DATAMAX UV-1
Zgrass LESSONS

October 27, 1981

DATAMAX UV-1 Zgrass LESSONS

October 27, 1981

LESSON 0 READ ME FIRST

Zgrass is a graphics programming language. It is probably closer to BASIC than any other language, yet it is much more flexible and general than BASIC. The presumption in these lessons is that you already know how to program BASIC (at least BALLY BASIC) and are familiar with loops, IF's, GOTO's, variables, and so on, and are ready to learn what makes Zgrass tick. The essential differences between Zgrass and BASIC are:

1. Zgrass allows any number of programs and subroutines, each named, and they can run in series or parallel. BASIC has one unnamed program and a lot of GOSUB's.

2. Zgrass has an interactive full-screen editor. BASIC edits with line numbers.

3. Zgrass has good ways of passing arguments to subroutines; BASIC has none.

4. Zgrass can construct programs and run them with string manipulation features; BASIC cannot.

5. Zgrass has excellent debugging aids: single step, run-time listing, and error trapping; BASIC doesn't.

6. Zgrass has fast, advanced graphics commands; most BASICs use peek and poke.

7. BASIC has FOR/NEXT. Zgrass does without.

In order to learn Zgrass, you will have to explore it. Fortunately, this is not hard and is very rewarding. These lessons are to help you start exploring. They don't teach you how to program or write games, they just present the tools to you and encourage you to build your own. The first six lessons concentrate on defining the playing field. Pay close attention!

If you find a word being used that you do not understand, consult the Glossary. Once you get through the lessons, read the Glossary in detail. You'll find yourself understanding some of the esoterica. Some of the advanced features you may never use or understand and it may take awhile for you to see why some of the diversions from BASIC were necessary. Before long, however, you will find going back to BASIC unbearable.

Just to get you started, there's a test program called NB in the system. Press the red RST on the UV-1 front panel and answer Y. Then type the two letters NB and press the RETURN key. An image will appear. Type NB and press RETURN again. The image will undo itself. For more action, type NB.B and press RETURN. You can stop this by pressing the CTRL (called CONTROL on some keyboards) key, holding it down and simultaneously pressing the C key. Have fun!

End of Lesson 0.

LESSON 1 GETTING STARTED

When Zgrass first starts up, you see a '>' on the terminal screen. This is the "attention mark" and it means Zgrass is waiting for you to type something. To make sure it's listening, press the RETURN key. It should put an attention mark on the next line (if not, push the RST button on the UV-1 front panel). Whenever there is an attention mark, you can type a COMMAND.

Aside from graphics output, the primary means of communication from Zgrass to you is the PRINT COMMAND. It is your window into Zgrass. Whenever you what to find out what something will evaluate to, type in PRINT plus that thing. Separate PRINT and the thing by a single space and press the RETURN key to end the command. (The RETURN key performs the same function as the GO key in BALLY BASIC, you might observe.) Unless you've already pressed the RETURN key, mistakes can be corrected by typing RUB to erase and re-typing correctly. Try these:

```
        PRINT 5
        PRINT 5*5
        PRINT 5*2+2
        PRINT 5*(2+2)
        PRINT (5*2)+3
        PRINT 100/3
```

The above examples illustrate using Zgrass as an overweight pocket calculator. When doing arithmetic in Zgrass, you must observe the PRECEDENCE of OPERATORS. The OPERATORS above are +,* and /. To discover the other operators in Zgrass and their precedence, look up both OPERATOR and PRECEDENCE in the Glossary now.

Zgrass has some operators that don't exist in BASIC. The random number operator is the percent sign: %. It takes the two numbers on either side of it and yields (returns) a number randomly chosen between them. The lower bound is sometimes chosen but the upper never is, although it can get very close. Try:

```
    PRINT 1%5
```

several times. Notice that you get fractions.

Another operator in Zgrass is the ASSIGNMENT operator, "=". You can type:

```
    PRINT A=10%100
```

and the number randomly chosen between 10 and 100 will be printed and stored in VARIABLE A. Normally you do not want to see the printout everytime you store something in a variable so you leave out the PRINT:

```
    A=10%100
```

Zgrass does not like extraneous spaces, except at the beginning of a line, so

```
    A = 10%100
```

will generate ERROR #20. Spaces on the end of a line are tough to see because you can't tell where the NEXTLINE character is. If you press CTRL down and press Y at the same time (hereafter referred to as CTRL+Y) an "!" will be printed where the NEXTLINE's are so you can see them. Another CTRL+Y turns this feature off, so it is called a toggle after its similarity to toggle light switches.

In any event, you can always find out the current value of A by typing:

```
    PRINT A
```

The concept of an EXPRESSION is central to Zgrass. All the things you have typed following the PRINT's above are expressions. PRINT always gives the value of an expression. The smallest expression is a single number or variable and larger expressions are made up of smaller expressions glued together with operators. In fact, even "PRINT 10" is an expression, as is every legal thing you can type in ZGRASS besides CTRL characters. You can verify that "PRINT 10" is an expression by:

```
    PRINT PRINT 10
```

which will print a 10 then a 1. The one is the value of the expression "PRINT 10" which the leftmost PRINT gives. It prints a 1 because the PRINT command and all other commands which have nothing more meaningful to evaluate to give 1's to indicate "success".

Evaluating to a value is often referred to as "returning" a value depending on the context. In these lessons, we will talk about returning values, which should never be confused with the RETURN key. To avoid confusion between the RETURN command which returns values and the RETURN key, we normally call the character generated by the RETURN key a "NEXTLINE." That's why we say CTRL+Y puts "!"'s where NEXTLINES are instead of where RETURN's are. (Many people refer to NEXTLINES as "carriage returns" which makes no sense whatsoever on a cathode ray tube terminal.)

To drive home the point of returning values which PRINT prints out, try:

```
    PRINT PRINT PRINT 10
```

Any Zgrass command can have its expressions put in parentheses instead. Try:

```
    PRINT(10)
    PRINT(PRINT(PRINT(10)))
```

Note the lack of spaces. The 10 is called an "argument" by computer folk and is an indication that computer languages were developed by mathematicians, not social psychologists or artists. At any rate, arguments are always separated by commas:

```
    PRINT 10,20,30
    PRINT(10,20,30)
```

The parentheses are used to clarify the nesting. The only reason we do not require them always is that they are a pain to type all the time (if we did require parentheses, by the way, you would be able to type spaces anywhere, but we don't so you can't). People seem to have an inordinate amount of trouble accepting two different formats for enclosing arguments to commands. Please make sure you re-read the above few paragraphs until they are clear to you.

Try this last PRINT evaluation:

```
    PRINT(PRINT(10)+PRINT(5))
```

Each print inside the parentheses returns a 1 after printing its argument and the sum of the two 1's is printed by the leftmost PRINT which prints 2. (The 1 it returns is lost because it is not assigned or passed on to anything.)

There are, of course, more simple ways to arrive at one and one are two, but understanding the connections between arguments, commands and values returned is critical to developing a feel for zooming around in Zgrass and seeing how Zgrass is far more powerful than BASIC.

WHAT COMMANDS DO

In addition to printing 10 on the terminal, PRINT 10 returns a 1 to whatever called it because it had nothing better to return than "success." That's reasonable because it did its work as a side effect of returning that 1. Commands generally are interesting because of what they do rather than what they return. For example, type:

```
    HELP
```

Look at the last line. Now type (make sure there's no space between the parentheses):

```
    PRINT HELP()
```

and you'll see a 1 there. The point of this is that commands cause other things to happen besides returning a value. FUNCTIONs are commands that return a value but otherwize don't change anything. It is possible to have expressions which contain both commands and functions since the distinction is, in essence, artificial, but such constructions are confusing, especially to someone else trying to read your programs.

Functions are generally used as parts of complex expressions and were invented because once you get past about ten operators, you simultaneously run out of punctuation symbols and the capacity to remember what they do. Functions have names like commands (you are probably familiar with SIN, COS, TAN, LOG, and SQRT, for example) and you can create your own functions and commands in Zgrass quite easily. We'll show you how in the next lesson.

Meanwhile, let's do some graphics commands. Try:

```
    BOX 0,0,300,200,3
    BOX 75,0,20,100,2
    BOX -75,0,20,100,1
    BOX 0,0,200,10,0
```

The BOX command draws a filled-in rectangle defined by its arguments:

```
    BOX XCENTER,YCENTER,XSIZE,YSIZE,COLORMODE
```

X and Y are the horizontal and vertical coordinates and have the ranges:

```
    -160<X<159
    -100<Y<100
```

The last argument to BOX is the COLORMODE. There are lots of these, but for now, assume 0 is white, 1 is red, 2 is green and 3 is blue.

The POINT command allows you to draw points on the TV screen. An individual point (from now on called a "pixel" to avoid confusion with the POINT command) is sometimes hard to see, especially if you are using a regular TV. Try:

```
    BOX 0,0,320,200,3;POINT 100,20,0
```

Look up the POINT command if the arguments are not obvious to you.

While looking up the POINT command, you might have noticed a POINT function,too. When the COLORMODE is not specified, Zgrass assumes you want it returned to you as a value. The values range 0-3 if the coordinate is on-screen and -1 is returned if the pixel specified is off-screen. Try:

```
    CLEAR
    BOX 0,0,100,100,3
    BOX 0,0,50,50,2
    POINT 0,0,1
```

and:

```
    PRINT POI(0,0),POI(0,1),POI(30,-30),POI(200,200)
```

So, POINT used as a function only returns a value while POINT used as a command changes the screen

One last command for this lesson: CLEAR. CLEAR erases the TV screen. CLEAR.CRT (or CL.C for short) erases the terminal screen. The ".CRT" is called a switch. Switches are used to modify some system commands and are separately documented in the Glossary.

End of Lesson 1.

LESSON 2 WRITING MACROS

It's pretty difficult to find a programming language that doesn't deal with numbers coherently. However, the use and handling of alphanumeric text (called "character strings") so fundamental to our natural language communication, is, in fact, a clumsy add-on to most popular programming languages. Only the most primitive and ad-hoc constructs are available to the user in BASIC or FORTRAN, for example. Zgrass, on the other hand, uses character strings as its way of building and storing user programs so string manipulation is as much a part of the language as numerical computation.

STRINGs (look up the definition in the Glossary if it's a new concept to you) are defined much the same as numbers except you need to specify delimiters (special punctuation) which first, indicate that they are strings and second, say where they end. The string delimiters in Zgrass are:
    ",',[,],{ and }
the last four of which can be nested. Try:
```
    PRINT "HELLO!"
```
or
```
    PRINT [HELLO!]
```

To assign a string to a variable, type:
```
    A=[HELLO!]
```
and then try:
```
    PRINT A,A,A,A,A
```
Unlike in BASIC, you do not need to use a '$' to indicate a string variable; Zgrass is pretty good at figuring it out. You may use '$' in a name if you find it comfortable. For the time being, avoid names that start with a $ and have exactly two additional characters to eliminate possible conflict with DEVICE VARIABLE names (see LESSON 7 and the Glossary). Also, do not use $CHAR1 as a name, since that is used by the TEXT command.

Note that there are spaces between the HELLO!'s when printed as above. PRINT always puts a space after the thing it prints. To eliminate the spaces, try:
```
    PRINT A&A&A&A&A
```
The '&' is the concatenation operator and it creates a string expression.

As stated before, Zgrass stores your programs as strings. We call programs in Zgrass "MACROs." So, to create a macro, you assign the string that contains the commands to a variable. Try:

```
    Q=[BOX 0,0,100,100,2
    BOX 50,50,100,100,3]
```

(Note the plus sign (+) which appears when a multi-line string is being typed and Zgrass is waiting for you to type the matching delimiter.)

To run this macro, just type Q and press the RETURN key.

You should wonder why the letter Q was used instead of A. If you use A as a name above, you will get ERROR #39 (try it). The reason is very important and the result of a great design difference between Zgrass and other languages: abbreviation. Abbreviation allows you to interact with Zgrass faster than if you had to spell everything out all the time.

What you are doing in assigning the string to Q is creating a macro, as we said. You can also think of it as creating your own command named Q. There aren't any system commands starting with Q so Zgrass can tell right away that it is your command you want to run. Since Zgrass allows you to abbreviate, you will notice that H is enough for HELP, P for PRINT, etc. (HELP will tell you what the first command in each alphabetical grouping is and you can always get at it with its first letter.) Since H by itself always gets to HELP, you cannot have a macro named H and expect to be able to run it. You can have a string variable or number variable named H without any problems, but you can't run it if it's supposed to be a macro.

A few examples might help clarify:

```
    P=100
    PRINT P
    P P
    P P 10
```

The PRINT P and P P are the same. However, P P 10 is the same as PRINT PRINT 10 or PRINT(PRINT(10)). It's that innocent little space between the P and the 10 that causes Zgrass to look for commands instead of variables. And it always looks at system commands first. Watch out for extraneous spaces!

Why all this confusion? It's to help you avoid lots of typing. You can create a macro like this one:

```
 QUANTIZED$SECOND$DEGREE$SPACE$INVADERS=[BOX 0,0,100,100,3]
```

and run it by typing Q, assuming it's your only thing starting with a Q. Even if you are a good typist, you will appreciate abbreviation once you get used to it. HELP can tell you what the system names are, and you can avoid abbreviations of them for macro names. If you do name a macro BO, which is a abbreviation for BOX, just assign it to another name:

```
    MYBO=BO
```

and run MYBO.

Particularly error-prone names are WAIT, TIME, GET, PUT, and ONE. Use HELP to figure out what commands these are abbreviations for.

The command USEMAP lists all the names currently in use. Once you start using names longer than one letter, they show up in the USEMAP command. Try the following:

```
    ABC=10
    DEF=[STRINGYTHINGY]
    USEMAP
```

DELETE will remove them:

```
    DEL ABC
    USEMAP
```

Let's do a simple macro:

```
    MANYBOXES=[CLEAR
    A=300
    10 A=A-3
    IF A>0,BOX 0,0,A,A,A\4;GOTO 10]
```

Run MANYBOXES. Notice that a comma is used to delimit the conditional part of the IF command. The semi-colon allows a second command (GOTO) to be in the scope of the IF command. The '\' is the modulus operator and assures the COLORMODE will not exceed 3.

If you are a terrible typist and cannot get this program in straight, persevere. The next lesson will show a much more humane way to create macros. Line numbers are used only as labels for GOTO's and have nothing to do with line ordering. In fact, GOTO labels can have letters too, as long as they start with a number. 1AGAIN, 3DEATH, 4TEEN are all legal labels.Another example:

```
        RANDOMCROSSES=[CLEAR
        1MORE X=-160%159
        Y=-100%100
        LINE X-10,Y,4
        LINE X+10,Y,3
        LINE X,Y-10,4
        LINE X,Y+10,3
        GOTO 1MORE]
```

This macro puts little crosses on the screen until you press CTRL+C or BREAK. The 4 as colormode above places the starting point of the line to be drawn each time. See the LINE command in the Glossary.

Note that if you like to have your programs formatted so that labels are at the left edge and commands indented a little, you can use spaces as the first characters of the lines you choose. You cannot use tabs, however. Use of spaces does chew up memory, but you can do it.

Zgrass encourages you to write several small macros rather than one large program as BASIC requires you to do. Small macros are easier to configure as software tools. In case you are tempted to write enormous programs, Zgrass, for internal reasons, limits you to 99 SKIPS, GOTOS and IFS in a single macro. Macros cannot exceed 4000 characters either. Even the EDIT command (next lesson) works best on small programs. So break your bad BASIC habits!

End of Lesson 2.

LESSON 3 EDITING

Editing is the act of creating and changing a character string. In the process of developing a complex graphics simulation, for example, you do a lot of editing. In BASIC, editing is done with line numbers. Zgrass allows you to roam around a page of text on your terminal screen with the EDIT command.

To start, type:

```
    EDIT TEST
```

The screen should clear. Just start typing the following:

```
    PR "THIS IS LINE 1"
    PR "THIS IS LINE 2"
    PR "THIS IS LINE 3"
```

Note that you do not put brackets or other string delimiters in. EDIT does that automatically.

You move the cursor with the arrow keys. Now practice getting the cursor under any character you choose.

Change the quotes in the first line to single quotes by positioning the cursor under them one at a time and typing a single quote each time. You can change any character by typing over it.

Now press RETURN. There's an open line now. Type:

```
            THIS IS THE NEW LINE 2
```

Move the cursor down and change old line 2 to line 3, etc.

To change the 1 in the first line to ONE, position the cursor under the 1 and type ONE. You have wiped out the end " though, so type it in again. It's easy to type characters at the end of a line.

To change it to "THIS IS NOT LINE ONE", position the cursor under the L, press the HOME key and type NOT plus a space. Move the cursor down a line to get out of insert mode.

To delete the word NOT, position the cursor under the N and press ESC three times. Type it one time more to delete the extra space.

Some other EDIT functions:

-To delete a whole line, press the TAB key.
-To exit from EDIT, CTRL+E.
-To exit but ignore all changes made in EDIT, press BREAK.
-To insert before the first line, press the HOME key
and type away.
-To insert a NEXTLINE in the middle of a line,
press the HOME key first.

EDIT also allows you to move and copy lines or parts of lines. In order to do this, you must set two pointers, one indicating the start and one past the last character to be copied or moved by positioning the cursor and typing CTRL+S each time. A single quote will appear temporarily to indicate the pointer. (If you make a mistake, type CTRL+T to erase the pointers.) Then position the cursor again to where you want the text to go and type CTRL+D to move the text there or CTRL+F to copy it there. If you confuse things horribly, press BREAK and try EDIT again.

After a while, using EDIT will become easy and you will be forever spoiled.

End of Lesson 3.

LESSON 4 MORE ON MACROS

All the macros in this lesson should be typed in with EDIT. Remember to only type the characters between the square brackets when in EDIT. Try:

```
TRIANGLES=[CLEAR
PRINT "LOOK AT THE TV!"
SIZE=0
1UP SIZE=SIZE+2
LINE 0,SIZE*2,4
LINE -SIZE,-SIZE,1
LINE SIZE,-SIZE,2
LINE 0,SIZE*2,3
IF SIZE<50,GOTO 1UP]
```

Run TRIANGLES. If there are any errors, check your macro carefully and EDIT it.

You can speed up TRIANGLES a bit by COMPILing it:

```
COMPILE TRIANGLES,CT
```

and then run CT as you would any macro or command.

For too many reasons to go into here, Zgrass has no FOR/NEXT construct. You must build loops out of IF's and GOTO's, explicitly testing conditions. The iteration and testing step can be combined, if you prefer:

```
IF (SIZE=SIZE+2)<50,GOTO 1UP)
```

although this would necessitate SIZE to be initially set to 2 instead of 0, and the removal of the "SIZE=SIZE+2" in the line starting with 1UP, in this particular case. It's ok to have a label without anything following it, by the way.

You should be wondering about that assignment inside the IF. It's possible because Zgrass does not use the same operator for assignment and logical equals as BASIC does. Zgrass uses '==' (double equals) for logical equality testing. It's another minor deviation from BASIC to get used to. Note its use:

```
COUNTDOWN=[A=10
1LESS PRINT A
A=A-1
IF A==5,PR "HALFWAY THERE"
IF A>0,GOTO 1LESS
PRINT "BLASTOFF"]
```

Run this one and then edit it so the A==5 is A=5. The loop will never end since A is continually assigned 5. To further clarify:

```
        PRINT 5==5
        PRINT 5==6
```

So, "true" is 1 and "false" is zero.

IF takes the rest of the line if the stuff between it and the comma evaluates to non-zero. Therefore, IF 1,PRINT "HI!" would always print HI!

The next macro will draw sweeping lines of random colors forming an ellipse 300*150 pixels:

```
        SWEEP=[ANGLE=0
        CLEAR
        X=SIN(ANGLE)*150
        Y=COS(ANGLE)*75
        LINE X,Y,4
        LINE -X,-Y,1%4
        IF (ANGLE=ANGLE+2)<180,SKIP -4]
```

Note the SKIP -4 in the last line. It is a shorthand way of doing GOTO's without the need for labels. It goes back four lines. SKIP 2 would go down two lines (not skip the next two lines as you might think, that would be SKIP 3). SKIP 0 keeps executing the same line it is on.

As in BASIC, the INPUT command is used to get responses from the user of the macro:

```
        HOWMANYBOXES=[CLEAR
        PROMPT "NUMBER OF BOXES TO DRAW?"
        INPUT QUAN
        XCENTER=-160
        BOX XCENTER=XCENTER+20,0,18,18,3
        IF (QUAN=QUAN-1)>0,SKIP -1]
```

If you answer with too big a number for QUAN, boxes will be drawn off-screen.

To wrap the boxes around in a grid, try:

```
        GRIDBOXES=[CLEAR
        PROMPT "NUMBER OF BOXES TO DRAW?"
        INPUT QUAN
        YCENTER=80
        XCENTER=-160
        BOX XCENTER=XCENTER+20,YCENTER,18,18,3
        IF (QUAN=QUAN-1)==0,RETURN
        IF XCENTER==140,YCENTER=YCENTER-20;SKIP -3
        SKIP -3]
```

after running GRIDBOXES, try:

```
        GRIDBOXES 30
```

and notice that no prompt appears. If a macro is passed arguments like the 30 above, the PRINT and PROMPT commands are automatically suppressed. You can tell if there are arguments passed with the ANYARGS command, just look it up.

Now try:

```
SINECURVE=[PROMPT "WHAT'S THE OFFSET TO BE?"
INPUT OFFSET
X=-160
ANGLE=0
POINT OFFSET+X,SIN(ANGLE)*80,3
ANGLE=ANGLE+2
IF (X=X+1)<159,SKIP -2]
```

with:

```
SINECURVE 0
SINECURVE 30
```

To show how macros can be used as subroutines for other macros, try:

```
MANYSINES=[PROMPT "FIRST OFFSET?"
PROMPT "LAST OFFSET?"
PROMPT "HOW MANY?"
INPUT FIRST,LAST,QUAN
INCREMENT=(LAST-FIRST)/QUAN
SINECURVE FIRST
IF (FIRST=FIRST+INCREMENT)<=LAST,SKIP -1]
```

You can answer the questions by typing:

```
MANYSINES
```

or specify arguments:

```
MANYSINES 0,100,5
```

If you choose to speed up SINECURVE, you can compile it, but remember to change MANYSINES to refer to the compiled name:

```
COMPILE SINECURVE,FASTSINE
```

As a further note, you can also input strings. Use the INPUT.STR command:

```
NAMES=[PROMPT "WHAT'S YOUR NAME?"
INP.STR XXXX
PRINT "THAT'S FUNNY!, ",XXXX," IS MY NAME TOO!"]
```

End of Lesson 4.

LESSON 5 STORING MACROS ON TAPE AND DISK

With luck, by this point, you may have some macros worth saving. If you do not have a disk yet, or want to send someone a tape copy, you need to know how to use GETTAPE and PUTTAPE.

Zgrass's audio tape storage has some advanced features when compared with standard BASIC tape handling. Zgrass allows you to easily store several copies of a macro so that if an error is detected while reading it back, the next copy can be automatically retrieved. It also will print out a directory of the tape as it is looking for a file by name, if you wish.

PUTTAPE works on macros, arrays, swap modules and screen dumps. So far, you've only used macros. You cannot PUTTAPE compiled macros or number variables. Screen dumps are 16K byte blocks which are memory dumps of the screen, useful for storing pictures instead of the instructions to draw the pictures. A screen dump gets PUTTAPED when you use the .TV switch and a name.

The required cable hookups for tape storage are described in the hardware manual for the UV-1.

To store a macro on tape, you need to specify three things:

```
    PUTTAPE NUMBER,MACRONAME,[SOME DESCRIPTIVE MESSAGE]
```

where the NUMBER is the number of times to write the file out (2 or more is suggested), and the message in brackets is supposed to be descriptive so that when you look at the tape three months from now you know what it's for. You can see the number count down in binary in console lights 8-15 as the copies are being put on tape.

For example, to PUTTAPE a macro called SHIRLEY, get the tape deck ready, type in the following line but don't press the RETURN key yet:

```
    PUTTAPE 2,SHIRLEY,[SHIRLEY DRAWS PINK ELEPHANT HATS]
```

set the tape deck in RECORD with the tape moving and then press RETURN. When you get the attention mark back, stop the deck.

Since Zgrass programming usually involves creating several macros as software tools, it's a good idea to have macros for each major task that do nothing but GETTAPE and

PUTTAPE all the pieces. You cannot conveniently reclaim data space on an audio cassette, so always work in the mode of reading all the parts in, change them as necessary for debugging and write them out in order.

To GETTAPE a file named SHIRLEY, simply type:

```
GETTAPE SHIRLEY
```

Press the RETURN key and start the audio tape up, making sure that the cables are connected, of course.

If there is an error detected in the tape read, you will see

```
BAD-AUTO RETRY
```

and the next copy will be gotten. You can test this feature by turning the audio level down all the way during a GETTAPE for an instant.

It's actually a good idea to rewind your tape, and type GETTAPE XXXXX after you've PUTTAPED your macros, before you RESTART the system. Assuming XXXXX is not a name on the tape, GETTAPE will scan the entries, print the directory information, verify the integrity of the data (if it doesn't say BAD DATA, it's ok), and keep going. If you encounter some errors, you can re-PUTTAPE the macros again. Get out of GETTAPE or PUTTAPE by pressing the red button marked "RST" on the UV-1 front panel.

If you are thorough and methodical about saving your Zgrass macros, arrays and so on, you will not lose your temper when the power company glitches, or you find a bug in Zgrass that erases all your macros in memory.

USING THE MICROPOLIS DISK AND DGET/DPUT

The Micropolis 5" floppy disk system supplied with Zgrass units has a lot of software support. All disk commands begin with a D. and often only require two characters (for example, DU works for DUSEMAP, DG for DGET, etc.). These disks will hold approximately 180,000 bytes on each surface giving a total of over 700,000 bytes of storage on-line.

The primary difference between tape and disk is random access. Disks have diskmaps (sometimes called directories) which tell you what is on them and the system where the data is stored on the disk. Unlike audio tape, access to different parts of the disk is easy so it can jump around alot. You do not have to store stuff sequentially as you do with audio tape.

Another difference is deletion. You can update stuff simply by putting it out again. The DPUT command automatically keeps one backup for you for safety. Of course, the disk is much faster than audio tape as well.

One note: names stored on disks are generally referred to as 'files' so do not get confused by the terminology. Files are just things stored on the disk, nothing more.

The first thing you should do is insert your system disk into drive 0. You do this by gently pushing the disk in the slot, keeping the label up and the little notch to the left. Then push the lever with the blue on it down until it catches. Then type:

```
     DSETUP 0
     DUSEMAP
```

and you will see a listing of all the swap modules on your system disk. Now put a blank disk in drive 1. Type:

```
     DSETUP 1
     DINIT 200
```

This will erase the disk and then initialize it to accept a maximum of 200 names. Type DUSEMAP and you will see how many sectors (each is 512 bytes) are left.

If you get an error when trying to DINIT, the disk may not be formatted, in which case you must first DFORMAT it. See the Glossary.

Now type:

```
     DS 1  (the system starts up with disk 0 setup)
     SAM=NB
     DPUT SAM,[SAM IS A COPY OF NB]
     DUSEMAP
```

and you will see SAM in the disk map. If you RESTART then type:

```
     DGET SAM
     USEMAP
```

you will see SAM in memory.

If you DPUT SAM and then type DUSEMAP, you will see SAM and a backup copy of SAM (type is listed as BAK). The second and subsequent times you DPUT something you do not have to include a message (the stuff in square brackets) unless you want to change the message. DPUT always maintains one and only one BAK.

You can DGET a BAK with DGET.BAK NAME.

You can delete a name on the disk with DDELETE NAME. If you want to save space on the disk, you can get rid of BAKs with DDELETE.BAK NAME. There is also a command DBAKS which removes all BAKs from the disk. You only need to remove BAKs if you need the space, which is not normally the case.

The reverse side of disk 0 is called disk 4, so you use DSETUP 4 to get at it. Of course, it must have been DINIT'd at some previous time. The reverse of disk 1 is called disk 5. If you have two Micropolis dual drives, you get disks 2,3 and 6,7 as well.

The Glossary has descriptions of the rest of the disk commands. One concept that has to be explained in detail, though, is the submap. Submaps are essentially little disk maps which you can use to partition your disk into areas specific to individual projects you are working on. Once you are working within a submap, all disk commands reference only that submap and cannot get from or put into any other submap. The only exception to the rule is that if DGET cannot find a name within the submap, it will go look through the regular diskmap (but not any other submaps) for the name, so you can DGET swap modules and any other 'tools' you often use that are stored in the regular disk map.

The way you get into a submap the first time is to type:

```
     DCREATE SUBMAPNAME,[MESSAGE]
```

For example, try:

```
     DCREATE PAINT,[PAINT PROGRAM SUBMAP]
     DUSEMAP
```

Nothing is there at present. If you DPUT a name, it will show up with DUSEMAP. To get back to the regular disk map, type:

```
     DSETUP n
```

where n is the disk you're using. To get back to the submap PAINT, type:

```
     DSETUP n,PAINT
```

The regular disk map holds entries for all the submaps so you can tell what their names are. You can actually store things under a submapname without having created the submap with DCREATE but you will not know they are there unless you possess an extraordinary memory. You have to pay attention when you change disks, in particular, so you do not DPUT into submaps which have not been DCREATE'd on that disk. It would take an enormous amount of overhead to have Zgrass check each time for whether the submap has been created.

DDSMAP, a swap module, will remove a submap and all its entries.

The disk commands turn off the rest of the system while operating so you cannot type ahead as you normally can.

There are also two other artifacts: the stripe command temporarily freezes and the system timer device variables are suspended during disk access time. Control characters are not listened to either, so if you want to use CTRL+W with DUSEMAP, make sure you press the keys before the command starts going.

You should also check out DLOAD.

Please read about the other disk commands in the Glossary at your leisure.

End of Lesson 5.

This page intentionally left blank

LESSON 6 DEBUGGING

This lesson is about what to do when your program doesn't do what you expect it to do. There are several classes of reasons for unexpected behavior. The first class involves SYNTAX errors.

Syntax errors are essentially errors in spelling, punctuation, abbreviation or specifying arguments to a command. Syntax errors can always be fixed with EDIT. Examples of syntax errors are:
Spelling errors:
```
    HILP instead of HELP
    help instead of HELP   (is caplock key on?)
    POINT instead of POINT  (zero instead of O)
```
Punctuation errors:
```
    POINT 14;30,4  (';' should be a ',')
    SKIP-2            (needs a space)
    POINT (14,30,2) (if parentheses, no space allowed)
    TEST="PRINT '"'"   (double quote inside double quotes)
    IF A=10,PRINT "huh?" (this always prints. Use '==')
```
Be particularly careful to match parentheses and brackets. Quotes, single and double, cannot be nested like brackets and parentheses can. Although you might at first be more comfortable with quotes from experience with BASIC, strings defined with brackets (both curly and square) seem easier to locate when you're debugging.

RECKLESS ABBREVIATION:

```
    DOODLE CR1,CR2,CR3,CR4,CR5,CR6,CR7,CR8,CR9,CR10,CR11
```
will cause confusion because CR1 is an abbreviation for CR10 and CR11 as well as, in this case, probably a variable name itself. Use names like CR01 if you have a names like CR10.
```
    SIN FIRST,ORIGINAL
```
SIN is a system function so you can't use it for a macro name. If you get into this situation, reassign the macro name by typing:
```
    SIN1=SIN
```
You can delete SIN then, if you want to clean things up. (Astute note: since you created SIN1 after SIN, the reference to SIN in DELETE SIN will find the first SIN, not SIN1. If you typed DELETE SIN again, it would get rid of SIN1, of course. Since there is no way to pass system command names as arguments, you can't actually delete the SINE command, even if you try.)

MISSING ARGUMENTS:

Often you omit something a command wants as an argument. HELP gives you the arguments in brief for each system command and function, and the Glossary gives information in detail. Sometimes unbalanced parentheses or brackets will confuse the argument scanner, other times it's a misplaced comma or semi-colon. Proofreading is essential in Zgrass since there is not nearly the redundancy present in English or BASIC.

OTHER TYPES OF ERRORS:

There are three other types of errors you will encounter: logic errors, incorrect assumptions, and running out of space and time. We will discuss all of these in detail.

Incorrect assumptions arise from poor documentation or instruction on our part and bad guessing by you. It is impossible to describe everything a programming language does or can do (which is why they're so intriguing, of course) so the best we can do is get you to a level at which you can tell whether it is your problem or a shortcoming of Zgrass when the unexpected happens.

DEBUGGING STEPS:

First, check for syntax errors. If there are none apparent, check that the commands you are using actually do what you expect them to do. If possible, use the command outside a macro in its simplest form. Use numbers instead of expressions since the error will often be in your expression, not in the command, something you should discover as soon as possible.

Always debug in interpretive mode (non-compiled). Anything that works as a macro should work as a compiled macro (if it doesn't, it's our fault unless it's documented).

Macros which make you unhappy do so because they generate errors or don't do what they're supposed to. The former are easier to detect because Zgrass points them out and gives you an error number which you can look up in the Glossary. You can list the program as it is executing with CTRL+X and you will see the last lines executed before the error is generated. You can print the values of variables and see what's wrong. A second CTRL+X cancels list mode, as will pressing CTRL+C or BREAK.

If you are still having trouble, once you get to the part of the program causing the error, press CTRL+D. This puts you into single step mode. Each time Zgrass sees a NEXTLINE or semi-colon, it prints a '#' and acts like it's in attention mode. You can type commands to see what's in variables, and/or press RETURN to continue one step at a time. A second CTRL+D plus a RETURN gets you out of single step.

There is also a once-only CTRL+D. You get it by pressing CTRL+Z. A RETURN key press gets you back into the macro.

Since it is easy to EDIT in Zgrass, put PRINT's in crucial places to test your variables in a loop. You can then use the print control characters (CTRL+O, CTRL+Q) to alter execution or printing of the variables. CTRL+O will suppress output to the terminal but otherwize allow the macro to execute so if you put a PRINT in a loop and don't want to see it all the time, type CTRL+O to turn printing off, another one to resume printing. CTRL+Q stops not only the printing but the execution as well when your macro is trying to print. It's good for waiting while you scratch your head trying to figure out the problem. Another CTRL+Q will resume printing. You can't type anything when stopped with a CTRL+Q, by the way, as you can in CTRL+Z or CTRL+D modes.

CTRL+W is also useful for seeing a terminal screen-full of information at a time. Press the RETURN key to get 20 lines of text at a time after you press CTRL+W. Another CTRL+W will get you out of this mode. (We used CTRL+W in Lesson 1 to page through the HELP command.)

SPACE AND TIME PROBLEMS:

Zgrass has a lot of memory for a small system. If you use a lot of arrays, system swap modules, or many, many large macros, you can run out of space. The only recourse is to delete some of the stuff that's taking up all the space. You may have to design some graphics sequences to GETTAPE or DGET parts while running, deleting stuff that is no longer needed.

Getting things to run faster is the real problem, though. The compiler speeds things up, of course. If you're really pressed for time, try modifying your macros to do less computation, use less array references. etc. Games programmers get paid very well for their cleverness.

You now have the tools necessary to debug your programs. It still isn't easy, but it's most of what we call programming.

End of Lesson 6.

LESSON 7   DEVICE VARIABLES AND PORTS

Devices (also sometimes called "peripherals") are hardware gadgets that hang off the computer system and do the communication with humans or other computers. So far you've communicated with Zgrass by typing on a terminal and it talked back with graphics on the TV and characters on the terminal. There are several other devices available in Zgrass some of which you access via DEVICE VARIABLES and others you read/write via PORTs. Ports are a more primitive way of accessing the hardware. Device variables have been provided for hardware features that share ports in a complex way and for timing-related software/hardware programming. Simpler devices (like the LED lights and switches, for instance) can be set or read easily by the PORT command so the overhead in providing device variables for them is not justified.

Device variables do not show up in USEMAP. They all start with a dollar sign and have two more letters.
You can have your own variable names with dollar signs as the first character, so you might want to always have three or more characters following the dollar sign to avoid conflict with device variables. Many device variables are acted upon by the system every 1/60 second.

For instance, find a joystick and plug it into the leftmost joystick hole in the front panel. There are four device variables which report the values received at each hole sixty times a second. For hole 1 (notice that the number on top of the joystick knob only corresponds if you take the care to plug it into the correct hole), the device variables are as follows:

```
    $T1 is zero if the trigger is out, 1 if in.
    $K1 ranges from -128 to +127 when you turn the knob as
        you would a volume control.

    $X1 is 0 if the knob is in the center position,
           1 if the knob is pushed to the right and
          -1 if the knob is pushed to the left
    $Y1 is 0 if the knob is in the center position,
           1 if the knob is pushed away from you and
          -1 if the knob is pulled toward you
```

assuming you hold the joystick with the trigger pointing away from you.

The easiest way to discover the values of a device variable is to print it in a loop while changing its values:

```
    RANGE=[PRINT $T1;SKIP 0]
```

and so on.

Let's write a simple drawing routine which puts a point down whenever the trigger is pulled in:

```
    DRAW=[IF $T1==1,POINT X=X+$X1,Y=Y+$Y1,3
    SKIP -1]
```

This program needs improvement, though. You can't tell where the point is going to be unless the trigger is in. You also can't erase any points put down in error. Try:

```
    DRAW=[POINT X=X+$X1,Y=Y+$Y1,3
    IF $T1==0,POINT X,Y,0
    SKIP -2]
```

This DRAW will erase the point you see at the current accumulated X,Y position unless the trigger is in. It's still pretty hard to see the dot, though, so let's put a cursor in:

```
    DRAW=[K=0
    X=X+$X1;Y=Y+$Y1
        BOX X,Y,1,20,5
        BOX X,Y,20,1,5
    IF (K=K+1)\2==1,SKIP -2
    IF $T1==1,BOX X,Y,3,3,$K1/64+2
    SKIP -5]
```

This DRAW uses several tricks. First, COLORMODE 5 is used with LINE to allow the red lines to flash without overwriting already drawn blue points. If we used COLORMODE 1, the red lines would erase the blue points. Secondly, COLORMODE 5 applied twice erases the red lines. Color modes 4-7 are done by special hardware in the system and will be discussed in detail in the next lesson. The four LINE commands are done twice by the testing of K modulo 2 which evaluates to 0 or 1 each time. Finally, the $K1/64+2 evaluates to 0,1,2, or 3 and sets the COLORMODE of the point drawn so you can erase with 0 or draw with red, green or blue by turning the knob.

It's less easy to figure out how to use device variables you set instead of read. The hue and brightness associated with the four colors on the screen are set by Zgrass sixty times a second based on what values you have stored in $L0, $L1, $L2 and $L3. Print out the values of these variables:

```
   PRINT $L0,$L1,$L2,$L3
```

Zgrass automatically puts these values in $L0-$L3 when you restart the system or type CTRL+E.

The colors are arranged in 8 brightness levels of 32 colors yielding a choice of 256 colors. Setting $L0 to 127, for example, turns the screen to yellow. Try:

```
    COLORS=[$LO=$LO+$X1;SKIP 0]
```

Let's make finding your favorite colors easier:

```
    CHOOSY=[PRINT "MOVE THE KNOB SIDE TO SIDE
       TO CHANGE COLOR
    AND TURN IT TO CHANGE BRIGHTNESS
    PULL TRIGGER TO PRINT VALUE"
    A=0
        A=A+$X1*8;IF A<0,A=0
        IF A>248,A=248
        $LO=A+$K1/32+4
    IF $T1==1,PRINT $LO
    SKIP -4]
```

The $K1/32+4 evaluates to a number in the range of 0-7 plus a fraction. Since $LO is used as an integer value, the fractional part is tossed away.

Now, look in the Glossary under DEVICE VARIABLES. Of primary interest are $HB and $BC. Try setting them to numbers within their ranges and see the results. When you set $HB to 20, for example, the screen now shows the right half colors $R0-$R3. Draw a set of color bars:

```
    CBARS=[CL;A=-149;C=0;$HB=21
    $R0=0;$R1=82;$R2=43;$R3=249
    $L0=7;$L1=213;$L2=126;$L3=164
    IF A<115,BOX A=A+45,0,46,202,C=(C+1)\3+1;SKIP 0]
```

You can easily make a pocket watch out of Zgrass with the system time device variables:

```
    CLOCK=[PRINT "INPUT HOUR,MINUTE,SECOND"
    INPUT $HR,$MN,$SC
    PRINT $HR,$MN,$SC;SKIP 0]
```

There are also ten system timers $Z0-$Z9 which you can use to control things over time. Many device variables are used for setting options in the software. See the Glossary for complete details. $RD, for example, if 0, sets the system to use degrees for angle movement; if 1, radians are used. All device variables except the $L0-$R3 and $HB are set to zero on RESTART, so the default settings are the ones that correspond to the device variable being 0. See the examples under DEVICE VARIABLES.

PORTS:

Use of the PORT command and function are documented in the Glossary.
You must use ports to get at the console switches, light up

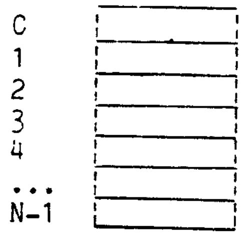
the console lights, and to directly access the music synthesizer. The rest of the ports you diddle at your own software risk--you can't break the hardware by setting ports, of course.

End of Lesson 7

LESSON 8 ARRAYS (Optional Lesson)

Computer programming languages have been largely built around what computers do well. One thing they do well is manipulate lists of things. A string is a list of characters, as you know, one after another. Also common are lists of numbers, and these are called ARRAYs. (There are also string arrays in Zgrass, but let's ignore them for now.)

An array with N elements looks like:



Before we get into the mechanics of accessing individual array elements, let's discuss the benefits of using arrays.

First, it takes at least 16 bytes to store the name and value of a named numeric variable. So 100 named variables take up at least 1600 bytes. Second, you have to get fancier than you presently know how to change all 100 of them in a loop with less than 100 different assignment statements, and it's not very fast anyway. Besides, it's a pain to type in so many names.

To create an array, use the array command:

```
ARRAY STUFF,100
```

STUFF now has 100 elements called STUFF(0), STUFF(1), ...,STUFF(99). To print out the values of STUFF, you have to write a macro:

```
PRSTUFF=[N=0
PRINT STUFF(N)
IF (N=N+1)<100,SK -1]
```

Run PRSTUFF. Of course, all values are 0. Try:

```
STUFF(91)=12345
PRSTUFF
```

You'll see the 12345 in element 91. We'll pretty up the array printing later on.

The major negative aspects of using arrays are that first, STUFF(91) is not a very good name, compared with GUNANGLE or BIGNURSE, for example, and, second, you wind up typing a lot of parentheses.

Geometrical problems lend themselves to array-based solutions. We will show you how to rotate a pyramid in 3-D using arrays. Let's look at the x,y,z coordinates and the color. The base is blue, and the lines going to the apex are red and green:

X	Y	Z	color
50	0	0	4
0	0	50	3
-50	0	0	3
0	0	-50	3
0	50	0	1
0	0	50	1
0	0	-50	4
0	50	0	2
-50	0	0	2

We will store the 40 numbers above in an array called PYR. First, create it:

```
        ARRAY PYR,40
        X values are in PYR(0), PYR(4), PYR(8), ... ,PYR(36)
        Y values are in PYR(1), PYR(5), PYR(9), ... ,PYR(37)
        Z values are in PYR(2), PYR(6), PYR(10),... ,PYR(38)
        Colors   are in PYR(3), PYR(7), PYR(11),... ,PYR(39)
```

that is, for values of N from 0 to 9,

```
        X's are in PYR(N*4)
        Y's are in PYR(N*4+1)
        Z's are in PYR(N*4+2)
        Colors are in PYR(N*4+3)
        Of course, we have to load up PYR:
        ENTER=[PRINT "TYPE ENDPOINT NUMBER, X,Y,Z,COLOR
        VALUES:"
        INPUT N,X,Y,Z,K
        PYR(N*4)=X
        PYR(N*4+1)=Y
        PYR(N*4+2)=Z
        PYR(N*4+3)=K
        SKIP -6]
```

Run ENTER and type these values:

```
        0,50,0,0,4
        1,0,0,50,3
        2,-50,0,0,3
        3,0,0,-50,3
        4,50,0,0,3
        5,0,50,0,1
        6,0,0,50,1
```

```
    7,0,0,-50,4
    8,0,50,0,2
    9,-50,0,0,2
```

If you make a mistake, just retype the number, x,y,z and color. Use CTRL+C to get out.

Now let's print the values out in a table:

```
    PRPYR=[N=0
    L=0
    PROMPT "PYR("&(4*N+L)&")=",PYR(4*N+L)
    IF (L=L+1)<4,SKIP -1
    PRINT
    IF (N=N+1)<10,SKIP -4]
```

PRINT with no arguments prints just a NEXTLINE. Be careful of the punctuation! The '&'s are used to eliminate the spaces that commas in the same place would cause. If any of the elements is incorrect when you run PRPYR, change it with ENTER.

Store PYR on tape:

```
    PUTTAPE 2,PYR,[PYRAMID ENDPOINT ARRAY FOR LESSON8]
```

Rewind the tape and type CTRL+N then:

```
    GETTAPE XXXXXX
```

to verify that PYR is stored without errors. If it doesn't say BAD DATA, PYR is stored properly. If you have a disk, DPUT PYR instead.

Now let's draw the contents of PYR. To see a 3-D object on a 2-D screen, you have to do a projection. The easiest projection is done by throwing away the z-axis coordinates. Just use the x, y and color values as arguments to the LINE command:

```
      DRAWPYR=[CLEAR
      N=0
      LINE PYR(N*4),PYR(N*4+1),PYR(N*4+3)
      IF (N=N+1)<10,SKIP -1]
```

A straight-on projection like this is not very interesting, of course. So, we'll rotate the image. If you don't understand the SINE/COSINE math below, take it on faith.

To rotate PYR around the center of the screen (z-axis), you change the endpoints by the following formula:

```
    XNEW=X*COS(ANGLE)+Y*SIN(ANGLE)
    YNEW=-X*SIN(ANGLE)+Y*COS(ANGLE)
    ZNEW=Z
```

So our macro for z-axis rotation is:

```
    ZROT=[A=0
    BOX 0,0,110,110,0
    S=SIN(A)
    C=COS(A)
    N=0
```

```
        L=N*4
        LIN PY(L)*C+PY(L+1)*S,-PY(L)*S+PY(L+1)*C,PY(L+3)
        IF (N=N+1)<10,SKIP -2
        WAIT 1
        A=A+6;SKIP -8]
```

Note that PY is being used as an abbreviation for PYR. If you change the -8 to a -7 in the last line, you'll get a built-up image. Compile ZROT to make it go faster:

```
        COM ZROT,CZROT
        CZROT
```

We still can't see the z-axis information because, obviously, we haven't used any PYR(L+2)'s yet. We have to rotate around different axes. Rotation around the x-axis is given by:

```
        XNEW=X
        YNEW=Y*COS(ANGLE)-Z*SIN(ANGLE)
        ZNEW=Y*SIN(ANGLE)+Z*COS(ANGLE)
```

Of course, we aren't using ZNEW. But we will. Try:

```
        XROT=[A=0
        BOX 0,0,110,110,0
        S=SIN(A);C=COS(A)
        N=0
        L=N*4
        LINE PYR(L),PYR(L+1)*C-PYR(L+2)*S,PYR(L+3)
        IF (N=N+1)<10,SKIP -2
        A=A+6;SKIP -6]
```

Similarly, y-axis rotation is given by:

```
          XNEW=X*COS(ANGLE)+Z*SIN(ANGLE)
          YNEW=Y
          ZNEW=-X*SIN(ANGLE)+Z*COS(ANGLE)
```

Rotating around two axes at once is more visually interesting. First we compute the rotation around x then apply the rotation around y. We need to use the intermediate value of Z in the computation of the y rotation:

```
        DOUBLEROT=[A=0;B=0
        10CLEAR IF $T1==1,BOX 0,0,110,110,0
        SA=SIN(A);SB=SIN(B);CA=COS(A);CB=COS(B)
        N=0
         .COMPUTE X ROTATION NEW X,Y,Z (THIS IS A COMMENT)
        1MOVE XXROT=PYR(N*4)
        YXROT=PYR(N*4+1)*CA-PYR(N*4+2)*SA
        ZXROT=PYR(N*4+1)*SA+PYR(N*4+2)*CA
         .COMPUTE Y ROTATION WITH NEW X,Y,Z
        X=XXROT*CB+ZXROT*SB
        Y=YXROT
         .DRAW THE LINE
        LINE X,Y,PYR(N*4+3)
        IF (N=N+1)<10,GOTO 1MOVE
        A=A+6;B=B+12;GOTO 10CLEAR]
```

Note the addition of trigger control of image clearing. You can tighten up DOUBLEROT by combining expressions, at the expense of clarity. It will go faster compiled, of course.

Comments may be interspersed with code if the first character is a period. The entire line is taken as a comment and skipped, even if there is a ';' in the line.

Zgrass allows multi-dimensional arrays. We could have defined PYR by:

```
        ARRAY PYR 10,4
```

giving 40 elements:

```
        PYR(0,0), PYR(0,1), PYR(0,2), PYR(0,3)
        PYR(1,0), PYR(1,1), PYR(1,2), PYR(1,3)
         ...
        PYR(9,0), PYR(9,1), PYR(9,2), PYR(9,3)
```

This would eliminate the multiplication of N*4 each time, so PRPYR would look like:

```
        PRPYR=[N=0
        L=0
        PROMPT "PYR("&N&','&L&")=",PYR(N,L)
        IF (L=L+1)<4,SKIP -1
        PRINT
        IF (N=N+1),10,SKIP -4]
```

You may find this conceptually clearer, but maybe not. The following example uses a 2-D array well. Let's make an array to hold the positions of pieces in a checkerboard. Each array element will correspond to a square on the board. A 0 means no piece, 1 means red, 2 means red king, -1 means black piece, -2 means black king. The initial board setup can be done by:

```
        ARRAY CHECKERBOARD,8,8
        DATA=[Y=0
        INPUT CHECKERBOARD(X,Y)
        IF (Y=Y+1)<8,SKIP -1]
        CHECKFILL=[X=0
        DATA 1,0,1,0,0,0,-1,0
        X=X+1
        DATA 0,1,0,0,0,-1,0,-1
        IF (X=X+1)<8,SKIP -3]
```

Note the mimicking of BASIC's DATA statement. The rest of the checkers game is up to you!

End of Lesson 8.

LESSON 9 MORE ON GRAPHICS

Zgrass is a graphics language by design. It provides high-level commands for creating and manipulating visual information. The highest level commands in Zgrass are those operating on arrays called "snaps" (after "snapshots") which are saved parts of the tv screen, in essence. They are defined by the SNAP command and drawn by the DISPLAY command:

```
        STEST=[CLEAR
        BOX 0,0,20,20,3
        BOX 5,0,10,15,2
        BOX -5,0,10,15,1
        SNAP IT,0,0,20,20
        DISPLAY IT,-160%159,-100%100,0;SKIP 0]
```

The last argument of DISPLAY is the DISPLAYMODE. 0 means plop the snap on the screen erasing whatever was there before. DISPLAYMODE 0 works just like COLORMODE 0-3. Now, EDIT STEST so its last line reads:

```
    DISPLAY IT,X=X+$X1,Y=Y+$Y1,0;SKIP 0
```

Plug in a joystick into the leftmost joystick socket and run STEST. You'll notice that you "drag" the edge around. One way to avoid this is to make the snap larger to include some "white space" around it. Change STEST as follows and run it:

```
    SNAP IT,0,0,24,24
```

You can have any pattern stored in a snap as long as it fits in the largest memory segment left. The CORE command will give you a list of the memory segments. The largest snap you can store is about a quarter of the screen. Larger snaps display slower than smaller snaps, of course.

In the drawing program in Lesson 7, we used COLORMODE 5 to flash a crosshair cursor. We can do the same trick with DISPLAYMODE 1:

```
        DRAW=[CLEAR
        LINE 10,0,4
        LINE -10,0,1
        LINE 0,-10,4
        LINE 0,10,1
        SNAP CURS,0,0,20,20
        X=X+$X1;Y=Y+$Y1
        DISP CURS,X,Y,1;DISP CURS,X,Y,1
        IF $T1==1,POINT X,Y,$K1/64+2
        SKIP -3]
```

Two successive DISPLAYMODE 1's will draw and erase the crosshair cursor without affecting the points you have drawn by pulling the trigger.

WHAT'S ACTUALLY GOING ON:

You can. in fact, construct any image possible on the 320x201 pixel TV screen with the colormodes you've learned so far. However, to do the animated graphics required in video games, several features were designed into the custom Bally hardware. Zgrass uses these and adds software to give you a rather complete set of graphic capabilities for what you can do with two bits per pixel.

Plop (colormodes 0-3 and displaymode 0) works just like assignment. The old value is replaced with the new one. All other color and display modes are binary functions, that is, they take two values and return a new value which is placed on the screen. The first value taken is what is on the screen at the given pixel, the second is given or implied by the color/display mode, and the function itself is also given or implied by the color/display mode.

You might wonder just how many unique results you can get from the functions of two two-pixel values. A lot. We've tried to provide the most useful ones: 21 colormodes and 15 displaymodes with 9 options each.

Rather than do a paragraph on each mode, we will show you how to figure them out. Some are quite clear from a verbal description (for example, displaymode 60 is "plop only the green and blue portions of this snap"). Unless you've tinkered with Boolean Algebra, some functions will be new to you and unless you've done extensive graphics already, the visual ramifications will be surprising.

We will explain several of the functions with TRUTH TABLEs. Truth tables are like multiplication tables except they can show other things than show results of multiplication.
The truth table for multiplication of numbers up to 4 is:

*	0	1	2	3	4
0	0	0	0	0	0
1	0	1	2	3	4
2	0	2	4	6	8
3	0	3	6	9	12
4	0	4	8	12	16

You have no trouble looking up the value of 3*4, presumably.

Multiplication tables usually stop at 12*12, but could go on forever. The truth tables we use are limited to two bit results and thus have
only four values like the following truth table for addition modulo 4:

\4	0	1	2	3
0	0	1	2	3
1	1	2	3	0
2	2	3	0	1
3	3	0	1	2

The functions we will describe are commutative, that is, the order of the values can be reversed without affecting the result. (Addition and multiplication are commutative; subtraction and division are not, for example.) If you wish, you can adopt the convention that what's on the screen is looked up along the left edge of the table and what is indicated by the color/display mode is along the top edge.

Colormodes 4-7 and displaymode 1 are the hardest to understand so we'll skip them and come back later. For now, let's do logical OR and AND.

Colormodes 8-11 and display mode 2 do a logical OR function between the value on the screen and the value indicated by the colormode (or the value in the snap at that point for DISPLAY). Try the following:

```
BOX 0,0,100,100,2
BOX 0,-25,50,50,1
BOX 0,25,50,50,3
BOX -25,0,50,50,9
```

Colormode 9 is "OR with red" or, more precisely: "OR with 01." Let's explain that better. We have two bits at each pixel, giving four possible values: 0, 1, 2, and 3. In binary, they are 00, 01, 10 and 11 (pronounced zero-zero, zero-one, one-zero and one-one, respectively). You may have noticed that:

$L0 corresponds to a pixel value of 00
$L1 corresponds to a pixel value of 01
$L1 corresponds to a pixel value of 10
$L2 corresponds to a pixel value of 11

When Zgrass comes up, the default colors are white, red, green and blue. So when we refer to a pixel as green, we mean it has a 10 binary value. You can confuse yourself horribly if, for example, you switch the values of $L2 and $L3 like this:

```
        A=$L2
        $L2=$L3
        $L3=A
```

so don't do it, at least in this part of the lesson. Whenever we refer to white, red, green or blue, it assumes the default colors are in $L0-$L4.

Look at the boxes on the screen again. Now look up the truth table for OR (it's under OR!) You can see why the green box OR'ed with red turns blue by reading the table: 10 OR 01 is 11. You can also see that OR'ing anything with 00 doesn't change it and OR'ing anything with 11 changes it to 11. (Observation reveals that some of the colormodes are redundant, by the way.) In words, OR produces a zero only if both corresponding bits were zero. Otherwize it puts a one there.

Let's try the AND modes (12-15):

```
        BOX 0,0,100,100,1
        BOX 0,-25,50,50,2
        BOX 0,25,50,50,14
        BOX -25,0,50,50,14
```

14 is AND with green (10). You should be able figure out what's going on by looking at the truth table for AND. In words, AND leaves a 1 in the bit only if both values had a 1 in that bit, otherwize it's zeroed.

XOR (exclusive-or) is the trickiest but most useful color/display mode function besides plop. Look up the XOR truth table. XOR yields a one when the corresponding bits are different and a zero when they are the same:

```
        10 XOR 10 IS 00
        10 XOR 00 IS 10
        10 XOR 01 IS 11
        etc.
```

You can see that XOR on a blank screen is the same as plop or OR on a blank screen. But try:

```
        NB
        NB
```

To see why the second NB erased the first, print NB:

```
        PRINT NB
```

(NB is a system macro, by the way). Note that it assures the colormode is between 5 and 7 by using the modulus operator. The lowercase letters are local variables which we'll explain in Lesson 11. Drawing anything twice with XOR will undo it.

You ought to take some time out to experiment with the colormodes and displaymodes. Here's a simple program by Jane Veeder to do some drawing:

```
        JANEDRAW=[PROMPT "WHAT'S THE XSIZE,YSIZE OF THE
```

```
DRAWING BOX?"
INPUT WIZE,HIZE
PROMPT "WHAT'S THE SPACING FACTOR OF BOX CENTERS? (0
TO 10)"
INPUT SPACING
PROMPT "WHAT COLORMODE? (1 TO 15)"
INPUT KOLOR
PROMPT "CLEAR THE TV? (Y OR N)"
INPUT.STR ANSWER;IF ANSWER=='Y',CLEAR
PRINT "MOVE JOYSTICK KNOB TO POSITION, HOLD TRIGGER
TO DRAW"
X=0;Y=0
X=X+$X1*SPACING
Y=Y+$Y1*SPACING
BOX X,Y,WIZE,HIZE,7
BOX X,Y,WIZE,HIZE,7
IF $T1==0,SKIP -4
BOX X,Y,WIZE,HIZE,KOLOR;SKIP -5]
```

Now that you've experimented a while with XOR, clear the screen and we'll do something tricky. First type in:

```
BEHIND=[X=-100
Y=50
LINE X,Y,4
LINE X,-Y,2
IF (X=X+5)<51,SKIP -2
X=-100
BOX X,0,20,20,5
BOX X,0,20,20,5
IF (X=X+1)<51,SKIP -2]
```

When you run BEHIND, you will note that the green lines are turned blue where the box is but are restored when the box is erased. Type the following:

```
$L2=$L3
```

The lines turn blue. Run BEHIND again. The red box appears to be traveling behind the blue lines because we've made both $L2 and $L3 the same color. When the red box is behind the blue lines, the value of the blue pixels is 11; when it is not there, the value is 10. So, by giving up a color, you can make one color appear to pass behind another. Now type CTRL+B to restore the colors to default. Set $L1=$L3 and run BEHIND again. The box is now "in front of" the lines.

You now have the skills to decipher the rest of the color and display modes.

End of Lesson 9.

LESSON 10 SWAP MODULES

Zgrass is rather tightly crammed into 32K of read-only memory (ROM). Some commands and functions we wanted to have in ROM just wouldn't fit so we distribute them on tape to be read in and executed just like macros. Such commands and functions are called SWAP MDOULES because you sort of swap some of your random-access memory (RAM) for the privilege of using them.

Besides allowing for more commands and functions than can fit in 32K ROM, swap modules have two other benefits: they can be changed, updated, and added to by us without having to send you new ROM's and, with the Zgrass assembler (separate package and documentation), you or your friendly neighborhood Z-80 wizard can add your own commands written in Z-80 assembler. Replacing a macro with a well-coded swap module can result in speed increases of two to maybe a thousand times depending on what you're doing. A skilled person can even do things that Zgrass won't let you, like put some of the information USEMAP gives you into variables, for instance.

If you do not have disks, the swap modules are distributed to you in alphabetical order on audio tape. You GETTAPE them just like anything else. You should copy your swap module tape early on because repeated use of any revolving mechanical recording medium will eventually result in its failure. If you have disks, the swap modules are on the disk we send you and you use DGET to bring them in.

Read about the TXT command in the Glossary and try the following (substitute DGET for GETTAPE if you have disks):

```
GETTAPE TXT
XYAXES=[LINE 160,-80,4
LINE -140,-80,1
LINE -140,100,1
X=-146;Y=-88
N=0
TXT X,Y,1,1,3,0,0,N
X=X+20
IF (N=N+1)<15,SKIP -2
N=0;X=-146
TXT X,Y,1,1,2,0,0,N
Y=Y+20
IF (N=N+1)<10, SKIP -2]
```

Zgrass also has a rather complete string manipulation package. Besides the concatenation operator (&), the following are available: ASCII, STRING, BUMP, FORMAT, LEN, LPAD, MATCH, REPLACE, SUBSTR. Some of these are swaps and you should try out these and other swap modules once you find a use for them.

Many of the disk utilities are swap modules. You will find them under the D's in the Glossary.

End of Lesson 10.

LESSON 11 ADVANCED CONCEPTS

If you print out the system test macro NB:

```
   PRINT NB
```

you will notice the lowercase variables a and b. Variables that begin with lowercase letters (a-z) are LOCAL VARIABLES, that is, they are known only to the macro they're in, just like labels for GOTOs. These variables are stored in a list attached to the memory automatically allocated to keep track of each macro call and are deleted whenever the macro returns. CTRL+C will also automatically delete all local variables.

Local variables have the following benefits:

1. They are zeroed whenever the macro is called.

2. They go away when the macro returns.

3. Recursion is possible.

4. You can create software tools without having to worry about names conflicting with other macros.

5. They never conflict with system command names.

6. Local strings and arrays are allowed.

Local variables have the following difficulties, though:

1. They don't show up in USEMAP and cannot be interrogated by typing CTRL+Z and printing the values. You have to put the PRINT right in the macro.

2. They are not known to called macros so you have to pass the values (this is actually good programming practice, anyway). In order to pass local variables that are not part of expressions, use the ? operator to force evaluation in the current macro context:

```
   SAM=[a=10
   b=20
   PRINT SQUAREM(?a,?b)]
```

```
        SQUAREM=[INPUT a,b
        RETURN a*a+b*b]
```

The a,b in each case are different variables because they are in different macros.

PARALLELISM:

Once you start communicating using animation, you realize how important timing is. Zgrass has several advanced features for controlling timing and sequencing of execution.

You are, by now, quite familiar with typing commands and running macros. When you are running a macro, you cannot type a command and expect it to execute without typing CTRL+Z. Zgrass has two modes of operation that allow macros to run and accept commands from attention mode at the same time. These modes give you the capability of foreground and background parallelism. So far, you've only used the middleground!

You can run a macro in the background by using the .B switch on it. Try:

```
        CHANGE=[$L0=$L0+$X1*8
        $L1=$L1+$Y1*8
        $L2=$K1]
        CHANGE.B
        NB.B
```

Notice that even though NB is drawing, you can change $L0, $L1 and $L2 with the joystick. You can also clear the screen at will or type other commands, start other .B macros, etc. (Note that variable b in NB is not re-initialized so it continues to increment until ERROR#23 eventually happens.)

Zgrass interleaves the command lines in CHANGE with those in NB and also slips in anything you type at the keyboard. Press CTRL+X and see the interleaving (press CTRL+Q to stop/start the printing).

If you run a macro at normal (middleground) level, it will suspend the .B macros until it is done. Thus, regular macros have precedence over .B's. You can stop all .B macros with CTRL+C or stop them selectively with the STOP command. If you press CTRL+A, .B macros will be interleaved with regular macros. You can set CTRL+A with the CONTROL command, of course.

You can also run macros in the foreground with .F. A .F macro has precedence over .B and regular macros. .F macros are assumed to be short and not contain infinite loops. They are restarted every 1/60 second or as fast as possible. It is a good idea to compile .F macros for speed.

The TIMEOUT command allows you to have the .F macro execute at multiples of 1/60 second. Say you want to draw an XOR box every five seconds. Five times sixty is 300. Try the following:

```
      TIMEDBOX=[TIMEOUT 300
      BOX 0,0,200,200,7]
      COMPILE TIMEDBOX,CTIME
      CTIME.F
```

You can have regular or .B macros running at the same time (try NB!).
Some further notes:

1. .B macros start over from the beginning automatically unless STOPped. No SKIP or GOTO is needed.

2. The interleaving of .B macros is on a line-by-line basis. Semi-colons don't count, blank lines do, so you can fine-tune the interleaving.

3. .F macros don't interleave. They are assumed to be short.

4. Macros run from within a .B macro are not interleaved unless run as .B. If called without a .B, they are interleaved as if they were a single line. If run with a .B, they run in parallel with the macro they're run from. You can setup variables to cause one .B macro to wait for another to continue, of course.

5. You can .B or .F the same macro multiple times, up to 128 times. You can have any number of .B and .F macros running at once. Obviously, things get pretty slow after a while.

6. It's a good idea to use local variables in .B and .F macros, especially if you .B a macro twice at the same time.

7. When executing in .B or .F mode, the local variables are not re-initialized to zero when the macro restarts at the beginning.

8. If your .B and .F macros have a lot of time-consuming graphics, the interleaving will not

appear to be particularly smooth.

ERROR TRAPPING:

If you wish, you can trap error messages and process them yourself, a useful feature for bulletproofing software for naive users.

The ONERROR command takes a label like GOTO but the label is only branched to when an error occurs. For instance:

```
UGH=[ARRAY SAM,10
A=-1
ONERROR 1OUT
PR SAM(A=A+1);SKIP 0
1OUT PRINT "SAM OUT OF BOUNDS"]
```

You can get the error number and command line in error into variables with the GETERROR swap command. LOOPMAX is a command used to catch infinite loops.

LOOPMAX and ONERROR do not work when compiled.

End of Lesson 11 and End of Lessons.