the GrafExpress disk in the default disk drive, type **RUN "WAVE.MKR"** and press **ENTER**. For this program you need a standard joystick or mouse in the right joystick port. Wave-Maker does not have a quit option, so you must press **BREAK** when you want to exit the program.

The slide bars on the left half of the screen control the weights of each of the eight harmonics. The numbers on the far left are the exact values of each weight. A weight can be changed by placing the cursor (arrow) on a slide bar where you want the indicator to move to and pressing the button. For finer control the arrows on either end of the slide bars can be clicked on to change the values by 1.

The controls for frequency, decay, tempo and duration all work the same as each other. Clicking on the up/down arrows next to each value will change the value by one. Holding the joystick button down while on the button between the arrows allows you to quickly change the values by moving the joystick up and down.

Clicking on the single note plays the note with a volume of 255. This can also be accomplished by pressing the second button of a two button joystick without the need to point to the single note. Clicking on the three notes plays a major chord with all three notes each having a volume of 85. The base of the chord is set to the frequency you have set and if you get a double beep when you press on the three notes it means that the top note of the chord (which is 7 greater than the base frequency) is too high.

When you click on the MAX icon, the program tries to create a waveform with the largest amplitude that it can while still maintaining the same ratios between the harmonic weights. Maintaining the same ratios keeps the tone quality the same, and maximizing the amplitude creates the best signal to noise ratio.

Foldover distortion is when waveform values get too large and it results in a very rough tone quality. (Of course, for certain sound effects you may actually want foldover distortion.) You can tell that a waveform has foldover distortion when the waveform runs off the top of the display window and continues at the bottom. To get rid of this you need to either decrease one or more of the weights or click on the MAX function. (Sometimes increasing a nondominant weight will actually lower the overall amplitude because of cancellation.)

Necessary files: WAVE.MKR, WAVEMKR.BIN, CHORD.PIC, GE16.BIN

Inside the program

Unlike Pic-Maker, Wave-Maker does not use any abbreviations at all which makes it somewhat larger, but more readable.

Sprite number 1 is the arrow and sprite number 2 is a one dot sprite which is the tip of the arrow and is used for precise checking of what the arrow points to. Two different methods are

employed by this program to detect clicks on icons. The first method creates the icons as pictures, assigns those pictures to sticky sprites and sticks the sprites to the screen in the right places. Then those sprites are turned off so that they do not slow down screen updating. When the program needs to do collision checking, it turns those sprites on, checks for a collision, and turns them back off. The second method also creates the icons as pictures, but draws the pictures directly on the screen. For collision checking it uses a bunch of sprites defined with rectangle pictures whose positions correspond to the locations of the pictures on the screen. This way these sprites do not need to be turned on and off in order to avoid drawing them every time. Avoiding the unnecessary drawing of sprites does not affect how the program functions and is not required, but for this program (which could have up to 41 sprites on the screen at once) doing so keeps the program's response from becoming too sluggish. The slide bar indicators are normal sprites (they are not simply drawn directly onto the screen), which is necessary because they need to be able to move without destroying the background graphics.

Of interest to assembly language programmers is the routine WAVEMKR.BIN which updates the picture of the waveform. The source code is included in the file WAVEMKR.ASM and is thoroughly documented. One thing that could be done with this routine is to decrease its size and increase its speed by grouping its variables together and using direct page addressing. Some of the commands located in the BASIC part of the program just before WAVEMKR.BIN is called (such as setting the default window and clearing it) could be moved to the assembly language routine. Be careful that the label ENDMRK does not become greater than 20000 (the start address of GrafExpress). If it does, you will need to change the ORG directive (which marks the beginning of the memory used by WAVEMKR.BIN) to a smaller number. You will also have to change the BASIC program by allocating more space with the CLEAR instruction and changing the values of the variables WT (weights table),MX (maximum amplitude) and ST (start of WAVEMKR.BIN) to point to their new locations. You should also make sure that the sine wave location remains the same. After assembling WAVEMKR.ASM you will need to run the BASIC program SINEWAVE.MKR which creates the sine wave table.

ArtExpress 1.0

This program is a 256 color art program. It lets you mix and choose your colors, and place them on the screen using stretchy lines, rectangles, boxes and circles. It also gives you free hand drawing and area fills. A zoom function is provided for precision pixel editing. Your art can also be saved and loaded. (NOTE: This program, like any 256 color program, does not function properly on RGB monitors, and should *not* have the file CMP merged with it.)

Using the program

Put your copy of the GrafExpress disk in the default disk drive, type RUN "ART" and press ENTER. For this program you need a joystick or mouse connected to a hi-resolution joystick interface in the right joystick port.

The four shaded colors at the top-right corner of the screen are the four composite primaries. All 256 colors are just combinations of those four colors. The white boxes indicate how much of each is currently mixed in (they all start at 0). The box below the four shaded colors is the current color box and displays what the mixture looks like. There are three ways to select a color. The first method is to click on different shades of the four primaries to select the amount of each one to be added. The second method is to click on the mixed color box which will cause all 256 colors to be displayed, then you just click on the color you want. The third method is very necessary when you have this many colors. It allows you to lift a color off of the screen by pressing your second button or the letter "L".

Below the mixed color box are the drawing mode selection icons. The modes are freehand, line, rectangle and box. The mode that is currently selected is the one that looks like it is pressed down. In freehand you can press the mouse/joystick button in the drawing area to create dots, or you can drag (move the joystick or mouse while holding down the button) to draw lines and arcs freehand. The line, rectangle and box work by pressing the button where you want one end or corner and dragging to where you want the opposite end or corner.

You can make circles by moving the cursor to where you want the center to be, and then press "C". Then drag the mouse or joystick to expand the circle to the desired size, and release the button when finished. To do a fill, position the cursor to the desired location and press "F".

To use any of the text icons, just click on them. The SAVE, LOAD, and KILL functions will request a file name. If you choose these by accident, just press ENTER without entering a file name. Art from this program is saved as four picture files using 3 granules each for a total of 12 granules. Make sure that you have 12 free granules with the DIR function before trying to use SAVE.

The last function is zoom. When you click on its icon, a box will appear in the drawing area. Move this box to where you want to zoom in on, and press the button again. In zoom mode you can still change colors, but the only tool available is pixel editing. To leave zoom mode, click on the zoom icon again.

Necessary files: ART.BAS, ART.BIN, GE256.BIN

Inside the program

This program did the "easy way" some of the things that Pic-Maker did the "hard way".

Pic-Maker used BASIC subroutines for opening and closing the dialogue box, and the idea of a subroutine is good because these functions were needed by several different places in the program. However, BASIC subroutines have the disadvantage of requiring you to remember a line number -- a number which keeps changing every time you renumber your program. ArtExpress uses named subroutines to solve this problem. In lines 140..150 it defines DF$ as a draw frame subroutine, in lines 160..170 it defines OB$ as an open dialogue box routine, and in lines 180..190 it defines CB$ as a close dialogue box routine. The names are much easier to remember than BASIC line numbers.

Substrings were also used in another way to dramatically decrease size and required programming effort. If you look at lines 540..690 you will see that the stretchy lines, rectangles, boxes and circles all share a lot of code. This is done by putting the one command that is specific for each one in A$ and then using A$ as a substring from the main part.

The one thing that caused the most problems in writing this program was making it work on 128k machines. It would have been a lot easier to write if there were memory enough for 2 screens each for the main screen, a zoom screen, and a color selection screen. The program would have also been faster and smaller in size. The moral probably is, if you only have 128k it is still possible to write decent programs, but you can write even better ones easier if you have 512k.

For those of you who want to delve into the program, you will probably find the following descriptions useful:

```
Pictures
    1:  Cursor
    2:  Cursor hot spot (RECTangle picture)
    3:  Graphics icon (RECT)
    4:  Text icon (RECT)
    5:  Color marker
    6:  Depressed button
    7:  Save/Load picture
    8:  Mixed color box (RECT)
    9:  Cursor buffer
   10:  Zoom box

Sprites
    1:  Cursor
    2:  Cursor hot spot
 3..6:  Graphics icons
 7..12:  Text icons
13..16:  Color markers
   17:  Depressed button
   18:  Mixed color box

Windows
    1:  Whole screen
    2:  Dialogue box background (for frame)
    3:  Dialogue box
    4:  Draw area
    5:  Used for icon drawing
    6:  Zoom source window
```

Now you can learn a lesson from one of my mistakes. As I wrote the program, I found myself constantly looking up what a certain sprite number was or which picture number I

THIS PAGE

IS

MISSING

GrafExpress 2.0 is an complete graphics and music programming environment. From the beginner to the accomplished professional, you can use GrafExpress to create lightning fast arcade games, graphic applications and utilities, and windowing multimedia demonstrations! The GrafExpress package includes two incredible systems. GrafExpress 16 works on all monitor types and offers support in 12 graphic resolutions (from 128x192 to 320x225). GrafExpress 256 offers 6 resolutions (from 128x192 to 160x225 on a composite monitor) in an astounding 256 colors! Ever see a CoCo do that before? Both systems include standard graphics commands (CIRCLE, FILL, etc.) that blow away the competition. For example, the BOX command peaks out at over 2 MegaPixels/second; that's 300 times faster than BASIC! 255 separate spites of up to 100x100 pixels each are supported with window clipping and high-res pixel level collision checking. The 8-octave/4-voice music synthesizer has independant envelope, waveform, and volume controls, a 7+ KHz sampling rate, and much more. Other features include text/graphics mixing, different font sizes, fast window copying and scrolling, picture save/load, easy implementation from both BASIC and assembly language, multiple screen animation, and support for 128/512K, double speed, and the high-res joystick interface. The package also contains support programs that are worth the purchase price of GrafExpress alone! These include an introductory demo, a picture editor, a waveform editor, and an art program (that supports 256 colors!). GrafExpress also comes with a 50 page manual that fully explains all of its incredible features. If you do any graphics programming or simply want to see what your little CoCo is capable of, GrafExpress is a must! GrafExpress 2.0 requires a 128K CoCo 3 and disk drive. Joystick is recommended.

☞ **Warranty:** All of our products are sold on an as-is condition. They are guaranteed to load for one year, and Sundog Systems will replace any defective diskettes free of charge during this period. Sundog Systems specifically disclaims all other warranties, expressed or implied.

☞ **Publisher:** *SUNDOG SYSTEMS*
21 Edinburg Drive
Pittsburgh, PA 15235
(412) 372-5674