32-BIT PROGRAMMING & SOFTWARE MANUAL

Release Version 1.02

This Manual Provides programming & support for the following products:

Boss 32 Controller Family

Universal Machine Controller

Divelbiss Corporation
9778 Mt. Gilead Road
Fredericktown, OH 43019
Phone: (800) 245-2327
Fax: (740) 694-9035
E-mail: divelbiss@divelbiss.com
http://www.divelbiss.com

.

32-bit Controller Software Manual
Table of Contents

.

NOTICE: The contents and specifications contained in this manual are subject to change without notice.

Introduction
How To Use This Manual

At the present time, the Divelbiss family of 32-bit controllers consists of the Boss32 line of controllers and the Universal Machine Controller. This manual covers Divelbiss 32-bit BASIC programming for all units. Hardware manuals are available separately.

How to Use this Manual

In this manual, the following conventions are used to distinguish elements of text:

BOLD	Denotes hardware labeling, commands, and literal portions of syntax that must appear exactly as shown.
italic	Used for variables and placeholders that represent the type of text to be entered by the user.
EXAMPLE	Used for example programs, sample command lines, and text displayed by the Boss 32.
SMALL CAPS	Used to show key sequences, such as CTRL-C, where the user holds down the <Ctrl> key and presses the <C> key at the same time.
[ ]	Brackets are used to indicate optional elements of a command, such as LOAD [*program_num*] where *program_num* is optional.

Chapter Four, Language Reference, explains the commands, functions and statements used in programming Divelbiss 32-bit Controllers. The left margin of the first page of each command, function or statement, indicates which of the Divelbiss 32-bit Controllers support that particular feature. These appear as in the example below:

can be
used with:

Boss 32

UMC

In addition, the following symbols appear periodically appear in the left margins to call the readers attention to specific details in the text:

Indicates that the text contains information to which the reader should pay particularly close attention.

Warns the reader of a potential danger or hazzard that is associated with certain actions.

Appears when the text contains a tip that is especially helpful in programming the Divelbiss 32-bit Controllers.

.

Chapter One
Software Overview

The Divelbiss Corporations 32-bit Controller family comes standard with a customized version of BASIC as it operating system.

The Divelbiss 32-bit Controller operating system consists of two parts: the command line interface and the Divelbiss 32-bit BASIC compiler. The command line interface executes direct commands as they are typed in; it enters the Divelbiss 32-bit BASIC source code, loads and saves programs, executes programs, and starts the compiler. This chapter describes the direct commands that are available, the Divelbiss 32-bit BASIC compiler, the syntax of Divelbiss 32-bit BASIC, and the statements and functions that Divelbiss 32-bit BASIC supports. The Divelbiss 32-bit BASIC commands, statements, and functions are described individually in Chapter 4.

.

Direct Commands

Direct commands are used while programming the Divelbiss 32-bit Controller, and entered at the compiler prompt. They can be grouped into three categories: file commands, compiler commands, and miscellaneous commands.

File Commands

DIR	display a listing of files on the EPROM
DOWNLOAD	disable echo while loading a program
LOAD [filenum]	load a program from the EPROM
SAVE [CODE] [fname]	save source or compiled code to the EPROM

Compiler Commands

C	abbreviation for COMPILE
COMPILE or C	compile the BASIC program
ERROR	enable error checking in compiled BASIC
G	abbreviation for GO
GO or G	start the compiled program executing
NOERR	disable error checking in compiled BASIC
R	abbreviation for RUN
RUN or R	compile and execute the current BASIC program
STAT	display memory usage, compiler version and hard ware information

Miscellaneous Commands

BYE	reset the Controller
CLEARMEMORY	write 0's into all memory locations
CLR	erase the console display
E linenum	abbreviation for EDIT
EDIT linenum or E	enter the line editor to alter line
HELP	display help text
L	list the entire program
LIST [linenum] [,linenum]	list all or part of the program
LFDELAY	delay at the end of each line displayed
NEW	clear out the current BASIC program
NOLINE	used to change to no line number mode of operation
SETOPTION DAC	set DAC initialization values
SETOPTION SRL	set the default baud rate for the com1 while in edit mode on power up
SETOPTION TREMINAL	set the default terminal emulation for com1 on powerup
SETOPTION NOLINE	set the no line numbering on or off on powerup
SETOPTION LFDELAY	set the delay at the end of each line on or off on powerup
SETOPTION RUN	set the run mode on powerup to on or off

Organization of a Bear BASIC Program

The Divelbiss 32-bit BASIC language has a structured syntax that requires that particular elements of a program be placed in a specific order. Failure to follow this syntax could result in syntax errors, or, worse yet, an inoperable program.

Program Lines (using line numbers)

A Divelbiss 32-bit BASIC program consists of a series of program lines, sometimes referred to as lines of code, or source lines. Each program line has a line number followed by one or more statements. The line number is an integer between 1 and 32767. If there is more than one statement in a program line, then each statement is separated by a colon (':').

The following are valid Divelbiss 32-bit BASIC program lines:

```
100 X=4
35 print "Hi there"
32000 J=K*4 + 3: IF J>0 Then GOSUB 2000:j=0
```

The following are invalid program lines:

```
43000 PRINT X                Line number is too large

100 X=4 J=X                  No colon between statements
```

Divelbiss 32-bit BASIC does not require line numbers; if a line is entered without a line number, it will add 2 to the previous line number and assign that number to the new line. If used carefully, this can make the BASIC source code much more readable when it is viewed on a personal computer. BASIC stores lines in a tokenized format, where special codes are stored instead of the actual statement, in order to save space. A side effect of this is that the program will look different when it is LISTed than when it was entered. An example will demonstrate this; if the following is typed:

```
100 integer j,sum
    sum=0                    ' Initialize the sum
for j=1 to 100
     sum=sum+j               ' Add up the numbers
130 next j
    print "The sum is "; sum ' Display the result
```

when this program is LISTed, the following is displayed:

```
100 INTEGER J,SUM
102 SUM=0:                   ' Initialize the sum
104 FOR J=1 TO 100
106 SUM=SUM + J:             ' Add up the numbers
130 NEXT J
132 PRINT "The sum is ";SUM: ' Display the result
```

Notice that line numbers were added to the lines that didn't have them. Also, the only lower-case letters that remain are in text strings and comments. Divelbiss 32-bit BASIC is not case sensitive, so code can be entered in upper or lower case, but it is always stored in upper case.

Program Lines (no number mode)

In the Divelbiss 32-bit Controllers, the editor has been enhanced to facilitate the use of line labels instead or line numbers. Only lines that have to be jumped to need line labels. If there is more than one statement in a program line, then each statement is separated by a colon (':'). The colon however will be deleted and the line will list as a separate line. The following are valid Divelbiss 32-bit BASIC program lines in no line number mode:

```
integer j,sum
sum=0                              ‘ Initialize the sum
for j=1 to 100
sum=sum+j                          ‘ Add up the numbers
next j
gosub prnsum:stop
PRNSUM: PRINT”The sum is “; sum    ‘ Display the result
return
```

The following are invalid program lines:

100 PRINT X	In no line number mode line numbers are not allowed
X=4 J=X	No colon between statements

when this program is LISTed, the following is displayed:

```
INTEGER J,SUM
SUM=0:                             ‘ Initialize the sum
FOR J=1 TO 100
SUM=SUM + J:                       ‘ Add up the numbers
NEXT J
GOSUB PRNSUM
STOP
PRNSUM: PRINT “The sum is “;SUM:   ‘ Display the result
RETURN
```

Variable Declarations

Unlike many BASIC systems, all Divelbiss 32-bit BASIC variables must be explicitly declared. All variable declarations (i.e. INTEGER, REAL, and STRING statements) must be at the beginning of the program. In order to create efficient code, the compiler needs to know how many variables there are before it generates any machine code. Divelbiss 32-bit BASIC is limited to 8000 variable names in a program. No distinction is drawn between variables by mode. This means that the variable A2 is the same variable as A2$. If the variable is declared as both integer and string (both INTEGER A2 and STRING A2$ statements exist), a compilation error will result. Similarly, variable A is the same as dimensioned variable A(n). Divelbiss 32-bit BASIC supports a maximum of two dimensions for real and integer arrays; it supports single dimension string arrays.

DATA and READ Statements

All DATA statements must be located before the first READ statement in the program. DATA statements can be interspersed with other executable statements, but no DATA statement can follow a READ statement.

Placement of Tasks

In a multitasking program, the beginning of each task is identified with a TASK statement. The TASK statements must be in ascending numerical order starting with TASK 1. When a task is run, it begins execution with the statement following the TASK statement. The code located before the TASK 1 statement actually belongs to task 0; task 0 is present in every Divelbiss 32-bit BASIC program. The placement of tasks and subroutines can be critical. In general, the safest method is to not call subroutines that are located in another task. See Chapter 3 for further discussion of multitasking.

Numeric Constants

Constants are formed by combining decimal digits with an optional decimal point. Whenever a decimal point is included in a constant, the compiler assumes this constant is a real number. If the constant is expressed without a decimal point, the compiler assumes that the value is an integer. This has significance when combining real and integer values within an expression. Even though Divelbiss 32-bit Basic is smart enough to convert constants between integer and real as required, programs will run more efficiently if modes are not mixed. Divelbiss 32-bit Basic provides a facility for using hexadecimal constants. Hexadecimal constants are specified with a leading dollar sign. $1AB represents the hexadecimal constant 1AB.

Variables

Divelbiss 32-bit BASIC variables are formed by a letter followed by up to seven letters or digits. For example, A is a legal variable, as well as A0, A1, HI92, and JUMP. However, 9AB is not a legal variable, nor is A1234567899. String variables are formed the same way, but are followed by a dollar sign. HIYA$, A1$, A9$ are all legal string variables. A maximum of 8000 different variables may exist in any given program. These variables may be in any form within the rules given above. Note that integer or real variables may not have the same name as string variables. For example, the variable A may not be used in the same program that uses the variable A$. Divelbiss 32-bit BASIC will try to use A and A$ as the same variable, and a string variable error will result. Similarly, a dimensioned variable must not have the same name as an undimensioned variable (for example, you cannot use B and B(n) in the same program).

All variables must be declared at the beginning of the program before any executable code is encountered. In practice, this means that the variable declaration statements (INTEGER, REAL, and STRING) should be the first statements in the program. If undeclared variables are found during the compilation, the compiler will display an error message. For strings, the string length is specified in the STRING statement (for example, B$(80)). If no string length is specified, a string length of 20 will be assigned. The maximum allowable string length is 127.

String arrays are defined by specifying the length of each element, followed by the number of elements in the array. STRING A$(10,20) specifies 20 strings, each of length 10. Any ele-

ment can be accessed just like a singly dimensioned array. A$(3)=”123” assigns a value to the third element of the string array.
Subscripted variables are specified within the INTEGER and REAL statements. Note that subscripted variables start with the zero dimension, and extend to the maximum dimension specified. Therefore, the statement INTEGER A(10) defines a variable with eleven members, A(0) through A(10).

Integer variables are stored using a thirty-two bit two’s complement representation. An integer value can range from +2,147,483,647 to -2,147,483,648. Positive values which exceed 2,147,483,647 will appear as negative numbers. Real values are four byte (32 bit) IEEE compatible single precision real numbers. This means that approximately 6.5 digits of precision are maintained for real numbers (i.e. numbers between $\pm 3.4028235 \times 10^{38}$). Many BASIC interpreters and compilers use BCD mathematics or 64 bit representations resulting in high accuracy numbers that require lots of memory. Divelbiss 32-bit BASIC does not support either of these in the interest of maximizing speed. The user must be aware that a real number may not be exactly the number anticipated. For example, since real numbers are constructed by using powers of 2, the value 0.1 cannot be exactly represented. It can be represented very closely (within 2-23), but it will not be exact. Therefore, it is very dangerous to perform a direct equality operation on a real number. The statement IF A=0.123 (assuming A is real) will only pass the test if the two values are exactly equal, a case which rarely occurs. This is true for all real relational operators, including, for example, the statement IF A>B, if values very close to the condition being measured are being used. Be aware that the number you expect may not be exactly represented by the compiler. If necessary, use a slight tolerance around variables with relational operators.

Operators

Operators are connectors within expressions that perform logical or mathematical computations.

These operators work both with integer and real numbers:

+	addition
-	subtraction
*	multiplication
/	division

Some operators are relational. They generate a nonzero result if their condition is met; the nonzero value is unspecified (i.e. it isn’t necessarily 1 or -1, as in many languages). These operators may be used in mathematical expressions, but they are more frequently used with IF/THEN statements:

>	greater than
<	less than
<> or ><	not equal to
=	relational equality test
>=	greater than or equal (integers and reals)
<=	less than or equal (integers and reals)
AND	logical AND
OR	logical OR

.

Note that AND and OR are evaluated in integer mode. Real arguments are converted to integer before AND and OR are evaluated. The equals sign (‘=’) is used in two different ways in Divelbiss 32-bit BASIC: as the equality operator, and as the assignment statement. All operators are in a hierarchy that defines what operators will be evaluated first. The following is a list, from highest to lowest priority:

*, /
+,-
unary -, >, <, <>, ><
AND, OR

A variable may hold the result of a relational comparison. For example, A=R>0. Strings don’t support the >= and <= relational operators. Some BASICs use + for string concatenation, but Divelbiss 32-bit BASIC programs must use the CONCAT$ function.

Expressions

An expression is a mathematical, logical, or string calculation, such as 2+3, A/B, A=4, or N$>C$. Expressions are formed using constants, variables, operators, and functions. Expressions may be combined to form complex expressions, such as (A+B)/SIN(A*SQR(C)); parentheses are used to control the order of evaluation in a complex expression. The evaluation of an expression produces a numeric or string result that is used as an argument for a statement, or as part of another expression. An expression cannot stand alone; it must be an argument to a statement, function, or another expression.

The following examples demonstrate the features of expressions; the result of each expression is given to the right of the expression. In the examples, these variable values are used:

```
INTEGER J,K: J=4: K=7
REAL X,Y: X=2.3: Y=5.8
STRING A$,B$: A$=”ABC”: B$=”DEF”
```

Numeric expressions use integer or real arguments and return an integer or real result:

2+3	Result is 5
2.0+3	Result is 5.0. Because 2.0 is a real constant, 3 is converted to real before performing the addition.
J/K+2	Result is 2. J/K evaluates to 0.
X*Y	Result is 13.34.
SQR(X*X+Y*Y)	Result is 6.23969

String expressions use string arguments and return a string result::

CONCAT$(A$,B$)	Result is “ABCDEF”

Relational expressions are used to make logical decisions in a program; they use the relational operators (>, <, =, etc.) and return a 0 or nonzero result:

3>5	Result is 0, since 3 is not greater than 5.
A$<>B$	Result is nonzero, since A$ is not equal to B$.

Divelbiss 32-bit BASIC is unlike many BASIC dialects in that it forces the user to declare the mode of each variable, thereby optimizing the compiler's speed. With all variables predeclared, the compiler is not forced to evaluate all expressions in floating point at run time (which is a very slow procedure), and then convert to integer as the need arises. Instead, the algorithms used in Divelbiss 32-bit BASIC attempt to evaluate all expressions in the output mode (the mode of the variable to which the expression is being assigned). To make it easier to write programs, Divelbiss 32-bit BASIC provides automatic mixed mode expression evaluation. This means that an expression may consist of a combination of real and integer values. Divelbiss 32-bit Basic will automatically convert the components of the expression to the proper mode before evaluating it, and will convert the result to the mode of the variable to which the expression is being assigned. This is very convenient for programmers; however, there are some important implications arising from it.

Whenever an expression is to be assigned to a real variable, then every component of that expression is evaluated in real mode. Components of the expression which are integer (for example, integer variables), are automatically converted to real before any arithmetic is performed. This conversion takes place entirely within temporary values in the compiler; the integer values themselves are not changed. Whenever a constant is specified with no decimal point, the compiler assumes that it is an integer value. Any constant designated with a decimal point will be assumed to be real. Since the process of converting an integer to a real is relatively slow, faster code will result with real operations when all real operands are specified.

Expressions are defined in terms of parentheses. Whenever an expression in parentheses is encountered, this is treated as a new expression, although it may be part of a larger expression. This has significance when expressions are being evaluated which will be assigned to integer arguments. When the compiler encounters a new expression (one with parenthesis), it attempts to evaluate that expression in the mode of the variable to which it will be assigned. In the case of a real operator this is not important, since all values are converted to real before any operation takes place. With integer variables, however, if any component of an expression is real, the rest of that expression will be converted to real before the operation takes place. A few examples will make this clear.

```
100 INTEGER A
110 A=(1/2)*2
```

In this case, the expression will evaluate to the value zero. All operations specified are integer. Integer operations take place by truncating the result, so 1 divided by 2 evaluates to 0.

```
100 INTEGER A
110 A=(1.0/2)*2
```

This expression also evaluates to zero, but for a different reason. The inner 1.0/2 evaluates to .5, but after the value is calculated, the compiler attempts to convert this back to integer to be in the proper mode for variable A. The integer version of 0.5 is 0.

```
100 REAL A
110 A=(1.0/2)*2.0
```

In this case, the expression will evaluate to 1. Each of the operations is real, so all operations take place in real mode.

```
100 INTEGER MOTOR                        ‘ DAC value to send to motor controller
110 REAL SPEED                           ‘ Speed setpoint for motor
120 SPEED=750.0                          ‘ Set to 750 RPM
130 MOTOR=(SPEED/100.0)*32767/10         ‘ Convert RPM to DAC value
140 DAC 1,MOTOR
```

This example illustrates the problem that can occur with real/integer conversions in expressions. In the example, it is assumed that DAC channel 1 is attached to a motor controller which accepts a 0 to 10 VDC input; each 1 VDC corresponds to 100 RPM (i.e. 3.4 VDC is 340 RPM). Line 130 converts the speed, given in RPMs, to the DAC value necessary to set the motor controller speed. Since the DAC statement works with integer values, it seems reasonable that MOTOR should be an integer. SPEED is a real, so we want the calculation to be handled using real numbers (result=24575.25); the result should be truncated and stored in MOTOR (24575). Unfortunately, when Divelbiss 32-bit BASIC evaluates the parenthesis, it converts the result at that point (7.50) to an integer (7), and then evaluates the rest of the expression in integer mode. The integer multiplication of 7*32767/10 causes a final result of 22936.

By changing MOTOR to a real, the calculation in line 130 is performed as intended, providing a result of 24575.25 to be stored in MOTOR. When the DAC statement is executed in line 140, MOTOR is truncated to 24575, which is the expected result.

If MOTOR is left as an integer, and the parentheses in line 130 are removed, then the correct result is also obtained. In this case, the entire calculation is performed in real mode, and then the result (24575.25) is truncated and stored in MOTOR.

Statements

Statements describe the actions to be taken by the Divelbiss 32-bit BASIC program. Multiple statements can be placed in a line, separated by colons (‘:’). Statements can be grouped into seven categories: program flow, data storage, memory access, I/O, function definition, multitasking, and miscellaneous.

Program Flow Statements

Divelbiss 32-bit BASIC normally executes statements in linear order, as they are stored in the program. The following statements modify the program flow to allow code to be executed multiple times, to allow code to be shared, and to allow the program to make decisions.

CALL laddr [,arg...]	call an assembly language subroutine
CHAIN filenum or “fname”	load and run another program from EPROM
FOR num=expr TO expr [STEP expr]	beginning of a program loop; see NEXT
GOSUB linenum	call a subroutine; see RETURN
GOTO linenum	jump to a line number
IF expr THEN [:ELSE:]	conditional execution of rest of line
NEXT	end of a program loop; see FOR
ON ERROR linenum	GOTO linenum if an error occurs
ON expr, GOSUB linenum [,linenum...]	call subroutine based on value of expr
ON expr, GOTO linenum [,linenum...]	jump to linenum based on value of expr

RETURN	continue at line following GOSUB
STOP	halt execution of program

Data Storage Statements

These statements are used to declare the variables that are used in the program, and to embed data values in the program.

DATA arg [,arg...]	store data for READ to access
INTEGER varname [,varname...]	declare signed integer variables
READ arg [,arg...]	read values from DATA statements
REAL varname [,varname...]	declare floating-point variables
RESTORE	reset READ pointer to first DATA statement
STRING varname [,varname...]	declare text string variables

Memory Access Statements

These statements allow the programmer to directly access the Divelbiss 32-bit Controller memory.

EEPOKE ee_addr,int	store a word in EEPROM
POKE laddr,byte	store byte at laddr
WPOKE laddr,int	store integer value at laddr

Input/Output Statements

Control applications depend upon input and output with the real world, and the Divelbiss 32-bit Controllers support a wide variety of I/O devices, so this is the largest group of Divelbiss 32-bit BASIC statements.

CLS [arg]	erase the current FILE display device
CNTRMODE chan,int	set operating mode for counter
DAC chan,int	set D/A output level
DOUT chan,bin_val	control digital output channel
ERASE	erase the current FILE display device
FILE int [,arg...]	set current file I/O device
FINPUT fmt_str,arg	formatted INPUT from current FILE
FPRINT fmt_str,arg [,arg...]	formatted PRINT to current FILE
GETDATE month,day,year,wday	get current date from real time clock
GETIME hour,minute,second	get current time from real time clock
INPUT arg [,arg...]	wait for user input from current FILE
INPUT$ arg [,arg...]	allow commas in user input
INTERRUPT device,chan,task	attach a task to a hardware interrupt source
LINE (arg,arg)-(arg,arg) [,BF]	graphics line draw
LOCATE row,col	set cursor position for current FILE device
NETMSG	send/receive network messages
NETWORK 0,unit,int,int,int,st	initialize 32-bit Controller network handler
NETWORK 1,type,reg,int,unit,reg,st	send registers to another 32-bit Controller
NETWORK 2,type,reg,int,unit,reg,st	read registers from another 32-bit Controller
NETWORK 3,type,reg,expr,st	set the value of a network register
NETWORK 4,type,reg,varname,st	read the value of a network register
NETWORK 10,unit,mode,file,time_out	initializes network handler for MODBUS on 32-bit Controller

.

NETWORK 11,unit,func,add,data,stat	accesses other MODBUS units from 32-bit Controller
NETWORK 12 status	used to set the status of the 32-bit Controller on the MODBUS network
POLY arg,(arg,arg),arg	graphics polygon draw
PRINT arg [,arg...]	send output to current FILE device
PSET (arg,arg) [,arg]	graphics pixel draw
RDCNTR chan,int,varname	read counter value into varname
SETDATE month,day,year,wday	set current date of real time clock
SETIME hour,minute,second	set current time of real time clock
WRCNTR chan,int,expr	write to counter register(s)

Function Definition Statements

Divelbiss 32-bit BASIC allows the programmer to extend the language by adding functions. See section 3.9 for a complete description.

DEF funcname [,arg...]	mark beginning of a user defined function
FNEND	mark end of a user defined function

Multitasking Statements

These statements support the multitasking ability of Divelbiss 32-bit BASIC, allowing the programmer to control which tasks are running and how often they are running.

CANCEL task	halt the rescheduling of a task
EXIT	abort execution of current task
PRIORITY int	set priority of current task
RUN task [,int]	begin task execution; set reschedule interval
STOP task#	and execution of a task
TASK task	mark the beginning of a task area
WAIT int	suspend task execution for int tics

Miscellaneous Statements

DEBUG	halt compile process to display variables
INTOFF	disable all interrupt processing
INTON	enable all interrupt processing
RANDOMIZE	re-seed the random number generator
REM [string]	comment text
TRACEON	enable line number trace on the trace FILE
TRACEOFF	disable line number trace on the trace FILE

Functions

A function is called by referencing it in an expression. A function returns an integer, real, or string result. Functions can be grouped into five categories: math, string, memory access, I/O, and miscellaneous.

Math Functions

These include trigonometric, logarithmic, and bitwise logical functions.

ACOS (expr)	arccosine of expr

ASIN (expr)	arcsine of expr
ATAN (expr)	arctangent of expr
BAND (expr,expr)	bitwise logical AND of expr's
BOR (expr,expr)	bitwise logical OR of expr's
BXOR (expr,expr)	bitwise logical XOR of expr's
COS (expr)	cosine of expr
EXP (expr)	exponential function e**expr
LOG (expr)	natural logarithm (base e) of expr
LOG10 (expr)	common logarithm (base 10) of expr
RND	generate a pseudo-random number
SIN (expr)	sine of expr
SQR (expr)	square root of expr
TAN (expr)	tangent of expr

String Functions

These are used to work with strings and convert between strings and numbers.

ASC (strexpr)	ASCII equivalent of first char. in strexpr
CHR$ (expr)	one char. equivalent of expr
CONCAT$ (strexpr, strexpr)	appends second string after first
CVI (strexpr)	converts binary string to integer
CVS (strexpr)	converts binary string to real
LEN (strexpr)	length of strexpr
MID$ (strexpr,expr,expr)	substring of strexpr
MKI$ (expr)	converts an integer to a binary string
MKS$ (expr)	converts a real to a binary string
STR$ (expr)	converts a number to a string
VAL (strexpr)	converts a string to a number

Memory Access Functions

These functions allow the programmer to directly access the 32-bit controller memory.

ADR (varname)	address of specified variable
EEPEEK (ee_addr)	read a word from EEPROM
PEEK (laddr)	read a byte from laddr
WPEEK (laddr)	read an integer value from laddr

Input/Output Functions

Each of these functions returns a value read from an input device.

ADC (chan)	A/D value for chan
DIN (chan)	read digital input channel; 0 or 1
GET	waits for one byte from current FILE
KEY	reads one byte from FILE, doesn't wait

Miscellaneous Functions

ERR	last error number generated

User Defined Functions

Divelbiss 32-bit BASIC supports multi-line user defined functions. A function definition may be any number of lines long. All functions must be defined as follows:

```
100 DEF function_name [arguments]
110 function_definition
120 FNEND
```

DEF indicates the start of a definition. FNEND indicates the end of a definition. The function name can be up to seven characters, following the rules for variable names. The function name must be declared in an INTEGER, REAL, or STRING statement. The function can have arguments, which are part of the DEF statement, but are not part of the INTEGER, REAL, or STRING statement where the function is declared. The arguments are true variables and must be declared. When the function is called, the parameters passed to the function will be copied to these variables so they should have unique names.

NOTE: See INTOFF for more information and anomalies.

The following rules apply to user defined functions:

- Functions must be defined BEFORE they are used. It's a good idea to put all function definitions near the beginning of your program, after the variable definitions. If a function is referenced before it is declared, a FUNCTION ERROR will result.

Important

- A maximum of 64 functions can be declared in one program.

- Function definitions cannot be nested. A FUNCTION ERROR will result if a DEF statement is found inside of a function definition.

- Function names must be unique. Do not use other variable names or names of Divelbiss 32-bit BASIC statements or functions (even a function name very close to a Divelbiss 32-bit BASIC reserved word may not be acceptable).

- Arrays cannot be used as arguments to functions. Only simple strings, reals, or integers are legal.

- Do not attempt to also perform EEPROM console inputs or outputs inside of a function if it will be used in a multitasking program. Your program may hang up since Divelbiss 32-bit Basic blocks console access during some multitasking operations.

- Functions are not recursive. A function cannot call itself, either directly or indirectly.

- The result of a function can be assigned to the function name. To do this, in the function definition use an assignment statement to place the desired value in the function's name (i.e. reference the function name like it was a variable name). In the assignment statement, do not specify the function's arguments on the left hand side of the "=" sign.

For example, the following function returns the value 1:

```
100 INTEGER FN1
110 DEF FN1
120 FN1=1
130 FNEND
140 PRINT FN1                          ‘ The value 1 will be printed.
```

The following function returns the sum of its arguments:

```
100 INTEGER FN1,A,B,C
110 DEF FN1(A,B,C)
120 FN1=A+B+C
130 FNEND
140 PRINT FN1(1,2,3)                   ‘ The value 6 will be printed.
```

Note that in line 120 the function is referred to without its arguments.

Here’s another example. This function returns the left N characters of a string:

```
100 STRING LEFT$(127), A$(127)
110 INTEGER N
120 DEF LEFT$(A$,N)
130 LEFT$=MID$(A$,1,N)
140 FNEND
150 PRINT LEFT$(“ABCDEF”,3)            ‘ This prints “ABC”
```

One function can reference another. For example:

```
100 INTEGER FNA, FNB
110 DEF FNB : FNB=1 : FNEND
120 DEF FNA : FNA=FNB : FNEND
130 PRINT FNA                          ‘ This prints “1”
```

If functions that are using strings call each other, the STRING SPACE EXCEEDED error can result if too many functions have partial string results stored in internal temporary storage. If the message appears, you’ve called too many functions that need intermediate string storage. Simplify your code somewhat.

Chapter Two
Writing Control Applications

This chapter is an introductory tutorial on using the Divelbiss 32-bit Controller for real time control applications. An example is provided to help clarify the programming and operation of the Divelbiss 32-bit Controller in a real application; it is a moderately complex control system that uses multitasking and the Bear Direct network interface. The program makes use of features which are described later in this manual, so the reader is advised to make use of the index when topics come up which haven't been covered previously.

Programming for Real Time Control Systems

The term ‘real time control’ implies that certain operations must be performed at particular times. This differs from the general application program, which is not timing dependent. For example, it doesn’t matter too much whether a spreadsheet takes 1 second or 10 seconds to recalculate; on the other hand, it matters very much whether a motor controller ramps the motor speed down in 1 second or 10 seconds. Real time events fall into three categories: events that must happen before a specified time, events that must happen at a specified time, and events that must happen after a specified time. Real time control often involves managing multiple processes concurrently; in other words, several things may be happening at the same time.

The Primary Rule of Real Time Programming

At its simplest, real time programming can be reduced to one rule: the program cannot disregard any of the processes for too long. At any time that the program is waiting for an event to occur in one process, it must still be handling the other processes. The following example implements a system with two switches and two counters, with a counter being incremented when the corresponding switch is closed. This example breaks the rule by waiting for a switch to open, without continuing to check the other switch.

```
100 INTEGER C1, C2                  ‘ Counter variables
105 C1=0: C2=0                      ‘ Initialize the counters to 0
110 IF DIN(1)=0 THEN 140            ‘ Continue if switch 1 not pressed
120 C1=C1+1                         ‘ Switch 1 pressed, increment counter
125 PRINT “C1 = “; C1
130 IF DIN(1)=1 THEN 130            ‘ Wait for switch 1 to be released
140 IF DIN(2)=0 THEN 170            ‘ Continue if switch 2 not pressed
150 C2=C2+1                         ‘ Switch 2 pressed, increment counter
155 PRINT “C2 = “; C2
160 IF DIN(2)=1 THEN 160            ‘ Wait for switch 2 to be released
170 GOTO 110                        ‘ Loop forever
```

This program almost works correctly. When no switch is closed, the program will loop from line 110 to line 140 to line 170 and back to line 110. If switch 1 is closed, then it will increment C1 and print the new C1 value in lines 120 and 125. It will then sit in a loop in line 130 until switch 1 is opened; when this happens, it will resume looping from 110 to 140 to 170, looking for a switch closure. Switch 2 and C2 are handled in the same manner. The problem occurs when its waiting for a switch to be opened (lines 130 and 160). If it is looping in line 130, for instance, switch 2 could close and open while switch 1 remains closed, and the program would not detect the switch 2 closure. The following example handles this problem and counts the two switches correctly.

```
100 INTEGER C(1)                    ‘ Counters
110 INTEGER S(1)                    ‘ Switch states
120 INTEGER J,K
130 C(0)=0: C(1)=0                  ‘ Initialize the counters to 0
140 FOR J=0 TO 1                    ‘ Loop for both switches
150  GOSUB 300                      ‘ Check for switch closure
160  IF K=0 THEN 190                ‘ Jump if no closure
170  C(J)=C(J)+1                    ‘ Increment counter
```

```
180  PRINT “C”; J;” = “; C(J)      ‘ Print counter
190 NEXT J
200 GOTO 140                      ‘ Loop forever
210 ‘
300 ‘ Subroutine to check for switch ‘J’ closure. If the switch was previously
310 ‘ open and is closed now, then return a 1; otherwise, return a 0.
320 K=DIN(J+1)
330 IF K=1 AND S(J)=0 THEN S(J)=K: RETURN
340 IF K=0 AND S(J)=1 THEN S(J)=K
350 K=0: RETURN
```

This program is designed much better than the previous one; it is easily modified to handle more switches, by changing the size of the arrays ‘C’ and ‘S’ and the loop size in line 140. It constantly checks the state of both switches, incrementing the count when it detects that a switch has closed. Since it continues checking the state of both switches at all times, it won’t miss a switch closure. The PRINT statement in line 180 will take about 5 milliseconds to complete (5 characters at 9600 baud). This means that it could miss a switch closure that lasts less than 5 msec; normally, a switch closure this short would be interpreted as a glitch, anyway, but it is still worth noting. In real time programming, any time delay must be examined to see how it affects the operation of the system.

The first program could be made to work correctly by using the multitasking feature of Divelbiss 32-bit BASIC. In the second example, extra code was written to allow the processor to perform two functions at once; actually it switched between them quickly to give the appearance of concurrent operation. This is precisely what the Divelbiss 32-bit BASIC context switcher does, so the program could be simpler if it took advantage of the context switcher. Two tasks would be written, with each one devoted to monitoring a switch. The following example demonstrates this technique; note that the two tasks are very similar to the two loops from the first example. Because each task gets 50 percent of the processor time, by executing every other tick, a switch state that is shorter than 10 msec (one tick) may be missed. For example, if the program is executing in line 210, and switch 2 toggles low and then high again 5 msec later, then when task 2 executes it will see input 2 high as if nothing had happened.

```
100 INTEGER C1, C2                ‘ Counter variables
110 C1=0: C2=0                    ‘ Initialize the counters to 0
120 RUN 1
200 ‘ Handle first switch
210 IF DIN(1)=0 THEN 210          ‘ Wait for switch 1 to be pressed
220 C1=C1+1                       ‘ Switch 1 pressed, increment counter
230 PRINT “C1 = “; C1
240 IF DIN(1)=1 THEN 240          ‘ Wait for switch 1 to be released
250 GOTO 210                      ‘ Loop forever
300 ‘ Handle second switch
305 TASK 1
310 IF DIN(2)=0 THEN 310          ‘ Wait for switch 2 to be pressed
320 C2=C2+1                       ‘ Switch 2 pressed, increment counter
330 PRINT “C2 = “; C2
340 IF DIN(2)=1 THEN 340          ‘ Wait for switch 2 to be released
350 GOTO 310                      ‘ Loop forever
```

Managing Timing in a Control System

Obviously, one of the most important functions of a real time program is to ensure that control operations happen at the proper time. Divelbiss 32-bit BASIC provides three methods for handling real time scheduling of operations: the real time clock, the WAIT statement, and the hardware timer. Each of these covers different areas of real time scheduling.

With a resolution of one second, the real time clock is only useful for handling relatively long time periods. This makes it ideal for keeping track of down-time, run-time, shift totals, daily totals, and other long term production statistics. Care must be used when programming with a real time clock, because of the possibility of wrap-around at the end of the minute, end of the hour, and end of the day. Also, the programmer must be aware of the size of numbers that result when working with time of day values; for example, there are 86400 seconds in a day.

The following example shows how to handle the real time clock for long time periods:

```
100 ‘ Example to demonstrate using the real time clock to measure
102 ‘ long time intervals.
110 INTEGER HA, MA, SA
120 INTEGER J, K, FLAG1, END1
130 FLAG1=0
190 ‘ When input 1 turns on, turn on output 1. When input 1 turns off,
192 ‘ wait for 140 seconds, then turn off output 1.
200 J=DIN(1)
210 IF J=0 AND FLAG1=0 THEN 200             ‘ Input off, keep waiting
220 IF J=1 AND FLAG1=1 THEN 200             ‘ Input on, keep waiting
230 IF J=0 THEN 300                         ‘ Jump if input just turned off
240 ‘ Input just turned on, so turn on output
250 DOUT 1,1
260 FLAG1=1
270 GOTO 200
300 ‘ Input just turned off, so wait for 140 seconds, then turn off output 1.
310 GETIME HA,MA,SA                         ‘ Get current time
320 END1=MA*60+SA+140                       ‘ Calculate end time (cur.time+140)
330 IF END1>3599 THEN END1=END1-3600        ‘ Convert to 0-3599 range
340 GETIME HA,MA,SA
350 IF MA*60+SA <> END1 THEN 340            ‘ Wait for 140 seconds to pass
360 DOUT 1,0
370 FLAG1=0
380 GOTO 200
```

In order to execute a 140 second delay, the program reads the real time clock in line 310. It converts the minutes (0-59) and seconds (0-59) to just seconds (0-3599), and adds in the 140 second delay time. This could result in a number larger than 3599, so it checks for this in line 330, wrapping the result around if it is larger than 3599. The program then loops in lines 340 and 350, waiting for the 140 seconds to pass.

With a resolution of approximately 10 msec, the WAIT statement is suitable for medium time intervals, on the order of 20 msec to 300 seconds. In a multitasking program, it can be difficult to predict exactly what time delay a particular WAIT statement will generate, since it depends upon what the other tasks are doing at the time, as well as the relative priorities of all of the tasks.

For critical timing functions, the Divelbiss 32-bit Controller relies on the hardware timer, which has a resolution of a 34.722 microseconds. Because of the way Divelbiss 32-bit BASIC works internally, the fastest time interval that can be reliably handled is 1 millisecond. The timer can be read periodically by the BASIC program to perform its timing functions, or a timer interrupt handler can be set up. Note that the timer is a 32 bit integer number that rolls over approximately 20.7 hours. The following example shows how the WAIT statement timing can vary, and how to read the timer to measure small time intervals:

```
105 INTEGER X,RL,RLL,RLH
110 INTEGER TM
120 INTEGER T2,T3
200 GOSUB 1000                          ‘ Start the timer running
210 RUN 2
220 RUN 3,1
230 T2 = 0
300 TM = 0                              ‘ Reset the timer value
310 WAIT 50                             ‘ Wait for 50/100 second
320 PRINT TM                            ‘ Display length of WAIT in msec
330 GOTO 300
1000 INTERRUPT 3,0,1                    ‘ Set up timer vector to task 1
1020 RL=1/.000034722 / 1000.0           ‘ 1000 ints/sec reload value
1040 TIMER 5,0,RL                       ‘ Init reload value
1050 TIMER 1,0,RL                       ‘ Init timer value and start timer 0
1070 RETURN
1100                                    ‘ Timer interrupt task. Called 1000 times per
                                        ‘ second by hardware
1102                                    ‘ timer 0.
1105 TASK 1
1120 TM=TM + 1                          ‘ Increment global timer value
1130 EXIT                               ‘ End of timer interrupt task
2000 TASK 2
2010 T2 = T2 + 1
2020 IF T2 = 10000 THEN PRIORITY 1      ‘ Bump to higher priority for a while
2030 IF T2 = 20000 THEN T2=0:PRIORITY 0 ‘ Back to original priority
2040 GOTO 2010
2100 TASK 3
2110 FOR T3 = 1 TO 8000                 ‘ Delay for a while, then exit
2120 NEXT T3
2130 EXIT
```

This program sets up a timer interrupt task to execute 1000 times per second. Timer 0 and the interrupt are initialized in lines 1000 to 1070. Task 1 is the timer interrupt task; it simply increments the TM variable. Lines 300 to 330 perform a 50 tick (500 msec) WAIT, then print the actual number of milliseconds that the WAIT took to complete. Task 2 just increments a variable, and periodically sets its priority to a higher value, causing task 0 and task 3 to be suspended until task 2 resets its priority.

Task 3 executes a short loop (to waste some time) and then exits, to be restarted after 1 tick. When task 2 sets itself to a higher priority, the task 0 WAIT statement will take much longer than 50 ticks (about 300 ticks), because task 0 can’t be run, even though it is ready to run after the 50 tick waiting period. Periodically, task 3 will be ready to run when the task 0 WAIT statement completes; task 3 will run before task 0 gets to run, causing an extra tick to pass before task 0 executes.

The following output was produced when this program was run; it shows both of these effects:

```
502
495
506
495
505
496
495
505
496
505
2942
495
505
495
496
505
496
506
```

Project Specification

The project specification is an extremely important element of the control system design. A complete, well thought out specification will increase the reliability of the system, and can dramatically decrease the implementation time for the system. The ideal specification would completely describe all components of the system in enough detail to allow an independent party to construct the system; this should be the goal of the specification writer. In reality, of course, this is not possible; questions will always arise during the project implementation that had not been previously considered.

The exact contents of the specification depend on the project, but in general the following elements should be included:

- General description of application. Ideally, this will be detailed enough that someone who is not familiar with the application will be able to understand it.

- Input/Output requirements. List the inputs and outputs required to support the application, including parameters such as voltage, current, pulse rate, temperature, etc.

- Screen and report formats. List the layout of any display screens or printed reports. This should be done on grid paper to ensure that the text will fit in the available space.

- Timing requirements. List any timing requirements, including both timing required for hardware reasons and timing desired by the user. For a machine control application, this could include machine cycle time, setup time, motor speed, minimum speed ramp time, sample rate for a control parameter, etc. For an application that uses a personal computer, this could include file access times, network poll rate, screen update speed, etc. Remember that minimum and maximum times can both be important.

● Mechanical requirements. List any special mounting or size requirements. This may also include mechanical information about the machinery that will be interfaced to.

● Environmental requirements. List the operating and storage environment for the system. This includes temperature, humidity, radiation, etc.

● Hardware requirements. This is a list of hardware items to be used or interfaced with. Include a note as to the reason for the requirement (i.e. because it is a standard part already in inventory, or because it is the only part that meets a particular specification). Include exact part numbers, if possible.

● Software requirements. This is a list of software packages to be used or interfaced with. Include a note as to the reason for the requirement. Note any special hardware that may be required by this software (i.e. math coprocessor, modem).

● Initial system testing. List the items that should be tested “in the lab”, including the verification procedure for each item.

● Final system testing. List the items that should be tested “in the field”, including the verification procedure for each item.

Real Time Programming Example

Many aspects of software design only come to light in larger programs. This section demonstrates the use of Divelbiss 32-bit BASIC by implementing a realistic example program. The goal is to develop a program that will perform a typical industrial control operation. Portions of this program will be useful in the user’s own programs. The example goes through the project specification, program development, and system testing phases.

This example is based on a hypothetical rewind machine in which the product is drawn off of an input roll and rewound onto shorter output rolls. The output rolls are wound onto cores which feed in from a hopper. The product wraps onto the core as the machine speed ramps up. The machine runs at operating speed until it nears the end of the output roll, then the speed is ramped down. When the machine stops, a knife cuts the product and a gate opens, allowing the output roll to fall onto a conveyor belt to be taken away. Several pieces of production data must be maintained by the Divelbiss 32-bit Controller, in order to be uploaded to a personal computer at the end of each shift. The operator will enter operating parameters into the Divelbiss 32-bit Controller using the keypad; the display will indicate the current status of the rewind operation.

Example Project Specification

The following specification shows the level of detail that should be the goal of the system designer. In practice, much of this information is often only expressed verbally, which may be acceptable if the project is small or if everyone involved has experience with the application. In a larger project, or if some parties are unfamiliar with the application, then a detailed specification will help to prevent misunderstandings. Typically, a specification such as the following example would be arrived at through discussions between the client and the programmer.

The example is printed in courier 12cpi to distinguish it from the rest of the text.

.

General description of application.

A control system is needed for a new rewind machine which is being constructed. The machine pulls the product off of an input roll. The input roll will be approximately 3000 yards long and 2.5 feet in diameter. Each product number has a preset length, between 10 and 30 yards, associated with it; this length of product will be wound onto each output roll. The Divelbiss 32-bit Controller is attached to the main drive motor; it ramps the speed up, winds at a preset speed, and ramps the speed down. A shaft encoder (quadrature, with A, B, and MARK outputs) will be used to measure the main drive motion. The ramp down must be calculated so that the main drive stops within 0.25 inches of the selected length. When the main drive is stopped, the Divelbiss 32-bit Controller will cause a knife to cut the product and then retract. A gate is then opened to allow the output roll to drop onto a conveyor to be taken away; two optical sensors are positioned to detect that the roll drops. The gate is closed, then another gate is opened to allow a new core to drop into position from a hopper above the machine. The operator attaches the product tail to the core then presses the RUN pushbutton to start the rewind operation again.

If an error is detected, then the machine will be stopped and an error message will be displayed on the Divelbiss 32-bit Controller; the operator will correct the error condition, insert a new core, and press the RUN pushbutton to start a new roll.

The user interface will consist of the Divelbiss 32-bit Controller keypad and display, the RUN pushbutton, and the emergency stop switch (ESTOP). The operator enters the product number, batch number, input roll length, operator number, and downtime code from the keypad. The display will show the remaining product on the input roll and the number of output rolls produced. When the input roll is removed, the Divelbiss 32-bit Controller will display the number of yards left (if any) on the input roll; the operator will write this on a tag to go on the roll. If it takes longer than 60 seconds after a roll is produced for the operator to press the RUN button, then the Divelbiss 32-bit Controller will accumulate downtime (in seconds); the operator must enter a downtime code before the machine can be run again. One of the downtime codes is “maintenance”; if this is entered, then the machine can be run, but no production data will be collected.

Production data is collected and transferred over the Bear Direct network periodically to a personal computer (PC). Each time that the operator enters a new product number or batch number, the totals for the previous product/batch are sent to the PC. At the end of the shift, the shift totals are collected by the PC.

The table of product lengths will be stored as part of the Divelbiss 32-bit Controller program, since no new products will be added in the foreseeable future.

Input/Output requirements

Hardware I/O points:

- Shaft encoder. This is a 600 pulse/rev biphase optical shaft encoder. The mechanical coupling with the machine produces 50 pulses per inch of material. It is connected to the Divelbiss 32-bit Controller’s onboard counter.

- Motor control. This is a -10 to 10 VDC analog signal to control the main drive speed; -10 VDC corresponds to 0 RPM, and 10 VDC corresponds to approximately 10 feet/second. This is connected to a DAC module on the Divelbiss 32-bit Controller.

- Knife output. This is a discrete output that causes the knife to cut the material when the output turns on. This output should stay on for approximately 200 msec.

- Optical sensor input. This is a discrete input that is connected to two optical sensors (with open collector outputs), that show when a roll is in place in the machine.

- Conveyor gate output. This is a discrete output that causes the lower gate to open, allowing the output roll to fall onto the conveyor. This output should stay on for approximately 200 msec after both optical sensors show that the roll has moved.

- Core hopper gate output. This is a discrete output that causes the upper gate to open, allowing the core to fall into place. This output can turn off as soon as both optical sensors show that the core is in place.

- RUN pushbutton input. This is a discrete input attached to a momentary pushbutton mounted near the operator's position.

- Emergency stop pushbutton input. This is a discrete input attached to a push-pull switch near the operator's position. Other poles of the emergency stop switch are connected to hardware failsafe systems; this input just informs the Divelbiss 32-bit Controller that the system has been stopped.

User input values:

- Product number. This is an integer number (0..9999) which the operator enters at the start of a product run.

- Operator number. This is an integer number (0..999) which the operator enters at the start of the shift.

- Input roll length. This is an integer number (0..9999) which the operator enters when a new input roll is mounted on the machine. It is the length of the input roll in yards.

- Batch number. This is an integer number (0..999999) which the operator enters at the start of a new batch.

- Downtime code. This is a number entered by the operator when the machine is stopped for more than 60 seconds. The operator will be prompted for the following downtime codes: Maintenance, break time, input roll loading, and machine jam.

- Display output values:
Output roll count for product. This is an integer number (0..99999) which shows the number of output rolls produced for this product so far in this shift.

- Output roll count for batch. This is an integer number (0..99999) which shows the number of output rolls produced for this batch so far in this shift.

- Number of yards left on the input roll. This is an integer number (0..9999) which shows the number of yards left on the input roll. This is calculated by subtracting the number of yards run through the machine from the input roll length; if the result is less than 0, then 0 should be displayed.

.

Network transfer values:

- Current product number.
- Current batch number.
- Current operator number.
- Total downtime thus far in shift.
- For each product completed, the product number, total number of rolls, and total yards are sent to the PC.
- For each batch completed, the batch number, total number of rolls, and total yards are sent to the PC.
- The shift totals for number of rolls, number of yards, and downtime (per category) are retrieved by the PC at the end of each shift. The operator number for the completed shift is also retrieved by the PC at the end of the shift.

Screen and report formats.

The display formats are not critical. The normal operating display will be similar to this:

```
Product # XXXX  Batch # XXXXXX
Prod:XXXXX Batch:XXXXX Yds left: XXXX
```

The format of any other displays may be chosen by the programmer.

Timing requirements.

The throughput of the machine should be maintained at the highest practical value. The operator should be able to enter the maximum run speed from the keypad. Other than these requirements and any previous constraints, there are no critical timing requirements.

Mechanical requirements.

The Divelbiss 32-bit Controller, its power supply, and any auxiliary equipment (excluding sensors) will be mounted in a NEMA 4 enclosure, approximately 12 inches wide by 16 inches high by 6 inches deep. All wiring will enter the enclosure through 0.75 inch holes punched in the bottom surface. All field wiring will terminate at screw-type terminal strips.

Environmental requirements.

The machine will be located in a normal factory environment.

Hardware requirements.

The optical shaft encoder and motor drive controller are being supplied by the customer; specifications are enclosed separately for these items. 120VAC will be brought into the enclosure; the Divelbiss 32-bit Controller and all I/O devices will be powered from the 120VAC source. Proper steps should be taken to ensure noise immunity and electrical isolation.

.

Software requirements.

The data should be stored on the PC in a Lotus 123 compatible file format, to facilitate post-processing of the data.

Initial system testing.

As much testing as is feasible will be performed prior to mounting the system on the actual machine. A shaft encoder will be provided for initial testing. The motor drive will not be available for initial testing.

Final system testing.

A machine will be made available for approximately two working weeks for system testing. An electrician and maintenance technician will be available during this time. The machine will undergo a five day acceptance period, during which it will be run under normal production conditions.

Often, during the engineering implementation, the specification must be changed in response to problems, new information, parameter changes, personal whims, etc. The actual printed specification should be updated and distributed to all interested parties when changes occur. This will help to avoid nasty surprises on anyone's part. Unfortunately, it is often not until a project has started to fall apart that the specification (or lack of one) becomes important.

Initial Software Design

For an experienced programmer, most of the initial phase of the software design occurs while reading the specification and while sitting in meetings. This is the part of the design where the overall structure of the program is laid out and data structures are chosen. For the less experienced programmer, it is often useful to consider this as an independent phase. The natural inclination for most people is to jump in and start writing the program; this often leads to programs which are confusing to read and hard to debug. As with the specification, a little bit of thought up front can save hours of frustration later.

Many different techniques can be used to lay out the structure of a program, such as flow charting, data flow diagrams, state descriptions, etc. Each of these has been the focus of entire books, and so won't be discussed in detail here. Primarily, the structure of a program depends upon two things: determining which actions must be performed sequentially (one following another), and determining which actions must be performed concurrently (taking place at about the same time). Drawing a state diagram will point up any concurrency issues in a design. The example program can be divided into 8 main states of operation. Some of these states could be broken down again into another level of state diagrams, but for a simple program like this one, it won't be necessary. Here are the main states:

1. Program initialization. Set up the program variables and system hardware prior to performing any control functions. Unless problems are detected which prohibit the system from being used, this goes to state 4.

2. Running a product roll. Using the specified product length, calculate the motor ramping parameters. Ramp the motor speed up to the preset operating speed, run until it reaches the ramp-down point, and ramp the motor speed down to a full stop. Cut the product by enabling the Knife output for 200 msec. Drop the product onto the conveyor belt. Drop a new core into position. This goes to state 3.

3. Update current production data. If not in maintenance mode, then update the current production totals. Update the display with the current number of rolls produced and the amount of material left on the input roll. This goes to state 4.

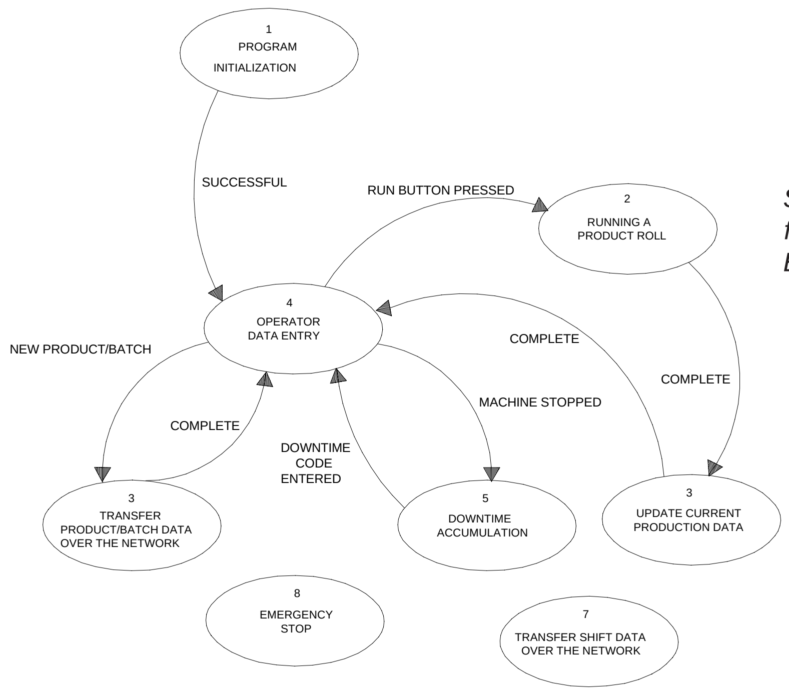
4. Operator data entry. Based on the function key that was pressed, prompt the operator to enter a product number, operator number, input roll length, or batch number. If a new product number or batch number is entered, then go to state 6. This goes to state 2 when the RUN pushbutton is pressed. If no operation is performed for 60 seconds, then this goes to state 5.

5. Downtime accumulation. Display the downtime codes. Accumulate downtime while waiting for the operator to enter a downtime code. When a downtime code is selected, this goes to state 4.

6. Transfer product/batch data over the network. When a new product or batch number has been entered, then the appropriate data is transferred to the central computer. This goes to state 4.

7. Transfer shift data over the network. When the central computer requests the shift end totals, transfer the data to it. This can occur while in any of states 2, 3, 4, 5, 6, or 8. Emergency stop. Put all outputs into a safe state. Wait for the emergency stop pushbutton to be released. This can occur while in any of states 2, 3, 4, 5, 6, or 7.

Figure 4 graphically shows the relationship between the states. Note that two of the states (“8 Emergency stop” and “7 Transfer shift data over the network”) aren’t shown connected to any other states. These two states can be entered at any time, based on outside events that occur asynchronously to the Divelbiss 32-bit Controller control program. In other words, they operate concurrently with the rest of the program. This makes them likely candidates for being implemented as individual tasks in the program.



State Diagram for Example Program

It is important to plan the program’s data structures before writing the program. The term “data structure” refers to the way that the program’s data is stored in variables. Data structure choices can have a huge effect on the operation of a program, both in terms of programming complexity and of runtime size and speed. Control applications generally have relatively simple data structures, but it still makes the programming go more smoothly if they are laid out before hand. A list of variables should be made to show the name of each variable, its type (integer, real, or string), a description, and a valid range (ie. 0.0 to 9.99). Note that the way a value is stored is often not the way that it is entered or displayed; for example, a number that may range from 0.0 to 99.9 may actually be stored as an integer between 0 and 999, to save space. This list doesn’t need to show the temporary variables used within the program, such as loop counters, intermediate results, etc.

The following is a preliminary list of variables that will be needed for the example program:

INLEN	Input roll length. A number in the range (0..9999); it is stored as an integer.
PRDNUM	Product number. A number in the range (0..9999); it is stored as an integer.
OPNUM	Operator number. A number in the range (0..999); it is stored as an integer.
BATNUM	Batch number. A number in the range (0..999999); it is stored as a text string.
LENSETP	Length setpoint for the current product, in shaft encoder pulses (1/50’s of an inch); over the 10 to 30 yard range of output roll length, this yields a number in the range (18000..54000). It is stored as a real.
CURLEN	Length of current roll being run, in shaft encoder pulses (1/50’s of an inch). A number in the range (18000..54000). It is stored as an integer.
DNTIME()	Array of current downtime totals. There are four downtime codes, for which downtime is accumulated separately in seconds over a shift, which is a range of (0..28800). This is an integer array.
TOTDNTM	Total downtime thus far in the shift, in seconds. A number in the range (0..28800); it is stored as an integer.
RROLCNT	Output roll count for current input roll. A number in the range (0..1000); it is stored as a real.
PROLCNT	Output roll count for product. A number in the range (0..99999); it is stored as a real.
PYARDS	Output roll total yards for product. A number in the range (0..9999); it is stored as a real.
BROLCNT	Output roll count for batch. A number in the range (0..99999); it is stored as a real.
BYARDS	Output roll total yards for batch. A number in the range (0..9999); it is stored as a real.
SROLCNT	Output roll count for shift. A number in the range (0..99999); it is stored as a real.
SYARDS	Output roll total yards for shift. A number in the range (0.9999); it is stored as a real.

Rewind Program Example

```
new
download

100 ‘ REWIND.BAS - Demo program for Divelbiss 32-bit Controller
      rewinder control
```

```
   ‘
   ‘
   ‘*****************************************************************
‘ Network register usage:
   ‘ I0   current product number
   ‘ I1   current operator number
   ‘ S0   current batch number
   ‘ I2   total downtime thus far in shift
   ‘
   ‘ I10   last product: new data available flag
   ‘ I11   last product: product number
   ‘ R10   last product: total number of rolls
   ‘ R11   last product: total yards
   ‘
   ‘ I20   last batch: new data available flag
   ‘ S1   last batch: batch number
   ‘ R20   last batch: total number of rolls
   ‘ R21   last batch: total yards
   ‘
   ‘ I30   shift end: end of shift flag
   ‘ I31   shift end: data available flag
   ‘ R30   shift end: total number of rolls
   ‘ R31   shift end: total yards
   ‘ I32   shift end: operator break downtime total
   ‘ I33   shift end: input roll downtime total
   ‘ I34   shift end: machine jam downtime total
   ‘ I35   shift end: maintenance downtime total

   ‘****************************************************************
‘ Variable declarations
   integer inlen          ‘ Input roll length. (0..9999)
   real ydsleft           ‘ Yards left on roll (0.0..9999.0)
   integer prdnum         ‘ Product number. (0..9999)
   integer opnum          ‘ Operator number. (0..999)
   string batnum$(6)      ‘ Batch number. (0..999999)
   real lnsetp            ‘ Length setpoint, 1/50’s of an inch
                            (18000..54000)
   integer curlen         ‘ Length of current roll, 1/50’s inch
                            (18000..54000)
   integer dntime(3)      ‘ Current downtime totals. (0..28800)
                          ‘  0 = operator break time
                          ‘  1 = change input roll
                          ‘  2 = machine jam
                          ‘  3 = machine maintenance
   integer totdntm        ‘ Total downtime for shift
   real rrolcnt           ‘ Output roll count for input roll (0..1000)
   real prolcnt           ‘ Output roll count for product.
   real pyards            ‘ Total yards for product.
   real brolcnt           ‘ Output roll count for batch.
   real byards            ‘ Total yards for batch.
   real srolcnt           ‘ Output roll count for shift.
   real syards            ‘ Total yards for shift.
   integer maintmd        ‘ 1 if in maintenance mode, otherwise 0
   integer estopmd        ‘ 1 if in emergency stop mode, otherwise 0
integer hour, min, sec, sttime
integer dhour, dmin, dsec

‘ Control loop variables
real curpos                          ‘ Counter value indicating posi-
                                       tion in roll
real spd
real accelrt
real decelrt
real maxspd
real decelpt

‘ I/O redirection variables. These are constant values that hold the
‘ channel number corresponding to each I/O function. This makes it
‘ easy to change the I/O layout, if necessary.

   integer IRUNBU                    ‘ Operator run pushbutton
   integer IESTOP                    ‘ Emergency Stop switch
   integer IOPTIC                    ‘ Optical sensor for core in
                                       position
   integer OMDRV                     ‘ Main drive enable
   integer OKNIFE                    ‘ Knife output
   integer OCONV                     ‘ Conveyor gate
   integer OCORE                     ‘ Core hopper gate

‘ Variables for user interface code.
   integer flush                     ‘ User defined function name
   integer flshtem                   ‘ Temp used by ‘flush’
   integer ch                        ‘ User interface temp
   integer timeout, delay            ‘ User interface timeout vari
                                       ables
integer K, uivlu, uilen
real uirvlu
string uifmt$(50), uifmt2$(50)
string uisvlu$

‘ Temporary scratchpad variables.
integer temp, a, b, c                ‘ Miscellaneous integer temps
integer done
integer t1tmp
integer t2, t2st
integer st
   real ftemp                        ‘ Real temp
   string tempstr$(40)               ‘ String temp

   ‘*****************************************************************
‘ DATA statements
   data 1000, 18000.0                ‘ Product # 1000, 360.0 inches
   data 1001, 36000.0                ‘ #1001, 720.0 inches
   data 1002, 5400.0                 ‘ #1002, 108.0
   data 1003, 5450.0                 ‘ #1003, 109.0

   data -1                           ‘ End of table flag
```

```
500 '*****************************************************************
' Function definitions
' Flush the keypad buffer and wait for a key to be pressed
' Returns: key pressed
' Modifies: flshtem
def flush
wait 10
520  if din($100) <> 0 then goto 520
    wait 20
530  if key <> 0 then goto 530
540  flshtem = din($100): if flshtem = 0 then 540
    flush = flshtem
   fnend

800 '*****************************************************************
' Initialization
file 6                         ' Use onboard display
' Set timeout delay to 30 seconds for waiting on the users input
delay = 300

   IRUNBUT = 1                 ' RUN button is on input 1
   IESTOP = 3                  ' Emergency Stop switch
   IOPTIC = 2                  ' Optical sensor
   OMDRV = 7                   ' Main drive enable is on output 7
   OKNIFE = 1                  ' Knife output
   OCONV = 2                   ' Conveyor gate output
   OCORE = 3                   ' Core hopper gate

   maintmd = 0                 ' Not in maint. mode to start with
for a = 0 to 3
dntime(a) = 0
next a
if len(batnum$) > 6 then batnum$ = "000000"
if prdnum < 0 or prdnum > 9999 then prdnum = 0
maxspd = 300.0
accelrt = 200.0
decelrt = 500.0
totdntm = 0
   estopmd = 0                 ' Not in emerg. stop mode to start with
   cntrmode 1,1                ' Quadrature X1 mode

   dout 5,1: dout 6,1          ' Enable shaft encoder inputs
   dout 0,1                    ' Set counter reset/latch to reset

   run 1,2                     ' Start emergency stop task
   run 2,2                     ' Start shift end task

   a = eepeek(1)               ' Get unit number
   network 0,a,50,50,2,b       ' Initialize network with 50 integers,
                               ' 50 reals, and 2 strings
```

```
1000 '****************************************************************
' Mainline loop. This is where we sit while waiting for the
' operator to press a key or hit the RUN pushbutton. If it sits
' here for more than 60 seconds, then gosub 4000 to accumulate
' downtime.
getime hour,min,sec
   sttime = min*60+sec                ' Save start time
   gosub 8100                         ' Display fixed part of main
                                        screen

1020 ch = key
if ch = $41 then gosub 1200: goto 1000
if ch = $42 then gosub 4300: goto 1000
if din (IRUNBUT) then gosub 2000: goto 1000

' Don't look for downtime if currently in maintenance mode.
if maintmd then 1020

' Has it been 60 seconds?      Don't forget to handle hour wrap-around.
                               getime a,b,c
a = b * 60 + c
if a-sttime < 0 then sttime = sttime-3600
if a-sttime > 60 then gosub 4000: goto 1000
goto 1020

1200 '****************************************************************
' Operator Data Entry
erase
uifmt$ = "Batch Number > ": uisvlu$ = batnum$: uilen = 6
gosub 8600: if K = 0 then 1250

   network 3,2,1,batnum$,a      ' Store batch number
   network 3,1,20,brolcnt,a     ' Store batch roll count
   network 3,1,21,byards,a      ' Store batch yards
   network 3,0,20,1,a           ' Set "data available" flag

batnum$ = uisvlu$
   brolcnt = 0.0                ' Clear batch roll count
   byards = 0.0                 ' Clear batch yards count
   network 3,2,0,batnum$,st     ' Store batch number in network var.

1250 uifmt2$ = "I4"
uifmt$ = "Product Number (0-9999) > ": uivlu = prdnum
gosub 8300: if K = 0 then 1400

' The operator typed a product number in, so we need to see if it
' is in the list.
restore
1300 read a
if a = uivlu then 1370
if a <> -1 then read a: goto 1300
```

```
‘ Looked through entire DATA table and didn’t find it, so display
‘ a message and let the operator try again.
erase
print “Unknown product number entered”;
wait 200: goto 1250

‘ Found it, so update the network registers, read the length setpoint,
‘ and continue.
1370 network 3,0,11,prdnum,           ‘ a Store product number
   network 3,1,10,prolcnt,a           ‘ Store product roll count
   network 3,1,11,pyards,a            ‘ Store product yards
   network 3,0,10,1,a                 ‘ Set “data available” flag

   prdnum = uivlu                     ‘ Update product number
   read lnsetp                        ‘ Read the new length setpoint
   prolcnt = 0.0                      ‘ Clear the roll count
   pyards = 0.0                       ‘ Clear the yards count
   network 3,0,0,prdnum,st            ‘ Store product number in
                                        network var.

1400 uifmt$ = “Input Roll Length (1-9999) > “: uivlu = inlen
gosub 8300: if K = 0 then 1450
inlen = uivlu
rrolcnt = 0
ydsleft = inlen
1450 uifmt$ = “Operator Number (0-9999) > “: uivlu = opnum
gosub 8300: if K = 0 then 1490
opnum = uivlu
network 3,0,1,opnum,st                ‘ Store operator number in
                                        network var.

1490 return

2000 ‘****************************************************************
‘ Run a Product Roll
if estopmd then 2990                  ‘ EStop, so don’t run!

‘ Start main drive moving slowly
wrcntr 1,0,0
dac 1,580
dout OMDRV,1                          ‘ Enable main drive
wait 10

‘ Main drive control loop to run specified length. This ramps the
‘ drive speed up to its running speed, runs until it gets close to
‘ the stop point, then ramps the drive back down to 0, stopping
‘ when it reaches the correct length.
2050  rdcntr 1,2,curpos               ‘ Read current position in roll
    spd = curpos / accelrt * 51.1
    if spd > maxspd then spd = maxspd
    decelpt = (lnsetp - curpos) / decelrt * 51.1
    if spd > decelpt then spd = decelpt
    if spd < 20.0 then spd = 20.0     ‘ Maintain a minimum speed
if spd > 500.0 then spd = 500.0
if estopmd then 2990                 ‘ EStop button pressed, so exit
dac 1,spd + 512.0
if curpos < lnsetp then 2050
‘ Reached the specified length, so stop the main drive.
dac 1,512
dout OMDRV,0                         ‘ Disable main drive

‘ Cut the product
dout OKNIFE,1
wait 20
if estopmd then 2990
dout OKNIFE,0

‘ Drop the product onto the conveyor
erase
print “Waiting for roll to drop onto conveyor”
dout OCONV,1
2100 if estopmd then 2990
if din(IOPTIC) = 1 then 2100
dout OCONV,0

‘ The roll was successfully produced, so update production statistics
gosub 3000
‘ Drop a new core into position
erase
print “Waiting for core to drop from hopper”
dout OCORE,1
2150 if estopmd then 2990
if din(IOPTIC) = 0 then 2150
dout OCORE,0

2990 ydsleft = ydsleft - lnsetp / 50.0 / 36.0
if ydsleft < 0.0 or ydsleft > inlen then ydsleft = 0.0
return

3000 ‘****************************************************************
‘ Update Current Production Data
rrolcnt = rrolcnt + 1
if maintmd then 3490

   ftemp = lnsetp / 50.0 / 36.0      ‘ Calc length of roll in yards

   brolcnt = brolcnt + 1.0           ‘ Update batch total
   byards = byards + ftemp           ‘ Update batch yards
   prolcnt = prolcnt + 1.0           ‘ Update product total
   pyards = pyards + ftemp           ‘ Update product yards
   srolcnt = srolcnt + 1.0           ‘ Update shift total
   syards = syards + ftemp           ‘ Update shift yards

3490 return
```

```
4000 ‘****************************************************************
‘ Downtime Accumulation. Wait for the operator to select a downtime
‘ code, then add current downtime to that downtime total. Note
‘ that maintenance downtime is a special case; it sets the ‘maintmd’
‘ flag and returns, so that the machine can still be run.
erase
print “F1-Break  F2-Input Roll  F3-Jam”
print “F4-Maintenance”;
dhour = hour
dmin = min
dsec = sec

4030 ch = get
if ch < $41 or ch > $44 then 4030
if ch = $44 then 4100
ch = ch - $41
gosub 4400                           ‘ Update downtime array

erase
if ch = 0 then print “Break”;
if ch = 1 then print “Input Roll”;
if ch = 2 then print “Jam”;
print “ downtime total > “; dntime(ch);
wait 500
goto 4190

4100 ‘ Maintenance downtime selected
maintmd = 1
4190 return

4300 ‘****************************************************************
‘ Maintenance Downtime Accumulation.
   if maintmd = 0 then 4390          ‘ Exit if not in maintenance mode
   ch = 3                            ‘ Select maintenance downtime
   gosub 4400                        ‘ Update downtime array
maintmd = 0
4390 return

4400 ‘****************************************************************
‘ Get current time and calculate downtime, then add to selected
‘ downtime array element.
‘ Input:
‘  ch = downtime array element to update (0-3)
‘ Output:
   ‘  dntime(ch) = updated
   ‘  ch is unmodified
getime a,b,c
a = a - dhour
if a < 0 then a = a + 24
b = b - dmin
if b < 0 then b = b + 60
c = c - dsec
if c < 0 then c = c + 60
```

```
a = a * 24 + b * 60 + C
dntime(ch) = dntime(ch) + a
totdntm = totdntm + a
network 3,0,2,totdntm,st                    ‘ Store total downtime in
                                              network var.
return

5000 ‘**************************************************************
‘ Emergency Stop
task 1
t1tmp = din(IESTOP)
if t1tmp = 0 then estopmd = 0: exit         ‘ EStop button not pressed
if t1tmp = 1 and estopmd = 1 then exit      ‘ Already in EStop mode

‘ Signal the rest of the program that EStop has occurred.
estopmd = 1

‘ Force the main drive to stop. This happens even if we don’t
‘ think that it is currently running.
   dac 1,512                                ‘ Stop main drive
   wait 100                                 ‘ Wait for it to stop
   dout OMDRV,0                             ‘ Disable main drive
exit

6000 ‘**************************************************************
‘ Transfer Shift Data over Network
task 2
   network 4,0,30,t2,t2st                   ‘ Read the “end of shift”
                                              flag
   if t2 = 0 then 6390                      ‘ If flag clear then exit

‘ Shift has ended, so copy data into network registers for PC to read.

network 3,1,30,srolcnt,t2st
srolcnt = 0.0                               ‘ Clear the roll count
network 3,1,31,syards,t2st
syards = 0                                  ‘ Clear the yards count
for t2 = 0 to 3
network 3,0,32+t2,dntime(t2),t2st
dntime(t2) = 0                              ‘ Clear the individual
                                              downtimes
next t2

‘ Now set the flags so that the PC will read the new data.
   network 3,0,30,0,t2st                    ‘ Clear the “end of shift”
                                              flag
   network 3,0,31,1,t2st                    ‘ Set “data available” flag

   totdntm = 0                              ‘ Clear the downtime total

6390 exit
```

```
 '*****************************************************************
' Need this TASK statement to prevent corruption of temporary vari-
   ables.

task 3
8000 '*****************************************************************
' Get a key with timeout
' Output: ch, timeout
timeout = 0
8010 if din($100) <> 0 then goto 8010
wait 10
8020 if key <> 0 then goto 8020
8030 wait 10: ch = key
if ch = 0 and timeout < delay then timeout = timeout + 1: goto 8030
return

8100 '*****************************************************************
' Display main screen
' Input:
   '  maintmd, prdnum, batnum$, prolcnt, brolcnt, rrolcnt, ydsleft
   '  lnsetp
' Modifies:
'  ftemp
erase
print "Product# ";
fprint "i4x4z",prdnum
print "Batch# "; batnum$; "  ";
if maintmd then print "Maint";
locate 2,1
print "Prod:";
fprint "f5.0z",prolcnt
print "  Batch:";
fprint "f5.0z",brolcnt
print "  Yds left:";
a = ydsleft
fprint "i4z",a
return

8300 '*****************************************************************
' Subroutine to get INTEGER user input.
cls: print uifmt$; uivlu;: wait 30
8310 K = din ($100): if K=0 then 8310
if K = $41 or K = 13 then K=key: K=0: return
if K<$30 or K>$39 then K=key: goto 8310
locate 1,len(uifmt$)+1: print "     ";
locate 1,len(uifmt$)+1: finput uifmt2$, uivlu
K=1: return

8400 '*****************************************************************
' Subroutine to get REAL user input.
cls: print uifmt$;: fprint "F4.2z",uirvlu
print "F8 is decimal point";: wait 30
8410 K = din ($100): if K=0 then 8410
```

```
if K = $41 or K = 13 then K=key: K=0: return
if K<$30 or K>$39 then K=key: goto 8410
locate 1,len(uifmt$)+1: print “                   “;
locate 1,len(uifmt$)+1: finput uifmt2$, uirvlu
K=1: return

8500 ‘***************************************************************
‘ Subroutine to get boolean (0 or 1) user input.
cls: print uifmt$; uivlu: print uifmt2$;: wait 30
8510 K = get
if K = $41 or K = 13 then return
if K <> $7F then 8530
locate 1,len(uifmt$)+1: print “0                  “;
uivlu=0: goto 8510
8530 if K<$30 or K>$31 then 8510
locate 1,len(uifmt$)+1: print chr$(K); “          “;
uivlu=K-$30
goto 8510

8600 ‘***************************************************************
‘ Subroutine to get string user input.
cls: print uifmt$; uisvlu$;: wait 30
8610 K = din ($100): if K=0 then 8610
if K = $41 or K = 13 then K=key: K=0: return
8620 locate 1,len(uifmt$)+1: print “              “;
locate 1,len(uifmt$)+1
a = 0
uisvlu$ = “”
8630 K = get
if K = 8 then 8620
if K = 13 then 8690
uisvlu$ = concat$(uisvlu$,chr$(K))
print chr$(K);
a = a + 1
if a < uilen then 8630
8690 K=1: return
```

Program Operating Instructions

The program has been designed to run on the Divelbiss Boss 32 Demonstration Case; the I/O numbers were chosen to match the wiring of this case. In order to run the program without a Demo Case, the following I/O devices must be supplied by the user:

Run pushbutton	input 1
Switch to simulate optical sensor	input 2
Emergency stop switch	input 3
Knife output	output 1
Conveyor gate open	output 2
Core hopper gate open	output 3
Main drive enable to turn on motor controller	output 7
Main drive speed select	DAC channel 1
Product length encoder	counter 1

The main drive motor must be coupled to the product length shaft encoder in order for the program to operate correctly, because the program uses the shaft encoder value to control the motor speed.

Before running the example program, EEPROM location 1 must be set to the Divelbiss 32-bit Controller unit number, so that the network works correctly. This is done by writing a short program to POKE the correct value into location 1. For example, if there is only one Divelbiss 32-bit Controller wired up to the network, the run the program 100 EEPOKE 1,1 (only one line).

Download the program into the Divelbiss 32-bit Controller and RUN it. The main screen will be displayed:

```
Product# 1002  Batch# 963895
Prod:  0  Batch:  0  Yds left:8532
```

If no action is taken for 60 seconds (no keypad buttons pressed, Run button not pressed, and Emergency Stop button not pressed), then the downtime screen will be displayed:

```
F1-Break        F2-Input Roll
F3-Jam          F4-Maintenance
```

Press F1, F2, or F3 to enter a downtime code; the number of seconds attributed to that downtime code thus far in the shift will be displayed. Press F4 to enter maintenance mode; the main screen will be redisplayed with “Maint” showing in the top right corner; the system may be run, but any rolls produced will not count towards production totals. When in maintenance mode, press F2 from the main screen to exit maintenance mode.

While in the main screen, press F1 to enter the “Operator Data Entry” state. It will ask for the batch number, product number, input roll length, and operator number. Enter a six digit number for the batch number; the function keys will enter the characters A through F. For the product number, it will only accept the numbers 1000, 1001, 1002, and 1003; these are the only numbers in the program DATA statements. Enter a number between 1 and 9999 for the input roll length. Enter a number between 0 and 9999 for the operator number. If the existing value of any parameter is still acceptable, then just press ENTER to leave the existing value in place.

Make sure that the optical sensor switch is on and press the run pushbutton to start a product roll. The motor speed will ramp up to a constant speed, hold that speed for a few seconds, then ramp down and stop. The knife output will turn on for 1/5 second, then the conveyor gate output will turn on. Turn off the optical sensor switch. The conveyor gate output will turn back off, and the core hopper gate output will turn on. Turn on the optical sensor switch; the core hopper gateoutput will turn back off. The production totals on the main screen display will be updated. Press the run pushbutton to do this all again.

.

Chapter Three
Multitasking

Many control problems can be separated into a small number of tasks that are mostly independent of each other, yet must be performed at the same time. Traditionally, these control systems have been constructed using several independent process controllers; for example, a system may have two temperature controllers, a motor speed controller, a flow controller, an operator's panel, and a general purpose programmable controller to interface all of the others. Each of these controllers handles its own small part of the system, which allows each part of the system to be easily understood.

When a system such as this is implemented on a single controller, however, it can quickly become extremely complex, because the controller must be programmed to perform all of the tasks concurrently. For example, while it is waiting for the operator to enter a number on the keypad, it must still be controlling the motor speed, process temperature, etc. It becomes especially difficult to modify the operation of the system, such as adding another controller or changing the update rate of one of the existing controllers. What is needed is some way to make the single controller act like multiple, individual control modules.

The Divelbiss 32-bit BASIC compiler supports multitasking, which is a powerful programming tool that allows portions of a program to execute concurrently. TheDivelbiss 32-bit Controller switches rapidly between different portions of the program, called tasks, giving the illusion that each task is being executed on it's own processor. If each of the controllers in the above example were implemented as separate tasks, then it would be similar to building the system out of individual controllers, making the software simpler to write and easier to modify in the future. This chapter explains how multitasking works, how to decide when it should be used, and how to use it correctly.

Multitasking Fundamentals

A Divelbiss 32-bit BASIC program may be divided into portions, called tasks, which can be thought of as independent programs which execute concurrently. Divelbiss 32-bit BASIC contains a piece of software called the context switcher, which manages all of the multitasking operations. Every 10 milliseconds (ie. 100 times per second), the Divelbiss 32-bit Controller suspends normal program execution and jumps to the context switcher. The context switcher updates its timing information, performs some system housekeeping (reads the keypad, handles the network, etc.), and then starts a task executing. The task that is started could be the one that was just interrupted, one that was interrupted at some earlier time, or one that had been waiting for an event to occur. Each 10 millisecond time interval is referred to as a tick in Divelbiss 32-bit BASIC. This is known as preemptive scheduling, since a task is preempted so that another task can execute.

All Divelbiss 32-bit BASIC programs consist of the main program, which is also known as the lead task or task 0. In order to form a multitasking program, one or more tasks must be created using the TASK statement, and task execution must be started with the RUN statement. The context switcher then causes the processor to be split among the executing tasks. Tasks are executed asynchronously with respect to each other; this means that it is impossible to predict exactly when a task will be interrupted so that another task can be run. It is possible for the programmer to synchronize the task execution, if necessary, by using variables as semaphores (this will be discussed later). If task 0 stops executing, all other tasks stop as well.

A simple example will show the basic operation of multitasking.

```
100 RUN 1                               ‘ Start task 1 executing
110 PRINT “|”;                          ‘ Task 0 prints vertical bars
120 GOTO 110                            ‘ Loop forever
200 TASK 1                              ‘ Declare following code to be task 1
210 PRINT “_”;                          ‘ Task 1 prints underscores
220 GOTO 210                            ‘ Loop forever
```

When this program is executed, it will alternate between printing “|” characters and printing “_” characters. Since each character will take about 1 millisecond to transmit at 9600 baud, it should print about 10 characters in each group, since each tick is 10 milliseconds long.

Here is the output produced on one sample run:

```
||||||__________||||||||||__________||||||||||__________||||||||||
__________|||||||||||__________
```

As far as each task is concerned, it gets to run all of the time; the context switcher takes care of switching between the tasks without any special BASIC code in the tasks. The RUN statement in line 100 is necessary to execute task 1.

Note that tasks are much different than subroutines. A subroutine is just a method of sharing a piece of code; it only executes when it is called with a GOSUB and it returns to the line following the GOSUB. A task, on the other hand, shares the processor with other tasks, appearing to execute at the same time as the other tasks. Tasks can be thought of as separate programs executing on the same processor. 32-bit Controller BASIC supports 32 tasks, numbered 0 to 31. Since task 0 is always present, the first user task is number 1. Tasks must be numbered sequentially, starting with 1; an error will be generated if a TASK statement is found out of order.

RUN and EXIT Statements

The example above shows the simplest situation, in which two tasks run continuously, each getting about 50 percent of the processor's time. In many cases, the programmer needs more control of the task execution; the RUN, EXIT, WAIT, CANCEL, and PRIORITY statements provide this control by causing task execution to be started, stopped, and postponed. The following example is a modification of the previous example; it shows the use of the RUN and EXIT statements.

```
100 RUN 1,2                ‘ Start task 1 executing, resched = 2
110 PRINT “|”;             ‘ Task 0 prints vertical bars
120 GOTO 110               ‘ Loop forever
200 TASK 1                 ‘ Declare following code to be task 1
210 PRINT “_”;             ‘ Task 1 prints an underscore
220 EXIT
```

The RUN statement in line 100 starts task 1 executing with a reschedule interval of 2. This means that when task 1 executes an EXIT statement, it will be rescheduled to begin execution again in 2 ticks. Since task 1 just prints a single character, it is finished in about 1 millisecond. When it executes the EXIT statement in line 220, task 1 is stopped and scheduled to start again in 2 ticks. The context switcher lets task 0 run for the rest of the tick (approximately 9 milliseconds). At thenext tick, task 0 gets to continue executing because task 1 is still waiting for the second tick of its 2 tick reschedule period. At the next tick, task 1 gets to execute, starting the process over again. The output from this program should be a series of about 19 vertical bars (in the 19 milliseconds that task 0 executes) followed by an underscore, repeated over and over. Here is a sample of the output:

```
| | | | | | _ | | | | | | | | | | | | | | | | | _ | | | | | | | | | | | | | | | | | | _ | | | | | | | | | | | | | | | | | | _ | | | | | | | | | | |
| | | | | | | _ | | | | | | | | | | | | | | | | | | _ | | | | | | | | | | | | | | | | | | _ | | | | | | | | | | | | | | | | | | _ | | | | | | | | |
| | | | | | | | | _ | | | | |
```

WAIT Statement

When it is necessary for a task to delay processing for a given time period, the WAIT statement is used. WAIT causes the task to suspend processing for the specified number of ticks; for example, WAIT 100 will delay for 1 second, since each tick is 1/100 second. While this task is suspended, other tasks continue to execute. When the context switcher executes at the beginning of each tick, it decrements the “wait counter” for any task that has executed a WAIT statement. When a task's counter reaches 0, that task is considered to be ready to run, and re-enters the normal scheduling system; this does not necessarily mean that this task will be the next one to be executed, unless it has the highest priority. If all of the tasks are waiting at the same time, then the context switcher just kills time until one of them is ready to run; background interrupt processing continues during this period. The WAIT statement is useful in non-multitasking programs, also, since it allows the programmer to easily delay for a fixed time period.

```
100 PRINT “*”;
110 WAIT 100
120 GOTO 100
```

This example uses the WAIT statement to cause a 1 second delay in a program. It will print a “*” character each second until the program is stopped by pressing CTRL-C. The next example shows a more complex use of the WAIT statement.

```
100 RUN 1,10                          ‘ Start task 1, resched = 0.1 sec
110 PRINT “_”;
120 WAIT 20                           ‘ Print “_” every 0.2 second
130 GOTO 110
200 TASK 1
210 PRINT “|”;
220 WAIT 15                           ‘ Delay for 0.15 second
230 PRINT “*”;
240 EXIT                              ‘ Done. It will run again in 0.1 sec
```

This example uses WAIT in a multitasking program. Note that task 1 performs a WAIT between printing the “|” and “*” characters, and that there is another delay caused by the 0.1 second reschedule interval that occurs when the EXIT is executed. The following output is produced when this program is run:

```
_|*_|_*|_*|_*_|*_|_*|_*|_*_|*_|_*|_*|_*_|*_|_*|_*|_*_|*_|_*|_*|_*_|*_|_*|_*|_*_|
_|_*|_*|_*_|*_|_*|_*|_*_|*_|_*|_*|_*_|*_|_*|_*|_*_|*_|_*|_*|_*_|*_|_*|
```

CANCEL Statement

In most situations, a task doesn’t need to execute all of the time; for example, a task may only need to execute when the machine is moving. The CANCEL statement stops the rescheduling of a task. It does not halt the execution of the task at it’s current location; instead, it prevents it from being started again after it executes the next EXIT statement. To stop a task in its current location, see STOP in Chapter Four. A task may CANCEL itself or any other task (except task 0, which always runs). The following example shows the use of the CANCEL statement:

```
100 INTEGER J
110 J=0
120 RUN 1,2                                ‘ Start task 1, resched = 0.02 sec
130 IF J=8 THEN CANCEL 1: GOTO 150         ‘ Wait for task 1 to count to 8
135 PRINT “*”;                             ‘ Print “*”s while waiting
140 GOTO 130
150 WAIT 200                               ‘ Delay to allow task 1 to finish
160 PRINT “Done”
170 STOP
200 TASK 1
210 J=J+1                                  ‘ Increment counter variable
220 PRINT J
230 EXIT
```

Line 120 starts task 1 executing with a reschedule interval of 2 ticks (2/100 second). Line 130 checks to see if J, which is being incremented by task 1, has reached 8 yet; if it has, then task 1 is CANCELed and the program jumps to line 150. If J hasn’t reached 8 yet, then it just prints “*” characters and continues waiting. Since the CANCEL statement only prohibits the task from being rescheduled, the task must execute one more time after the CANCEL statement is executed. Line 150 waits for 2 seconds to allow task 1 to finish executing; this could be a much shorter delay, since task 1 will run again after 2 ticks.

The following output is produced when this program is run:

```
******1
****************2
****************3
****************4
****************5
*****************6
****************7
****************8
9
Done
```

This example demonstrates one of the most important points about canceling a task: the task will execute one more time after the CANCEL statement is executed. This is because CANCEL setsa flag which tells the context switcher to not reschedule the task when it next executes an EXIT statement.

This example also shows an important point about task timing: below a certain resolution, it is unpredictable. In the sixth line of the output, an extra '*' character is printed, because of slight timing variations in the program execution and the serial port. This point will be brought up several times, as it is extremely important.

PRIORITY Statement

Sometimes, one task needs to exclude the other tasks from executing. For example, a task may be performing a time-critical section of code, and needs to guarantee that the context switcher won't switch to other tasks in the middle of this code, causing an unacceptable delay. The PRIORITY statement provides a solution to this problem. When the context switcher is choosing the next task to execute, it will execute the highest priority task that is ready to run. The task that has the highest priority will continue to execute until it hits a WAIT statement or an EXIT statement, or until it resets its priority level low enough to allow another task to run. If the highest priority task doesn't WAIT, EXIT, or reset its priority, then no other task will get a chance to execute. The priority level of each task is set to 0 when the program begins executing, causing the tasks to be executed in a round robin manner until PRIORITY statements are used to set higher priorities. The priority level ranges from 0 through 127, with a higher number indicating higher priority.

```
100 ‘ PRIORITY statement example
110 INTEGER J,K
150 RUN 1: RUN 2,1                        ‘ Start tasks running
190 J=0                                   ‘ Initialize counter variable
200 PRINT “0”;
205 FOR K=1 TO 52: NEXT K                 ‘ Delay for awhile
210 J=J+1: IF J=15 THEN PRIORITY 1        ‘ Bump the priority up
220 IF J=38 THEN PRIORITY 0: J=0: PRINT   ‘ Reset priority to 0
230 GOTO 200
300 TASK 1
310 PRINT “1”;
315 WAIT 1
320 GOTO 310                              ‘ Task 1 is an infinite loop
```

```
400 TASK 2
410 PRINT “2”;
420 EXIT                            ‘ Task 2 gets rescheduled
```

This example uses the PRIORITY statement to cause task 0 to lock out tasks 1 and 2 periodically. After task 0 has printed 15 “0”s, it sets its priority up to 1 in line 210, making it the highest priority task. After it has printed 38 “0”s, it resets its priority to 0, allowing tasks 1 and 2 to execute again. Following is a sample of the output:

```
00120000012000001200000000000000000000000000
01200000120000012000000000000000000000000000
00000120000012000001200000000000000000000000
00001200000120000012000000000000000000000000
00012000001200000120000000000000000000000000
0120000120000012000001200000000000000000000000
00001200000120000012000000000000000000000000
00012000001200000120000000000000000000000000
00120000120000012000000000000000000000000000
0120000120000012000001200000000000000000000000
00012000001200000120000000000000000000000000
```

Determining Task Timing

When the operation of the context switcher is viewed over a sufficiently long period of time, it gives the illusion that multiple tasks are running concurrently. As shown in the previous examples, however, when the operation is viewed over a smaller period of time, it is evident that the processor is switching rapidly between tasks. In some situations, the context switcher’s operation must be considered carefully while programming.

It can be very difficult to predict how the context switcher is going to execute the tasks. In fact, it is impossible to predict exactly what the context switcher is going to do, since there is no way to predict exactly where the program will be when the tick occurs. For example, if a program only has one task currently executing, and that task executes a WAIT 1 statement, then the delay will be between 200 µsec and 10 msec. If the tick occurs immediately after the WAIT statement is executed, then the task will continue executing after a 200 to 1000 µsec delay (the amount of time required for the context switcher to process everything and restart the task). If the WAIT statement is executed immediately after a tick has occurred, then it will be 10 msec until the next tick causes the task to be restarted.

The situation is even more complex if multiple tasks are currently executing, since one or more tasks may get to execute before a task that has executed a WAIT statement. For example, in a program with three tasks that are all at priority level 0 (the default level), suppose that task 1 executes a WAIT 10 statement. After 10 ticks in which tasks 0 and 2 continue to execute, task 1 is ready to run again. However, being ready to run does not mean that the task will run immediately, only that it will be run the next time that the context switcher gets to it. This means that it could execute both task 0 and 2 before it gets to task 1, causing the total time delay for the WAIT 10 statement to be 12 ticks. If task 1 were executing at a higher priority than tasks 0 and 2, however, then it would have executed before them, causing the delay to be 10 ticks.

The result of this is that the BASIC programmer shouldn't rely on the context switcher to provide accurate timing at the millisecond level. The Divelbiss 32-bit Controller is capable of performing operations with a timing accuracy of 1 msec, if necessary, using the timer interrupt. Also, the programmer can modify the task priorities to ensure that time critical sections of code get executed without interruptions from other tasks. If a section of code is extremely time critical (for example, two outputs must be turned on within microseconds of each other), then the INTON and INTOFF statements can be used to ensure that the code gets executed without any interruptions (including hardware interrupts).

Determining When and How to use Multitasking

In many cases, multitasking techniques can make a program much simpler to write and understand. It is not the proper choice for all programs, however. If a multitasking approach is attempted in a situation which is not well suited to it, the resulting program could be quite difficult to understand and debug. Programming experience is highly desirable in order to determine whether a multitasking approach should be used. The following guidelines can be used as a starting point when making this determination.

Fundamentally, multitasking should be used when a system consists of parallel subsystems; each subsystem can be programmed as a separate task. An example would be a control program that implements a temperature control and a speed control; each controller would be a separate task. Multitasking should not be used with systems that are primarily sequential; these should be handled as regular sequential BASIC code. For example, a system that picks up a part, moves it into position, drills a hole, and then drops it onto a conveyor, consists of a sequence of actions that must be performed in order; attempting to write this as a multitasking program would be a major mistake. In practice, most systems consist of both sequential and parallel components; multitasking should be used when the parallel components are large and relatively independent of each other.

Another primary consideration is program speed. The overhead imposed by the context switcher increases as the number of tasks increases. In some high speed applications it is necessary to avoid multitasking in order to get the greatest speed. To get the maximum performance from the Divelbiss 32-bit Controller, the program should be implemented using a combination of assembly language and C; Divelbiss offers a C compiler and an assembler for the Divelbiss 32-bit Controller.

A program's user interface is often difficult to implement, especially if the control system must continue to run while the operator is pressing keys. This is a situation where multitasking can make the program much simpler, since one task can handle the user interface while other tasks continue to control the system. In general, the program will be easier to write and debug if all user I/O to a particular file is contained in a single task. For instance, if two tasks could be accessing the display concurrently, then the program will have to ensure that they don't corrupt each others information; it doesn't do much good to display an alarm, only to have another task clear the display an instant later.

Interaction Between Tasks

Each task will be performing an independent function in a well-organized program. In most cases, though, some of the tasks will need to pass information to other tasks. An example would be a system where a bar code reader task would need to change the setpoint values being used by box folding task (when it detects different products coming in on a conveyor, for instance). A multitasking program will be easier to implement if the interaction between tasks can be minimized. In most programs, the tasks will have to interact somewhat, however. This can lead to some problems which the programmer needs to watch for. Several techniques are given here that allow reliable communication between tasks.

Computer systems usually include items that are limited to a small number of units; these items are referred to as “scarce resources”. For example, a mainframe computer system may have three identical line printers attached, allowing three print jobs to execute concurrently; any new print requests must wait until one of the first three finishes. The printers are a scarce resource. Software can also be a scarce resource; a subroutine can usually only be called from one task at a time, making that subroutine a scarce resource. In a multitasking system, the software must manage the scarce resources so that too many tasks don’t try to use a resource concurrently. This is another form of task interaction that the programmer must be aware of.

The programmer must be careful when tasks interact, because there is no way to predict when the context switcher will switch between tasks. This means that one task may be in the middle of updating a variable when another task gets to execute; if the second task uses the same variable, then unpredictable operation could result. The following example demonstrates this:

```
100 INTEGER J, K
140 J=0
150 RUN 1,10
210 K=J
220 IF K <> J THEN PRINT “Error”: STOP
240 GOTO 200
500 TASK 1
510 J=J+1
520 PRINT “*”;
530 EXIT
```

If the context switcher interrupts task 0 after line 200 or while it is evaluating line 210, then task 1 will modify the value of J. This will cause the test in line 210 to fail when task 0 gets to execute again. A simple solution to this particular problem involves using the PRIORITY statement to control access to the variable J, as follows:

```
100 INTEGER J, K
140 J=0
150 RUN 1,10
200 PRIORITY 1                          ‘ Lock out task 1
210 K=J
220 IF K <> J THEN PRINT “Error”: STOP
230 PRIORITY 0                          ‘ Allow task 1 to execute again
240 GOTO 200
500 TASK 1
510 J=J+1
520 PRINT “*”;
530 EXIT
```

Once task 0 sets its priority to 1 in line 200, task 1 can’t execute again until task 0 sets its priority back to 0 in line 230. A side effect of this is that task 1 executes less often, because task 0 is spending most of the time at priority 1. If this is a problem, then a WAIT 1 statement could be inserted at line 235 to allow task 1 time to execute. A more serious problem with this approach is that it only protects task 0 from task 1; it doesn't protect task 1 from task 0 (ie. task 0 can interrupt task 1 and could modify values being used by task 1). A solution to this problem involves using variables as flags to protect critical regions of code; these are normally called semaphores in computer literature.

```
100 ‘ Example to demonstrate the use of semaphores to protect a region of
102 ‘ code from other tasks. Task 0 and task 1 are both displaying numbers
104 ‘ on the terminal using the LOCATE and PRINT statements. The code must
106 ‘ ensure that one task can't write to the display in between the other
108 ‘ task's LOCATE and PRINT, thereby corrupting the display. The display
110 ‘ is the scarce resource in this example; it can only be used by one task
112 ‘ at a time (without displaying values at the wrong locations.
120 INTEGER COUNT1, COUNT2              ‘ Counter variables
130 INTEGER SEM1                        ‘ Semaphore flag
140 INTEGER LOOP1, LOOP2                ‘ Loop counters
200 SEM1=0                              ‘ Initialize semaphore to 0
210 ERASE
220 COUNT1=0: COUNT2=COUNT1             ‘ Initialize task counters to 0
230 RUN 1
300 ‘ Task 0 main loop.
310 COUNT1=COUNT1+1
315‘ Next two lines perform critical region protection, to lock the
316‘ console display.
320 INTOFF: IF SEM1 < 0 THEN INTON: WAIT 1: GOTO 320
330 SEM1=SEM1-1: INTON
340 LOCATE 10,10: PRINT COUNT1;         ‘ Display current count
350 INTOFF: SEM1=SEM1+1: INTON          ‘ Unlock the console display
360 FOR LOOP1=1 TO 500: NEXT LOOP1      ‘ Simulate execution of other code
370 GOTO 300
400 TASK 1
410 COUNT2=COUNT2+1
415‘ Next two lines perform critical region protection, to lock the
416‘ console display.
420 INTOFF: IF SEM1 < 0 THEN INTON: WAIT 1: GOTO 420
430 SEM1=SEM1-1: INTON
440 LOCATE 20,10: PRINT COUNT2;         ‘ Display current count
450 INTOFF: SEM1=SEM1+1: INTON          ‘ Unlock the console display
460 FOR LOOP2=1 TO 900: NEXT LOOP2      ‘ Simulate execution of other code
470 GOTO 410
```

This program has two tasks that are writing to different locations on the terminal (attached to COM1); it uses a semaphore to control access to the COM1 port, so that one task doesn't print at the other task's screen location. In task 0, lines 320, 330, and 350 implement the semaphore locking algorithm. In line 320, it checks to see if COM1 is available (indicated by SEM1=0); if it isn't, then it waits 1 tick to allow the task that is using COM1 to finish, and loops back to check again. When COM1 is finally available (SEM1=0), then it decrements SEM1 (to -1) to indicate to other tasks that COM1 is in use. The interrupts must be turned off while accessing SEM1, so that two tasks don't access it at the same time, which could allow both tasks to access COM1 concurrently. When the task is finished with COM1, it unlocks it by incrementing SEM1in line 350, allowing another waiting task to continue when it sees SEM1=0. In task 1, lines 420, 430, and 450 perform the same locking function that lines 320, 330, and 350 do. This technique will work with any number of tasks by using these three lines around any critical code that must be protected.

There is one more way that task interaction can be controlled. The Divelbiss 32-bit Controller has a LOCK statement that allows the user to define up to 32 different resources to lock from other tasks. This approach works similar to using semaphores but better. The LOCK statement is used to lock and unlock resources. When locking a resource, the LOCK statement first checks to see the that resource is free. If the resource is free then it is flagged that the current task has use of the resource and returns to run the task. If on the other hand the resource is already in use the current task is put in a queue to be the next task to run when the resource is free, and then it is block from running. Program execution returns to the task that has the resource locked. When a task is unlocked, the queue list is looked at. If the list is empty then the resource is marked free and the returns to the current task. When the queued list is not empty then the resource is marked being used by first task in the list and the task is run. This approach allows for faster task switching and frees the task scheduler the chore of switching to tasks that can't run because of a used resource.

```
100 ‘ Example to demonstrate the use of the LOCK statement to protect a region
102 ‘ of code from other tasks. Task 0 and task 1 are both displaying numbers
104 ‘ on the terminal using the LOCATE and PRINT statements. The code must
106 ‘ ensure that one task can’t write to the display in between the other
108 ‘ task’s LOCATE and PRINT, thereby corrupting the display. The display
110 ‘ is the scarce resource in this example; it can only be used by one task
112 ‘ at a time (without displaying values at the wrong locations.
120 INTEGER COUNT1, COUNT2              ‘ Counter variables
130 INTEGER LOOP1, LOOP2                ‘ Loop counters
200 ERASE
210 COUNT1=0: COUNT2=COUNT1             ‘ Initialize task counters to 0
220 RUN 1
300 ‘ Task 0 main loop.
310 COUNT1=COUNT1+1
315‘ Next lines perform critical region protection, to lock the
316‘ console display.
320 LOCK 1,0                            ‘ lock the console display (resource zero)
330 LOCATE 10,10: PRINT COUNT1;         ‘ Display current count
340 LOCK 0,0                            ‘ Unlock the console display (resource zero)
350 FOR LOOP1=1 TO 500: NEXT LOOP1      ‘ Simulate execution of other code
360 GOTO 300
400 TASK 1
410 COUNT2=COUNT2+1
415‘ Next lines perform critical region protection, to lock the
416‘ console display.
420 LOCK 1,0                            ‘ lock the console display (resource zero)
430 LOCATE 20,10: PRINT COUNT2;         ‘ Display current count
440 LOCK 0,0                            ‘ Unlock the console display (resource zero)
450 FOR LOOP2=1 TO 900: NEXT LOOP2      ‘ Simulate execution of other code
460 GOTO 410
```

This program, just like the previous one, has two tasks that are writing to different locations on the terminal (attached to COM1); it uses the LOCK statement to control access to the COM1 port, so that one task doesn't print at the other task's screen location. In task 0, lines 320 and 340 implement the LOCK statement. In line 320, the LOCK statement checks to see if resource zero, COM1, is available; if it isn't, then it runs the task that has the resource locked, task 1. When the task is finished with COM1, it unlocks it in line 340, allowing another waiting task to run. In task 1, lines 420 and 440 perform the same locking function that lines 320 and 340 do. This technique will work with any number of tasks by using these two lines around any critical code that must be protected.

Organization of Tasks and Subroutines

This section discusses the manner in which a multitasking program should be organized. It is extremely important that the tasks and subroutines are put in the proper locations; if they aren't, then the BASIC program will operate unpredictably. Incorrect organization of the program can cause some very subtle problems that can be quite difficult to find. Fortunately, the problem can be avoided in most cases by following a few simple rules.

Before getting into the details, a few comments must be made about tasks. Tasks must be declared in numerical order starting with number 1; also note that the code before the 'task 1' statement actually belongs to task 0. All of the code between two TASK statements is part of the same task, as far as the compiler is concerned; everything up to the 'task 1' statement is in task 0 and everything from the 'task 1' statement up to the 'task 2' statement is in task 1, for example.

In general, a Divelbiss 32-bit BASIC program must have its tasks declared at the end of the program, after the variable declarations and mainline code; this includes any tasks that are handling hardware interrupts (ie. referred to in a VECTOR statement). The following example illustrates the proper form for a program; the program would contain code that has been left out for clarity, but all TASK and GOSUB statements are included here (there are no GOSUBs in this program). Note the test after line 300 ('if test <> 0 then ...'); this will never print because 'test' is always 0.

The following example demonstrates the basic form that a Divelbiss 32-bit BASIC program should follow:

```
100 ' Variable declarations
integer test
integer x1, x2
200 ' Start of program code. This is task 0.
    run 1,10                        ' Run task 1 10 times/second
    run 2,5                         ' Run task 2 20 times/second
    test = 0                        ' Initialize test flag

' Mainline code loop
300 if test <> 0 then print "Failed mainline"
goto 300
10000   task 1
' code for task 1
x1 = x1 + 1
exit
```

```
11000   task 2                          ‘ Last task in program
‘ code for task 2
x2 = x2 + 1
exit
```

The existence of subroutines in a program complicates matters. The simplest situation occurs when each subroutine is only called from within a single task. In this case, each subroutine should be within the boundaries of the task that calls it; for example, a subroutine that is called by task 4 must be between the ‘task 4’ and ‘task 5’ statements. Let’s expand the preceding program to illustrate this point; again, even though a lot of code is missing from this example, all of the TASK and GOSUB statements are shown here. The important point in this example is that the subroutine at 1000 (which is in task 0) is only called from within task 0, and the subroutine at 10200 (which is in task 1) is only called from task 1. The ‘test’ variable is checked in each task; since it is always 0, no messages will be printed.

```
100‘ Variable declarations
integer test
integer x0, x1, x2
200‘ Start of program code. This is task 0.
    run 1,10                            ‘ Run task 1 10 times/second
    run 2,5                             ‘ Run task 2 20 times/second
test = 0
‘ Mainline code loop
300if test <> 0 then print “Failed mainline”
gosub 1000
goto 300

1000    ‘ Subroutine called only by task 0 (mainline)
‘ Code for subroutine goes here
x0 = x0 + 1
return

10000   task 1
‘ Code for task 1
gosub 10200
x1 = x1 + 1
if test <> 0 then print “Failed task 1”
exit

10200   ‘ Subroutine called only by task 1
‘ Code for subroutine goes here
‘ This is still part of task 1
return

11000   task 2                          ‘ Last task in program
‘ Code for task 2
x2 = x2 + 1
if test <> 0 then print “Failed task 2”
exit
```

If the two rules outlined above are followed then your program will work as expected. Once again, the rules are:

Important

1. All tasks must be at the end of the program.

2. Never call a subroutine that is outside of the task that contains the GOSUB to that subroutine.

Unfortunately, the second rule is quite limiting, because it means that multiple tasks can't call a common subroutine. In order to arrive at a solution to this problem, you must understand how the compiler works and why the above rules are necessary.

The situation comes about because of the method that the Divelbiss 32-bit Controller BASIC compiler uses to allocate memory for a program. As each BASIC statement is executed, there are temporary values that are calculated and stored in a set of temporary variables; these values are only needed during the execution of a single statement. Each task has its own set of temporary variables; every statement within a task uses the same set of temporary variables, which is the cause of the difficulty. Let's assume that a statement is being executed when the timer tick occurs; one or more tasks will execute before that statement gets to complete. If any one of those tasks is using the same temporary variables as the statement, then when it finally gets to complete execution, its temporary values will have been altered, resulting in incorrect operation of the program. How could one of the intervening tasks use the same temporary variables as the original statement? The simplest situation would be if one of the intervening tasks called a subroutine that was in the same task as the statement that was interrupted. Here are two examples that illustrate a few ways that this could happen:

```
10   ‘ This example contains a task that calls a subroutine that is
     ‘ located within another task (task 1 calls the subroutine at 1000,
     ‘ which is in task 0). If task 0 is executing the statement in
     ‘ line 300 when the timer tick occurs, allowing task 1 to run, then
     ‘ when task 0 finishes executing line 300, an erroneous conclusion
     ‘ will be reached. When task 1 calls 1000, it will have altered the
     ‘ temporary variables being used in line 300, because 300 and 1000
     ‘ both use the same temporary variables.

100  ‘ Variable declarations
integer test
integer x0, x1, x2
200  ‘ Start of program code. This is task 0.
     run 1,10                        ‘ Run task 1 10 times/second
     run 2,5                         ‘ Run task 2 20 times/second
test = 0
‘ Mainline code loop. Line 300 will sometimes operate
‘ incorrectly, printing the message even though “test” is 0.
300  if test <> 0 then print “Failed mainline”
gosub 1000
goto 300

1000 ‘ Subroutine called from multiple tasks
‘ Code for subroutine goes here
x0 = x0 + 1
return
```

```
10000   task 1
‘ Code for task 1
gosub 10200
gosub 1000
x1 = x1 + 1
if test <> 0 then print “Failed task 1”
exit

10200 ‘ Subroutine called only by task 1
‘ Code for subroutine goes here
‘ This is still part of task 1
return

11000 task 2                          ‘ Last task in program
‘ Code for task 2
x2 = x2 + 1
if test <> 0 then print “Failed task 2”
exit
```

Here is the second example:

```
10   ‘ This example breaks rule number 1 by putting the mainline code
     ‘ after task 2, which means that the mainline code will be using
     ‘ the same temporary variables as task 2. When a timer tick causes
     ‘ task 2 to execute, it will modify its temporary variables and
     ‘ possibly cause an expression in the mainline to be evaluated
     ‘ incorrectly. One possible solution to this would be to put a
     ‘ ‘task 3’ statement at line 20000; this would cause the compiler
     ‘ to use a new set of temporary variables for the mainline code.

100  ‘ Variable declarations
integer test
integer x0, x1, x2
200  ‘ Start of program code. This is task 0.
     run 1,10                         ‘ Run task 1 10 times/second
     run 2,5                          ‘ Run task 2 20 times/second
test = 0
     goto 20000                       ‘ Jump to mainline loop

1000    ‘ Subroutine called from mainline
‘ Code for subroutine goes here
x0 = x0 + 1
return

10000   task 1
‘ Code for task 1
gosub 10200
x1 = x1 + 1
if test <> 0 then print “Failed task 1”
exit
```

```
10200   ‘ Subroutine called only by task 1
‘ Code for subroutine goes here
‘ This is still part of task 1
return

11000   task 2                          ‘ Last task in program
‘ Code for task 2
x2 = x2 + 1
if test <> 0 then print “Failed task 2”
exit

‘ Mainline code loop. This is still part of task 2. Line
‘ 20000 will sometimes operate incorrectly, printing the
‘ message even though ‘test’ is 0.
20000   if test <> 0 then print “Failed mainline”
gosub 1000
goto 20000
```

So, is there a solution to the difficulties produced by rule number 2 above? Yes, it is possible to call a subroutine that is located in a different task, which means that it is possible to have a common subroutine that is called from multiple tasks. In order for it to work correctly, you must be able to guarantee that the subroutine can’t interrupt any other code that is in the task that the subroutine is in. The simplest way to do this is to put the subroutine in a separate task and disable interrupts at the beginning and end of the subroutine. Of course, since the interrupts are disabled, the subroutine must be short and fast so that no hardware interrupts are lost. Multiple subroutines can be put into the same task, but they must all have interrupts disabled while executing.

Remember that some BASIC statements and functions can’t be used while interrupts are disabled, because they turn the interrupts back on prematurely; these include PRINT, INPUT, GET, string operations, and user defined functions. If a subroutine must use one of these and also be called from multiple tasks, then the subroutine should be placed in a task by itself. The following example illustrates these techniques.

```
100     ‘ Variable declarations
integer test
integer x0, x1, x2

200     ‘ Start of program code. This is task 0.
        run 1,10                        ‘ Run task 1 10 times/second
        run 2,5                         ‘ Run task 2 20 times/second
test = 0
‘ Mainline code loop.
300     if test <> 0 then print “Failed mainline”
        gosub 20000                     ‘ Subroutine called from multiple tasks
        gosub 21000                     ‘ Also called from multiple tasks
goto 300
10000   task 1
‘ Code for task 1
    gosub 10200                         ‘ Only called from task 1
gosub 20000                          ‘ Called from multiple tasks
x1 = x1 + 1
if band (x1, $7F) = 0 then gosub 22000
if test <> 0 then print “Failed task 1”
exit

10200   ‘ Subroutine called only by task 1
‘ Code for subroutine goes here
‘ This is still part of task 1
return

11000   task 2                       ‘ Last task in program
‘ Code for task 2
11010   x2 = x2 + 1
if band (x2, $7F) = 0 then gosub 22000
gosub 21000
if test <> 0 then print “Failed task 2”
goto 11010

‘ The next task statement is used to separate the subroutines
‘ from the rest of the code; everything after the “task 3”
‘ statement will use the same set of temporary variables.
‘ Since the subroutines disable interrupts while executing,
‘ they can be called from multiple tasks without any problems.
task 3

20000   ‘ Subroutine called from multiple tasks
intoff
‘ Code for subroutine goes here
x0 = x0 + 1
inton
return

21000   ‘ Subroutine called from multiple tasks
intoff
x0 = x0 - 1
inton
return

task 4
22000   ‘ Subroutine called from multiple tasks. This routine uses
‘ the PRINT statement, so it can’t turn interrupts off. This
‘ means that it must be put into a task by itself, so that it
‘ doesn’t share temporary variables with any other routine.
print “x0 = “; x0
return
```

Important

So, we can add a third rule to the list:

1. All tasks must be at the end of the program.

2. Never call a subroutine that is outside of the task that contains the GOSUB to that subroutine.

3. If the program contains subroutines that must be called from multiple tasks, then put all of these subroutines into a separate task, and disable interrupts during the subroutine execution. The subroutines must be short, and they can't use any of the PRINT, GET, or INPUT statements.

If subroutines can't disable interrupts while executing (because they are too long or execute statements that enable interrupts) then the subroutines should be treated as a scarce resource and protected with a semaphore. This technique is described in section 5.4 above. The subroutines are placed in a separate task, and then all accesses to any subroutine in that task must be protected with a semaphore flag. This will prevent multiple tasks from executing subroutines simultaneously, thereby preventing the subroutine temporary variables from being corrupted.

Chapter Four
Language Reference

can be used with:

Boss 32

UMC

= Statement

Summary:
The = statement assigns a value to a variable.

Syntax:
variable = expr

Arguments:
expr a numeric or string expression. The expression type must match the type of *variable*.

Description:
The = statement is used to store the result of an expression into a variable. If *variable* is an integer or real, then the result of the expression is converted to the proper type before being stored. Care should be taken when assigning the result of an expression containing real values to an integer variable. In this case, Divelbiss 32-bit BASIC may lose precision during the expression evaluation; see section 3.6 for a discussion of this problem.

Note that the character '=' is also used to represent the equality operator. This means that the statement X=A=B is legal; it sets X to a nonzero value if A is equal to B.

In multitasking programs, the context switcher will not switch tasks during the assignment operation. For example, if a program has a real variable X and a task executes the statement X=3.1416, the task will always update all 4 bytes of X as a group, without allowing another task to break in. This is also true of integers and strings. In an expression evaluation, however, the context switcher may allow another task to corrupt a variable. If one task is executing the statement X=X*1.5 and another task interrupts and executes X=4.3, then the value of X depends on the exact spot that the context switcher interrupted. The programmer should avoid this type of situation by using different variables or using the PRIORITY statement to control the operation of the context switcher.

Example:

```
100 INTEGER J
110 REAL X
120 X=2.7*4.0
130 J=(X*10000)-21234
140 PRINT J
```

ABS Function

can be used with:

Boss 32

UMC

Summary:

The ABS function calculates the absolute value of the expression.

Syntax:

x = ABS (*expr*)

Arguments:

expr an integer expression

Description:

The ABS function returns the absolute value of its argument, which must be a numeric expression. The result is returned as a INTEGER value.

Example:

```
100 INTEGER X,Y
110 PRINT ABS(-23)
120 X=-4
130 Y=ABS(X)
140 PRINT “Absolute value of “;X;” is “;Y
```

This produces the following output when run:

```
23
Absolute value of -4 is 4
```

ACOS Function

can be used with:

Boss 32

UMC

Summary:
The ACOS function calculates the arccosine function.

Syntax:
x = ACOS (*expr*)

Arguments:

expr	a numeric expression. (-1 to 1)

Description:
The ACOS function returns the arccosine of its argument, which must be a numeric expression. The result is returned as a REAL value, in degrees. If *expr* >1 or if *expr* <-1 then x=0.

Example:

```
100 REAL X,Y
110 PRINT ACOS(.23)
120 X=.4
130 Y=ACOS(X)
140 PRINT "Arccosine value of ";X;" is ";Y
```

This produces the following output when run:

```
76.70812
Arccosine value of .40000 is 66.42183
.
```

can be used with:

Boss 32

UMC

ADC Function

Summary:
The ADC function is used to read an analog input value from one of the A/D channels.

Syntax:
advl = ADC(*chan*)

Arguments:
chan a numeric expression. The A/D channel number to read starting with 1.

Description:
The ADC function performs an analog conversion on the specified A/D channel, and returns the result as an INTEGER value between 0 and 32767. The value returned by the A/D hardware is scaled to cover the full range of 0 to 32767; this allows A/D converters of different resolutions (ie. 12 bit, 14 bit, etc.) to be substituted without altering the user's program. The ADC function takes approximately 50 microseconds to complete. The A/D channels are assigned based on the hardware available on the Boss 32; channel 1 could be on the onboard A/D, or in any of the expansion ports (if there is no onboard A/D converter). The order of precedence for assigning A/D channels is: onboard A/D followed by J3 followed by J4 followed by J5.

The ADC function can also be used to read the last value written to the D/A converter. The *chan* number to read the last written D/A value is a number between 257 and 256+total number of DAC channels ($101 and $1XX).

Example:

```
100 ' Display the first 5 A/D channels
110 INTEGER NUM
120 FOR NUM = 1 TO 5
130  PRINT "Channel ";NUM;" = "; ADC(NUM)
140 NEXT NUM
```

This produces the following output when run:

```
Channel 1 = 0
Channel 2 = 0
Channel 3 = 13434
Channel 4 = 0
Channel 5 = 0
```

Channels 1, 2, 4, and 5 indicate that 0 volts are present. Channel 3 indicates that 2.05 volts is present (13434/32767*5.0=2.05).

can be used with:

Boss 32

UMC

ADR Function

Summary:
The ADR function returns the address of a variable in the BASIC program.

Syntax:
addr = ADR(*name*)

Arguments:
name a character constant. The name of a variable declared in the BASIC program.

Description:
The ADR function returns the address of a variable; it is returned as an integer value. This can be used to find the address of a string or array in which an assembly language program will be POKEd. It can also be used to access a variable without using the normal BASIC operations.

Example:

```
100 ‘ Swap the bytes in a variable
110 INTEGER J,K
120 K = $12345678
130 POKE ADR(J), PEEK(ADR(K)+3)
140 POKE ADR(J)+1, PEEK(ADR(K)+2)
150 POKE ADR(J)+2, PEEK(ADR(K)+1)
160 POKE ADR(J)+3, PEEK(ADR(K))
170 FPRINT “S2H8S4H8”, “J=”, J, “ K=”, K
```

This produces the following output when run:

```
J=78563412   K=12345678
```

The key to understanding this example lies in remembering that an INTEGER occupies 4 bytes in memory. In line 130, we put the forth byte of K into the first byte of J. In line 140 we put the third byte of K into the second byte of J. In line 150 we put the second byte of K into the third byte of J. In line 160 we put the first byte of K into the forth byte of J. The numbers are represented in hexadecimal to make the swap operation a little more obvious.

can be used with:

Boss 32

UMC

ASC Function

Summary:
The ASC function calculates the ASCII numeric equivalent of the first character of a string.

Syntax:
x = ASC (st$)

Arguments:

st$	a string expression.

Description:
The ASC function returns the ASCII equivalent of the first character in the specified string. The result is returned as an INTEGER value. ASC is the converse of the CHR$ function.

Example:

```
100 STRING A$
110 INTEGER N
120 PRINT "ASCII value of X: ";ASC("X")
130 A$="012ABCabc ."
140 FOR N = 1 TO LEN(A$)
150  PRINT N; " "; ASC(MID$(A$,N,1))
160 NEXT N
```

This produces the following output when run:

```
ASCII value of X: 88
1       48
2       49
3       50
4       65
5       66
6       67
7       97
8       98
9       99
10      32
11      46
```

ASIN Function

can be used with:

Boss 32

UMC

Summary:
The ASIN function calculates the arcsine function.

Syntax:
x = ASIN (*expr*)

Arguments:
expr a numeric expression. (-1 to 1)

Description:
The ASIN function returns the arcsine of its argument, which must be a numeric expression. The result is returned as a REAL value, in degrees. If *expr* >1 or if *expr* <-1 then x=90.

Example:

```
100 REAL X,Y
110 PRINT ASIN(.23)
120 X=.4
130 Y=ASIN(X)
140 PRINT “Arcsine value of “;X;” is “;Y
```

This produces the following output when run:

```
13.29188

Arcsine value of .40000 is 23.57816
```

ATAN Function

can be used with:

Boss 32

UMC

Summary:
The ATAN function calculates the arctangent function.

Syntax:
x = ATAN (*expr*)

Arguments:
expr a numeric expression.

Description:
The ATAN function returns the arctangent of its argument, which must be a numeric expression. The result is returned as a REAL value, in degrees.

Example:

```
100 REAL X,Y
110 PRINT ATAN(1.23)
120 X=4.0
130 Y=ATAN(X)
140 PRINT “Arctangent value of “;X;” is “;Y
```

This produces the following output when run:

```
50.88856

Arctangent value of 4.00000 is 75.96370
```

can be used with:

Boss 32

UMC

BAND Function

Summary:
The BAND function performs a bitwise logical AND function.

Syntax:
x = BAND (expr1,expr2)

Arguments:

expr1	a numeric expression.
expr2	a numeric expression.

Description:
The BAND function logically ANDs *expr1* and *expr2* as 32 bit integers; each of the bits is individually ANDed together. This is useful for clearing bits in an integer variable, such as when accessing hardware registers. This differs from the AND operator, which compares two numbers as boolean values (ie. true or false). For example, BAND(2,6) returns 2, while 2 AND 6 evaluates to an undefined non-zero value.

Example:

```
100 INTEGER J,K
110 PRINT "  ";
120 FOR K = 0 TO 7
130  FPRINT "I3Z",K
140 NEXT K
150 PRINT
160 FOR J = 0 TO 7
170  FPRINT "I3Z",J
180  FOR K = 0 TO 7
190   FPRINT "I3Z", BAND(J,K)
200  NEXT K
210  PRINT
220 NEXT J
```

This produces the following output when run:

```
        0    1    2    3    4    5    6    7
0       0    0    0    0    0    0    0    0
1       0    1    0    1    0    1    0    1
2       0    0    2    2    0    0    2    2
3       0    1    2    3    0    1    2    3
4       0    0    0    0    4    4    4    4
5       0    1    0    1    4    5    4    5
6       0    0    2    2    4    4    6    6
7       0    1    2    3    4    5    6    7
>
```

BOR Function

can be used with:

Boss 32

UMC

Summary:
The BOR function performs a bitwise logical OR function.

Syntax:
x = BOR (expr1,expr2)

Arguments:

expr1	a numeric expression.
expr2	a numeric expression.

Description:
The BOR function logically ORs *expr1* and *expr2* as 32 bit integers; each of the bits is individually ORed together. This is useful for setting bits in an integer variable, such as when accessing hardware registers. This differs from the OR operator, which compares two numbers as boolean values (ie. true or false). For example, BOR(2,5) returns 7, while 2 OR 5 evaluates to an undefined non-zero value.

Example:

```
100 INTEGER J,K
110 PRINT "  ";
120 FOR K = 0 TO 7
130  FPRINT "I3Z",K
140 NEXT K
150 PRINT
160 FOR J = 0 TO 7
170  FPRINT "I3Z",J
180  FOR K = 0 TO 7
190   FPRINT "I3Z", BOR(J,K)
200  NEXT K
210  PRINT
220 NEXT J
```

This produces the following output when run:

```
        0       1       2       3       4       5       6       7
0       0       1       2       3       4       5       6       7
1       1       1       3       3       5       5       7       7
2       2       3       2       3       6       7       6       7
3       3       3       3       3       7       7       7       7
4       4       5       6       7       4       5       6       7
5       5       5       7       7       5       5       7       7
6       6       7       6       7       6       7       6       7
7       7       7       7       7       7       7       7       7
>
```

BXOR Function

can be used with:

Boss 32

UMC

Summary:
The BXOR function performs a bitwise logical exclusive OR function.

Syntax:
x = BXOR (expr1,expr2)

Arguments:

expr1	a numeric expression.
expr2	a numeric expression.

Description:
The BXOR function logically XORs *expr1* and *expr2* as 32 bit integers; each of the bits is individually XORed together. This is useful for inverting bits in an integer variable, such as when accessing hardware registers.

Example:

```
100 INTEGER J,K
110 PRINT "  ";
120 FOR K = 0 TO 7
130  FPRINT "I3Z",K
140 NEXT K
150 PRINT
160 FOR J = 0 TO 7
170  FPRINT "I3Z",J
180  FOR K = 0 TO 7
190   FPRINT "I3Z", BXOR(J,K)
200  NEXT K
210  PRINT
220 NEXT J
```

This produces the following output when run:

```
     0    1    2    3    4    5    6    7
0    0    1    2    3    4    5    6    7
1    1    0    3    2    5    4    7    6
2    2    3    0    1    6    7    4    5
3    3    2    1    0    7    6    5    4
4    4    5    6    7    0    1    2    3
5    5    4    7    6    1    0    3    2
6    6    7    4    5    2    3    0    1
7    7    6    5    4    3    2    1    0
>
```

BYE Direct Command

can be used with:

Boss 32

UMC

Summary:
The BYE direct command resets the Divelbiss 32-bit Controller.

Syntax:
BYE

Arguments:
BYE needs no arguments.

Description:
BYE performs a software reset of the Divelbiss 32-bit Controller. If the SETOPTION RUN is set to ON, then the last program on the EPROM will be loaded and run.

.

can be used with:

Boss 32

UMC

CALL Statement

Summary:
The CALL statement is used to link Divelbiss 32-bit BASIC to an assembly language subroutine.

Syntax:
CALL addr, [arg1], [arg2]...

Arguments:
addr a numeric expression. The address of the subroutine to call.
argx a numeric or string expression. An argument to be passed to the assembly language routine.

Description:
The CALL statement begins execution of an assembly language subroutine starting at *addr*. Execution of the assembly language subroutine continues until a RET instruction is encountered, at which point execution will continue at the point following the CALL statement in the Divelbiss 32-bit BASIC program. No registers need be preserved while in the assembly language subroutine.

If optional arguments are given after the address of the routine being called, then the address of these arguments are passed to the routine, allowing any BASIC variable or value to be passed to the assembly routine. Since the addresses of the values are passed, the assembly routine can alter the values before returning to the calling program.

If optional arguments are supplied, then these arguments will be stored in a table following the assembly language CALL instruction, so the return address on the stack will point at the table. The table will consist of zero or more entries, with each entry consisting of a mode byte followed by the argument address. The table will end with a 0 byte in place of a mode byte. The mode byte is assigned as follows: 1 for integer, 2 for real, and 3 for string. The arguments can be arrays, constants, or variables; array elements and constants are passed using temporary variables, and so should not be modified. Variables can be modified.

For example, the code

```
100 INTEGER J
110 REAL X(10)
120 STRING A$(40)
‘ Other code goes here
500 CALL $C000, X(3), J, A$
```

would generate assembly language that looks like this:

```
CALL   $C000
.BYTE  2                              ; X(3) mode is real
.WORD  address of temp holding value of X(3)
.BYTE  1                              ; J mode is integer
.WORD  address of J
```

```
.BYTE   3                               ; A$ mode is string
.WORD   address of A$
.BYTE   0                               ; End of table flag
                                        ; Next line of BASIC code.
```

If it is necessary to update an array element from a subroutine, then it must be done using an intermediate variable. For example:

```
600 K=KARRAY(4)
610 CALL $B800,K
620 KARRAY(4)=K
```

In order to write robust code, the assembly language programmer must not assume that the BASIC programmer used the correct number or type of arguments. The assembly code should always look through the table for the 0 (end of table flag), and return to the next address past it.
Even if the assembly routine requires no arguments (ie. 800 CALL $C000), the “end of table flag” is still stored after the CALL instruction, so the assembly routine must look at the table to determine the correct return address.

Example:

```
100 INTEGER NUM,BYTE
110 DATA $E1          ‘ pop  hl       Get table address
111 DATA $23          ‘ inc  hl       Skip mode byte
112 DATA $5E          ‘ ld  e,(hl)    Get low byte of arg address
113 DATA $23          ‘ inc  hl
114 DATA $56          ‘ ld  d,(hl)    Get high byte of arg address
115 DATA $23          ‘ inc  hl       Point to 0 byte
116 DATA $23          ‘ inc  hl       Point to next code byte
117 DATA $E5          ‘ push hl       Store new return address
118 DATA $1A          ‘ ld  a,(de)    Get low byte of argument
119 DATA $C6,$05      ‘ add  a,5      Add 5
120 DATA $12          ‘ ld  (de),a    Store low byte back
121 DATA $13          ‘ inc  de
122 DATA $1a          ‘ ld  a,(de)    Get high byte of argument
123 DATA $CE,$00      ‘ adc  a,0      Propagate carry into high byte
124 DATA $12          ‘ ld  (de),a    Store high byte back
125 DATA $C9          ‘ ret           Return to BASIC
130 FOR NUM = 0 TO 17
140  READ BYTE
150  POKE $B000+NUM,BYTE              ‘ Store assembly program at $B000
160 NEXT NUM
160 NUM=90
170 CALL $B000,NUM                    ‘ Call assembly program
180 PRINT NUM
```

This example calls a simple subroutine that gets the address of the argument, adds 5 to the argument, and returns. In the interest of space and clarity, this was programmed very badly; no argument checking is done in the assembly language. If $B000 is called with the wrong number of arguments, the system will almost certainly crash.

can be used with:

Boss 32

UMC

CANCEL Statement

Summary:
The CANCEL statement stops a task from being restarted when the schedule interval given in the RUN statement elapses.

Syntax:
CANCEL *task*

Arguments:
task   an integer expression. The number of the task to cancel.

Description:
CANCEL stops the specified *task* from being started again when the schedule interval given in the RUN statement elapses. CANCEL does not abort the task - only EXIT or STOP can stop the execution of a task. When a RUN statement is entered, a schedule interval is specified; each time the task EXITs, it is scheduled to restart after that many ticks have elapsed. CANCEL causes the scheduling of the task to cease. When CANCEL is executed, the task continues executing normally until it EXITs. The task will not run again until it is specifically commanded to via another RUN statement. CANCEL is useful when a task need only be run a fixed number of times; the task can CANCEL itself, as in the example below.

Example:

```
100 INTEGER J, K
105 K = 0                             ‘ Initialize counter for task 1.
110 RUN 1,10                          ‘ Set reschedule interval to 10 ticks.
120 FOR J = 1 TO 1000
130  PRINT “*”;
140  WAIT 1
150 NEXT J
160 STOP
200 ‘ Task 1 prints 5 ‘.’ characters.
210 TASK 1
220 PRINT “.”;
230 K = K + 1                         ‘ Increment number of times we’ve run.
240 IF K >= 5 THEN CANCEL 1           ‘ If 5 times then done, don’t reschedule.
250 EXIT
```

can be used with:

Boss 32

UMC

CHAIN Statement

Summary:

The CHAIN statement loads and runs another program from the user's EPROM.

Syntax:

CHAIN filenum
or
CHAIN "*filename*"

Arguments:

filenum	an integer expression. The number of the compiled code file to load and execute.
filename	a text string up to 10 characters long. The file name of the compiled code file to load and execute.

Description:

CHAIN causes another program to start executing. This allows a program that is too large to fit into memory to be broken into multiple smaller programs. Because Divelbiss 32-bit BASIC doesn't overwrite variables when it loads the new program, values can be passed between programs. The variables to be passed between programs must be the first variables declared, and they must be declared in the same order.

When CHAIN is used with a file name, the last file on the EPROM with the specified name is executed; this allows a file to be modified and stored on the same EPROM, since the newest file is always executed. When CHAIN is used with a file number, then that specific file is executed, even if there is a newer one with the same name.

The CHAIN statement disables interrupt processing. When the new program starts executing, it will reset some of the onboard hardware. The serial ports get reset, timer 1 is turned off, and the onboard display (LCD/VFD) is cleared. The high speed counter's value and mode is left unchanged.

Examples:

```
100 PRINT "Hi there"
110 CHAIN 4                          ' Jump to program 4 on EPROM
```

This example shows how to use CHAIN to run a specific program number.

```
100 ' First program. Save on EPROM as "FIRST"
110 PRINT "This is the first program"
120 WAIT 100
130 CHAIN "SECOND"

100 ' Second program. Save on EPROM as "SECOND"
110 PRINT "This is the second program"
120 PRINT "*************************"
130 WAIT 100
140 CHAIN "FIRST"
```

Type in the first program, compile it, and type SAVE CODE FIRST to save it on the EPROM. Then type in the second program, compile it, type SAVE CODE SECOND, and run the program. It will execute, then CHAIN to the first program, which will execute and CHAIN back to the second program.

CHR$ Function

can be
used with:

Boss 32

UMC

Summary:
The CHR$ function returns a string that corresponds to an integer value.

Syntax:
st$ = CHR$ (expr)

Arguments:
expr an integer expression between 0 and 255.

Description:
CHR$ returns a 1 character long string that corresponds to the ASCII value of the specified expression. CHR$ is the converse of ASC.

Example:

```
100 ‘ Print the character equivalents of the values 32 to 126.
110 INTEGER J
120 FOR J=32 TO 126
130  FPRINT “i3x1s1x3z”,J,CHR$(J)
140 NEXT J
```

This produces the following output when run:

```
 32      33 !    34 “    35 #    36 $    37 %    38 &    39 ‘    40 (    41 )

 42 *    43 +    44 ,    45 -    46 .    47 /    48 0    49 1    50 2    51 3

 52 4    53 5    54 6    55 7    56 8    57 9    58 :    59 ;    60 <    61 =

 62 >    63 ?    64 @    65 A    66 B    67 C    68 D    69 E    70 F    71 G

72 H     73 I    74 J    75 K    76 L    77 M    78 N    79 O    80 P    81 Q

82 R     83 S    84 T    85 U    86 V    87 W    88 X    89 Y    90 Z    91 [

92 \     93 ]    94 ^    95 _    96 `    97 a    98 b    99 c    100 d   101 e

102 f    103 g   104 h   105 i   106 j   107 k   108 l   109 m   110 n   111 o

112 p    113 q   114 r   115 s   116 t   117 u   118 v   119 w   120 x   121 y

122 z    123 {   124 |   125 }   126 ~   >
```

CLEARFLASH Direct Command

can be used with:

Boss 32

UMC

Summary:
The CLEARFLASH direct command erases the FLASH EPROM that the user's code is stored on.

Syntax:
CLEARFLASH

Arguments:
CLEARFLASH needs no arguments.

Description:
The Divelbiss 32-bit Controller stores the user's programs on a FLASH EPROM which can be erased electrically, without being removed from the Divelbiss 32-bit Controller. The CLEARFLASH command performs this erase operation, removing all data from the EPROM. This command takes about 10 seconds to complete. The FLASH EPROM can be erased at least 100000 times.

Example:
```
CLEARFLASH
```

CLEARMEMORY Direct Command

can be used with:

Boss 32

UMC

Summary:
The CLEARMEMORY direct command fills the entire RAM memory area with zeroes.

Syntax:
CLEARMEMORY

Arguments:
CLEARMEMORY needs no arguments.

Description:
The Boss 32 and UMC store source code, compiled code, and variables in memory. CLEARMEMORY will clear the memory on the unit by writing zeros to the entire RAM memory area.

EXAMPLE:
```
CLEARMEMORY
```

.

CLR Direct Command

can be used with:

Boss 32

UMC

Summary:
The CLR direct command erases the console terminal display.

Syntax:
CLR

Arguments:
CLR needs no arguments.

Description:
CLR sends the control sequence to erase the display on a supported terminal. If an incompatible terminal is attached, then this won't erase the display.

.

can be used with:

Boss 32

UMC

CLS Statement

Summary:
The CLS statement erases the current display device and/or graphics screen.

Syntax:
CLS [*n*]

Arguments:

n	an optional integer value of 0, 1, or 2. This value is used to clear the graphics screen or the current active FILE.

0 or no arg. = current FILE
1=the graphics on the GVFD
2=Both .

Description:
CLS sends the clear-screen command to the currently active FILE. It works for the console terminal (FILE 0) and the front panel display (FILE 6, which is the VFD). Note that in order for this to work correctly with FILE 0, the terminal must be set to emulate a supported terminal.

Example:

```
100 INTEGER J
110 FOR J = 1 TO 1000
120  FILE 0: PRINT “*”;
130  FILE 6: PRINT “*”;
140 NEXT J
150 WAIT 100
160 CLS
170 FILE 0: CLS
```

CNTRMODE Statement

can be used with:

Boss 32

UMC

Summary:
The CNTRMODE statement sets the operating mode of the high speed counter.

Syntax:
CNTRMODE *num*, *mode*

Arguments:

num	an integer expression. The counter number to set up: 1 through 13. If there are onboard counters then it is 1 (and 2), and the counter expansion modules would begin with the last on-board Channel H, and continue for the entire counter mod ule(s). If there is no onboard counter, then the counter expansion modules would be 1 through 12 (or 4 or 8, depending upon how many modules are installed).
mode	an integer expression. The mode to set the counter to; one of the following:

0 - disable the counter
1 - X1 quadrature mode
2 - X4 quadrature mode
3 - a pulse on counter input A causes the count to increase, and a pulse on input B causes the count to decrease.
4 - counter input B sets the direction of counting (increase or decrease), and a pulse on input A causes the counter to count by 1. *(see note)*
5 - X1 quadrature mode. RES input will enable/disable counter.
6 - X4 quadrature mode. RES input will enable/disable counter.
7 - a pulse on counter input A causes the count to increase, and a pulse on input B causes the count to decrease. RES input will enable/disable counter.
8 - counter input B sets the direction of counting (increase or decrease), and a pulse on input A causes the counter to count by 1. RES input will enable/disable counter. *(see note)*

Important

NOTE: Modes 3 and 7 are not available on UMC. In modes 4 and 8, on the UMC, if input B is high the count direction is up, and if input B is low, the count direction is down. In modes 4 and 8, on the Boss 32, if input B is high the count direction is down, and if input B is low, the count direction is up.

Description:
TheDivelbiss 32-bit Controller's high speed counter circuit is extremely flexible; it is capable of operating in several modes, depending upon how the software and hardware is configured. The hardware is configured using various jumpers. The software is configured with the CNTRMODE statement. See chapter 7 for a description of the counter hardware.

Example:

```
100 CNTRMODE 1,2                ‘ Set counter 1 to X4 quadrature mode
```

COMPILE Direct Command

can be used with:

Boss 32

UMC

Summary:
The COMPILE direct command compiles the BASIC program currently in memory.

Syntax:
COMPILE (may be abbreviated to C)

Arguments:
COMPILE needs no arguments.

Description:
COMPILE causes the Divelbiss 32-bit BASIC compiler to convert the BASIC source code entered by the programmer into the native machine code used by the Divelbiss 32-bit Controller's 80380 processor. This is necessary before the program can be executed. For a large program, it may take as much as 20 seconds to compile. If the program is successfully compiled, then the message COMPILED will be displayed, otherwise an error message will be displayed. After the program has compiled successfully, the command STAT can be used to show some compilation statistics.

.

CONCAT$ Function

can be used with:

Boss 32

UMC

Summary:
The CONCAT$ function returns a string that is the concatenation of two strings.

Syntax:
st$ = CONCAT$ (st1$, st2$)

Arguments:

st1$	a string expression.
st2$	a string expression.

Description:
CONCAT$ combines two strings into one, appending *st2$* immediately after *st1$*. In some other implementations of the BASIC language, strings are concatenated using the ‘+’ operator.

Example:

```
100 STRING A$(40), B$                    ‘ A$ must be large enough to hold both.
110 A$=”This is one string, “
120 B$=”and this is another.”
130 A$=CONCAT$(A$, B$)
140 PRINT “<”;A$;”>”
```

COS Function

can be used with:

Boss 32

UMC

Summary:
The COS function calculates the cosine function.

Syntax:
x = COS (*expr*)

Arguments:
expr a numeric expression.

Description:
The COS function returns the cosine of its argument, which must be a numeric expression, in degrees. The result is returned as a REAL value.

Example:

```
100 REAL X,Y
110 PRINT COS(45.0)
120 X=68.3
130 Y=COS(X)
140 PRINT “Cosine value of “;X;” is “;Y
```

This produces the following output when run:

```
.70711

Cosine value of 68.29999 is .36975
```

CVI Function

can be used with:

Boss 32

UMC

Summary:
The CVI function converts a binary string to an integer.

Syntax:
x = CVI (st$)

Arguments:

st$	a string expression.

Description:
CVI performs the opposite function of MKI$; it takes the first four bytes of a string and returns them as an integer value. See MKI$ for a full explanation of binary strings.

Example:

```
100 STRING A$
110 A$=MKI$(12345678)
120 PRINT CVI(A$)
```

This produces the following output when run:

```
12345678
```

can be used with:

Boss 32

UMC

CVS Function

Summary:
The CVS function converts a binary string to a real.

Syntax:
x = CVS (st$)

Arguments:

st$	a string expression.

Description:
CVS performs the opposite function of MKS$; it takes the first four bytes of a string and returns them as a real value. See MKI$ for a full explanation of binary strings.

Example:

```
100 STRING A$
110 A$=MKS$(123.456)
120 PRINT CVS(A$)
```

This produces the following output when run:

```
123.45599
```

can be used with:

Boss 32

UMC

DAC Statement

Summary:
The DAC statement is used to control analog output channels.

Syntax:
DAC chan, value

Arguments:
chan an integer expression. The number of the channel to access, starting at 1.
value an integer expression. The level to set on the analog output, ranging between 0 and 32767 for the Divelbiss 32-bit Controller.

Description:
The DAC statement is used to assign an output level to each DAC channel. The onboard D/A channels are set for a 0 to 10 volt or 0 to 20mA out. Each DAC channel of the modules can be set up for a variety of output voltage and current ranges, such as 0 to 10 volts, -10 to 10 volts, 4 to 20 mA, etc. This means that the actual output supplied by a particular DAC statement value depends upon the configuration of the analog output module. The D/A channels are assigned based on the hardware available on the 32-bit controller; channel 1 could be on the onboard D/A, or in any of the expansion ports (if there is no onboard D/A converter). The order of precedence for assigning D/A channels is: onboard D/A followed by J3 followed by J4 followed by J5 when three expansion ports are available.

Example:

```
100 ‘ Program to output a sine wave on DAC channel 1.
110 INTEGER J,Y
120 FOR J=0 TO 628                    ‘ Go through an entire cycle.
130  Y=32767.0*SIN(J/100.0)           ‘ Scale sine to fit Boss 32 DAC range.
140  DAC 1,Y                          ‘ Output the value.
150 NEXT J
160 GOTO 120                          ‘ Loop forever.
```

Notice that in line 130, the constants (32767.0 and 100.0) are given as real numbers (with a decimal point). By doing this, the program runs slightly faster, since the compiler doesn’t have to convert the numbers from integer to real each time that the line is executed.

can be used with:

Boss 32

UMC

DATA Statement

Summary:
The DATA statement defines constant values to be accessed with READ statements.

Syntax:
DATA data_list

Arguments:

data_list	list of constant values separated by commas. The values may be numeric or string. Strings must be surrounded by double quotes, and may contain commas, colons, spaces, and other punctuation.

Description:
DATA statements are used to define a block of constants which will be read by READ statements. All DATA statements must be in the program before the first READ statement; when the first READ statement is encountered, BASIC searches for all DATA statements. While the program is executing, each READ statement accesses the next DATA element in order; the DATA elements are treated as a continuous list of items, regardless of how they are arranged into statements. The variable type given in the READ statement must agree with the corresponding constant in the DATA statement.

Example:

```
100 INTEGER J,K
110 REAL X
120 STRING A$
130 PRINT “DATA statement example”
140 ‘ Data table for program
150 DATA 4.77,2,3,4,5,23,83,8              ‘ Must be before first READ
160 DATA 999,3.14159,1.6667
170 DATA 893.664,999,”Data 1”
180 DATA “This is a test: Hi, everybody”
190 FOR J=1 TO 3
200  READ K: PRINT K                       ‘ Print first 3 elements
210 NEXT J
220 READ J
230 IF J<>999 THEN PRINT J: GOTO 220       ‘ Print elements until a 999
240 READ X
250 IF X<999.0 THEN PRINT X: GOTO 240      ‘ Print elements until a 999
260 READ A$: PRINT A$                      ‘ Print first string
270 READ A$: PRINT A$                      ‘ Print second string
280 READ X: PRINT X                        ‘ Wrap around, do first one again
```

. This produces the following output when run:

```
DATA statement example
4
2
3
4
5
23
83
8
3.14159
1.66670
893.66381
Data 1
This is a test: Hi,
4.77000
```

The two 999 values in lines 160 and 170 are used as flags to indicate the end of portions of the table; this makes it easy to add to the DATA table without needing to change the program where the READ statements are executed. In line 270, it prints "This is a test: Hi, " because the string variable A$ defaults to 20 character maximum length, so it truncates the rest of the string when it is read. Note that in line 280, it reads beyond the end of the DATA table, so it wraps around and reads the first value in the table.

can be used with:

Boss 32

UMC

DEF Statement

Summary:
The DEF statement marks the beginning of a user defined function.

Syntax:
DEF funcname, [arg1] [,arg2]...

Arguments:

funcname	a string constant. The name to use for the function that will be defined on the following lines; this must follow the rules for variable names.
argx	a numeric or string expression. An argument to be passed to the function.

Description:
The DEF statement marks the beginning of a user defined function.

Example:

.

can be used with:

Boss 32

UMC

DELETE Direct Command

Summary:
The DELETE direct command deletes a file from the FLASH EPROM.

Syntax:
DELETE [CODE] [filenum] [filename]

Arguments:

CODE	IF code is supplied it deletes file TYPE of CODE, otherwise it deletes file TYPE of SOURCE.
filenum	filenumber of file to delete
filename	filename of file to delete

Description:
DELETE marks the specified file on the FLASH EPROM as deleted. A listing of the deleted files can be displayed by using direct command DIRDEL.

Example:

DELETE filename	Deletes BASIC source file filename
DELETE filenum	Deletes BASIC source file number filenumber
DELETE CODE filename	Deletes BASIC CODE file filename
DELETE CODE filenum	Deletes BASIC CODE file number filenumber

DIN Function

can be used with:

Boss 32

UMC

Summary:
The DIN (Digital Input) function reads input expander modules attached to the Divelbiss 32-bit Controller.

Syntax:
x = DIN (*addr*)

Arguments:
addr an integer expression. The address of the input channel to read.

Description:
One or more Divelbiss I/O expander boards can be attached to the Divelbiss 32-bit Controller; DIN is used to read the input modules. It returns an INTEGER value; a 0 if the input is off, or a 1 if the input is on. *addr* is the input address to read; it will be a number between 0 and 127 ($00 and $7F).

DIN can also be used to read the current keypad status; for example, so that one of the keypad buttons can be used as a “jog” button. The keypad is accessed when *addr* is 256 ($100); the INTEGER returned is the key number (same thing returned by KEY). When the KEY function is used to read the keypad, it has an autorepeat delay which makes it hard to determine how long a button has been pressed. DIN just returns the status of the keypad at the current time, disregarding the autorepeat logic. DIN reads the keypad regardless of what the current FILE is, unlike KEY, which only reads the keypad if a FILE 6 statement has been executed.

DIN can also be used to read what was written to digital outputs. The output state is accessed when *addr* is between 512 and 639 ($200 and $27F).

DIN can also be used to read the timer values. The timer is accessed when *addr* is between 1024 and 1245 ($400 and $4DD).

Examples:

```
100 ‘ Program to read all 16 inputs on I/O page 3.
110 INTEGER J,K
120 FOR J=$30 TO $3F
130  K=DIN(J)
140  PRINT J,K
150 NEXT J
100 ‘ Program to display the current keypad status.
110 INTEGER J
120 LOCATE 1,1
130 FPRINT “I3Z”, DIN($100)
140 GOTO 120
```

DIR Direct Command

can be used with:

Boss 32

UMC

Summary:
The DIR direct command displays a list of files contained on the user's FLASH EPROM.

Syntax:
DIR

Arguments:
DIR needs no arguments.

Description:
DIR displays a list of files stored on the FLash EPROM. It displays the file name, file number (files are numbered sequentially), file type (source or code), file size (truncated to the nearest KB), and date and time the file was saved. It also displays the amount of free space on the EPROM, truncated to the nearest KB.

Example:

.

```
DIR
```

DIRDEL Direct Command

can be used with:

Boss 32

UMC

Summary:
The DIRDEL direct command displays a listing of deleted files contained on the user's FLASH EPROM.

Syntax:
DIRDEL

Arguments:
DIRDEL needs no arguments

Description:
Use same descrption as dir, but change change file(s) to deleted file(s)

Example:
DIRDEL

.

can be used with:

Boss 32

UMC

DOUT Statement

Summary:
The DOUT (Digital Output) statement controls output expander modules attached to the Divelbiss 32-bit Controller.

Syntax:
DOUT addr, state

Arguments:

addr	an integer expression between 0 and 127. The address of the output channel to turn off or on.
state	an integer expression. A value of 0 turns the output off; any other value turns the output on.

Description:
One or more Divelbiss I/O expander boards can be attached to the Divelbiss 32-bit Controller; DOUT is used to control output modules.

DOUT can also be used to reset the HDIO bus which turns off all outputs. The *addr* to reset the HDIO buss is 256 ($100). The *state* does not matter.

DOUT can also be used to toggle the internal watchdog timer. This allows the basic programmer to keep the watchdog timer happy when interrupts are off for extended periods of time. The *addr* to toggle the watchdog is 257 ($101). The *state* does not matter.

Example:

```
100 ‘ Program to turn all 16 outputs on I/O page 2 off, then delay
110 ‘ for 2 seconds, then turn them on.
120 INTEGER J,K
130 FOR J=0 TO 1                    ‘ Turn outputs off, then on
140  FOR K=$20 TO $2F               ‘ Access all outputs on I/O page 2
150   DOUT K,J
160  NEXT K
170  WAIT 200
180 NEXT J
```

can be used with:

Boss 32

UMC

DOWNLOAD Direct Command

Summary:
The DOWNLOAD direct command disables echoing to the serial port to provide for more reliable program downloading.

Syntax:
DOWNLOAD

Arguments:
DOWNLOAD needs no arguments.

Description:
Normally, the Divelbiss 32-bit Controller echoes all characters entered at the command line prompt back to the terminal, to allow the user to see what is being typed. When downloading a program, however, this causes errors to be scrolled off of the screen. The DOWNLOAD command solves this by disabling the console echo until either an END command or a CTRL-Z is received; any errors that are encountered are still displayed. After the END or CTRL-Z is received, the message Warning: duplicate line numbers detected is displayed if two lines were entered with the same line number; this may or may not be a problem, depending upon the programmer’s style.

The easiest way to use DOWNLOAD is to put the DOWNLOAD command as the first line in the file containing the BASIC program (don’t put a line number in front of DOWNLOAD) and put END as the last line in the file (again, with no line number). Then, when the file is sent to the Divelbiss 32-bit Controller, only error messages will be displayed. Many editors put a CTRL-Z at the end of files; if this is the case, then the END does not need to be entered into the file, since DOWNLOAD will enable echoing again when it sees the CTRL-Z.
Another advantage of using DOWNLOAD is that the file transfer to the Boss 32 will be faster, since the Divelbiss 32-bit Controller must use processor time to echo the characters back.

Note that if an error is detected while downloading a file, the Divelbiss 32-bit Controller may miss some characters while it is printing the error message, which will cause other errors to be displayed that aren’t actually valid.

Example:

The following shows how to create a file that will download as described above.

```
DOWNLOAD
100 ‘ This is the BASIC program to be downloaded
110 PRINT “This is a BASIC program”
END
```

can be used with:

Boss 32

UMC

EDIT Direct Command

Summary:
The EDIT direct command allows the programmer to modify the BASIC source code.

Syntax:
EDIT linenum
or
E linenum

Arguments:
linenum a valid BASIC line number. The number of the line to be modified.

Description:
EDIT invokes the Bear Basic line editor. The line is displayed with the cursor positioned at the first character. The following function keys can be used:

Control S -	Move the cursor left one position.
Control D -	Move the cursor right one position.
Control G -	Delete the character at the cursor.
Backspace -	Delete the character one space to the left of the cursor.
Control A -	Beginning of Line.
Control F -	End of Line.
Control V -	Toggle insert mode on and off.

If insert mode is off, any character entered will replace the character at the cursor and will move the cursor to the right one space. If insert mode is on, any character entered will be added at the cursor position, causing the entire line at the cursor and to the right of the cursor to move right one space. A carriage return will cause the modified line to be sent to the compiler. The editor will only work with a line short enough to fit on a single 80 character line.

Note that EDIT will not work correctly unless the console terminal is set up to emulate a supported terminal type.

EEPEEK Function

can be used with:

Boss 32

.

Summary:
The EEPEEK function returns the value stored at the specified address in the EEPROM memory.

Syntax:
x = EEPEEK (*addr*)

Arguments:

addr	an integer expression between 0 and 1999 (for the 8K EEPROM), between 0 and 8143 (for the 32K EEPROM. The address to be read from.

Description:
The Divelbiss 32-bit Controller can be purchased with an optional EEPROM memory device installed; this can be either a 8KB or 32KB device. The EEPEEK function operates in a similar fashion to the WPEEK function, except that it reads from the EEPROM instead of from the RAM. EEPEEK returns the INTEGER value stored at *addr* in the EEPROM; note that the address is specified as an offset from the beginning of the EEPROM. Unlike the EEPOKE statement, EEPEEK can be performed an unlimited number of times, and it operates very quickly (approximately 40 microseconds). The EEPEEK statement takes about 100 microseconds to execute.

The second example demonstrates how to store real variables in the EEPROM.

Examples:

```
100 ‘ Program to read the first 20 values from EEPROM.
110 INTEGER J,K
120 FOR J=0 to 19
130  K=EEPEEK(J)
140  PRINT J,K
150 NEXT J
```

```
100 REAL X,Y,HOLD
110 INTEGER FIRST
120 X=3.45
130 FIRST=WPEEK(ADR(X))
140 EEPOKE 10,FIRST
150 HOLD=EEPEEK(10)
160 WPOKE ADR(Y),HOLD
200 PRINT”Value of Y is”:Y
300 PRINT”Value of 10=”;:PRINT EEPEEK(10)
```

EEPOKE Statement

can be used with:

Boss 32

Summary:

The EEPOKE statement stores a value at the specified address in the EEPROM memory. Note that this statement takes approximately 20 milliseconds to return. Also, each byte of the EEPROM can only be written to approximately 10,000 times before the byte fails.

Syntax:

EEPOKE *addr*, *value*

Arguments:

addr	an integer expression between 0 and 1999 (for the 8K EEPROM), between 0 and 8143 (for the 32K EEPROM). The address to be written to.
value	an integer expression. The value to write into the EEPROM.

Description:

The Divelbiss 32-bit Controller can be purchased with an EEPROM memory device installed; this can be either a 8KB or 32KB device. The EEPOKE statement operates in a similar fashion to the WPOKE statement, except that it writes to the EEPROM instead of to the RAM. EEPOKE stores the INTEGER *value* at *addr* in the EEPROM; note that the address is specified as an offset from the beginning of the EEPROM. Remember that, due to the characteristics of EEPROM technology, each byte in the EEPROM can only be written to approximately 10,000 times before failing; if the value to be written matches the value already stored at that location, then EEPOKE won't perform the write operation, so that it doesn't waste one of the 10,000 write cycles. Note also that it takes about 20 msec for this statement to return, since EEPROMs write each byte in about 10 msec.

See EEPEEK for an example demonstrating how to store real numbers in the EEPROM.

Example:

```
100 ‘ Program to write 20 values into EEPROM.
110 INTEGER J,K
120 FOR J=0 to 19
130  K=J*3+7
140  EEPOKE J,K
150 NEXT J
```

END Direct Command

can be used with:

Boss 32

UMC

Summary:
The END direct command enables the echoing to the serial port after DOWNLOAD.

Syntax:
END

Arguments:
END needs no arguments

Description:
The END direct command will reanble the echoing of charachters to the serial port after a DOWNLOAD
direct command.

.

Example:

```
DOWNLOAD
100 ‘This is the BASIC program to be downloaded
110 PRINT “This is a BASIC program”
END
```

ERR Function

can be used with:

Boss 32

UMC

Summary:
The ERR function returns the number corresponding to the last error that was detected.

Syntax:
x = ERR

Arguments:
ERR needs no arguments.

Description:
Divelbiss 32-bit BASIC detects two kinds of errors at runtime: I/O errors and program errors. An I/O error indicates that an I/O device reported an error during a transfer; this happens most often with serial data transfer. A program error indicates that a serious program error was detected, such as Return Without Gosub or Subscript Out of Range. The ERR function returns the integer number that corresponds to the most recently detected error:

0	$0	No error
2	$2	Failure programming EEPROM
3	$3	EEPROM address out of range
10	$A	COM1 serial transmission timeout
11	$B	COM2 serial transmission timeout
12	$C	COM3 serial transmission timeout
13	$D	COM4 serial transmission timeout
16	$10	COM1 serial receive error (overrun, parity, framing)
17	$11	COM2 serial receive error (overrun, parity, framing)
18	$12	COM3 serial receive error (overrun, parity, framing)
19	$13	COM4 serial receive error (overrun, parity, framing)
256	$100	Undefined error
263	$107	Expression Error
264	$108	String Variable Error
266	$10A	Line Number Does Not Exist
267	$10B	Task Error
271	$10F	RETURN Without GOSUB
272	$110	Subscript out of Range
273	$111	Overflow
275	$113	Task Mismatch
276	$114	Function Error
277	$115	String Length Exceeded
280	$118	Illegal Print/Input Format
281	$119	String Space Exceeded
282	$11A	Illegal File Number
283	$11B	Improper Data to INPUT Statement
285	$11D	Data Statement Does Not Match Read
286	$11E	Failure Programming EPROM or EEPROM

The error numbers greater than 255 ($FF) are fatal program errors that will cause the program execution to stop. The error numbers up to 255 ($FF) are nonfatal I/O errors that can be handled with the ON ERROR statement or by checking the ERR function periodically. Note that when ERR is referenced, it returns the most recent I/O error detected and resets the I/O error to 0. This means that an error on one device could overwrite an error on another device if ERR isn't checked often enough. It also means that the ERR value must be read into another variable if it will be needed again (see the example below). A successful I/O operation won't reset the I/O error value to 0; if a character is received on COM1 with a parity error and then other characters are received successfully, when ERR is checked it will return 16 to indicate that a receive error occurred on COM1.

Example:

```
100 INTEGER J,K
110 K=KEY
120 IF K<>0 THEN PRINT CHR$(K);
130 J=ERR
140 IF J<>0 THEN FILE 6: PRINT "Error ";J: FILE 0
150 GOTO 110
```

Run this example with the terminal set to 9600 bps, even parity, 8 data bits, and 1 stop bit. Some of the ASCII characters sent to the Divelbiss 32-bit Controller will generate an error 16, because those characters will cause a framing error when received by the Divelbiss 32-bit Controller, which is using 9600 bps, no parity, 8 data bits, and 1 stop bit. The value of ERR is stored in the variable J in line 130 because it may be needed twice in line 140. If line 130 were removed and J in line 140 were replaced with ERR, then it would always print "Error 0", because the first reference to ERR would set the error status back to 0.

can be used with:

Boss 32

UMC

ERROR Direct Command

Summary:
The ERROR direct command enables error checking in the compiled BASIC program.

Syntax:
ERROR

Arguments:
ERROR needs no arguments.

Description:
ERROR turns on the compiler's runtime error checking software. It is the converse of NOERR. When Divelbiss 32-bit BASIC is first started, all runtime error checking is on. It stays on until explicitly disabled by NOERR. Runtime error checking is essential in most programs to insure that mistakes don't "blow up" the compiler. For example, if a "SUBSCRIPT OUT OF RANGE" error were not detected, a program could write over itself, causing unpredictable and undesirable results.

Like NOERR, ERROR modifies the code generated during compilation of the program, so it must be specified before the program is RUN or COMPILEd (if you have used NOERR in the Divelbiss 32-bit BASIC session before RUN or COMPILE). ERROR will then remain in effect unless disabled by NOERR or NEW.

Unfortunately, the runtime error checking software makes the compiled code larger and slower. If code size and speed are important, then the program should be compiled with NOERR enabled after it has been debugged.

EXIT Statement

can be used with:

Boss 32

UMC

Summary:
The EXIT statement causes the currently executing task to abort. When the schedule interval specified in the RUN statement has elapsed, the task will be started at its beginning.

Syntax:
EXIT

Arguments:
EXIT needs no arguments

Description:
EXIT causes the currently executing task to abort. When the EXIT is encountered, the task stops until the schedule interval specified in the RUN statement elapses, at which point the task starts over again from its beginning. Note that EXIT does not stop the task from being rescheduled; only CANCEL can do that. Whenever EXIT is executed, it automatically turns interrupts back on (ie. simulates an INTON). This feature is needed by hardware device interrupt handlers to insure that the device interrupt handler can return just as interrupts are re-enabled.

Example:

```
100 INTEGER J,K
110 PRINT “Press any key to stop”
120 J=0: RUN 1,100                    ‘ Start task 1 with 1 sec reschedule.
130 K=GET                             ‘ Wait for a key.
140 CANCEL 1                          ‘ Cancel task 1.
150 PRINT “Wait 2 seconds”: WAIT 200  ‘ Wait for it to stop.
160 STOP                              ‘ Halt the program.
200 TASK 1
210 PRINT J;
220 J=J+1
230 WAIT 50                           ‘ Wait for .5 seconds.
240 PRINT “*”;CHR$(13);               ‘ Stay on same line - print CR.
250 EXIT                              ‘ Abort task, go get rescheduled.
```

can be used with:

Boss 32

UMC

EXMOD Statement

Summary:
The EXMOD statement is used to communicate with any of the intelligent expander modules.

Syntax:
EXMOD port, st$ [, numbytes]

Arguments:

port	expansion port number. 3, 4, or 5 corresponding to J3, J4, or J5. (Port 3 for UMC)
st$	string containing data to transfer to the module. If we are reading from the module, then the return data will be passed back in this string, also.
numbytes	this parameter is only specified when reading from the module. It is the number of bytes to be returned back from the module.

Description:
Some of the Divelbiss I/O expander modules are classed as intelligent expanders, which means that they are a complete microprocessor system that acts as a slave processor to the Divelbiss 32-bit Controller. These modules offload complex I/O processing tasks from the Divelbiss 32-bit Controller's processor, allowing faster system throughput and better I/O timing accuracy. Since these modules are custom-programmed to perform their specified task, the commands and parameters for each module are unique. The EXMOD statement performs the communications between the Divelbiss 32-bit Controller's processor and the module's processor. The exact form of this communications depends on the module being used; refer to the module's data sheet to find detailed information.

EXMOD writes a command (stored in *st$*) to the module plugged into the expansion connector referred to by *port*. If the command causes a response from the module, then EXMOD reads *numbytes* of data from the module, which is then returned in *st$*.

Example:

```
100 ‘ Example to show EXMOD communicating with stepper module.
110 INTEGER PORT,CHNL: STRING EX$(10)
200 PORT=4: CHNL=1                           ‘ Module in J4, stepper channel 1
300 ‘ Set acceleration and deceleration rates for channel 1.
310 EX$=CONCAT$(CHR$($83),CHR$(CHNL))  ‘ Command $83 and channel #
320 EX$=CONCAT$(EX$,MKI$(2000))            ‘ Accel rate = 2000 pulse/second
330 EX$=CONCAT$(EX$,MKI$(1500))            ‘ Decel rate = 1500 pulse/second
340 EXMOD PORT,EX$                         ‘ Send data to module
400 ‘ Read status back from stepper module.
410 EX$=CONCAT$(CHR$($84),CHR$(CHNL))  ‘ Command $84 and channel #
420 EXMOD PORT,EX$,7                       ‘ Read status from module (7 bytes)
430 FPRINT “H2”,ASC(EX$)                   ‘ Display module mode in hexadecimal
440 PRINT CVI(MID$(EX$,2,2))               ‘ Display low word of step count
450 PRINT CVI(MID$(EX$,6,2))               ‘ Display current pulse rate
```

EXP Function

can be used with:

Boss 32

UMC

Summary:
The EXP function calculates the exponential function.

Syntax:
x = EXP (*expr*)

Arguments:
expr a numeric expression.

Description:
The EXP function returns e^{expr}, which must be a numeric expression. The result is returned as a REAL value. This is the converse of the natural logarithm function (LOG).

By using the properties of logarithms, a number X can be raised to a power Y (XY) using EXP(LOG(X)*Y).

Example:

.

```
100 REAL X,Y
110 PRINT EXP(0.35)
120 X=-1.82
130 Y=EXP(X)
140 PRINT “Exp value of “;X;” is “;Y
150 PRINT “4 cubed = “; EXP(LOG(4)*3)
```

This produces the following output when run:

```
1.41834
Exp value of -1.82000 is .16171
4 cubed = 64.00003
```

can be used with:

Boss 32

UMC

FILE Statement

Summary:

The FILE statement causes all subsequent character I/O to be performed with the specified device.

Syntax:

FILE *fnum [,[baud][,[parity][,[data][,[stop]]]]]*
or
FILE *6 [,[display] [,[“mode”]]]*

Arguments:

fnum an integer expression. The file number to access; one of the following:

0	COM1 serial port (the “console” port)
1	COM2 serial port
2	future expansion
3	future expansion
4	COM3 serial port
5	COM4 serial port
6	onboard display and keypad
7	future expansion

baud an optional integer expression. The baud can be set to 150, 300, 600, 1200, 2400, 4800, 9600, 19200, 38400, 57600 on all COM ports.

parity a optional string expression. The valid expressions are: N (none), E (even), O (odd), S (space), M (mark).

data an optional integer expression. The data length can be set to 5, 6, 7, or 8 bits.

stop an optional integer expression. The number of stop bits can be 1 or 2.

display an optional integer expression. This is used for FILE 6 to set which display is active on the GVFD. 0 = no change, 1 = graphics display on / text display off, 2 = both displays on, 3 = text display on / graphics display off, 4 = both displays off.

mode an optional string expression. This is used for FILE 6 to set the mode for the graphics on the GVFD. “O” (OR) text and graphics, “X” (XOR) text and graphics, ” A” (AND) text and graphics.

Description:

The FILE statement causes all subsequent I/O from PRINT, FPRINT, INPUT, INPUT$, FIN-PUT, GET, KEY, and LOCATE, to be performed with the specified device. It does not effect the graphics I/O (LINE, PSET, POLY, ZPRINT) from being performed on the Graphics Vacuum Fluorescent Display (GVFD). The two optional expressions on FILE 6 only are valid for the GVFD, these allow the display to be turned on and off and set the mode (OR, XOR, AND). The FILE statement sets the file number for the task that issues the statement; each task maintains its own current file number. The file number for each task defaults to file 0, not to the file number used in the preceding task. The optional expressions for the FILE statement are used for the COM ports to change the communication settings.

Important

NOTE: File 6 is not available on UMC.

(continued on following page)

.

Example:

```
100 PRINT “This is task 0”                    ‘ Output to console (COM1).
110 RUN 1,100: RUN 2,250: RUN 3,325
120 WAIT 1000: STOP                           ‘ Allow tasks to run a few times.
200 TASK 1
210 FILE 5                                    ‘ Output to COM2
220 PRINT “This is task 1”: EXIT
300 TASK 2
310 FILE 6                                    ‘ Output to onboard display.
320 PRINT “This is task 2”: EXIT
400 TASK 3
410 PRINT “This is task 3”                    ‘ Output to console (COM1).
420 FILE 6                                    ‘ Also to onboard display.
430 PRINT “Also task 3”: EXIT
```

can be used with:

Boss 32

UMC

FINPUT Statement

Summary:
The FINPUT statement provides a simple way to get numeric entry from the user.

Syntax:
FINPUT format, variable

Arguments:

format	a string expression. This defines the format of the number to get from the user. The following are valid:
	Ux unsigned integer, where x is the number of digits allowed.
	Ix signed integer, where x is the number of digits allowed, not including the minus sign (ie. “I2” will allow the user to type -23).
	Fx.y real, where x is the number of digits to the left of the decimal point, and y is the number to the right.
variable	integer or real variable name. The value entered will be put into this variable.

Description:
FINPUT provides a simpler way to get numeric input from the user. It will only accept entries that follow the specified format; the user will not be allowed to deviate from this format. For example, if a 2 digit product code must be entered, the statement FINPUT “I2”, PRDCOD will only allow 2 digits to be entered. If the user presses backspace (when using a terminal with FILE 0, 1, 4, or 5) or CLEAR (when using the built in keypad) after entering some digits, then these digits will be erased and the user can start again; the cursor will be at its original starting position. If the format string does not follow the form recognized by Divelbiss 32-bit BASIC, then FINPUT will instantly return zero, without waiting for input from the user.

FINPUT should not be used in an interrupt task, because it re-enables interrupts and allows multitasking to continue. Depending upon what the other tasks are doing at the instant that the interrupt task executes, FINPUT could cause the system to lock up.

Example:

```
100 INTEGER J
110 REAL X
120 CLS: LOCATE 10,1
140 PRINT “Enter product number (0-99) >  “;
150 LOCATE 10, 31: FINPUT “I2”, J
160 LOCATE 12,1
170 PRINT “Enter temperature setpoint (180.00-209.99) >    “;
180 LOCATE 12,46: FINPUT “F3.2”, X
190 IF X < 180.0 OR X >= 210.0 THEN GOTO 160
```

FNEND Statement

can be used with:

Boss 32

UMC

Summary:
The FNEND statement marks the end of a user defined function.

Syntax:
FNEND

Arguments:
FNEND needs no arguments

Description:
FNEND is used in conjunction with DEF to create a user defined function: DEF marks the beginning of the function, and FNEND marks the end.

Example:

```
100 INTEGER RIGHTN
110 STRING TEST$, RIGHT$(127), RIGHTA$(127)
120 ‘ This function returns the N rightmost characters of A$.
130 DEF RIGHT$ (RIGHTA$, RIGHTN)
140  RIGHT$=MID$(RIGHTA$, (LEN(RIGHTA$)-RIGHTN+1, RIGHTN)
150 FNEND
160 TEST$=”This is a test”
170 PRINT TEST$;” <”;RIGHT$ (TEST$, 4);”>”
```

This produces the following output when run:

```
This is a test <test>
```

FOR Statement

can be used with:

Boss 32

UMC

Summary:

The FOR statement is used with the NEXT statement to implement a loop control structure to execute a range of lines multiple times.

Syntax:

FOR *variable* = *expr1* TO *expr2* [STEP *expr3*]

Arguments:

variable	an integer or real variable, to be used as the loop counter. This must be a non-subscripted variable.
expr1	a numeric expression. This is the starting value of the loop. *variable* will be set to this value before executing the body of the loop for the first time.
expr2	a numeric expression. This is the ending value of the loop. *variable* is tested against this value after executing the body of the loop; if *variable* is less than *expr2* then the loop will be executed again.
expr3	an optional numeric expression. If specified, this value will be added to *vari able* each time through the loop. If this isn't specified, then *variable* will be incremented by 1 each time.

Description:

The FOR..NEXT construct specifies that a series of lines should be executed multiple times in a loop. Program lines following the FOR statement are executed until the NEXT statement is encountered, at which time *variable* is incremented by *expr3*, or by 1 if *expr3* is not specified. If *variable* is less than *expr2*, then BASIC jumps back to the statement immediately following the FOR statement, and this process repeats. If *variable* is greater than or equal to *expr2*, then BASIC continues execution with the statement following the NEXT statement. For example, the statement FOR X=4 TO 13 STEP 2 will cause the loop to be executed 5 times, for 4, 6, 8, 10, and 12.

In order to cause a loop to count downwards, a negative STEP value may be used. If *expr3* is negative, then *variable* will be decremented each time through the loop. For example, the statement FOR J=10 TO 1 STEP -1 will cause the loop to be executed 10 times, for 10, 9, 8, ..., 1.

Note that the body of the loop is always executed once, because the test *variable<expr2* is performed when NEXT is encountered.

It is possible to alter the value of *variable* inside the body of the loop. Extreme care should be used when doing this, however, as it can result in very subtle problems. In general, *variable* should not be altered, except by the FOR statement itself.

A FOR..NEXT loop may be used inside of another FOR..NEXT loop; this is called a nested loop. Each loop must have a unique variable name in the *variable* field. The inside FOR..NEXT loop must be entirely enclosed in the outer one.

.

Example:

```
100 INTEGER J,K
110 REAL X,Y
120 FOR J=3 TO 8
130  FOR K=1 TO 7 STEP 2
140   FPRINT "I2I2X2Z", J, K
150  NEXT K: PRINT
160 NEXT J
170 PRINT
180 FOR X=360.0 TO 0.0 STEP -22.5
190  FPRINT "F3.1F7.3", X, SIN(X)
200 NEXT X
```

This produces the following output when run:

```
 3 1    3 3    3 5    3 7
 4 1    4 3    4 5    4 7
 5 1    5 3    5 5    5 7
 6 1    6 3    6 5    6 7
 7 1    7 3    7 5    7 7
 8 1    8 3    8 5    8 7

360.0     .000
337.5    -.383
315.0    -.707
292.5    -.924
270.0   -1.000
247.5    -.924
225.0    -.707
202.5    -.383
180.0     .000
157.5     .383
135.0     .707
112.5     .924
 90.0    1.000
 67.5     .924
 45.0     .707
 22.5     .383
   .0     .000
```

FPRINT Statement

can be used with:

Boss 32

UMC

Summary:
The FPRINT statement allows formatted printing to the current FILE.

Syntax:
FPRINT format, arg1 [,arg2]...

Arguments:
format a string expression. This specifies how the following arguments are to be printed. The following symbols are valid:

Fn.x	prints n digits before the decimal point and x digits following it. Leading zeros are converted to spaces, and trailing zeros are left as zeros. No more than 6 positions following the decimal point are allowed.
Hn	prints n hexadecimal digits. Leading zeros are printed.
In	prints n integer digits. Leading zeros are converted to spaces.
Sn	print n characters of a string. If the string more than n characters long, then the first n characters are printed. If the string is shorter than n characters, then trailing spaces are printed.
Un	print n unsigned integer digits. Leading zeros are converted to spaces.
Xn	prints n spaces.
Z	suppresses the carriage return at the end of the line. This is only legal at the end of the format string.
argx	a numeric or string expression. Each argument must match the corresponding entry in the format string.

Description:
FPRINT is a more sophisticated version of the PRINT statement which allows the programmer to control the format of the data output by the program. FPRINT prints each argument in the list of expressions using the corresponding control information from the format string (*format*, above). The type of each expression must match it's corresponding control string in *format*; a runtime error will occur if the types do not match. The specified field widths are strictly enforced; each expression is forced to fit in it's field. If a numeric value is too large to fit in the specified field, then the field is filled with asterisks instead. For example, if a format of "H2" is given, and the number to be output is $123, then "**" will be printed. For a string field, the string is truncated to fit in the specified field.

FPRINT should not be used in an interrupt task, because it re-enables interrupts and allows multitasking to continue. Depending upon what the other tasks are doing at the instant that the interrupt task executes, FPRINT could cause the system to lock up.

Examples:

```
100 INTEGER J, K
110 REAL X
120 J=123: K=$F7: X=38.8746
130 FPRINT "I5X2H4F5.2", J, K, X
```

This produces the following output when run:

```
123 00F7  38.87
```

Two spaces are printed before the value of J because the field width is set to 5. Two space are printed by the “X2” format in order to separate J and K. Two zeros are printed before the value of K because the field width is set to 4. The fractional part of X is truncated to two digits because the field with is “F5.2”.

```
100 INTEGER J
110 STRING A$, B$
120 J=40000
130 FPRINT “I3I7U7”, J, J, J
140 A$=”Hello”: B$=”There it is.”
150 FPRINT “S5S2S5”, A$, “, “, B$
```

This produces the following output when run:

```
*** -25536    40000
Hello, There
```

In this example, it is important to remember that the representation of a number is different than the value of a number; the same numeric value may be represented many different ways. In particular, the hexadecimal number $9C40 can be represented as a signed integer as -25536, or it can be represented as an unsigned integer as 40000. Three asterisks are printed because -25536 won’t fit into the 3 digits allowed by the “I3” format. The next line shows how strings are handled; B$ is truncated to the specified 5 characters.

can be used with:

Boss 32

UMC

FUZZY Statement

Summary:
The FUZZY statement is a control algorithm for use on the Boss 32.

Syntax:
FUZZY array address, input 1, input 2, output 1, output 2, status

Arguments:

array address	an integer value of the address of the first element of the parameter array. The array contains all the mid values, sensitivity values, and rules tables for the FUZZY control algorithm
input 1	an integer expression from -32767 to +32767. This is the first input variable passed to the FUZZY routine.
input 2	an integer expression from -32767 to +32767. This is the second input vari able passed to the FUZZY routine.
output 1	an integer expression from -32767 to +32767. This is the first output vari able passed from the FUZZY routine.
output 2	an integer expression from -32767 to +32767. This is the second output variable passed from the FUZZY routine.
status	an integer expression. This will hold the status of the operation; a 0 will indi cate a successful operation, a 1 indicates output 2 was beyond the range of an integer, a 2 indicates output 1 was beyond the range of an integer, and a 3 indicates both outputs were beyond the range of an integer. When an out put is beyond the range of an integer the return value from the FUZZY rou tine will be +32767 or -32767 depending on the actual sign of the output.

Description:
The FUZZY statement supports two inputs, two outputs, seven fuzzy sets with symmetric triangles, user-definable fuzzy set values and a return status. Typical applications include temperature, climate, speed, position, pressure, flow, and fluid depth just to name a few. The statement utilizes a data array which contains 106 integer values which are used by the FUZZY statement to generate the outputs. The array address is the first value required by the statement and provides the starting address of all of the data. Inputs 1 and 2 are integer expressions which must be obtained and passed to the statement. In most process control applications both inputs are obtained from the ADC statement which converts an analog input (i.e., speed, pressure, angle, position, temperature, ...) to the Divelbiss 32-bit Controller into discrete numbers from 0 to 32767. The statement has three return values; output 1, output 2, and status. Outputs 1 and 2 are integer expressions which contain the results of the Fuzzy Logic processing. The status expression will hold the status of the operation; a 0 will indicate a successful operation, a 1 indicates output 2 was beyond the range of an integer, a 2 indicates output 1 was beyond the range of an integer, and a 3 indicates both outputs were beyond the range of an integer. When an output is beyond the range of an integer the return value from the FUZZY routine will be +32767 or -32767 depending on the actual sign of the output.

As mentioned, the statement requires 106 integer values in order to do the Fuzzy Logic processing. Specifically there are 8 values for fuzzy set mid-points and sensitivities, 49 for the rules for output 1 and 49 rules for output 2 for a total of 106. The data must be contained in a data array in the following order:

```
input 1 midpoint, input 1 sensitivity
input 2 midpoint, input 2 sensitivity
output 1 midpoint, output 1 sensitivity
output 2 midpoint, output 2 sensitivity

rule output 1 [in1=1, in2=1]        ...  rule output 1 [in1=1, in2=7]
.
.
rule output 1 [in1=7, in2=1]        ...  rule output 1 [in1=7, in2=7]

rule output 2 [in1=1, in2=1]        ...  rule output 2 [in1=1, in2=7]
.
.
rule output 2 [in1=7, in2=1]        ...  rule output 2 [in1=7, in2=7]
```

Example:

The following example is for a motor speed control problem with two inputs (speed error and rate of change of speed) and 1 output (change in speed).

```
80      ‘ Declare variables
100     INTEGER SPDERR, SPDRATE, OUT1, OUT2, ST, DRIVE, FUZLOOP, FUZ-
        DAT(106)
110     INTEGER SETPNT, SPD, LASTSPD

115     ‘ Midpoint and sensitivity data
120     DATA 0,65
140     DATA 0,100
160     DATA 0,320
170     DATA 0,0

175     ‘ Rules for output 1
180     DATA 7,7,7,7,6,5,4
200     DATA 7,6,6,6,5,4,3
220     DATA 6,6,5,5,4,3,2
240     DATA 5,5,5,4,3,3,3
260     DATA 6,5,4,3,3,2,2
280     DATA 5,4,3,2,2,2,1
300     DATA 4,3,2,1,1,1,1

310     ‘ Rules for output 2
320     DATA 0,0,0,0,0,0,0
340     DATA 0,0,0,0,0,0,0
360     DATA 0,0,0,0,0,0,0
380     DATA 0,0,0,0,0,0,0
400     DATA 0,0,0,0,0,0,0
420     DATA 0,0,0,0,0,0,0
```

```
440     DATA 0,0,0,0,0,0,0

450     ‘ Initialize values to zero
460     SPDERR = 0: SPDRATE = 0: LASTSPD = 0: SETPNT = 0: DRIVE = 0

490     ‘ Read in the fuzzy data into the fuzzy array
500     FOR FUZLOOP = 0 TO 105
520     READ FUZDAT(FUZLOOP)
540     NEXT FUZLOOP

590     ‘ Setup task 1 to run 10 times a second
600     RUN 1,10

990     ‘ Infinite loop for the main loop
1000    GOTO 1000

3900    ‘ Task 1 to read setpoint, speed, calculate input 1 and 2, reserve last
3925    ‘ speed, use the FUZZY statement, use return value and add to drive, 3950       ‘
        then output the drive.
4000    TASK 1
4020    SETPNT = ADC(1)
4040    SPD = ADC(2)
4060    SPDERR = SPD - SETPNT
4080    SPDRATE = SPD - LASTSPD
4100    LASTSPD = SPD
4120    FUZZY ADR(FUZDAT(0)),SPDERR,SPDRATE,OUT1,OUT2,ST
4140    DRIVE = OUT1 + DRIVE
4150    IF DRIVE < 0 THEN DRIVE = 0
4175    IF DRIVE > 16000 THEN DRIVE = 16000
4160    DAC 1,DRIVE
4180    EXIT
```

The example code above is for a speed control problem. Lines 80 - 110 are for variable declarations. Note that all variables related to the FUZZY statement are integers. Lines 115 - 170 contain the data for the midpoints and sensitivities for both inputs and outputs. Lines 175 - 440 contains the rules for output 1 and output 2. Note that although output 2 is not being used in the system the data is still required (all 0's). Lines 490 - 540 demonstrates the easiest way to get the fuzzy data into the data array. Typically this type of setup will make the program easier to read and to modify. Lines 590 and 600 setup the fuzzy control task to break the infinite loop of the main task 10 times per second. Lines 3900 - 4180 contains the fuzzy control task. The first step is to acquire the setpoint which is read in from analog channel 1 using the ADC statement. Next, the current speed is read from analog channel 2. Input 1 (speed error) and input 2 (rate of change of speed) are then calculated for use as inputs to the controller. The current speed is then stored as the last speed for use during the next execution of the task on line 4100. Finally, the FUZZY statement is utilized with the array address, inputs 1 and 2, outputs 1 and 2, and the return status. Output 1 is the change in the speed and thus the drive signal is added to the output to obtain the next drive signal (line 4140) which is then limited to ensure valid data. Lastly, the drive signal is output to the motor through D/A channel 1.

GET Function

can be used with:

Boss 32

UMC

Summary:
The GET function returns one character from the current file.

Syntax:
x = GET

Arguments:
GET needs no arguments.

Description:
The GET function reads data from the current file; it waits for the next available character and returns it as an INTEGER value without performing any conversions or filtering. Specifically, unlike INPUT and INPUT$, which treat the carriage return as a delimiter, GET will return the carriage return using it's ASCII value, 13 ($0D). GET waits for the next character to become available. In a multi-tasking program, if there isn't a character available immediately, GET will allow another task to execute, minimizing the processor overhead while waiting; when a character becomes available, GET will return it the next time that the task gets to execute.

GET should not be used in an interrupt task, because it re-enables interrupts and allows multitasking to continue. Depending upon what the other tasks are doing at the instant that the interrupt task executes, GET could cause the system to lock up.

Example:

```
100 ' Program to get keys from the console and the keypad.
110 INTEGER J, K
120 FILE 0: GOSUB 200          ' Read keys from the console.
130 FILE 6: GOSUB 200          ' Read keys from the keypad.
130 STOP
200 ' Subroutine to get 5 characters from the current file and
210 ' echo them back.
220 FOR J = 1 TO 5
230  K=GET
240  PRINT K,CHR$(K)
250 NEXT J
260 RETURN
```

can be used with:

Boss 32

UMC

GETDATE Statement

Summary:
The GETDATE statement retrieves the current date from the Real Time Clock.

Syntax:
GETDATE month, day, year, wday

Arguments:

month	an integer variable. The month is stored in this variable. The months are represented as 1-Jan, 2-Feb, ..., 12-Dec.
day	an integer variable. The day of the month is stored in this variable. The days are represented as 1..31.
year	an integer variable. The year is stored in this variable. The years are represented as 0..99.
wday	an integer variable. The day of the week is stored in this variable. The days are represented as 1-Sunday, 2-Monday, ..., 7-Saturday.

Description:
GETDATE reads the current date from the Real Time Clock. GETDATE takes about 200 microseconds to execute.

Example:

```
100 INTEGER MONTH, DAY, YEAR, WDAY
110 STRING WD$(22)
120 WD$=”SunMonTueWedThuFriSat”
130 GETDATE MONTH, DAY, YEAR, WDAY
140 PRINT “The date is “; MID$(WD$, WDAY*3-2, 3); “ “;
150 PRINT MONTH; “/”; DAY; “/”; YEAR
```

This produces the following output when run:

```
The date is Wed 10/24/90
```

can be used with:

Boss 32

UMC

GETIME Statement

Summary:

The GETIME statement retrieves the current time from the Real Time Clock.

Syntax:

GETIME hours, minutes, seconds

Arguments:

hours	an integer variable. The hours are stored in this variable. The hours are rep resented as 0..23, with 0 being midnight.
minutes	an integer variable. The minutes are stored in this variable. The minutes are represented as 0..59.
seconds	an integer variable. The seconds are stored in this variable. The seconds are represented as 0..59.

Description:

GETIME reads the current time from the Real Time Clock. GETIME takes about 200 microseconds to execute.

Example:

```
100 INTEGER HOUR, MIN, SEC
110 GETIME HOUR, MIN, SEC
120 PRINT “The time is “; HOUR; “:”; MIN; “:”; SEC
```

This produces the following output when run:

```
The time is 17:25:56
```

can be used with:

Boss 32

UMC

GO Direct Command

Summary:
The GO direct command starts the compiled code executing.

Syntax:
GO (may be abbreviated G)

Arguments:
GO needs no arguments.

Description:
GO begins execution of the most recently COMPILEd or RUN program. If the source code has been modified, a NEW command has been executed, or an error was detected in the last compile, then the No Compiled Code error will be displayed. GO is typically used to run a program previously compiled via the RUN or COMPILE commands. If the program is very long, then GO is faster than using RUN repeatedly, since RUN must first recompile the program.

.

can be used with:

Boss 32

UMC

GOSUB Statement

Summary:
The GOSUB statement is used to call a subroutine.

Syntax:
GOSUB linenum

Arguments:
linenum a line number in the BASIC program or line label if in no line number mode.

Description:
It is often useful in a program to be able to execute a section of code from many places in the program. Instead of using duplicate code in each spot that needs it, a subroutine may be created and then called with GOSUB. When a GOSUB is encountered, program execution continues at *linenum*, until a RETURN statement is executed, at which point BASIC will transfer execution to the statement following the GOSUB. Subroutines have three main advantages: they make the program smaller by eliminating redundant code, they allow the logic to be debugged once and used many places, and they can make the program easier to understand by hiding the details of the lower levels of the program.

Example:

```
100 REAL SLOPE(1), OFFSET(1), ADJST(1)
110 INTEGER CHANL
120 SLOPE(0)=0.432: OFFSET(0)=11.5        ‘ Set up coefficients for chan 0
130 SLOPE(1)=1.28: OFFSET(1)=-5.29        ‘ Set up for chan 1
140 ADJST(0)=ADC(2)                       ‘ Read value for chan 0
150 ADJST(1)=ADC(5)                       ‘ Read value for chan 1
160 FOR CHANL=0 TO 1
170  GOSUB 1000                           ‘ Perform adjustment
180  PRINT “Adjusted value = “; ADJST(CHANL)
190 NEXT CHANL
200 STOP
1000 ‘ Subroutine to perform a Y=MX+B adjustment.
1010 ADJST(CHANL)=SLOPE(CHANL)*ADJST(CHANL)+OFFSET(CHANL)
1020 RETURN
```

This produces the following output when run:

```
Adjusted value = 1241.83600
Adjusted value = 465.75000
```

can be used with:

Boss 32

UMC

GOTO Statement

Summary:
The GOTO statement transfers execution to a specific program line.

Syntax:
GOTO linenum

Arguments:
linenum a line number in the BASIC program or line label if in no line number mode.

Description:
GOTO *linenum* causes execution of the BASIC program to continue at line *linenum*.

Example:
```
100 PRINT “Line 100”
110 GOTO 200
120 PRINT “Line 120”
130 STOP
200 PRINT “Line 200”
210 GOTO 120
```

This produces the following output when run:

```
Line 100
Line 200
Line 120
```

HELP Direct Command

can be used with:

Boss 32

UMC

Summary:
The HELP direct command displays on-line help screens.

Syntax:
HELP [*command*]

Arguments:

command	This optional command will display just the help text for the command given.

Description:
HELP causes a set of on-line help screens to be displayed. The space bar must be pressed at the end of each screen to cause the next screen to be displayed; any other key will abort the help command and return to the compiler prompt. If the optional argument is given then help searches for the command given and display just the text for that command.

.

can be used with:

Boss 32

UMC

HEXVAL Function

Summary:
The HEXVAL function converts a hexidecimal string into a number.

Syntax:
x=HEXVAL (strexpr$)

Arguments:

strexpr$	a hexadecimal string expression

Description:
The HEXVAL function returns an integer result that is the numeric conversion of the number at the beginning of *strexpr$*. If *strexpr$* has a leading $ it will be ignored. If *strexpr$* can't be interpreted as a number, the HEXVAL returns 0. Note that this is much different than CVI or CVS, which convert a binary string into a number. Binary strings are not human readable, and, in fact, they are just arrays of bytes.

Example #1 (With line Numbers):

```
100 INTEGER J: INTEGER X
110 STRING A$(60), B$(60)
120 A$=”123” : B$=”ABC”
130 J=HEXVAL(A$) : X=HEXVAL(B$)
140 PRINT “NUMERIC VALUE OF $”;A$;”is “;J
150 PRINT “NUMERIC VALUE OF $”;B$;”is “;X
```

This produces the following output when run:

```
Numeric value of $123 is 291
Numeric value of $ABC is 2748
```

can be used with:

Boss 32

UMC

IF..THEN Statement

Summary:
The IF..THEN statement controls program flow based on a relational expression.

Syntax:
IF rel_expr THEN statement [: statement]...[:ELSE: statement] [: statement]...

Arguments:

rel_expr	a numeric or string expression. This is evaluated as a TRUE or FALSE value, with nonzero values being TRUE and zero being FALSE.
statement	BASIC statement to be executed if *rel_expr* is TRUE.

Description:
In general, the power of the computer is based upon its ability to make decisions based on its input data. The IF..THEN statement is used in Divelbiss 32-bit BASIC to control the flow of program execution based on the result of a calculation. *rel_expr* is evaluated; if it is TRUE (nonzero), then *statement* is executed. If *rel_expr* is FALSE (zero), then program execution continues at the next line. Usually, *rel_expr* will be a comparison between variables or expressions; for example: J=5, K>=N+3, or A$<>B$. However, since it is being evaluated as a TRUE or FALSE value, *rel_expr* may just be a variable by itself; for example, the statement IF J THEN... will be executed if J is nonzero. For integer and real expressions, the relational operators are: AND, OR, >=, <=, =, <>, >, and <. For string expressions, the relational operators are: =, <>, >, and <.

Note that if *rel_expr* is TRUE, then the rest of the line is executed. This means that multiple statements can be placed after THEN; the statements will only be executed if the expression is TRUE.

Often, *statement* will be a GOTO statement; for example: IF X>3.5 THEN GOTO 260. The compiler accepts a shortened form of this, in which the word GOTO is left out; ie. IF X>3.5 THEN 260. The compiler will insert the GOTO into the statement, so when the line is listed it will read IF X > 3.5 THEN GOTO 260

Example #1 (With line Numbers):

```
100 INTEGER J,K,N
110 J=3: K=5: N=0
120 PRINT “J=3: “; J=3; “ J>3: “; J>3
130 IF J=3 THEN PRINT “J = 3”;: PRINT “...”;
140 PRINT
150 IF J>3 THEN PRINT “J > 3”;: PRINT “...”;
160 PRINT
170 IF J<5 AND K>2 THEN PRINT “Passed line 170”
180 IF K THEN PRINT “K is TRUE”
190 IF N THEN PRINT “N is TRUE”
```

.

This produces the following output when run:

```
J=3: 257 J>3: 0
J = 3...
Passed line 170
K is TRUE
```

Example #2 (With line numbers):

```
        Interger
START:  J=0
        If J <=5 Then
                Print “J= “; J
                J=J+1
        Else
                Print “J= “; J
                Goto End
        End If
        Goto Start
END:    Stop
```

This produces the following output when run:

```
1
2
3
4
5
6
7
```

INPUT Statement

can be used with:

Boss 32

UMC

Summary:

The INPUT statement waits for a data value to be entered on the current FILE device.

Syntax:

INPUT variable [,variable]...

Arguments:

variable a BASIC variable name. The value entered is stored in this variable.

Description:

The INPUT statement is used to enter data while the program is running. It waits for a data value, followed by a carriage return, to be entered on the current FILE, then it stores that value into a BASIC variable. The BACKSPACE key may be used to modify the input data before pressing ENTER. In a multitasking program, other tasks will continue executing while this task is waiting for input; multiple tasks can even execute INPUT at the same time on different FILEs. INPUT is particularly well suited to reading data that is being sent over one of the serial ports; if data is to be entered by the user, then FINPUT will probably work better.

If multiple *variable*s are specified, then multiple data values can be entered separated by commas. For example, INPUT J,K,X will accept 4,12,3.76 as a valid input. Note, however, that it does not verify that the correct number of values was entered; with the previous example, if 10,8 were entered, then X would retain its original value, since no new value was entered. Likewise, if 3,4,10.32,8 were entered, then the 8 would be thrown away, since there is no variable for it to be assigned to; J would be set to 3, K to 4, and X to 10.32, and no error would be generated.

If *variable* is a string, then commas and control characters cannot be read by INPUT. Commas are used as delimiters between input data values; if necessary, the INPUT$ statement allows commas and most control characters to be entered.
If *variable* is an integer or real and a string is entered, then the error message *** Bad Input: Please Re-enter *** will be displayed and it will wait for another value to be entered. Note that this could make a mess out of the screen display, if the input is being typed by a user; for this reason, it is probably better to get user input using FINPUT. If multiple numeric variables are to be input, then this error will also be displayed if any spaces are entered.

INPUT should not be used in an interrupt task, because it re-enables interrupts and allows multitasking to continue. Depending upon what the other tasks are doing at the instant that the interrupt task executes, INPUT could cause the system to lock up.

Examples:

```
100 STRING A$
110 PRINT “What is your name? “;
120 INPUT A$
130 PRINT “Hello, “; A$
```

This example shows how text can be INPUT into a string variable. Run this and try using BACKSPACE to modify the string. Also try entering commas in the string.

```
100 INTEGER J,K
110 REAL X
120 PRINT “Enter J, K, X > “;
130 INPUT J,K,X
140 PRINT J,K,X
```

This example shows how multiple items can be entered with one INPUT statement. Run this and try entering more or less than three numbers.

```
100 INTEGER J,K
110 RUN 1
120 INPUT J
130 PRINT “The value entered is “; J
140 STOP
150 TASK 1
160 FILE 6
170 INPUT K
180 PRINT “K = “; K
190 EXIT
```

This example shows how INPUT can be used concurrently in multiple tasks.

INPUT$ Statement

can be used with:

Boss 32

UMC

Summary:
The INPUT$ statement is identical to INPUT, except that it allows commas and some control characters to be entered in strings.

Syntax:
INPUT$ variable [,variable]...

Arguments:
variable a BASIC variable name. The value entered is stored in this variable.

Description:
The INPUT$ statement is identical to INPUT, except in its handling of strings. Whereas INPUT won't accept commas and control characters in strings, INPUT$ accepts commas and most control characters. The only delimiter used by INPUT$ is the carriage return, which signals the end of data entry. For example, INPUT$ COLOR$ would accept red, green, blue and leave COLOR$ holding the string "red, green, blue".
Note that there should only be one string *variable* field, and that no integer or real *variable*s can follow the string, since there is no way for it to detect the end of the string except for the carriage return. For example, with the code INPUT$ J,A$,K there is no way to enter a value into K, because, if the user types 12,widget7,9 then J will be 12, A$ will be "widget7,9", and K will be unchanged.

INPUT$ should not be used in an interrupt task, because it re-enables interrupts and allows multitasking to continue. Depending upon what the other tasks are doing at the instant that the interrupt task executes, INPUT$ could cause the system to lock up.

Examples:

```
100 INTEGER J,K
110 STRING A$(80)
120 INPUT$ J,K,A$
130 PRINT J,K,A$
```

INTEGER Statement

can be used with:

Boss 32

UMC

Summary:
The INTEGER statement is used to declare integer variables.

Syntax:
INTEGER variable [,variable]...

Arguments:

variable	a text string. The name to use for the variable. Variable names in Bear BASIC consist of an alphabetic character followed by up to 7 alphanumeric characters. To declare an array, this will be followed by one or two numbers enclosed in parenthesis. The following are valid integer variable names: J, J(20), PRESSURE, CHAN5, TIME4(10), J(10,7).

Description:
Integer variables are used to hold numeric data that ranges between -2147483648 and 2147483647. Integers are stored in 32 bit, two's complement form, and therefore take up 4 bytes each. All variables in a Bear BASIC program must be declared as INTEGER, REAL, or STRING; all variable declarations must occur before any executable statements in the BASIC program.

Integer arrays may be declared with the INTEGER statement, as well. Divelbiss 32-bit BASIC allows one and two dimensional arrays. The variable name is followed by parenthesis enclosing one or two array dimensions, such as J(5) or J(7,5). Note that the array dimension is not the number of array elements, but the number of the last array element; arrays always start with element 0. The array J(5) actually contains 6 variables, J(0) through J(5).

Example:

```
100 INTEGER J,K
110 INTEGER CHAN(9)                 ‘ Array of 10 integers: 20 bytes
120 INTEGER TDAT(4,19)              ‘ 5 by 20 array: 200 bytes
130 FOR J=0 TO 9
140  CHAN(J)=J                      ‘ Shows how arrays are accessed
150  TDAT(0,J)=J*4
160 NEXT J
```

can be used with:

Boss 32

UMC

INTERRUPT Statement

Summary:

The INTERRUPT statement attaches a task to a hardware interrupt source, so that the task becomes the handler for that interrupt.

Syntax:

INTERRUPT *device*, *chan*, *task*

Arguments:

device	an integer expression. The type of device to work with: 1 = High speed counter 2 = Fatal ERROR 3 = Timer 0 4 = Future expansion
chan	an integer expression. The channel number of the device, from 1 up to the number of channels available.
task	an integer expression between 1 and 31. The number of the task to be attached to the device.

Description:

The INTERRUPT statement provides a simple way to write interrupt service routines in Divelbiss 32-bit BASIC. It causes a task to be executed when a specific hardware interrupt occurs. For example, INTERRUPT 1,5,3 will cause task 3 to be executed when high speed counter number 5 causes an interrupt. When using the INTERRUPT statement for Timer 0 or a Fatal ERROR, the *chan* argument can be set to anything since the value is not needed for these types interrupts.

Example:

```
100 ‘ Program to demonstrate INTERRUPT using the high speed counter
110 INTEGER J, RELOAD, COUNT
120 CNTRMODE 1,4                    ‘ A count, B direction
130 WRCNTR 1,0,0                    ‘ Set count to 0
140 RELOAD=5: COUNT=0
150 WRCNTR 1,1,RELOAD               ‘ Set counter interrupt value
160 INTERRUPT 1,1,1                 ‘ Set counter interrupt to task 1
170 J=0
180 FILE 6
200 ‘ Main program loop.
210 LOCATE 1,1
220 FPRINT “U5X5U5Z”, J, COUNT
230 J=J+1                           ‘ Increment junk variable
240 GOTO 210
300 ‘ Counter interrupt handler task.
310 TASK 1
320 RDCNTR 1,0,COUNT                ‘ Get new count
330 RELOAD=RELOAD+5                 ‘ Set up new reload
```

```
340 WRCNTR 1,1,RELOAD                ‘ Set counter interrupt value
350 EXIT
```

This example sets up task 1 as the interrupt handler for counter 1. In lines 120 and 130 it initializes counter 1, setting its mode to A-count, B-direction and writing a 0 to the counter. In lines 140 and 150 it sets the counter reload variable to 5 and writes this reload value into the counter’s compare register. Line 160 uses INTERRUPT to set task 1 as the interrupt handler for the counter. Lines 210 to 240 form the program’s main loop, which just displays the latest counter value (COUNT) from the last interrupt; it also increments and displays J, just to cause some action on the display. Line 300 to 350 form task 1, the interrupt handler; it reads the current counter value and updates the reload value.

In task 1, the first thing that it does is read the current counter value. At low pulse rates, this will be the same as RELOAD, since it is reading the same value that caused the interrupt. As the pulse rate increases, however, the counter will increment before the task 1 gets to read the counter. At a very high pulse rate, a problem will occur when the counter has already gone beyond the new RELOAD value before RELOAD is written to the counter (ie. COUNT=783 and RELOAD=780); the counter will need to wrap completely around before the interrupt will occur again.

INTOFF Statement

can be used with:

Boss 32

UMC

Summary:
The INTOFF statement disables all processor interrupts.

Syntax:
INTOFF

Arguments:
INTOFF needs no arguments.

Description:
It is sometimes necessary to temporarily stop the processor from handling hardware interrupts, for two reasons: to stop an interrupt handler from modifying a variable that another routine needs to access, and because a section of code needs to execute extremely quickly and interrupt processing would slow it down. INTOFF stops the processor from handling interrupts; INTON enables the interrupt processing again. Interrupts should only be turned off for very short periods; if they are off for too long, then characters coming in on the serial ports could be lost, the timing of tasks with WAIT statements could slow down, or hardware events could be handled erratically.

The Divelbiss 32-bit Controller uses a hardware timer interrupt to run the multitasking context switcher, so executing INTOFF stops the multitasking operation. This can be used to keep one task from corrupting a variable used in another task, for example. Some instructions enable interrupt processing as part of their execution: INTON, WAIT, EXIT, INPUT, INPUT$, GET, FINPUT, PRINT, FPRINT, NETWORK 1, NETWORK 2, and any string operations. These instructions should not be used after an INTOFF, since they will defeat the purpose by re-enabling interrupts.

If interrupts are left off for too long (more than 0.5 second) then the watchdog may reset the system. Some BASIC statements reset the watchdog, and so this will not happen if those statements are executed.

Example:

```
100 INTEGER D,J,K,N
110 RUN 1,1
120 FOR J=1 TO 1000
130  INTOFF
140  K=2
150  N=J*K/2
160  D=J+K-N
170  INTON
180  FPRINT “U5Z”,D
190 NEXT J
200 PRINT: STOP
210 ‘ Task to demonstrate variable corruption
220 TASK 1
```

```
230 K=9999
240 EXIT
```

This example shows that a task can corrupt a variable that another task is using. In this case, task 1 modifies the value of K, which the main routine is using in lines 140 to 160. Without the INTOFF in line 130, the context switcher could start task 1 running at any time, including while the main routine is in line 140, 150 or 160, thus changing the value of K from 2 to 9999. This would drastically affect the outcome of the calculation. To demonstrate this, run the program as it is shown; it should print “2” 1000 times. Then remove lines 130 and 170 and run the program again; this time it will print mostly “2”, with other numbers interspersed periodically. The other numbers are caused when task 1 executes while task 0 (the main routine) is in line 140, 150, or 160.

NOTE: With resource locking and function: If a function (F1) was started before INTOFF, and a different function (F2) from a different task is called after the INTOFF, the first function (F1) finishes before the second function (F2) starts, even with the INTOFF. This is because of the resource locking of the function’s temporary variables.

can be used with:

Boss 32

UMC

INTON Statement

Summary:
The INTON statement enables processor interrupts.

Syntax:
INTON

Arguments:
INTON needs no arguments.

Description:
INTON is used to re-enable interrupts after an INTOFF statement has disabled interrupt processing. See INTOFF for a detailed description.

Example:

```
100 INTEGER J
110 RUN 1,1
120 INTOFF: DOUT 1,1: DOUT 2,1: INTON
130 WAIT 10
140 INTOFF: DOUT 1,0: DOUT 2,0: INTON
150 WAIT 5: GOTO 120
200 TASK 1
210 PRINT “*”;
220 EXIT
```

This example uses INTOFF and INTON to ensure that the two DOUT statements take place at almost the same time. If the interrupts weren’t disabled, then task 1 could break in between the two DOUT statements, causing them to occur a few milliseconds apart.

KEY Function

can be used with:

Boss 32

UMC

Summary:
The KEY function reads one byte from the current FILE; it doesn't wait if no byte is available.

Syntax:
x = KEY

Arguments:
KEY needs no arguments.

Description:
KEY returns a byte from the current FILE if one is available; the byte is returned as an integer value. 0 is returned if no byte is available; if the byte's value is 0, then it is returned as 256 ($100). KEY returns all bytes entered, no control codes are filtered out.

Since KEY goes through the normal FILE routines, it provides an autorepeat function when accessing the onboard keypad (FILE 6). Sometimes, it is desirable to access the keypad directly, as if it were an input module. The DIN function can be used to get the current keypad state, without performing the autorepeat. See the DIN description for more information.

KEY should not be used in an interrupt task, because it re-enables interrupts and allows multitasking to continue. Depending upon what the other tasks are doing at the instant that the interrupt task executes, KEY could cause the system to lock up.

Example:

```
100 INTEGER N,K: N=0
110 K=KEY                              ‘ Get a key, if one is available
120 FILE 6: LOCATE 1,1: PRINT N        ‘ Display counter on onboard display
130 FILE 0: N=N+1
140 IF K=0 THEN 110                    ‘ If no key then continue waiting
150 IF K=256 THEN K=0                  ‘ Check for 0 byte value
160 PRINT “Key = “; K: GOTO 110
```

This example gets characters from the console and displays the value of the characters; it also displays a counter on the onboard display, to show that KEY is not waiting. Line 110 gets the byte from the console. Line 120 prints the counter on the onboard display; note that it changes to FILE 6, prints the counter, then changes back to FILE 0. Line 140 loops back if no byte was retrieved. Line 150 handles the 0 value, which would have been returned as 256; in this example, it is unlikely that a 0 would have been entered over the console port.

LEN Function

can be used with:

Boss 32

UMC

Summary:
The LEN function returns the length of a string.

Syntax:
x = LEN (st$)

Arguments:

st$	a string expression.

Description:
LEN returns the number of characters currently contained in *st$* as an integer value. It does not return the maximum size of *st$*; for example, if a string was declared as STRING NAME$(40), it will not return 40 unless there is a 40 character string stored in NAME$.

Example:

```
100 STRING A$(80)
110 PRINT “Enter a string > “;
120 INPUT A$
130 PRINT “String length = “; LEN(A$)
```

LFDELAY Direct Command

can be used with:

Boss 32

UMC

Summary:
The LFDELAY direct command causes a slight delay to be performed with each carriage return, line feed that is sent to the console.

Syntax:
LFDELAY

Arguments:
LFDELAY needs no arguments.

Description:
Some video terminals cannot scroll the screen fast enough to keep up at 9600 baud; they will lose the first few characters at the beginning of each line. The LFDELAY command causes the Divelbiss 32-bit Controller to pause at the beginning of each line to allow the terminal to catch up. Once the LFDELAY command is issued, it remains in effect until the Divelbiss 32-bit Controller is reset by executing the BYE command or turning the power off and back on.

can be used with:

Boss 32

LINE Statement

Summary:
The LINE statement draws a line, box, or filled box on the Graphics Vacuum Fluorescent Display (GVFD) .

Syntax:
LINE (*x1*, *y1*) - (*x2*, *y2*) [, B [F]]

Arguments:

x1	a numeric expression. The horizontal position to start the line at.
y1	a numeric expression. The vertical position to start the line at.
x2	a numeric expression. The horizontal position to end the line at.
y2	a numeric expression. The vertical position to end the line at.
B	an optional B tells the line statement to draw a box instead of a line.
BF	an optional BF tells the line statement to draw a filled box instead of a line.

Description:
LINE is one of the statements that can be used to draw graphics on the GVFD. This statement only works with the GVFD. The (x,y) coordinate system puts (0,0) at the lower left corner.

Example:

```
100 ‘Example to display a box the GVFD
110  FILE 6,0                    ‘ set the graphics screen to be ORed with the text
                                   screen
120 CLS 2                        ‘ clear both text and graphics
130 LINE (0,0)-(255,63),B        ‘ draw a box around the whole screen
140 LOCATE 2,2
140 PRINT “New graphics”
```

This produces the following output when run:

```
new graphics
```

LIST Direct Command

can be used with:

Boss 32

UMC

Summary:
The LIST direct command displays the current BASIC program.

Syntax:
LIST [*start*] [, *end*]

Arguments:

start	a valid BASIC line number. The first line number to display.
end	a valid BASIC line number. The last line number to display.

Description:
LIST displays the program currently in memory on the system's console terminal. If LIST is typed with no arguments, or just L is typed, then the entire program will be listed. If LIST is typed and only *start* is specified, then only that line will be listed (if it exists). If both *start* and *end* are specified, then the program from *start* through *end* will be listed.

Examples:

LIST	Lists the entire program.
L	Lists the entire program.
LIST 190	Lists line 190, if it exists.
LIST 190,240	Lists lines 190 through 240.

LOAD Direct Command

can be used with:

Boss 32

UMC

Summary:

The LOAD direct command loads a saved BASIC source program or compiled code that is stored on the FLASH EPROM and stores it in memory.

Syntax:

LOAD [CODE] [filename] [filenum]

Arguments:

CODE	If supplied loads compiled code
filename	The name of the file
filenum	The number of the file shown by DIR

Description:

The LOAD direct command loads the saved BASIC source progam from the FLASH EPROM into memory.
LOAD CODE loads the save BASIC compiled code from the FLASH EPROM into memory.
Once the source program is loaded it can be compiled and ran or modified. Once compiled code is loaded it can be ran with GO.

.

Example:

```
LOAD filename

LOAD CODE 1
```

LOCATE Statement

can be used with:

Boss 32

UMC

Summary:

The LOCATE statement positions the cursor on the current FILE output device.

Syntax:

LOCATE *row*, *column*

Arguments:

row	a numeric expression. The vertical position to put the cursor at, starting at 1.
column	a numeric expression. The horizontal position to put the cursor at, starting at 1.

Description:

LOCATE is used to position the cursor on the current FILE output device, using a row/column coordinate system with (1,1) at the upper left corner. Each task maintains its own cursor position, so that two tasks may print to a device at different locations at the same time. LOCATE works with the onboard display (FILE 6), as well as with COM1 (FILE 0) and COM2 (FILE 5); in order to work with FILE 0 and FILE 5, the terminal must be able to emulate the terminal type enabled.

Example:

```
100 'Example to display a sine wave on the terminal
110 REAL X,Y: INTEGER J
120 CLS
130 FOR J=0 TO 79                        ' Use entire display
140  X=J * 360.0 / 79.0                  ' Scale to 360 degrees
150  Y=SIN(X) * 8.0 + 9.0                ' Scale to +/- 8, with 0 in center
160  LOCATE Y+1,J+1: PRINT "*";          ' Draw the point
170 NEXT J
```

This produces the following output when run:

```
                                                                **************
                                                             **                ***
                                                          ***                     **
                                                         *                           **
                                                      **                               **
                                                    **                                    *
                                                  *                                         **
                                                **                                             *
                                              **                                                 *
**                                         **
  **                                      *
     *                                 **
      **                             **
         **                        *
           **                   ***
             ***              **
                ****************
```

can be used with:

Boss 32

UMC

LOCK Statement

Summary:
The LOCK statement locks and unlocks user definable resources.

Syntax:
LOCK *mode, num*

Arguments:

mode	an integer value of 0 or 1. 1 = lock, 0 = unlock.
num	an integer value of 0 to 31. This value is the resource number.

Description:
LOCK is used to lock and unlock resources from other tasks. When a lock statement is run the Divelbiss 32-bit Controller checks to see if the resources trying to be locked is available (not locked by another task). If the resource is available then it is marked locked and program execution is returned to the task that issued the lock. If the resource is locked then the task issuing the lock is put in a waiting que and the task that has the resource locked is restarted. This mechanism allows one task to run a resource at a time with next in line queuing. Resources are user definable by a number from 0 to 31. Resources that a programmer may want to lock: files, variables, A/D, D/A, digital I/O. Any device, variable or subroutine may be assigned a resource number to lock them.

Example:

```
100 'Example to lock and unlock file 0
110 LOCK 1,10                        ' resource 10 defined to be file 0
120 CLS : RUN 1,50
130 PRINT "TASK 0 has control of FILE 0"
140 LOCK 0,10                        ' unlock resource 10
150 WAIT 100
160 STOP
200 TASK 1
210 LOCK 1,10                        ' resource 10 or file 0
220 PRINT "TASK 1 has control of FILE 0"
230 LOCK 0,10                        ' unlock resource 10
240 EXIT
```

This produces the following output when run:

```
TASK 0 has control of FILE 0
TASK 1 has control of FILE 0
TASK 1 has control of FILE 0
```

LOG Function

can be used with:

Boss 32

UMC

Summary:
The LOG function calculates the natural logarithm (base e) function.

Syntax:
x = LOG (*expr*)

Arguments:

expr	a numeric expression.

Description:
The LOG function returns the natural logarithm of *expr*, which must be a numeric expression. The result is returned as a REAL value.

By using the properties of logarithms, a number X can be raised to a power Y (XY) using EXP(LOG(X)*Y).

To calculate the common logarithm (LOG_{10}) of X, use LOG10.

Example:

```
100 REAL X,Y
110 PRINT LOG(72.35)
120 X=0.82
130 Y=LOG(X)
140 PRINT “Logarithm of “;X;” is “;Y
150 PRINT “4 cubed = “; EXP(LOG(4)*3)
```

This produces the following output when run:

```
4.28151
Logarithm of .82000 is -.19845
4 cubed = 64.00003
```

can be used with:

Boss 32

UMC

LOG10 Function

Summary:
The LOG10 function calculates the common logarithm (base 10) function.

Syntax:
x = LOG10 (*expr*)

Arguments:
expr a numeric expression.

Description:
The LOG10 function returns the common logarithm of *expr*, which must be a numeric expression. The result is returned as a REAL value.

To calculate the natural logarithm of X, use LOG.

Example:

```
100 REAL X,Y
110 PRINT LOG10(72.35)
120 X=0.82
130 Y=LOG10(X)
140 PRINT “Common logarithm of “;X;” is “;Y
```

This produces the following output when run:

```
1.85944
Common logarithm of .82000 is -.08619
```

can be used with:

Boss 32

UMC

MID$ Function

Summary:
The MID$ function returns a substring of a specified string.

Syntax:
st$ = MID$ (strexpr$, start, num)

Arguments:

strexpr$	a string expression.
start	a numeric expression. The start of the substring.
num	a numeric expression. The number of characters in the substring.

Description:
MID$ returns a string that is a substring of *strexpr$*. The substring will begin at *start* in *strexpr$* and continue for *num* characters. MID$ must not go beyond the end of *strexpr$*; if *start* + *num* is greater then LEN(*strexpr$*), then a String Length Exceeded error will result.

Example:

```
100 STRING A$(60)
110 A$=”The quick brown fox jumped over the lazy dog”
120 PRINT “<”;MID$(A$,1,3);”>”
130 PRINT “<”;MID$(A$,20,10);”>”
140 PRINT “<”;MID$(A$,38,LEN(A$)-37);”>”
150 PRINT “<”;CONCAT$(MID$(A$,5,6),MID$(A$,17,3));”>”
```

This produces the following output when run:

```
<The>
< jumped ov>
<azy dog>
<quick fox>
```

In line 140, the LEN function is used to ensure that MID$ doesn’t try to go beyond the end of A$, which would cause an error.

MKI$ Function

can be used with:

Boss 32

UMC

Summary:
The MKI$ function converts an integer to a binary string.

Syntax:
st$ = MKI$ (intexpr)

Arguments:
intexpr an integer expression.

Description:
A binary string is a string that may contain more than just ASCII text data; it may contain bytes that are not valid ASCII characters. Because of this, they are not usually human-readable. A binary string is really just an array of numeric bytes. The main use for binary strings is to provide a more efficient way to transfer blocks of numeric data over the serial ports; numbers stored in binary strings take up fewer bytes than numbers stored as ASCII characters. The functions MKI$, CVI, MKS$, and CVS are provided to work with binary strings.

The MKI$ function returns *intexpr* as a four byte string. For example, MKI$($12345678) returns a string whose first byte is $78, second byte is $56, third byte is $34, and whose fourth byte is $12; note that $12345678 takes up 8 characters when stored in ASCII, but only 4 when stored in binary form.

Example:

```
100 INTEGER J
110 STRING A$: A$=""
120 FOR J=$30303030 TO $30303036
130  A$=CONCAT$(A$,MKI$(J))
140 NEXT J
150 PRINT A$
160 FOR J=1 to LEN(A$)
170  FPRINT "H2X2Z",ASC(MID$(A$,J,1))
180 NEXT J
190 PRINT
200 J=CVI(A$): FPRINT "H8",J
```

This produces the following output when run:

```
0000100020003000400050006000
30  30  30  30  31  30  30  30  32  30  30  30  33  30  30  30  34  30
  30  30  35  30  30  30  36  30  30  30
30303030
```

The first line printed shows how the 7 integers $30303030 through $30303036 were stored in A$; these numbers were chosen so that A$ would still be a printable string. The second line shows how the individual bytes are stored in the string. The third line shows that CVI can be used to convert the first two bytes of A$ back to an integer.

can be used with:

Boss 32

UMC

MKS$ Function

Summary:
The MKS$ function converts a real to a binary string.

Syntax:
st$ = MKS$ (realexpr)

Arguments:
realexpr	a real expression.

Description:
The MKS$ function returns *realexpr* as a four byte string. For example, MKS$(123.456) returns a string whose bytes are $42, $F6, $E9, $78; note that 123.456 takes up 7 characters when stored in ASCII, but only 4 when stored in binary form. See MKI$ for more information on binary strings.

Example:

```
100 INTEGER J: REAL X
110 STRING A$: A$=""
120 FOR X=1.0 TO 3.0
130  A$=CONCAT$(A$,MKS$(X))
140 NEXT X
150 FOR J=1 to LEN(A$)
160  FPRINT "H2X2Z",ASC(MID$(A$,J,1))
170 NEXT J
180 PRINT
190 X=CVS(A$): PRINT X
```

This produces the following output when run:

```
3F   80   00   00   40   00   00   00   40   40   00   00
1.00000
```

The first line printed shows how the 3 reals 1.0, 2.0, and 3.0 were stored in A$. The second line shows that CVS can be used to convert the first four bytes of A$ back to a real.

MODFUNCTION Function

can be used with:

Boss 32

UMC

Summary:
The MODFUNCTION function returns the last MODBUS function performed by the Divelbiss 32-bit Controller.

Syntax:
code = MODFUNCTION

Description:
The MODFUNCTION function returns the last MODBUS function performed by the Divelbiss 32-bit Controller on the network. If there was no function performed since the last time MODFUNCTION was run then the function returns a zero. It also sets the first register number to be read by the MODREGISTER function.

Examples:

```
100 INTEGER FUNC
110 NETWORK 10, 3, 1, 1, 1
120 FUNC = MODFUNCTION              ‘ Get the last FUNCTION performed
130 IF FUNC = 0 THEN GOTO 120       ‘ Loop until a FUNCTION is performed
140 PRINT “THE LAST MODBUS FUNCTION DONE WAS “;FUNC
```

MODREGISTER Function

can be used with:

Boss 32

UMC

Summary:
The MODREGISTER function returns the first register number of the MODBUS last MODBUS function performed by the Divelbiss 32-bit Controller.

Syntax:
code = MODREGISTER

Description:
The MODREGISTER function returns the first register number of the last MODBUS function performed by the Divelbiss 32-bit Controller on the network. The register number is set by MODFUNCTON, so MODFUNCTION must be run first to get the register number.

Examples:

```
100 INTEGER FUNC
110 NETWORK 10, 3, 1, 1, 1
120 FUNC = MODFUNCTION              ‘ Get the last FUNCTION performed
130 IF FUNC = 0 THEN GOTO 120       ‘ Loop until a FUNCTION is performed
140 PRINT “THE LAST MODBUS FUNCTION DONE WAS “;FUNC
150 PRINT “AND THE FIRST REGISTER ACCESSED WAS “;MODREGISTER
```

NETMSG Function

can be used with:

Boss 32

UMC

Summary:

The NETMSG function sends and receives messages over the network. It returns an error code.

Syntax:

ercode = NETMSG (message$,rdwr,unit)

Arguments:

message$	if *rdwr* is 0, then *message$* is a string expression; this string will be sent to the specified unit. If *rdwr* is 1, then *message$* is a string variable; if a mes sage is available in the receive buffer, then it will be returned in *message$*.
rdwr	an integer expression that evaluates to 0 or 1. 0 indicates that the message is to be sent to another unit, while 1 indicates that a message should be read from the network buffers, if one is waiting.
unit	if *rdwr* is 0, then *unit* is the network address of the unit to send the message to, and should be an integer expression that evaluates between 0 and 255. If *rdwr* is 1, then *unit* is an integer variable that will be set to the network address of the unit that sent the message.

Description:

The NETMSG function performs two different, but related, functions: it sends messages over the network, and retrieves messages that have arrived over the network. In both cases, it returns an integer error code that indicates the success or failure of the operation.

To send a message to another unit, the message is assembled into a string, then the string is sent with NETMSG(*message$*,*0*,*unit*), which will wait until a response is received or until it times out (in approximately 5 seconds). It returns a 0 if the message was successfully transferred, or a 1 if a timeout error occurred.

Any incoming messages are received and buffered by the low level network handler. NETMSG(*message$*,*1*,*unit*) will return a 0 if no message is waiting in the buffer. It will return a 1 if there is a message; the message will be returned in *message$*, and the number of the unit that sent the message will be returned in *unit*.

Examples:

```
100 INTEGER J,K
110 STRING NM$(127)
200 ‘ Initialization
210 NETWORK 0,1,1,1,1,J
220 IF J<>0 THEN PRINT “Network initialization failed”: STOP
300 ‘ Send a message to unit 2
310 NM$=CHR$(0)                        ‘ Message type = 0
320 NM$=CONCAT$(NM$,MKS$(20.0)) ‘ Send the value 20.0
330 J=NETMSG(NM$,0,2)                  ‘ Send the message to unit 2
```

```
340 IF J<>0 THEN PRINT "Network send error": STOP
400 ' Now wait for a response
410 J=NETMSG(NM$,1,K)
420 IF J=0 THEN GOTO 410            ' Loop if no response yet
430 IF K<>2 THEN GOTO 410           ' Loop if not from unit #2
440 PRINT "Response = ";CVS(MID$(NM$,2,4))
```

The program above is run on one Divelbiss 32-bit Controller on the network. It initializes the Divelbiss 32-bit Controller to be unit number 1, sends a message that consists of the number 20.0, and then waits for a response to arrive. The following program is run on another Divelbiss 32-bit Controller on the network to generate the response to the program above.

```
100 INTEGER J,K
110 STRING IN$(127)
120 REAL X
200 ' Initialization
210 NETWORK 0,2,1,1,1,J
220 IF J<>0 THEN PRINT "Network initialization failed": STOP
300 ' Wait for a message from unit 1
310 J=NETMSG(IM$,1,K)
320 IF J=0 THEN GOTO 310            ' Loop if no message yet
330 IF K<>1 THEN GOTO 310           ' Loop if not from unit #1
400 ' Generate the response message
410 X=SQR(CVS(MID$(IM$,2,4)))       ' Send back the square root
420 IM$=CHR$(0)                     ' Message type = 0
430 IM$=CONCAT$(IM$,MKS$(X))        ' Put the square root in the string
440 J=NETMSG(IM$,0,1)               ' Send the message to unit 1
450 IF J<>0 THEN PRINT "Network send error": STOP
460 GOTO 300                        ' Do it all again
```

This program initializes the Divelbiss 32-bit Controller to unit number 2, waits for a message from unit number 1, calculates the square root of the number in the message, and sends a message that consists of the square root.

can be used with:

Boss 32

UMC

NETSERVER Statement

Summary:
The NETSERVER statement starts the Divelbiss 32-bit Controller as a peer to peer network server.

Syntax:
NETSERVER units, autotime, debug, year, month, day, hour, minute

Arguments:

units	an integer value of 1 to 31. This indicates the number of networked units on the line.
autotime	an integer value of 0 or 1. This turns on or off the autotime feature. 0 for OFF, 1 for ON.
debug	an integer value of 0 or 1. This turns on or off the debug feature. 0 for OFF, 1 for ON.
year	an integer expression. This value specifies the year. This is used with the autotime feature.
month	an integer expression. This value specifies the month. This is used with the auto time feature.
day	an integer expression. This value specifies the day. This is used with the autotime feature.
hour	an integer expression. This value specifies the hour. This is used with the autotime feature.
minute	an integer expression. This value specifies the minute. This is used with the autotime feature.

Description:
The NETSERVER statement is used to set up a Divelbiss 32-bit Controller as a peer to peer network server. This statement should only be used in a dedicated task for serving the network because once this statement is run it will not return control to basic. It must be noted that this command can only be used on one unit in the network and must not be used when a computer with a network card is on the network.

Example:

```
100 INTEGER YEAR,MONTH,DAY,WDAY,HOUR,MINUTE,SECOND
110 GETDATE MONTH,DAY,YEAR,WDAY
120 GETTIME HOUR,MINUTE,SECOND
130 PRINT “STARTING NETWORK FOR TWO UNITS”
140 ‘Start the network with autotime enabled
150 NETSERVER 2,1,0,YEAR,MONTH,DAY,HOUR,MINUTE
```

NETWORK 0 Statement

can be used with:

Boss 32

UMC

Summary:
The NETWORK 0 statement initializes the network handler on the Divelbiss 32-bit Controller.

Syntax:
NETWORK 0, unit_id, num_int, num_real, num_string, status

Arguments:

unit_id	an integer expression that evaluates between 1 and 254. This is the unit number of this Divelbiss 32-bit Controller in the network. Each unit in the network must have a unique number.
num_int	an integer expression. This is the number of INTEGER registers to allocate space for.
num_real	an integer expression. This is the number of REAL registers to allocate space for.
num_string	an integer expression. This is the number of STRING registers to allocate space for.
status	an integer variable. This will hold the status of the NETWORK operation; a
0 will	indicate successful completion, while a nonzero value will indicate that an error occurred.

Description:
This sets the unit ID for this Divelbiss 32-bit Controller and allocates space for the network registers. Each unit on the network must have a unique unit ID; if two units have the same ID, then neither of them will be able to access the network reliably. The unit ID may range between 1 and 254; for maximum network efficiency, the unit numbers should be allocated sequentially from 1 to N. If there are any unused unit numbers, then the network master will waste time trying to talk to those units periodically.

The network master is always unit number 0. The network master uses a significant amount of processor time processing the network operations, so it should be a unit that is lightly loaded. In a network that includes a personal computer with a Divelbiss Network Interface Card, the PC should be the network master, so that the other units on the network maintain a higher processing efficiency. In this case, none of the Divelbiss 8-bit Controllers will be set to unit 0. In a network that consists entirely of Divelbiss 8-bit Controllers, one Divelbiss 8-bit Controller is set to unit number 0, which causes that Divelbiss 8-bit Controller to become the network master. This Divelbiss 32-bit Controller will continue to operate as usual, except that it will spend a large percentage of its processing power to operate the network.

BASIC accesses the network using a set of network registers; these are just values that are shared between the BASIC program and the network. There are three sets of network registers: one each for INTEGERs, REALs, and STRINGs. The BASIC program controls how many registers are allocated of each type. To maintain compatibility with the Divelbiss 8-bit Controller network, each INTEGER register uses 2 bytes, each REAL uses 4 bytes, and each STRING uses 41 bytes. The total space allocated for all three register sets must be less than the variable space left in BASIC.

The network operates through the COM4 serial interface, so FILE 5 (which uses COM4) should not be accessed after the network is initialized. This could slow the network down and possibly corrupt data transfer on the network.

Example:

```
100 INTEGER ST
110 ‘ Set up this unit’s network handler to be unit #3, with
112 ‘ 200 integer registers, 50 real registers, and 10 string registers.
120 NETWORK 0, 3, 200, 50, 10, ST
130 IF ST<>0 THEN PRINT “Network initialization failed”
```

NETWORK 1 Statement

can be used with:

Boss 32

UMC

Summary:

The NETWORK 1 statement sends a block of network registers to another unit on the network.

Syntax:

NETWORK 1, vartype, source_reg, number, unit_id, dest_reg, status

Arguments:

vartype	an integer value of 0, 1, or 2. This indicates the type of network register to send: 0 for INTEGER, 1 for REAL, and 2 for STRING.
source_reg	an integer expression. This is the starting register number of the register block to send.
number	an integer expression. This is the number of registers in the block to send.
unit_id	an integer expression that evaluates between 1 and 254. This is the unit number of the Divelbiss 32-bit Controller to send to. Each unit in the network must have a unique number.
dest_reg	an integer expression. This is the register number to store the register block at in the destination unit.
status	an integer variable. This will hold the status of the NETWORK operation; a 0 will indicate successful completion, while a nonzero value will indicate that an error occurred.

Description:

The NETWORK 1 statement is used to transfer data to another unit on the network. It sends a block of network registers from this Divelbiss 32-bit Controller to a register block in another unit. The next time that the network master polls this unit, the registers are transferred; when the destination unit receives the registers, it will acknowledge the reception, and this statement will return. It will wait for at least five seconds for a response before timing out and returning an error; in a multitasking program, it may take longer than five seconds to time out, perhaps as much as 30 seconds or more.

It is often useful to send a register to a different register number in the destination unit. The *source_reg* and *dest_reg* fields determine which registers are sent and where they are stored in the destination unit. For example, NETWORK 1,0,12,2,3,36,ST will send INTEGER registers 12 and 13 to unit 3, where they will be stored in INTEGER registers 36 and 37. Of course, a register can be sent to the same register number; for example, NETWORK 1,1,7,1,2,7,ST will send REAL register number 7 to unit 2, where it will be stored in register 7.

No range checking is performed. If an attempt is made to read from a register that does not exist, then invalid data will be transferred. If an attempt is made to store into a register that does not exist, then other registers will be overwritten and unpredictable operation may occur.

This statement may not be used concurrently in a multitasking program; in other words, it can't be called from two tasks at the same time. It is highly recommended that all NETWORK 1 and NETWORK 2 statements be located in the same task.

.

Example:

```
100 INTEGER J, K, ST
110 NETWORK 0,1,20,30,0,ST            ‘ Initialize the network
120 IF ST THEN PRINT “Init Error”
130 FOR J=0 TO 19
140  NETWORK 3,0,J,J+100,ST           ‘ Store 100..119 in registers 0..19
150 NEXT J
160 NETWORK 1,0,0,20,2,0,ST           ‘ Send integer registers to unit 2
170 IF ST THEN PRINT “Timeout”
180 NETWORK 1,0,18,1,3,28,ST          ‘ Send reg 18 to unit 3, reg 28
190 IF ST THEN PRINT “Timeout”
```

In line 110, the network driver is initialized to unit 1 with 20 INTEGER registers, 30 REAL registers, and 0 STRING registers. In lines 130 to 150, the 20 integer registers (0 to 19) are filled with the values 100 to 119. In line 160, all 20 integer registers are sent to unit 2, where they are stored in the same register locations. In line 180, integer register 18 is sent to unit 3, where it is stored in register 28. Note that in line 160 we are assuming that unit 2 has at least 20 integer registers are available, and in line 180 we are assuming that unit 3 has at least 28 integer registers available.

NETWORK 2 Statement

can be used with:

Boss 32

UMC

Summary:
The NETWORK 2 statement reads a block of network registers from another unit on the network.

Syntax:
NETWORK 2, vartype, dest_reg, number, unit_id, source_reg, status

Arguments:

vartype	an integer value of 0, 1, or 2. This indicates the type of network register to read: 0 for INTEGER, 1 for REAL, and 2 for STRING.
dest_reg	an integer expression. This is the first register number to store into in the destination unit.
number	an integer expression. This is the number of registers in the block to read.
unit_id	an integer expression that evaluates between 1 and 254. This is the unit number of the Divelbiss 32-bit Controller to read from. Each unit in the network must have a unique number.
source_reg	an integer expression. This is the first register number to read from the source unit.
status	an integer variable. This will hold the status of the NETWORK operation; a 0 will indicate successful completion, while a nonzero value will indicate that an error occurred.

Description:
The NETWORK 2 statement is used to transfer data from another unit on the network. It reads a block of network registers from the source Divelbiss 32-bit Controller to a register block in this unit. The next time that the network master polls this unit, a register request is transferred; when the destination unit receives the request, it will return the register data, and this statement will return. It will wait for at least five seconds for a response before timing out and returning an error; in a multitasking program, it may take longer than five seconds to time out, perhaps as much as 30 seconds or more.

It is often useful to read a register from a different register number in the source unit. The *dest_reg* and *source_reg* fields determine which registers are read and where they are stored in this unit. For example, NETWORK 2,0,12,2,3,36,ST will read INTEGER registers 36 and 37 from unit 3 and store them in INTEGER registers 12 and 13 in this Divelbiss 32-bit Controller. Of course, a register can be read from the same register number; for example, NETWORK 2,1,7,1,2,7,ST will read REAL register number 7 from unit 2 and store it in REAL register 7 in this Divelbiss 32-bit Controller.

No range checking is performed. If an attempt is made to read from a register that does not exist, then invalid data will be transferred. If an attempt is made to store into a register that does not exist, then other registers will be overwritten and unpredictable operation may occur.

This statement may not be used concurrently in a multitasking program; in other words, it can't be called from two tasks at the same time. It is highly recommended that all NETWORK 1 and NETWORK 2 statements be located in the same task.

.

Example:

```
100 INTEGER J, K, ST
110 NETWORK 0,1,20,30,0,ST           ‘ Initialize the network
120 IF ST THEN PRINT “Init Error”
130 FOR J=0 TO 19
140  NETWORK 3,0,J,0,ST              ‘ Zero out registers 0..19
150 NEXT J
160 NETWORK 2,0,0,20,2,0,ST          ‘ Read integer registers from unit 2
170 IF ST THEN PRINT “Timeout”
180 FOR J=0 TO 19
190  NETWORK 4,0,J,K,ST              ‘ K = register J value
200  PRINT J,K
210 NEXT J
220 NETWORK 2,0,18,1,3,28,ST         ‘ Read reg 28 from unit 3 into reg 18
230 IF ST THEN PRINT “Timeout”
240 NETWORK 4,0,18,K,ST
250 PRINT “Register 18 = “; K
```

In line 110, the network driver is initialized to unit 1 with 20 INTEGER registers, 30 REAL registers, and 0 STRING registers. In lines 130 to 150, the 20 integer registers (0 to 19) are filled with 0. In line 160, all 20 integer registers are read from the same registers in unit 2. Lines 180 to 210 print out the 20 values just read. In line 180, integer register 18 is read from unit 3, register 28; this value is printed in lines 230 and 240. Note that in line 160 we are assuming that unit 2 has at least 20 integer registers are available, and in line 180 we are assuming that unit 3 has at least 28 integer registers available.

NETWORK 3 Statement

can be used with:

Boss 32

UMC

Summary:

The NETWORK 3 statement writes data into a network register.

Syntax:

NETWORK 3, vartype, regnum, value, status

Arguments:

vartype	an integer value of 0, 1, or 2. This indicates the type of network register to read: 0 for INTEGER, 1 for REAL, and 2 for STRING.
regnum	an integer expression. This is the register number to be written to.
value	an numeric or string expression. This is the value to be written to *regnum*.
status	an integer variable. This will hold the status of the NETWORK operation; a 0 will indicate successful completion, while a nonzero value will indicate that an error occurred.

Description:

The network registers are stored in an area that is normally inaccessible to the BASIC program as variable space. The NETWORK 3 statement writes data into a network register. This does not cause the data to be sent over the network, however. Data is only transferred when this unit performs a NETWORK 1 statement, another Divelbiss 32-bit Controller performs a NETWORK 2 statement, or the network master reads or writes data in the register. There are three sets of network registers, INTEGERs, REALs, and STRINGs; the number of each of these is set up with the NETWORK 0 statement. Each group of registers is numbered starting at 0; for example if there is a NETWORK 0,1,25,15,5,ST statement, then there are 25 INTEGER registers (numbered 0 to 24), 15 REAL registers (0 to 14), and 5 STRING registers (0 to 4). If an attempt is made to write outside of these ranges, then an error will be returned; for example, NETWORK 3,0,30,99,ST would return an error (ST will be nonzero) because only 25 INTEGER registers were declared above.

Example:

```
100 INTEGER J, K, ST
110 NETWORK 0,1,20,30,0,ST          ‘ Initialize the network
120 IF ST THEN PRINT “Init Error”
130 FOR J=0 TO 19
140  NETWORK 3,0,J,J+100,ST         ‘ Store 100..119 in registers 0..19
150 NEXT J
```

NETWORK 4 Statement

can be used with:

Boss 32

UMC

Summary:
The NETWORK 4 statement reads data from a network register.

Syntax:
NETWORK 4, vartype, regnum, variable, status

Arguments:

vartype	an integer value of 0, 1, or 2. This indicates the type of network register to read: 0 for INTEGER, 1 for REAL, and 2 for STRING.
regnum	an integer expression. This is the register number to be read from.
variable	a numeric or string variable. The value read from *regnum* will be stored in this vari able.
status	an integer variable. This will hold the status of the NETWORK operation; a 0 will indicate successful completion, while a nonzero value will indicate that an error occurred.

Description:
The network registers are stored in an area that is normally inaccessible to the BASIC program as variable space. The NETWORK 4 statement reads data from a network register into a BASIC variable. There are three sets of network registers, INTEGERs, REALs, and STRINGs; the number of each of these is set up with the NETWORK 0 statement. Each group of registers is numbered starting at 0; for example if there is a NETWORK 0,1,25,15,5,ST statement, then there are 25 INTEGER registers (numbered 0 to 24), 15 REAL registers (0 to 14), and 5 STRING registers (0 to 4). If an attempt is made to read outside of these ranges, then an error will be returned; for example, NETWORK 4,0,30,J,ST would return an error (ST will be nonzero) because only 25 INTEGER registers were declared above.

Example:

```
100 INTEGER J, K, ST
110 NETWORK 0,1,20,30,0,ST            ‘ Initialize the network
120 IF ST THEN PRINT “Init Error”
130 FOR J=0 TO 19
140  NETWORK 4,0,J,K,ST               ‘ Read registers 0..19
150  PRINT J,K
160 NEXT J
```

NETWORK 10 Statement

can be used with:

Boss 32

UMC

Summary:

The NETWORK 10 statement initializes the network handler for MODBUS on the Divelbiss 32-bit Controller.

Syntax:

NETWORK 10, unit_id, comm_mode, file #, time_out

Arguments:

unit_id	an integer expression that evaluates between 0 and 247. This is the unit number of this Divelbiss 32-bit Controller in the network. Each unit in the network must have a unique number. If *unit_id* is 0 then the Divelbiss 32-bit Controller will be in MODBUS master mode.
comm_mode	an integer expression. This number determines the communication mode and how the serial port is used. 0=ASCII RS232, 1=RTU RS232, 2=ASCII RS485, 3=RTU RS485.
file #	an integer expression. This number sets the file for MODBUS operation. File 0 and 1 will only operate in RS232 mode, file 4 and 5 can be RS232 or RS485.
time_out	an integer expression. This number is only used if the *unit_id* is 0. This is the time out value, in mS, for the master pulling slaves.

Description:

This sets the unit ID and MODBUS mode for the Divelbiss 32-bit Controller. It also allocates about 19.53K of space for 10,000 two byte integer network registers. Each unit on the network must have a unique unit ID; if two units have the same ID, then neither of them will be able to access the network reliably. The unit ID may range between 0 and 247; for maximum network efficiency, the unit numbers should be allocated sequentially from 1 to N. ID 0 is used only for making the Divelbiss 32-bit Controller operate as a MODBUS master.

The network master is always unit number 0. The network master uses a significant amount of processor time processing the network operations, so it should be a unit that is lightly loaded. In a network that includes a personal computer with a Divelbiss Network Interface Card, the PC should be the network master, so that the other units on the network maintain a higher processing efficiency. In this case, none of the Divelbiss 8-bit Controllers will be set to unit 0. In a network that consists entirely of Divelbiss 8-bit Controllers, one of the Divelbiss 8-bit Controllers is set to unit number 0, which causes that Divelbiss 8-bit Controller to become the network master. This Divelbiss 8-bit Controller will continue to operate as usual, except that it will spend a large percentage of its processing power to operate the network.

BASIC accesses the network using a set of network registers; these are just values that are shared between the BASIC program and the network. There are 10,000 integer registers allocated for the MODBUS network. The network registers are read by BASIC using a NETWORK 4 statement and are written to by using a NETWORK 3 statement. The registers are predefined for the MODBUS network as follows:

1-4500	Input Registers. These are read on the network by a FUNCTION 4 com mand.
4501-9000	Holding Registers. These are accessed on the network by a FUNCTION 3, 6, or 16 commands.
9001-9256	Coil Registers. These are written to on the network by a FUNCTION 5 or 15 command.
9257	Exception Register. This register is read on the network by a FUNCTION 7 command.
9501-9700	Master Transmit scratch pad. These registers are used only if the Boss 32's ID is set to 0 (master mode). These registers would be used for a FUNC TION 15 or 16.
9701-9900	Master Receive scratch pad. These registers are used only if the Boss 32's ID is set to 0 (master mode). These registers would be used for a FUNC TION 1, 2, 3, or 4.

The Divelbiss 32-bit Controller supports MODBUS functions 1, 2, 3, 4, 5, 6, 7, 15 and 16.

● Function 1	Read Coil Status
● Function 2	Read Input Status
● Function 3	Read Holding Registers
● Function 4	Read Input Registers
● Function 5	Force Single Coil
● Function 6	Preset Single Register
● Function 7	Read Exception Status
● Function 15	Force Multiple Coils
● Function 16	Preset Multiple Registers

The network operates through the file number given in the NETWORK 10 statement. This file must be set to the correct baud rate, data configuration with a FILE statement, before the NETWORK 10 statement is used. The file used for the network should not be accessed after the network is initialized. This could slow the network down and possibly corrupt data transfer on the network.

Example:

```
100 FILE 1,9600,"E",8,2' Set up the comport for 9600 Even 8 data 2 stop
110 ' Set up this unit's network handler to be unit #3, using file 1 in RTU mode
120 NETWORK 10, 3, 1, 1, 1
```

can be used with:

Boss 32

UMC

NETWORK 11 Statement

Summary:
The NETWORK 11 statement is used to access other MODBUS units from the Divelbiss 32-bit Controller.

Syntax:
NETWORK 11, unit_id, function, address, data, status

Arguments:

unit_id	an integer expression that evaluates between 0 and 247. This is the unit number to access on the MODBUS network. Number 0 is for broadcast, and can only be used with writes.
function	an integer expression (1,2,3,4,5,6,7,15,16). This number is the MODBUS function to request on the network.
address	an integer expression. This number is the first register for the function requested.
data	an integer expression. Depending on the function being requested, it is either data or the number of registers to request.
status	an integer variable. This will hold the status of the NETWORK operation; a 0 will indicate successful completion, while a nonzero value will indicate that an error occurred.

Description:
This statement allows a Divelbiss 32-bit Controller, setup as a master, to read and write to other MODBUS units on the network. Data received is stored in registers 9701-9900. Data written to other MODBUS units must be stored in registers 9501-9700 before this statement is to be used. When data is receive or written it will start at register 9X01 and end with 9X01+*data.*

The network registers are read by BASIC using a NETWORK 4 statement and are written to by using a NETWORK 3 statement. The registers are predefined for the MODBUS network as flows:

1-4500	Input Registers. These are read on the network by a FUNCTION 4 com mand.
4501-9000	Holding Registers. These are accessed on the network by a FUNCTION 3, 6, or 16 commands.
9001-9256	Coil Registers. These are written to on the network by a FUNCTION 5 or 15 command.
9257	Exception Register. This register is read on the network by a FUNCTION 7 command.
9501-9700	Master Transmit scratch pad. These registers are used only if the Boss 32's ID is set to 0 (master mode). These registers would be used for a FUNC TION 15 or 16.
9701-9900	Master Receive scratch pad. These registers are used only if the Boss 32's ID is set to 0 (master mode). These registers would be used for a FUNC TION 1, 2, 3, or 4.

.

The Boss 32 supports MODBUS functions 1, 2, 3, 4, 5, 6, 7, 15 and 16.

● Function 1	Read Coil Status
● Function 2	Read Input Status
● Function 3	Read Holding Registers
● Function 4	Read Input Registers
● Function 5	Force Single Coil
● Function 6	Preset Single Register
● Function 7	Read Exception Status
● Function 15	Force Multiple Coils
● Function 16	Preset Multiple Registers

Status Values:

0 = Successful transmission
1 = Illegal Function
2 = Illegal address
3 = Illegal data
4 = Slave device failure
5 = Acknowledge (not supported)
6 = Slave device busy
7 = Negative Acknowledge (not supported)
8 = Memory parity error (not supported)
9 = Slave timeout error
10 = Slave packet error

Example:

```
100 INTEGER STATUS
110 ‘ Set up this unit’s network handler to be unit #0 master, using file 1 in RTU mode
120 NETWORK 10, 0, 1, 1, 1000
130 NETWORK 11, 1, 4, 1, 20, STATUS          ‘ Read the first 20 INPUT registers
                                             ‘ from UNIT 1 into 9701-9720
```

can be used with:

Boss 32

UMC

NETWORK 12 Statement

Summary:
The NETWORK 12 statement is used set the status of theDivelbiss 32-bit Controller on the MODBUS network.

Syntax:
NETWORK 12, status

Arguments:
status an integer value 0,1,2. This will hold the status of the Divelbiss 32-bit Controller.

Description:
This statement allows a Divelbiss 32-bit Controller, setup as a slave, to set its status as read on the MODBUS network. A status value or 0 indicates the unit is OK. A status value of 1 set the unit to busy, and a value of 2 sets the unit to failure. This status is to let the master know that the unit is busy and can't be accessed at this time or that there is a failure. When the NETWORK 10 statement is run the status is initialized to 0.

.

Example:

```
100 NETWORK 12, 1    ‘ Let the master know we are BUSY
```

NEW Direct Command

can be used with:

Boss 32

UMC

Summary:
The NEW direct command erases the program currently in memory.

Syntax:
NEW

Arguments:
NEW needs no arguments.

Description:
NEW erases the BASIC source code that is currently in memory. It enables runtime error checking, clears the duplicate line numbers flag, and erases the compiled code.

.

NEXT Statement

can be used with:

Boss 32

UMC

Summary:
The NEXT statement marks the end of a FOR...NEXT control structure.

Syntax:
NEXT

Arguments:

variable	an integer or real variable. This must match the variable referenced in the corresponding FOR statement.

Description:
See the FOR statement for a description of the FOR...NEXT control structure.

NOERR Direct Command

can be used with:

Boss 32

UMC

Summary:
The NOERR direct command disables the runtime error checking.

Syntax:
NOERR

Arguments:
NOERR needs no arguments

Description:
NOERR turns off much of the runtime error checking of Divelbiss 32-bit BASIC, resulting in a program that runs much faster, and takes less memory. To insure that a program doesn't run wild and destroy itself, Divelbiss 32-bit BASIC inserts quite a lot of error checking code into the compiled program. For example, tests are made for “SUBSCRIPT OUT OF RANGE”, “NEXT WITHOUT FOR”, etc. NOERR also removes the test for a CTRL-C. (CTRL-C aborts a running program, so a program compiled under NOERR can't be killed.) Most of these tests are removed by specifying NOERR. Some error checking routines remain, since in some cases the error checking code does not greatly effect execution speed. An increase in execution speed of several hundred percent can often be realized by using NOERR. The compiled program will also be significantly shorter. Be warned, though- NOERR permits the user's program to execute erroneously, possibly trashing Bear BASIC. Only fully debugged programs should be compiled under NOERR. When the compiler starts, all error checking is enabled and remains on until NOERR is specified. NOERR affects the way a program is compiled, so it must be specified before the program is COMPILEd or RUN.

NOLINE Direct Command

can be used with:

Boss 32

UMC

Summary:
The NOLINE direct command to set the Divelbiss 32-bit Controller into no line numbering mode.

Syntax:
NOLINE

Arguments:
NOLINE needs no arguments

Description:
NOLINE turns on the no line numbering mode of operation. In this mode of programming line labels are used instead of line numbers. This allows the programmer to put meaningful names to subroutines and jumps in the basic code. Only lines that are jumped to with GO, GOSUB, ON GO, ON GOSUB need labels, but any line can have a label. In no line numbering mode, block IF..THEN..ELSE statements are also enabled.

.

ON ERROR Statement

can be used with:

Boss 32

UMC

Summary:
The ON ERROR statement traps I/O errors.

Syntax:
ON ERROR *linenum*

Arguments:

linenum	a valid BASIC line number. This must be a line number that is in task 0 (ie. before the TASK 1 statement in the program).

Description:
ON ERROR causes the execution of the program to transfer to *linenum* when an I/O error is detected. ON ERROR is limited to operation in task 0. Both the function (or statement) that encounters the error and *linenum* must be before the TASK 1 statement.

Example:

```
100 INTEGER K
110 ON ERROR 900                      ‘ Set up error handler
120 K=GET                             ‘ Get a character from COM1
130 PRINT CHR$(K);                    ‘ Echo the character
140 WAIT 10: GOTO 120                 ‘ Do it again
900                                   ‘ Error handler
910 FILE 6                            ‘ Display error on LCD/VFD
920 PRINT “Error “;ERR;” detected”    ‘ Print error number
930 FILE 0: GOTO 120                  ‘ Back to COM1 and continue above
```

This example will get bytes from COM1 and echo them back on COM1 at a rate of 10 per second. If an I/O error occurs, then execution will transfer to line 900, where it will display the error number on the onboard display, then jump back to line 120.

can be used with:

Boss 32

UMC

ON GOSUB Statement

Summary:
The ON GOSUB statement calls one of a number of subroutines based on the value of an expression.

Syntax:
ON *expr*, GOSUB *linenum* [,*linenum*]...

Arguments:

expr	an integer expression between 1 and the number of line numbers specified in the statement.
linenum	a valid BASIC line number or line label if in no line number mode.

Description:
ON GOSUB causes the program to GOSUB to a line number based on the value of *expr*, which must evaluate to a number from 1 to the number of line numbers given in the statement. If *expr* evaluates to 1, then the program branches to the first line number. If it is 2, then the program branches to the second line number, and so on. If *expr* evaluates to a number larger or smaller than the number of line numbers given, then the error message Line Number Does Not Exist will appear.

Example:

```
100 INTEGER J
110 PRINT “Enter a number (1 to 5) > “;
120 FINPUT “U1”, J
130 PRINT
140 ON J, GOSUB 200,300,400,500,600
150 GOTO 110
200 PRINT “Line 200”: RETURN
300 PRINT “Line 300”: RETURN
400 PRINT “Line 400”: RETURN
500 PRINT “Line 500”: RETURN
600 PRINT “Line 600”: RETURN
```

This example calls a subroutine based on the number entered by the user in line 120. Note that there is no range checking on the value entered, so it is possible to enter a number greater than 5, which will cause an error to occur.

can be used with:

Boss 32

UMC

ON GOTO Statement

Summary:
The ON GOTO statement jumps to line number based on the value of an expression.

Syntax:
ON expr, GOTO linenum [,linenum]...

Arguments:

expr	an integer expression between 1 and the number of line numbers specified in the statement.
linenum	a valid BASIC line number or line label if in no line number mode.

Description:
ON GOTO causes the program to GOTO to a line number based on the value of *expr*, which must evaluate to a number from 1 to the number of line numbers given in the statement. If *expr* evaluates to 1, then the program branches to the first line number. If it is 2, then the program branches to the second line number, and so on. If *expr* evaluates to a number larger or smaller than the number of line numbers given, then the error message Line Number Does Not Exist will appear.

Example:

```
100 INTEGER J
110 PRINT “Enter a number (1 to 5) > “;
120 FINPUT “U1”, J
130 PRINT
140 ON J, GOTO 200,300,400,500,600
150 STOP
200 PRINT “Line 200”
300 PRINT “Line 300”
400 PRINT “Line 400”
500 PRINT “Line 500”
600 PRINT “Line 600”
```

This example jumps to a line based on the number entered by the user in line 120. Note that there is no range checking on the value entered, so it is possible to enter a number greater than 5, which will cause an error to occur.

can be used with:

Boss 32

UMC

PEEK Function

Summary:
The PEEK function reads a byte value from the specified address.

Syntax:
x = PEEK (*logaddr*)

Arguments:

logaddr	an integer expression. The logical address to read from.

Description:
The PEEK function returns an integer from 0 through 255 that is the byte value at memory location *logaddr.*

Example:

```
100 INTEGER K
110 POKE $A000,$12                    ‘ Write a $12 into $A000
120 PRINT “$A000=”;PEEK($A000)
130 K=500
140 FPRINT “H2”,PEEK(ADR(K))          ‘ Print the low byte of K
150 FPRINT “H2”,PEEK(ADR(K)+1)        ‘ Now print the high byte of K
```

This produces the following output when run:

```
$A000=18
F4
01
```

Line 120 prints $A000=18 because 18 is the decimal equivalent of $12. Line 140 reads the low byte of the variable K; K is 500 ($1F4), so this prints “F4”. Line 160 reads the high byte of K, which is 1.

Note: For PEEK to work correctly, Address must be within variable memory mapped area.

PID Function

can be used with:

Boss 32

UMC

Summary:
The PID function is utilized to perform the calculations required for PID closed-loop control.

Syntax:
x = pid (*error, id*)

Arguments:

x	real - pid calculation results
error	real - system error
id	integer - loop id

Description:
The PID function handles all of the necessary calculations and programming maintenance required to perform PID (proportional, integral, derivative) closed-loop control. The PID function must be preceded by the PIDSETUP function in order to perform correctly.

To utilize one, or multiple, PID loops the programmer needs to setup each loop individually using the PIDSETUP statement. Each loop must have a unique *id* number. Once complete, system error calculations need to be determined and then passed to the PID function along with the appropriate loop *id*. The function will then return the results of the PID calculation to be used for control. The following example will demonstrate the function's use.

Example:

```
100     REAL    setpnt1, setpnt2, syserr1, syserr2, feedbck1, feedbck2, out1, out2, drive1,
        drive2
110     INTEGER result
120     result = pidsetup(.03, 3.0, .001, .0, 10, 0)       ! Setup Loop 0
125     GOSUB 200                                           ! Check for setup error
130     result = pidsetup(.02, 2.5, .002, .0, 10, 1)       ! Setup Loop 1
140     GOSUB 200                                           ! Check for setup error
150     setpnt1 = 200.0: setpnt2 = 300                      ! Initialize setpoint values
160     drive1 = 0.0: drive2 = 0.0                          ! Initialize drive values
170     RUN 1,50                                            ! Start loop 1 running twice a sec
180     RUN 2,10                                            ! Start loop 2 running 10 times a
                                                            ! sec
190     WAIT 10: GOTO 170                                   ! Dummy main loop

200     IF result = 1 then GOTO 230                         ! Check for passed result
210     FILE 6: PRINT "PID Setup Failure";                  ! If fail then tell user
220     STOP                                                ! And stop
230     RETURN                                              ! Return

500     TASK 1                                              ! Task 1 statement
510     PRIORITY 1                                          ! Higher priority then main
520     feedbck1 = ADC(1)                                   ! Get system feedback value from
                                                            ! A/D 1
```

```
530     syserr1 = setpnt1 - feedbck1     ! Calculate system error
540     out1 = PID(syserr1, 0)           ! Get control value
550     drive1 = drive1 + out1           ! Calculate new drive
560     DAC 1,drive1                     ! Send out control signal with D/A 1
570     EXIT                             ! Exit task

500     TASK 2                           ! Task 1 statement
510     PRIORITY 2                       ! Higher priority then main and loop 1
520     feedbck2 = ADC(2)                ! Get system feedback value from A/D 2
530     syserr2 = setpnt2 - feedbck2     ! Calculate system error
540     out2 = PID(syserr2, 1)           ! Get control value
550     drive2 = drive2 + out2           ! Calculate new drive
560     DAC 2,drive2                     ! Send out control signal with D/A 2
570     EXIT                             ! Exit task
```

can be used with:

Boss 32

UMC

PIDSETUP Function

Summary:
The PIDSETUP function is utilized to setup a user-defined PID loop.

Syntax:
x = pidsetup *(kp, kd, ki, init, samples, id)*

Arguments:

x	integer - setup results (1= setup passed, 0 = setup failed)
kp	real - PID proportional term
kd	real - PID derivative term
ki	real - PID integration term
init	real - initial PID loop value
samples	integer - number of samples to be utilized in integration calculation
id	integer - loop id

Description:
The PIDSETUP function performs the operation of setting up a user-defined PID loop to be utilized for closed-loop control. The values *kp*, *kd*, and *ki* are the proportional, derivative, and integral loop coefficients standard in all PID loop calculations. The *samples* entry is the number of samples which will be included in the integration calculation. An integration is simply a sum over a certain period of time. The higher the *samples* term the larger the effect previous system setpoint errors will have on the current control value. The value of *samples* is limited between 0 and PID_MAX_SAMPLES. The *id* is a user defined ID between 0 and PID_MAX_LOOPS. The use of unique id's will allow a programmer to setup and utilize up to PID_MAX_LOOPS PID loops in a single program.

kp, *kd*, and *ki* are determined in one of two ways; prior knowledge of the system or by performing system tuning. The value of *samples* may normally be able to be fixed at 10 except for in some outstanding circumstances. As stated, *samples* causes previous system errors to have an effect on current control values. For high inertia systems (slow effect to control changes) the value of *samples* may be increased for better system control. On the other hand, for low inertia systems the integral term may need to be reduced to avoid system instability.

Example:

```
100     REAL            kp, kd, ki
110     INTEGER         samples, id, result
120     kp = .03: kd = 1.0: ki = .002
130     samples = 10: id = 0
140     result = pidsetup(kp, kd, ki, .0, samples, id)
150     IF result = 1 THEN PRINT “PID setup passed”;
160     IF result = 0 THEN PRINT “PID setup failed”;
```

can be used with:

Boss 32

UMC

POKE Statement

Summary:
The POKE statement writes a byte value to the specified address.

Syntax:
POKE logaddr, value

Arguments:
logaddr an integer expression. The logical address to write to.
value an integer expression between 0 and 255. The value to write.

Description:
The POKE statement writes a byte (*value*) into memory at *logaddr.*

Example:

```
100 INTEGER J,K
110 POKE $A000,$12                  ‘ Write a $12 into $A000
120 PRINT “$A000=”;PEEK($A000)
130 J=0: K=500
140 POKE ADR(K),J                   ‘ Write a $00 into low byte of K
150 PRINT K
160 POKE ADR(K)+1,J                 ‘ Now write to high byte of K
170 PRINT K
```

This produces the following output when run:

```
$A000=18
256
0
```

Line 120 prints $A000=18 because 18 is the decimal equivalent of $12. Line 140 stores a 0 into the low byte of the variable K; K was originally 500 ($1F4), so this changes it to 256 ($100). Line 160 stores a 0 into the high byte of K.

NOTE: For POKE to work correctly, Address must be within variable memory mapped area.

POLY Statement

can be used with:

Boss 32

.

Summary:

The POLY statement draws a multiple line object on the Graphics Vacuum Fluorescent Display (GVFD) .

Syntax:

POLY *n , (* *x*, *y*) , *r*

Arguments:

n	a integer value of 4, 8, 16, or 32. This is the number of sides that the object dis played will have.
x	a integer expression with a value of 0 to 255. The horizontal position to start the center of the object.
y	a integer expression with a value of 0 to 63. The vertical position to start the center of the object.
r	a integer expression with a value of 0 to 63. This is the radius of the object.

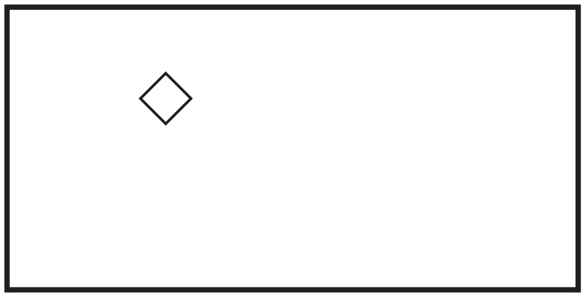
Description:

POLY is one of the statements that can be used to draw graphics on the GVFD. This statement only works with the GVFD. The (x,y) coordinate system puts (0,0) at the lower left corner.

Example:

```
100 ‘Example to display a box the GVFD
110     FILE 6,0                                  ‘ set the graphics screen to be Ored
                                                  ‘ with the text screen
120 CLS 2                                         ‘ clear both text and graphics
130 POLY 4,(50,30),5                              ‘ draw a diamond on the screen
```

This produces the following output when run:



can be used with:

Boss 32

UMC

PRINT Statement

Summary:
The PRINT statement sends output to the current FILE device.

Syntax:
PRINT *expr* [;*expr*]...[;]

Arguments:
expr a numeric or string expression.

Description:
PRINT outputs data in a human-readable form to the current FILE device, as set by the last FILE statement executed. The console (FILE 0, COM1) is the default FILE. Commas or semicolons may separate *expr* values. A comma causes *expr* to be printed at the next tab stop; tab stops are 16 characters wide. A semicolon causes *expr* to be printed without inserting any extra spaces; if the last thing on the PRINT line is a semicolon, then no carriage return or line feed will be printed. The question mark can be used as an abbreviation for PRINT when typing in a program; the '?' will be replaced with PRINT when the program is LISTed.

Integer numbers are printed using the minimum number of characters. Real numbers are printed with 5 digits to the right of the decimal point.

PRINT should not be used in an interrupt task, because it re-enables interrupts and allows multitasking to continue. Depending upon what the other tasks are doing at the instant that the interrupt task executes, PRINT could cause the system to lock up.

Example:

```
100 INTEGER J: REAL X: STRING A$
120 J=12: X=1.234
130 PRINT “Boss 32”
140 PRINT “Integrated “; “controller”
150 PRINT “J = “; J, “X = “; X
160 PRINT “Tan(X) = “;TAN(X)
170 A$=”ABCDEFGHIJK”
180 PRINT A$;” “;
190 PRINT MID$(A$,2,3)
```

Line 140 uses the semicolon (‘;’) to cause the two strings to be printed together; note that the first string ends with a space. Line 160 shows the use of PRINT with an expression; the expression is evaluated and the result is printed. Line 180 ends with a semicolon, which causes the following PRINT (line 190) to appear on the same line.

.

This produces the following output when run:

```
Boss 32
Integrated controller
J = 12    X = 1.23400
Tan(X) = .02154
ABCDEFGHIJK BCD
```

can be used with:

Boss 32

UMC

PRIORITY Statement

Summary:
The PRIORITY statement controls the order of execution of tasks in a multitasking program by setting a priority level for the current task.

Syntax:
PRIORITY pri_num

Arguments:

pri_num	an integer expression between 0 and 127. This is the priority level to assign to the current task.

Description:
The PRIORITY statement assigns a priority level to a task. It is used to alter the manner in which tasks are executed. It allows the user to assign a priority, or degree of importance, to a task. PRIORITY is useful only in multitasking programs. The PRIORITY statement must be issued from within the task whose priority level is to be altered.

PRIORITY takes one argument, which is the relative priority level for that task. The larger the number, the more important the task is. These numbers may range from 0 to 127. Therefore, 0 is least important, and 127 is most important. Note that Divelbiss 32-bit BASIC assumes all tasks run at priority level 0 if no PRIORITY statement is issued.

In a multitasking program with no PRIORITY statements, each task is executed in a “round robin” fashion. This means that upon receipt of a tic, Divelbiss 32-bit Basic stops running the current task and starts running the next sequential task, if it is ready to be run. PRIORITY can be used to alter this sequence.

Whenever a tic is received, Divelbiss 32-bit Basic examines the priority levels of all tasks that are ready to execute. The first highest priority task is then run. This means that if task 1 is level 2, task 2 is level 3, and task 3 is level 3, then task 2 will run until it completes. If all the tasks have the same priority level, than the lowest number task that is ready to run will continue to run until it relinquishes control (by executing EXIT, WAIT, INPUT, etc.). If the task with the highest priority level does not EXIT or WAIT, then it will use all of the computer time, effectively disabling multitasking, until it EXITs, goes into a WAIT, or reduces its priority level.

Divelbiss 32-bit Basic reevaluates the priorities on each tic, so tasks can dynamically change their level and the compiler will alter the scheduling as demanded.
Whenever a CANCEL statement is encountered, the priority level for that task is dropped to 0. This yields faster context switching.

Example:
```
100 RUN 1,10
110 RUN 2,20
120 GOTO 30
130 TASK 1
```

```
140 PRIORITY 3
150 PRINT ‘Task 1’
160 EXIT
170 TASK 2
180 PRINT ‘Task 2’
190 EXIT
```

This program shows task 1 issuing a PRIORITY 3. This means that if this task is ready to run, and if task 2 is also ready, then task 1 will execute first, and will continue to execute until it is complete. When it is done, task 2 will run, since task 1 has been told to wait 10 tics (via the RUN statement in line 10) before starting again.

PSET Statement

can be used with:

Boss 32

.

Summary:
The PSET statement draws a pixel on the Graphics Vacuum Fluorescent Display (GVFD) .

Syntax:
PSET (*x*, *y*) [, *s*]

Arguments:

x	a integer expression. The horizontal position to put the pixel.
y	a integer expression. The vertical position to put the pixel.
s	a optional integer value of 0 or 1. This indicates weather to turn on the pixel or turn it off. 0 = off , 1 = on , novalue = on

Description:
PSET is one of the statements that can be used to draw graphics on the GVFD. This statement only works with the GVFD. The (x,y) coordinate system puts (0,0) at the lower left corner.

RANDOMIZE Statement

can be used with:

Boss 32

UMC

Summary:
The RANDOMIZE statement reseeds the random number generator.

Syntax:
RANDOMIZE

Arguments:
RANDOMIZE needs no arguments.

Description:
The RND function generates pseudo-random numbers, which means that they are generated by a mathematical algorithm. This algorithm starts with a seed number; the first call to RND returns a number that is based on this seed. Each succeeding number is based on the previous number, and becomes the new seed. The RANDOMIZE statement changes the seed value by reading the processor refresh register, which provides a constantly changing value.

Example:

```
100 INTEGER J
110 RANDOMIZE                          ‘ Reseed the random number generator
120 FOR J=1 TO 20
130  FPRINT “U2X3U5”,J,RND             ‘ Print 20 random numbers
140 NEXT J
```

can be used with:

Boss 32

UMC

RDCNTR Statement

Summary:

The RDCNTR statement reads the count value from the high speed counter. It also can reset the counter's high speed output.

Syntax:

RDCNTR chan, flag, variable

Arguments:

chan	an integer expression between 1 and the number of counter channels installed. This is the counter channel to read.
flag	an integer expression from 0 through 3. This determines whether to read the current count or latched count, and whether or not to reset the high speed output: 0 to latch and read the current count. 1 to read only the last latched count value; this is used to read a value that was latched when the marker input was activated. 2 to latch and read the current count; also reset the high speed output. 3 to read only the last latched count value; also reset the high speed output. *variable* an integer variable.

.

Description:

The RDCNTR statement reads the one of the counter's registers, and optionally resets the high speed output. The counter hardware provides two 24 bit count registers: one holds the current count, and one holds the count value when the latch input was activated. In order to read the current count, the hardware first loads it into the latched count register.

The counter channels are assigned based on the hardware available on the Divelbiss 32-bit Controller; channel 1 could be the onboard counter, or in any of the expansion ports (if there is no onboard counter. The order of precedence for assigning counter channels is: onboard counter followed by J3 followed by J4 followed by J5.

The counter automatically wraps around when it reaches the 24 bit limit ($FFFFFF).

Examples:

```
100 INTEGER COUNTI
120 CNTRMODE 1,4                    ‘ A count, B direction
130 WRCNTR 1,0,65500.0              ‘ Set count to 65500
140                                 ‘ Main program loop
150 RDCNTR 1,0,COUNTI               ‘ Read current count as integer
170 PRINT COUNTI
180 WAIT 10: GOTO 150
```

This example demonstrates reading counter 1. It simply sits in a loop, reading and displaying the counter value.

READ Statement

can be used with:

Boss 32

UMC

Summary:
The READ statement loads variables with values stored in DATA statements.

Syntax:
READ variable [,variable]...

Arguments:
variable an integer, real, or string variable name. The next DATA item will be stored in this variable.

Description:
READ loads the variables in its argument list with values from DATA statements. As successive READs are executed, data is taken from each DATA statement in the program. All DATA statements must appear before the first READ in the program.

Example:

```
100 INTEGER J,K
110 REAL X
120 STRING A$
130 PRINT "DATA statement example"
140 ' Data table for program
150 DATA 4.77,2,3,4,5,23,83,8             ' Must be before first READ
160 DATA 999,3.14159,1.6667
170 DATA 893.664,999,"Data 1"
180 DATA "This is a test: Hi, everybody"
190 FOR J=1 TO 3
200  READ K: PRINT K                      ' Print first 3 elements
210 NEXT J
220 READ J
230 IF J<>999 THEN PRINT J: GOTO 220      ' Print elements until a 999
240 READ X
250 IF X<999.0 THEN PRINT X: GOTO 240     ' Print elements until a 999
260 READ A$: PRINT A$                     ' Print first string
270 READ A$: PRINT A$                     ' Print second string
280 READ X: PRINT X                       ' Wrap around, do first one again
```

This produces the following output when run:

```
DATA statement example
4
2
3
4
5
23
```

```
83
8
3.14159
1.66670
893.66381
Data 1
This is a test: Hi,
4.77000
```

The two 999 values in lines 160 and 170 are used as flags to indicate the end of portions of the table; this makes it easy to add to the DATA table without needing to change the program where the READ statements are executed. In line 270, it prints “This is a test: Hi, “ because the string variable A$ defaults to 20 character maximum length, so it truncates the rest of the string when it is read. Note that in line 280, it reads beyond the end of the DATA table, so it wraps around and reads the first value in the table.

REAL Statement

can be used with:

Boss 32

UMC

Summary:
The REAL statement is used to declare floating point variables.

Syntax:
REAL variable [,variable]...

Arguments:

variable	a text string. The name to use for the variable. Variable names in Bear BASIC consist of an alphabetic character followed by up to 6 alphanumeric characters. To declare an array, this will be followed by one or two numbers enclosed in parenthesis. The following are valid real variable names: J, J(20), PRESSURE, CHAN5, TIME4(10), J(10,7).

Description:
Real variables are used to hold numeric data that ranges between -1.7x10^{38} and 1.7x1038. Reals are stored in 32 bit IEEE single precision format, taking up 4 bytes. All variables in a Bear BASIC program must be declared as INTEGER, REAL, or STRING; all variable declarations must occur before any executable statements in the BASIC program.

Real arrays may be declared with the REAL statement, as well. Bear BASIC allows one and two dimensional arrays. The variable name is followed by parenthesis enclosing one or two array dimensions, such as J(5) or J(7,5). Note that the array dimension is not the number of array elements, but the number of the last array element; arrays always start with element 0. The array J(5) actually contains 6 variables, J(0) through J(5).

Approximately 6.5 digits of precision are maintained for real numbers. Many BASIC interpreters and compilers use BCD mathematics or 64 bit representations resulting in high accuracy numbers that require lots of memory. Divelbiss 32-bit BASIC does not support either of these in the interest of maximizing speed. The user must be aware that a real number may not be exactly the number anticipated. For example, since real numbers are constructed by using powers of 2, the value 0.1 cannot be exactly represented. It can be represented very closely (within 2**-23, or about 0.0000001), but it will not be exact. Therefore, it is very dangerous to perform a direct equality operation on a real number. The statement IF A=0.123 (assuming A is real) will only pass the test if the two values are exactly equal, a case which rarely occurs. This is true for all real relational operators, including, for example, the statement IF A>B, if values very close to the condition being measured are being used. Be aware that the number you expect may not be exactly represented by the compiler. If necessary, use a slight tolerance around variables with relational operators.

Example:

```
100 INTEGER J
110 REAL X
110 REAL CHAN(9)                  ‘ Array of 10 integers: 40 bytes
120 REAL TDAT(4,19)               ‘ 5 by 20 array: 400 bytes
130 FOR J=0 TO 9
140  CHAN(J)=J                    ‘ Shows how arrays are accessed
150  TDAT(0,J)=J*1.234
160 NEXT J
```

REM Statement

can be used with:

Boss 32

UMC

Summary:
The REM statement is used to mark the rest of the line as comment text.

Syntax:
REM *text*

Arguments:

text	any text string.

Description:
The REM statement allows the programmer to put informational comments (also called remarks) into the BASIC program. The compiler disregards anything between a REM statement and the end of the line, including the colon (:). Comment text increases the size of the BASIC source code, but does not increase the size of the compiled code. REM is equivalent to ‘, which also mark comment text; REM statements will be displayed as single quotes (‘) when the program is LISTed. If a (!) is used then the comment line will be thrown out and no comment will be saved.

Example:

```
100 REM This program demonstrates the REM statement
110 INTEGER J: REM Declare an integer : this is still a comment
120 J=3600              ‘ This comment starts with the quote character
130                     ! This comment starts with the exclamation point
140 J=J/3               ! Note that there doesn’t have to be a colon before the quote
```

If this example were typed in (or downloaded) exactly as shown above and then LISTed, it would be displayed like this:

```
100 ‘ This program demonstrates the REM statement
110 INTEGER J: ‘ Declare an integer : this is still a comment
120 J=3600: ‘ This comment starts with the quote character
140 J=J / 3
```

can be used with:

Boss 32

UMC

RESTORE Statement

Summary:
The RESTORE statement resets the DATA pointer to the first DATA statement in the program.

Syntax:
RESTORE

Arguments:
RESTORE needs no arguments.

Description:
As READ statements are executed to load values from DATA statements, a pointer is incremented to point to the succeeding DATA statement. RESTORE sets this pointer to the first DATA statement in the program. The next READ will load the first DATA item.

Example:

```
100 INTEGER J, K
110 DATA 1,2,3,4
120 DATA 5,6
130 FOR J= 1 to 5
140  READ K: PRINT K
150 NEXT J
160 RESTORE                         ‘ Set back to first DATA, which is 1
170 READ K: PRINT K
```

This produces the following output when run:

```
1
2
3
4
5
1
```

can be used with:

Boss 32

UMC

RETURN Statement

Summary:
The RETURN statement ends a subroutine started by a GOSUB.

Syntax:
RETURN

Arguments:
RETURN needs no arguments.

Description:
The RETURN statement causes program execution to continue at the statement following the GOSUB that called the subroutine containing the RETURN. If a RETURN is encountered without a corresponding GOSUB, then the error message RETURN Without GOSUB will be displayed.

Example:

```
100 GOSUB 200: PRINT “We’re back”   ‘ Call subroutine at line 200
110 STOP                             ‘ Don’t fall into subroutine
200 PRINT “Line 200”
210 RETURN                           ‘ Return from subroutine
```

This example just calls a subroutine and displays messages to indicate what is being executed. Line 110 is needed so that it doesn’t continue executing and fall into line 200, which would generate an error when it got to line 210. Try removing line 110 and running the program.

can be used with:

Boss 32

UMC

RND Function

Summary:
The RND function returns a pseudo-random integer.

Syntax:
x = RND

Arguments:
RND needs no arguments.

Description:
The RND function generates pseudo-random numbers, which means that they are generated by a mathematical algorithm. It returns an integer value between -2147483648 and 2147483647. This algorithm starts with a seed number; the first call to RND returns a number that is based on this seed. Each succeeding number is based on the previous number, and becomes the new seed. The seed may be initialized using the RANDOMIZE statement.

Example:

```
100 INTEGER J
110 RANDOMIZE
120 FOR J=1 TO 32
130 FPRINT “I10Z”,RND
140 NEXT J
150 PRINT
```

This produces the following output when run:

```
11546    28119    11026    19169    -21366    11495    10626    -
   25297
26118    19959    30194    1855     -19094    4359     -12190
   13135
16678    -18921   -1838    24927    30282     32039    -20670   -
   11921
25670    -22985   -19534   -15489   -4310     28999    -14814
   30607
```

RUN Direct Command

can be used with:

Boss 32

UMC

Summary:
The RUN direct command compiles and executes a program.

Syntax:
RUN

Arguments:
RUN needs no arguments.

Description:
RUN compiles the BASIC program in memory. If no errors are detected, then it executes the program. It can take up to 20 seconds to compile a long program.

RUN Statement

can be used with:

Boss 32

UMC

Summary:

The RUN statement starts execution of a task in a multitasking program.

Syntax:

RUN *task* [,*resched*]

Arguments:

task	an interger expression from 1 through 31. The task number to begin executing.
resched	an interger expression from 1 through 32767. The reschedule interval for task. If this isn't specified, then the schedule for the task interval is undefined.

Description:

The RUN statement makes a task ready to run, so that the context switcher will execute it at some future time. If the task EXITs, then it will be rescheduled to start executing again after *resched* tics have elapsed. This provides a convenient way of making something happen periodically.

The RUN satement should not be used if the task is already executing. This will cause the task to be restarted at the beginning, regardless of where it is currently executing. This would cause erratic operation of the task.

Examples:

```
100 RUN 1,100                'Start task 1 running, resched once/sec
110 GOTO 110                 'Loop forever
200 TASK 1
210 PRINT "task 1 running"
220 EXIT
```

Line 100 sets up task 1 to start running with a reschedule interval of 100 tics, or once per second. The main program then sits in a loop forever. When task 1 runs, it prints a message then EXITs, which causes it to be rescheduled for 100 ticks later.

```
100 RUN 1,100                'Start task 1 running, resched once/sec
110 GOTO 110                 'Loop forever
200 TASK 1
210 PRINT "task 1 running"
220 GOTO 210                 'Loop and print again
```

The difference between this example and the previous one is in line 220. In the second example, task 1 doesn't EXIT, but instead just loops and prints the message again. Since it doesn't EXIT, the reschedule interval has no effect, so line 100 could have just been Run 1.

can be used with:

Boss 32

UMC

S19DOWNLOAD Direct Command

Summary:
The S19DOWNLOAD direct command disables echoing to the serial port and loads code directly from an S19 file to memory.

Syntax:
S19DOWNLOAD

Arguments:
S19DOWNLOAD needs no arguments.

Description:
Normally, the Divelbiss 32-bit Controller echoes all characters entered at the command line prompt back to the terminal to allow the user to see what is being typed. When downloading an S19 file, however, there is no need to show what is in the file. The S19DOWNLOAD command is used to download code compiled and linked off-board with a Z380 compiler/assembler or on-board by the basic compiler. The S19 file holds all the information needed to download assembled code into the Divelbiss 32-bit Controller. If the file is not correct in any way, the download will be aborted and an error will be displayed. The description is usually the name of the first file linked to make the S19 file. The S19 file format holds all the addressing information (where to place the file in the Divelbiss 32-bit Controller's memory), and has checksums for each line downloaded for reliable coding.

If the S19 file is compiled basic code, then the system versions must be the same or the download will be aborted. After basic code is downloaded it can be ran the same as it it had been compiled on-board.

S19GO Direct Command

can be used with:

Boss 32

UMC

Summary:
The S19GO direct command starts code downloaded with S19DOWNLOAD executing.

Syntax:
S19GO

Arguments:
S19GO Needs no arguments.

Description:
S19GO begins execution of the most recently downloaded S19 code. If the S19 code has not been downloaded, or a NEW command has been executed, then the No Compiled Code error will be displayed. S19GO is typically used to run a program previously compiled and linked off-board with a cross compiler and then downloaded with the S19DOWNLOAD command. A S19 file is a hex file created by the linker. The S19 format was chosen because of its ability to address memory larger han 64K, and the compiler for the Z380 supports this format.

.

can be used with:

Boss 32

UMC

S19UPLOAD Direct Command

Summary:
The S19UPLOAD direct command dumps basic compiled code as an S19 file to FILE 0.

Syntax:
S19UPLOAD [FLASH] [FILENAME]

Arguments:

Flash	If flash is entered the file will be taken from FLASH EEPROM.
Filename	The name of the file to be uploaded. If FLASH is entered then *FILENAME* is dumped from the FLASH EEPROM, otherwise the compiled code in memory is dumped using *FILENAME* for the file header.

Description:
S19UPLOAD dumps basic compiled code to FILE 0 as an S19 file so that it can be uploaded to a Divelbiss 32-bit Controller. The compiled code can be dumped from the FLASH EEPROM or from memory.

.

SAVE Direct Command

can be used with:

Boss 32

UMC

Summary:

The SAVE direct command saves the BASIC source program or compiled code that is in memory to the user's EPROM.

Syntax:

SAVE [CODE] [*filename*]

Arguments:

filename	a text string, up to 10 characters long. The filename to be stored on the EPROM. The name will be stored in uppercase, even if it is entered in lowercase.
CODE	if supplied saves complied code to Flash EPROM, otherwise saves source.

Description:

SAVE stores the current BASIC source program onto the user's Flash EPROM. SAVE CODE stores the current compiled code onto the user's Flash EPROM. If *filename* is specified, then it is stored with the file. The file name for a source program is just used as a comment, to indicate what the program does. With compiled code, however, the file's name can be used with the CHAIN statement. It is possible to have more than one file on an Flash EPROM with the same name; these files will be differentiated by their file numbers.

Example:

SAVE	Saves the BASIC source with no filename.
SAVE CODE	Saves compiled code with no filename.
SAVE Program1	Saves the BASIC source as PROGRAM1.
SAVE CODE test	Saves compiled code as TEST.

can be used with:

Boss 32

UMC

SERIALDIR Statement

Summary:
The SERIALDIR statement sets the direction and RTS level of the serial ports.

Syntax:
SERIALDIR *file*, *direction*

Arguments:

file	an integer expression. The file number of the serial port: 0 for COM1, 1 for COM2, 4 for COM3, 5 for COM4.
direction	an integer expression.

0 = both RTS & DTR off - enables the RS485/422 transmitter and the RS422 receiver
1 = RTS on / DTR off - enables the RS232 receiver
2 = DTR on / RTS off - enables the RS422 receiver
3 = RTS & DTR on - enables the RS485 receiver

Description:
For an RS-232 serial ports (COM1 & 2), SERIALDIR sets the RTS & DTR output level. For an RS-422/485 serial ports (COM3 & 4), SERIALDIR sets up the mode. Note that the RS232 transmitter is always enabled.

Upon power-up:

FILE 0 & 1 - the hardware sets the RTS & DTR active.
FILE 4 & 5 - the hardware is set for RS-232.

Example:

```
100 SERIALDIR 0,0            ‘ Set COM1 RTS inactive
110 SERIALDIR 5,0            ‘ Set COM4 to RS485/422 transmit & RS422 receive
```

can be used with:

Boss 32

UMC

SERIALSTAT Function

Summary:

The SERIALSTAT function reads the level of CTS, DSR, DCD, RI on the serial ports.

Syntax:

x = SERIALSTAT (*file*)

Arguments:

file	an integer expression. The file number of the serial port: 0 for COM1, 1 for COM2, 4 for COM3, 5 for COM4.

Description:

For an RS-232 serial port, SERIALSTAT reads the CTS, DSR, DCD, RI levels. The value returned is 8-bits.

	Boss 32	UMC
Bit 0	change in CTS	change in CTS
1	change in DSR	
2	change in DCD	
3	change in RI	
4	CTS (clear to send)	CTS (clear to send)
5	DSR (data set ready)	
6	DCD (data carrier detect)	
7	RI (ring indicator)	

For an RS-422/485 serial port, SERIALSTAT will return a value but these lines are not supported and the value means nothing.

Example:

```
100 X = SERIALSTAT(0)              ‘ Read COM1 lines
```

can be used with:

Boss 32

UMC

SETDATE Statement

Summary:
The SETDATE statement sets the current date on the Real Time Clock.

Syntax:
SETDATE month, day, year, wday

Arguments:

month	an integer expression. The month in the year, represented as 1-Jan, 2-Feb, ..., 12-Dec.
day	an integer expression. The day of the month, represented as 1..31.
year	an integer expression. The year, represented as 0..99.
wday	an integer expression. The day of the week, represented as 1-Sunday, 2-Monday, ..., 7-Saturday. This value is not stored on the UCP, but an expression must still be supplied to avoid generating a syntax error.

Description:
SETDATE sets the current date on the Real Time Clock. SETDATE takes about 200 microseconds to execute.

Example:

```
100 SETDATE 4,1,91,2            ‘ Set Monday, April 1, 1991
```

SETIME Statement

can be used with:

Boss 32

UMC

Summary:
The SETIME statement sets the current time on the Real Time Clock.

Syntax:
SETIME hours, minutes, seconds

Arguments:

hours	an integer expression. The hours, represented as 0..23, with 0 being midnight.
minutes	an integer expression. The minutes, represented as 0..59.
seconds	an integer expression. The seconds, represented as 0..59.

Description:
SETIME sets the current time on the Real Time Clock. SETIME takes about 200 microseconds to execute.

Example:

```
100 SETIME 14,23,45                    ‘ Set time to 2:23:45 pm
```

.

can be used with:

Boss 32

UMC

SETOPTION DAC Direct Command

Summary:
The SETOPTION DAC command sets the initial power-up value for a DAC output channel.

Syntax:
SETOPTION DAC *chan*,*value*

Arguments:

chan	the DAC channel number, between 1 and 12.
value	the initial DAC value, between 0 and 32767.

Description:
The initial DAC values are stored in the reserved area of the EEPROM; when the Divelbiss 32-bit Controller is turned on, each DAC channel is set to the corresponding EEPROM value. These EEPROM values are set using the SETOPTION DAC command. The STAT command will display the setting of all of the SETOPTION values.

Examples:

.

```
SETOPTION DAC 1,16383
```

This sets the EEPROM DAC table so that channel 1 will be set to 16383 when the Boss 32 is turned on. If the DAC module is configured so that channel 1 is -5V to +5V, then this causes the output to be 0V at power-up, for example.

```
SETOPTION DAC 3,0
```

This causes DAC channel 3 to be initialized to 0. If channel 3 is configured as a 4-20mA current loop, then the output would be 4mA at power-up.

SETOPTION LFDELAY Direct Command

can be used with:

Boss 32

UMC

Summary:
The SETOPTION LFDELAY command sets the initial power-up value for the line feed delay.

Syntax:
SETOPTION LFDELAY *value*

Arguments:

value	ON or OFF.

Description:
The default setting for a new unit is LFDELAY = OFF. The STAT command will display the setting of all of the SETOPTION values. For more information about LFDELAY see the LFDELAY direct command.

Examples:

```
SETOPTION LFDELAY ON
```

.

can be used with:

Boss 32

UMC

SETOPTION NOLINE Direct Command

Summary:
The SETOPTION NOLINE command sets the initial power-up value for no line numbering mode.

Syntax:
SETOPTION NOLINE *value*

Arguments:
value ON or OFF.

Description:
The default setting for a new unit is NOLINE = OFF. The STAT command will display the setting of all of the SETOPTION values.

Examples:

```
SETOPTION NOLINE ON
```

.

can be used with:

Boss 32

UMC

SETOPTION RESET Direct Command

Summary:
The SETOPTION RESET command sets the initial power-up value for resetting or not resetting all outputs on program termination.

Syntax:
SETOPTION RESET *value*

Arguments:
value ON or OFF.

Description:
On power-up, if RESET is set to ON, the Divelbiss 32-bit Controller will reset all outputs at program termination. The digital outputs are reset to 0 and the DAC outputs are reset to the values set with the SETOPTION DAC command. The STAT command will display the setting of all of the SETOPTION values.

Examples:
```
SETOPTION RESET ON
```

can be used with:

Boss 32

UMC

SETOPTION RUN Direct Command

Summary:
The SETOPTION RUN command sets the initial power-up value for running or not running the unit with a program stored in FLASH EPROM.

Syntax:
SETOPTION RUN *value*

Arguments:
value BASIC, ON, OFF, S19

Description:
On power-up if RUN is set to ON, theDivelbiss 32-bit Controller will run the last stored program in FLASH EPROM. The STAT command will display the setting of all of the SETOPTION values. BASIC and ON arguments are the same.

Examples:
SETOPTION RUN ON

.

SETOPTION SRL Direct Command

can be used with:

Boss 32

UMC

Summary:
The SETOPTION SRL command sets the initial power-up value for the baud rate of COM1 in command line entry mode.

Syntax:
SETOPTION SRL *value*

Arguments:
value the initial SRL value, 2400, 4800, 9600, 19200, 38400, 57600.

Description:
The initial SRL value is used to set the baud rate for the COM1 in command line entry mode. This allows more flexibility for downloading programs into the Divelbiss 32-bit ControllerP 202
. The STAT command will display the setting of all of the SETOPTION values.

Examples:

.

```
SETOPTION SRL 19200
```

can be used with:

Boss 32

UMC

SETOPTION TERMINAL Direct Command

Summary:
The SETOPTION TERMINAL command sets the initial power-up value for the terminal emulation on COM1.

Syntax:
SETOPTION TERMINAL *value*

Arguments:

value	the initial TERMINAL value, ADM5, VT52, VT100, VT220, ANSI

Description:
The initial TERMINAL value is set to ADM5 unless this command is used to change it. The STAT command will display the setting of all of the SETOPTION values.

Examples:
SETOPTION TERMINAL VT100

.

This command for use in programming

Boss 32

UMC

SETOPTION TIC Direct Command

Summary:
The SETOPTION TIC command sets the multiplier times 2.5milliseconds for the switching rate of the context switcher. This timing affects all commands that rely on tic timing (WAIT, RUN, etc).

Syntax:
SETOPTION TRACE *value*

Arguments:
value 1,2,3, or 4.

Description:
The initial TRACE value is set 4 unless this command is used to change it.
Typing SETOPTION TIC will display the current multipler setting.

Examples:

.

SETOPTION TIC 1

can be used with:

Boss 32

UMC

SETOPTION TRACE Direct Command

Summary:
The SETOPTION TRACE command sets the file number that all trace line numbers will be displayed on .

Syntax:
SETOPTION TRACE *value*

Arguments:
value ON or OFF.

Description:
The initial TRACE value is set to file 0 unless this command is used to change it. The STAT command will display the setting of all of the SETOPTION values.

Examples:

```
SETOPTION TRACE 1
```

.

SETOPTION VARSIZE Direct Command

can be used with:

.

UMC

Summary:
The SETOPTION VARSIZE direct command sets the number of KBytes to resevere for variable storage.

Syntax:
SETOPTION VARSIZE kbytes

Arguments:

kbytes	The number of KBytes to reserve for variable storage.

Description:
SETOPTION VARSIZE sets the number of KBytes to reserve for variable storage. The inital value is 32 KBytes. If the error message “2 Not Enough Variable Memory to Compile Program” is displayed the setting can be increased to allow for more variables. Also the variable storage can be decreased to allow for more code storage.

For the changes to take effect you must reset the unit, type BYE, or CLEARMEMORY.

Example:
```
SETOPTIONVARSIZE 16
```

SETOPTION VFD Direct Command

can be used with:

Boss 32

.

Summary:
The SETOPTION VFD command sets the initial power-up value for the intensity of the vacuum flourescent display.

Syntax:
SETOPTION VFD *value*

Arguments:
value the initial luminance setting for the VFD is value from 1 to 4. A value of 1 is the brightest.

Description:
The initial VFD value is set to 1 unless this command is used to change it. The STAT command will display the setting of all of the SETOPTION values.

Examples:

```
SETOPTION VFD 2
```

SIN Function

can be used with:

Boss 32

UMC

Summary:
The SIN function calculates the sine function.

Syntax:
x = SIN (*expr*)

Arguments:

expr	a numeric expression.

Description:
The SIN function returns the sine of its argument, which must be a numeric expression, in degrees. The result is returned as a REAL value.

Example:

```
100 REAL X,Y
110 PRINT SIN(45.0)
120 X=68.3
130 Y=SIN(X)
140 PRINT "Sine value of ";X;" is ";Y
```

This produces the following output when run:

```
.70711
Sine value of 68.29999 is .92913
```

can be used with:

Boss 32

UMC

SQR Function

Summary:
The SQR function calculates the square root function.

Syntax:
x = SQR (*expr*)

Arguments:
expr a numeric expression.

Description:
The SQR function returns the square root of its argument, which must be a numeric expression. The result is returned as a REAL value.

Example:

```
100 REAL X,Y
110 PRINT SQR(2.0)
120 X=144.0
130 Y=SQR(X)
140 PRINT “Square root of “;X;” is “;Y
```

This produces the following output when run:

```
1.41422
Square root of 144.00000 is 12.00000
```

STAT Direct Command

can be used with:

Boss 32

UMC

Summary:
The STAT direct command displays the Bear BASIC software version number, memory usage and hardware on board.

Syntax:
STAT

Arguments:
STAT needs no arguments.

Description:
The STAT command can be issued at any time. It displays the Divelbiss 32-bit Controller's software version number. It also displays the memory usage information and the hardware installed on the unit. If a program was compiled and generated errors, then some of the numbers displayed by STAT may be incorrect.

Example:

STAT displays data such as the following:

```
                    Divelbiss 32-bit Controller status of memory and hardware

  Code Start: 0084C101       Code End: 0084C101
Variable Start: 0040FFFF     Variable End: 0040FFFF

   0k Variable Bytes Used              64k Variable Bytes Free
   0k Temporary & Constant Bytes Used
   0k Source Bytes Used                208k Code Bytes Free
   0k Runtime Bytes Used
  123k FLASH EPROM Bytes Used          133k FLASH EPROM Bytes Free

        press the space bar to continue

Serial baud rate set to:                    19.2k
No Line Numbering is                        OFF
Delay Carriage Return Line Feed is          OFF
Terminal Emulation is                       VT52
RUN on power up is                          OFF
VFD LEVEL is set to                         1
RESET outputs on program termination is     ON
TRACEON file number is set to                0

ON BOARD HARDWARE:    A/D                   PRESENT
                      D/A                   ABSENT
                      COUNTER               PRESENT
                      REAL TIME CLOCK       PRESENT
```

```
                                   DISPLAY                     2 X 40 VFD
                                   EEPROM                      8k

EXPANSION MODULES:                 J3 -                        OPEN
                                   J4 -                        OPEN
                                   J5 -                        OPEN

BOSS 32 BASIC Compiler Version 1.02F

>
```

can be used with:

Boss 32

UMC

STOP Statement

Summary:
The STOP statement stops execution of the program or task.

Syntax:
STOP [TASK]

Arguments:
TASK an interger expression. The number of the task to stop.

Description:
If no task number is given STOP immediately halts execution of the program and returns to the compiler prompt. If a task number is given STOP immediately halts execution of the specified task. The specified task will not run again until it is specifically commanded to via another run statement.

Example:

```
100 RUN 1,S
110 WAIT 10
120 STOP 1
130 WAIT 100
200 TASK 1
210 PRINT"TASK 1"
220 EXIT
```

This produces the following output when run:

```
TASK 1
TASK 1
```

can be used with:

Boss 32

UMC

STR$ Function

Summary:
The STR$ function converts a number into a string.

Syntax:
st$ = STR$ (expr)

Arguments:

expr	a numeric expression.

Description:
The STR$ function returns a STRING result that is the human readable, decimal text representation of *expr*. It formats the result as a real number (ie. 12.34000, 5.00000, etc.). Remember that a number may have many different representations; for example, the decimal number 100 is represented in hexadecimal as $00000064. The result of STR$(100) is a 9 character string: “100.00000”.

Note that this is much different than MKI$ or MKS$, which convert a number into a binary string. Binary strings are not human readable, and in fact are just arrays of bytes.

Example:

```
100 INTEGER J: REAL X
110 STRING A$(60)
120 J=45: X=38.014
130 PRINT STR$(J)
140 A$=CONCAT$(“The value of X is “,STR$(X))
150 PRINT A$
```

This produces the following output when run:

```
45.00000
The value of X is 38.01399
```

STRING Statement

can be used with:

Boss 32

UMC

Summary:
The STRING statement is used to declare string variables.

Syntax:
STRING variable [,variable]...

Arguments:
variable a text string. The name to use for the string variable. String variable names in Divelbiss 32-bit BASIC consist of an alphabetic character followed by up to 6 alphanumeric characters followed by a '$'. To declare an array, this will be followed by one or two numbers enclosed in parenthesis. The following are valid string variable names: A$, A$(20), NAME$, TIME4$(10), A$(10,7).

Description:
String variables are used to hold text data. All variables in a Divelbiss 32-bit BASIC program must be declared as INTEGER, REAL, or STRING; all variable declarations must occur before any executable statements in the BASIC program.

Variables in the STRING statement may have a maximum string length specified by enclosing the maximum length in parenthesis; no more than 127 may be specified. If no maximum is given, it will default to 20 characters. If an attempt is made to access beyond the end of the string, the error message String Length Exceeded will be displayed. A string array may be specified by giving two parameters enclosed in parenthesis: the first is the length of each element, and the second is the number of elements in the array. When accessing a string array, however, only one parameter is given inside the parenthesis; for example, if STRING RL$(10,24) is used to declare an array of 25 strings, each 10 characters long, then the 11th array element is accessed using RL$(10).

Example:

```
100 INTEGER J
110 STRING A$                       ‘ 20 character string
120 STRING RDLIN$(80)               ‘ 80 character string
130 STRING DAY$(3,6)                ‘ Array of 7 strings, 3 chars long
140 DATA “Sun”,”Mon”,”Tue”,”Wed”,”Thu”,”Fri”,”Sat”
150 PRINT “Enter a string: “
160 INPUT RDLIN$
170 A$=MID$(RDLIN$,1,3)
180 PRINT A$
170 FOR J = 0 TO 6
180  READ DAY$(J)                   ‘ Shows how arrays are accessed
190  PRINT DAY$(J)
200 NEXT J
```

.

This produces the following out out when run:

```
Enter a string:
Test
Test
Sun
Mon
Tue
Wed
Thu
Fri
Sat
```

SYSADC Direct Command

can be used with:

Boss 32

UMC

Summary:
The SYSADC direct command reads an analog input value from one of the A/D channels at the command prompt.

Syntax:
SYSADC chan [, num chans]

Arguments:

chan	The A/D channel number to read starting with 1
num chans	The number of A/D channels to read

Description:
SYSADC can be used to read the A/D channels at the command prompt. It performs an analog conversion on the specified A/D channel(s) and displays the value between 0 and 32767. For more information see ADC Function.

.

EXAMPLE:

```
sysadc 1,8

I1:0
I2:0
I3:0
I4:0
I5:0
I6:0
I7:0
I8:0
```

can be used with:

Boss 32

UMC

SYSDAC Direct Command

Summary:
The SYSDAC direct command controls an analog output channel from the command prompt.

Syntax:
SYSDAC chan, value, [, num chans]

Arguments:

chan	The D/A channel number to control starting with 1
value	The level to set on the analog output, ranging between 0 and 32767.
num chans	The number of D/A channels to control

Description:
SYSADC can be used to read the A/D channels at the command prompt. It performs an analog conversion
on the specified A/D channel(s) and displays the value between 0 and 32767. For more information
see ADC Function.

.

Example:
```
SYSDAC 1, 16383

SYSADC 1, 1500, 2
```

SYS DATE Direct Command

can be used with:

Boss 32

UMC

Summary:
The SYS DATE direct command displays the date read from the on board real time clock.

Syntax:
SYS DATE

Arguments:
SYS DATE needs no arguments.

Description:
The SYS DATE direct command displays the day, month, date, and year.

Example:

```
>SYS DATE

Thursday, April 25, 1996

>
```

SYS SET DATE Direct Command

can be used with:

Boss 32

UMC

Summary:
The SYS SET DATE direct command sets the date in the on board real time clock.

Syntax:
SYS SET DATE [*month/date/year,day of week*]

Arguments:
SYS SET DATE needs no arguments but will accept them.

Description:
The SYS SET DATE direct command will set the real time clock.

Example:

```
>SYS SET DATE
Enter the date MM/DD/YY/WD -
```

.

can be used with:

Boss 32

UMC

SYS SET TIME Direct Command

Summary:
The SYS SET TIME direct command sets the time in the on board real time clock.

Syntax:
SYS SET TIME [*hours:minutes:seconds*]

Arguments:
SYS SET TIME needs no arguments but will accept them.

Description:
The SYS SET TIME direct command sets the time in the real time clock.

Example:

```
>SYS SET TIME
Enter the time HH:MM:SS -
```

.

SYS TIME Direct Command

can be used with:

Boss 32

UMC

Summary:
The SYS TIME direct command displays the time read from the on board real time clock.

Syntax:
SYS TIME

Arguments:
SYS TIME needs no arguments.

Description:
The SYS TIME direct command displays the current time read from the real time clock.

Example:

```
>SYS TIME

7:49:57 am

>
```

SYSDIN Direct Command

can be used with:

Boss 32

UMC

Summary:
The SYSDIN direct command reads input expander modules attached to the controller.

Syntax:

SYSDIN	addr [, num]

Arguments:

addr	The address of the input channel to read.
num	Optional, the number of inputs to read.

Description:
SYSDIN starts reading inputs at addr and reads num inputs if specified. See the DIN function for more information.

Example:

```
SYSDIN 1,8
I1:1
I2:1
I3:1
I4:1
I5:1
I6:1
I7:1
I8:1
```

can be used with:

Boss 32

UMC

SYSDOUT Direct Command

Summary:
The SYSDOUT direct command controls output expander modules attached to the controller.

Syntax:
SYSDOUT addr , state [, num]

Arguments:

addr	The address of the input channel to read.
state	A value 0 turns the output off, any other value turns the output on.
num	Optional, the number of inputs to set.

Description:
SYSDOUT sets the state of output channel addr. See the DOUT function for more information.

.

Example:
```
SYSDOUT 0,1,2

SYSDOUT 4,0
```

can be used with:

Boss 32

UMC

TAN Function

Summary:
The TAN function calculates the tangent function.

Syntax:
x = TAN (*expr*)

Arguments:
expr a numeric expression.

Description:
The TAN function returns the tangent of its argument, which must be a numeric expression, in degrees. The result is returned as a REAL value.

Example:

```
100 REAL X,Y
110 PRINT TAN(45.0)
120 X=68.3
130 Y=TAN(X)
140 PRINT “Tangent value of “;X;” is “;Y
```

This produces the following output when run:

```
1.00000
Tangent value of 68.29999 is 2.51288
```

can be used with:

Boss 32

UMC

TASK Statement

Summary:
The TASK statement marks the beginning of a task.

Syntax:
TASK *task*

Arguments:

task	an integer number from 1 through 31. This is the number of the task in the program.

Description:
In a multitasking program, all tasks, other than task 0 (the main program), must begin with a TASK statement. The TASK statement must be at the start of a BASIC line; it cannot be in the middle of a multi-statement line. The tasks must be numbered sequentially starting at 1; the error message Task Error is displayed by the compiler if a task is numbered out of order.

Each task has its own set of temporary variables that are used by BASIC to perform operations; they must be separate so that a task doesn't corrupt other task's variables when it starts to execute. A task must not GOTO or GOSUB a line that is part of another task without taking special precautions; see Chapter 5 for further details.

Example:

```
100 RUN 1,100                   ‘ Start task 1, reschedule once/second
110 PRINT “Main program”
120 WAIT 90: GOTO 110
200 TASK 1                      ‘ Mark beginning of task 1
210 PRINT “Task 1”
220 EXIT
```

NOTE: In noline mode, task line must have a label.

can be used with:

Boss 32

UMC

TIMER Statement

Summary:
The TIMER statement is used to set, read, start and stop the timers.

Syntax:
TIMER *mode*, *x* [, *value*]

Arguments:

mode	an integer number from 0 to 5. This is the number defines the action of the statement. Timer modes: 0 - Stop and optionally read the timer 1 - Start and optionally write to the timer 2 - Read the timer 3 - Write to the timer 4 - Set the HDIO putout number and value (for timer 0 only) 5 - Set the reload value for timer 0
x	an integer value. This is the timer number or HDIO number
value	an integer value from 1 through 2,417,483,647. This is the value written or read from the timer or output state.

Description:
There are 221 different timers. Timer 0 is a special hardware timer with a resolution of 34.722mS. Timer 0 can be used to set or reset a HDIO output at the end of its count. This function is only available for timer 0. Timer 1 - 20 have a resolution of 10mS. The rest of the timers (21 - 220) have a resolution of 100mS. All of the timers except timer 0 are not effected by turning off interrupts. All the timers are count down timers and stop when zero is reached.

Example:

```
100 INTEGER A
110 TIMER 2,10,A                    ‘ READ TIMER 10
120 TIMER 3,11,A                    ‘ WRITE TO TIMER 11
130 TIMER 1,11                      ‘ START TIMER 11
```

can be
used with:

Boss 32

TMADDR Statement

Summary:
The TMADDR statement sets the ID for succeeding Touch Memory accesses.

Syntax:
TMADDR *idstr*

Arguments:
idstr Touch Memory ID stored in first 8 bytes of string.

Description:
After TMSEARCH is used to find the ID's of all attached Touch Memories, TMADDR specifies which Touch Memory to communicate with. After TMADDR ID is executed, all succeeding TMREAD and TMWRITE accesses will go to the Touch Memory with the specified ID.

Example:

```
100 INTEGER NDEV, NUM, ST
110 STRING ID$(80), DAT$(30)
200 NDEV=TMSEARCH(ID$)                    ‘ Read device IDs.
210 IF NDEV=0 THEN WAIT 50: GOTO 200      ‘ Loop if none found.
220 FOR NUM=0 TO NDEV-1
230  TMADDR MID$(ID$,NUM*8+1,8)           ‘ Set up ID to write to.
240  ST=TMREAD(DAT$,0,20)                 ‘ Zero out data at location 0.
250  PRINT ST,DAT$                        ‘ Display data.
260 NEXT NUM
```

TMREAD Function

can be used with:

Boss 32

Summary:

The TMREAD function reads data from a Touch Memory device attached to the Touch Memory port.

Syntax:

stat=TMREAD(datstr$,location,length)

Arguments:

datstr$	string variable to store data into.
location	an integer expression. The starting byte location to read from within the Touch Memory.
length	an integer expression. The number of bytes to read from the Touch Memory.

Description:

TMREAD transfers data from a Touch Memory device into a BASIC string variable. It transfers *length* number of bytes starting at *location* in the Touch Memory.

Note that no range checking is performed on the *location* and *length* arguments. It is possible to read beyond the end of the Touch Memory device. This causes unpredictable values to be stored in *datstr$*.

Example:

```
100 INTEGER NDEV, NUM, ST
110 STRING ID$(80), DAT$(30)
200 NDEV=TMSEARCH(ID$)                  ‘ Read device IDs.
210 IF NDEV=0 THEN WAIT 50: GOTO 200    ‘ Loop if none found.
220 FOR NUM=0 TO NDEV-1
230  TMADDR MID$(ID$,NUM*8+1,8)         ‘ Set up ID to write to.
240  ST=TMREAD(DAT$,30,10)              ‘ Read bytes 30..39
250  PRINT ST,DAT$
260 NEXT NUM
```

can be used with:

Boss 32

TMSEARCH Function

Summary:
The TMSEARCH function reads the ID string for all Touch Memory devices attached. It returns the number of devices found.

Syntax:
num=TMSEARCH(*idstr*)

Arguments:

idstr	string variable to store any Touch Memory ID strings in.

Description:
Each Touch Memory device has a unique eight byte ID string stored permanently in it. In order to access the device, this ID must be sent as part of all commands. The TMSEARCH function scans the Touch Memory interface port, getting the ID strings for all attached Touch Memories. All of these ID strings are concatenated together in *idstr*. If two Touch Memories were found, for example, then *idstr* would be 16 characters long, with the first ID at the first character of the string, and the second ID starting at the eighth character of the string.

.

Example:

```
100 INTEGER NDEV, NUM, ST
110 STRING ID$(80), DAT$(30)
150 DAT$=""
160 FOR NUM=1 TO 30
170  DAT$=CONCAT$(DAT$,CHR$(0))           ' Build string of 0 bytes.
180 NEXT NUM
200 NDEV=TMSEARCH(ID$)                              ' Read device IDs.
210 IF NDEV=0 THEN WAIT 50: GOTO 200      ' Loop if none found.
220 FOR NUM=0 TO NDEV-1
230  TMADDR MID$(ID$,NUM*8+1,8)           ' Set up ID to write to.
240  ST=TMWRITE(DAT$,0)                   ' Zero out data at location 0.
250  PRINT "Write status=";ST
260 NEXT NUM
```

TMWRITE Function

can be used with:

Boss 32

Summary:
The TMWRITE function writes data to a Touch Memory device attached to the Touch Memory port.

Syntax:
stat=TMWRITE(datstr$,location)

Arguments:

datstr$	string variable containing the data to write.
location	an integer expression. The starting byte location to read from within the Touch Memory.

Description:
TMWRITE transfers data from a BASIC string variable into a Touch Memory device. It transfers *length* number of bytes starting at *location* in the Touch Memory.

Important

Note that no range checking is performed on the *location* and *length* arguments. It is possible to attempt to write beyond the end of the Touch Memory device. This causes data to be lost, since the data that won't fit in the Touch Memory is discarded.

Example:

```
100 INTEGER NDEV, NUM, ST
110 STRING ID$(80), DAT$(30)
200 NDEV=TMSEARCH(ID$)                     ‘ Read device IDs.
210 IF NDEV=0 THEN WAIT 50: GOTO 200       ‘ Loop if none found.
220 FOR NUM=0 TO NDEV-1
230  TMADDR MID$(ID$,NUM*8+1,8)            ‘ Set up ID to write to.
240  DAT$=”This is a test”
250  ST=TMWRITE(DAT$,73)                   ‘ Write to bytes 73..86
260  PRINT ST
270  ST=TMREAD(DAT$,73,14)
280  PRINT ST,DAT$
290 NEXT NUM
```

can be used with:

Boss 32

UMC

TRACEON and TRACEOFF Statements

Summary:

The TRACEON and TRACEOFF statements enable and disable the line number trace, respectively.

Syntax:

TRACEON
TRACEOFF

Arguments:

TRACEON and TRACEOFF need no arguments.

Description:

TRACEON causes the line number of each line to print as it is executed; the line numbers are printed on the FILE device set by, SETOPTION TRACE. TRACEOFF disables the trace.

Example:

```
100 INTEGER J
110 TRACEON
120 FOR J=1 TO 3
130  PRINT “Number “; J
140 NEXT J
150 TRACEOFF
160 PRINT “Done”
```

This produces the following output when run:

```
120
130
Number 1
140
130
Number 2
140
130
Number 3
140
150
Done
```

VAL Function

can be used with:

Boss 32

UMC

Summary:
The VAL function converts a string into a number.

Syntax:
x = VAL (strexpr$)

Arguments:

strexpr$	a string expression.

Description:
The VAL function returns a REAL result that is the numeric conversion of the number at the beginning of *strexpr$*. If *strexpr$* can't be interpreted as a number, then VAL returns 0. Note that this is much different than CVI or CVS, which convert a binary string into a number. Binary strings are not human readable, and in fact are just arrays of bytes.

Example:

```
100 INTEGER J: REAL X
110 STRING A$(60)
120 A$=”123”
130 J=VAL(A$)
140 PRINT “Numeric value of “; A$; “ is “; J
150 X=VAL(“83.298”)
160 PRINT “X = “; X
```

This produces the following output when run:

```
Numeric value of 123 is 123
X = 83.29799
```

can be used with:

Boss 32

UMC

WAIT Statement

Summary:
The WAIT statement delays execution of the current task for a specified period.

Syntax:
WAIT num_tics

Arguments:
num_tics a numeric expression from 1 to 32767. The number of tics (1/100 second) to delay. A tick equals 10msec which equals 1/100 second.

Description:
The WAIT statement provides an efficient way to put a delay into a program. It delays the execution of the current task for at least *num_tics* hundredths of a second; the task may be delayed for more than *num_tics*, depending upon how many tasks are ready to run when *num_tics* have elapsed. Any other tasks that are ready to run will continue to execute; if no other task is ready to run, then the processor will be idle until one of the tasks is done WAITing and can execute again. WAIT enables interrupts, since it depends upon the context switcher being called, so it can’t be used inside of a hardware interrupt service task. WAIT works fine in a single-task program, the processor just sits idle until *num_tics* have elapsed.

Example:

```
100 RUN 1                    ‘ Start task 1 executing
110 PRINT “*”;               ‘ Print a ‘*’ every 300 msec
120 WAIT 30: GOTO 110
130 TASK 1
140 PRINT “-”;: GOTO 140     ‘ Print a ‘-’ every 1 msec at 9600 baud
```

WPEEK Function

can be used with:

Boss 32

UMC

Summary:
The WPEEK statement reads an integer word value from the specified address.

Syntax:
x = WPEEK (*logaddr*)

Arguments:
logaddr an integer expression. The logical address to read from.

Description:
The WPEEK function reads a word from memory at *logaddr*; it returns the word as an INTEGER. PEEK and WPEEK are similar, except that WPEEK reads a word (4 bytes), while PEEK only reads a byte.

Example:

```
100 INTEGER J,K
110 WPOKE $A000,$1234               ‘ Write a $1234 into $A000
120 PRINT “$A000=”;WPEEK($A000)
130 J=0: K=500
140 PRINT WPEEK(ADR(K))             ‘ Print the value of K
```

This produces the following output when run:

```
$A000=4660
500
```

Line 120 prints $A000=4660 because 4660 is the decimal equivalent of $1234. Line 140 reads the variable K by WPEEKing the address of K.

WPOKE Statement

can be used with:

Boss 32

UMC

Summary:
The WPOKE statement writes an integer word value to the specified address.

Syntax:
WPOKE logaddr, value

Arguments:
logaddr an integer expression. The logical address to write to.
value an integer expression. The value to write.

Description:
The WPOKE statement writes a word (*value*) into memory at *logaddr*. POKE and WPOKE are similar, except that WPOKE writes a word (4 bytes), while POKE only writes a byte.

Example:

```
100 INTEGER J,K
110 WPOKE $A000,$1234              ‘ Write a $1234 into $A000
120 PRINT “$A000=”;WPEEK($A000)
130 J=0: K=500
140 WPOKE ADR(K),J                 ‘ Write a $0000 into low byte of K
150 PRINT K
```

This produces the following output when run:

```
$A000=4660
0
```

Line 120 prints $A000=4660 because 4660 is the decimal equivalent of $1234. Line 140 stores a 0 into the variable K.

WRCNTR Statement

can be used with:

Boss 32

UMC

Summary:
The WRCNTR statement writes the count value into the high speed counter.

Syntax:
WRCNTR chan, flag, expr

Arguments:

chan	an integer expression between 1 and the number of counter channels installed. This is the counter channel to write to.
flag	an integer expression of 0 or 1. This determines whether to write to the actual counter or to the compare register: 0 to write to both the actual counter and the compare register. 1 to write to only the compare register.
expr	a integer expression.

Description:
The counter hardware has an actual count register and a compare register; WRCNTR is used to write a value to these registers. In order to write to the actual count, the value is also written into the compare register, which means that the actual count should always be set before the compare value is set.

The counter channels are assigned based on the hardware available on the Divelbiss 32-bit Controller; channel 1 could be the onboard counter, or in any of the expansion ports (if there is no onboard counter. The order of precedence for assigning counter channels is: onboard counter followed by J3 followed by J4 followed by J5.

Examples:

```
100 INTEGER COUNTI
120 CNTRMODE 1,4                ‘ A count, B direction
130 WRCNTR 1,0,0                ‘ Set count to 0
140 WRCNTR 1,1,10               ‘ Set compare register to 10
150 ‘ Main program loop
160 RDCNTR 1,0,COUNTI           ‘ Read current count as integer
170 PRINT COUNTI
180 WAIT 10: GOTO 150
```

ZPRINT Statement

can be used with:

Boss 32

Summary:
The ZPRINT statement draws a single zoomed graphics character on the Graphics Vacuum Fluorescent Display (GVFD) .

Syntax:
ZPRINT *z* , (*x*, *y*) , *char* [, S]

Arguments:

z	a integer value (1 to 8). The zoom factor to increase the character.
x	a integer expression. The horizontal position to start the character.
y	a integer expression. The vertical position to start the character.
char	a ASCII string. It uses the first character in the string to print. ASCII values 32-126 are supported.
S	a optional S smoothes the zoomed character.

Description:
ZPRINT is one of the statements that can be used to draw graphics on the GVFD. This statement only works with the GVFD. The (x,y) coordinate system puts (0,0) at the lower left corner. The starting coordinate is the lower left corner if the character. Each character starts with a 6x8 pixel matrix. The zoom factor is used to enlarge the matrix to a maxim of 48x64 pixels.

Chapter Five
Using Fuzzy Logic

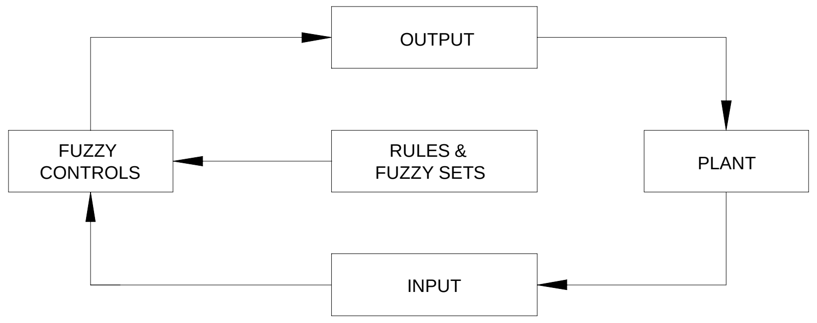
This document describes using Fuzzy Logic with the Divelbiss 32-bit Controller. Fuzzy Logic, as described in this document, is a methodology for using the know-how of experts in the implementation of control systems. In its most basic form, Fuzzy Logic is based on rigorous mathematical theory involving numerous calculations. Superior results to that of conventional control systems (PID ...) have been documented in applications ranging from elevator control to flight control in Navy aircraft. Because of this, Divelbiss Corporation has decided to make this exciting technology available using its general purpose controllers.

For simplicity, our Fuzzy Logic approach requires only variable declarations, minimal data setup, and an easy to use callable Divelbiss 32-bit BASIC statement, *FUZZY. FUZZY* will make all of the calculations required by Fuzzy Logic transparent to the user, thus requiring only the know-how of the user to implement a sophisticated control system.

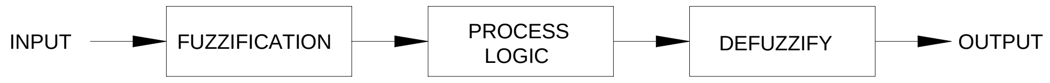
This document is organized into three sections; 1) Fuzzy Logic Fundamentals, 2) Using the *FUZZY* statement, and 3) Examples / Applications. Section 1 will give some of the basic definitions of Fuzzy Logic and describe the technology from the end-user's point-of-view. Section 2 describes how to use the Divelbiss 32-bit BASIC statement *FUZZY* to implement Fuzzy Logic on the Divelbiss 32-bit Controller. And finally, Section 3 will provide some simple, yet real-world examples of using Fuzzy Logic.

Fuzzy Logic Fundamentals

The first figure below shows a block diagram of a typical fuzzy system. The second figure describes in greater detail the Fuzzy Control Block of in the first figure.



Typical Fuzzy Control System



272Fuzzy Control Block

.

The definitions for each of the elements of the control system and control block have been provided below. Although confusing now, these definitions will become clear by the end of this section.

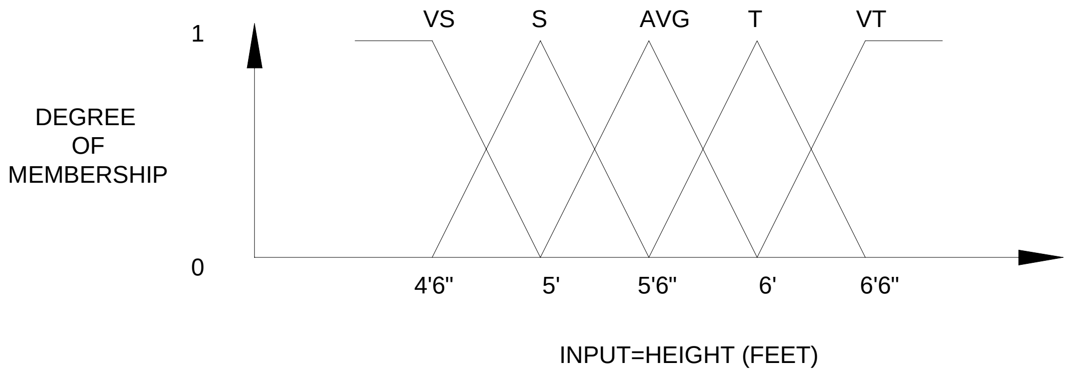
Plant:	Device or process to be controlled
Input:	Data feedback from the plant (i.e., pressure, speed, angle, position, ...)
Output:	Result of the control block which is output to the plant
Fuzzy Control:	Strategy for controlling the plant
Rules & Fuzzy Sets:	Your knowledge of the system being controlled
Fuzzification:	Transforming the input into values related to your fuzzy sets
Process Logic:	Your rules which guide the operation of the control system
Defuzzify:	Process to convert the output of the Process Logic to the actual output

“Fuzzy” Thinking

Fuzzy Sets and Membership Functions

Before a firm grasp of Fuzzy Logic basics can be obtained, an overview of some “fuzzy” thinking is required. Absolutes must be forgotten when considering fuzzy logic. Concepts like “IF YOU ARE 6 FEET IN HEIGHT YOU ARE TALL” must be replaced by “IF YOU ARE *AROUND* 6 FEET IN HEIGHT THEN YOU BELONG TO THE CLASS OF TALL PEOPLE”. The basic idea is that nothing is absolute but everything is fuzzy, and thus the name Fuzzy Logic. Consider a system where an input is the height of a person. As humans, we might describe height with language such as {VERY SHORT, SHORT, AVERAGE, TALL, VERY TALL}, and avoid saying that any one of these descriptions is an exact number. For instance, we might agree that someone who is VERY SHORT could have a height which is less than 5’. SHORT may vary from 4’ 6” to 5’ 6”, AVERAGE from 5’ to 6’, TALL from 5’ 6” to 6’ 6”, and VERY TALL anybody who is greater than 6’. Note that each description involves a range of heights and all of the different descriptions actually overlap. To draw this out graphically, each of our five fuzzy sets might look like Figure 33.

The horizontal axis of Figure 33 is the input values and the vertical axis is the degree of membership having values between 0 and 1. Take as an example someone who is 5’ 10” in height. In reality, that person is somewhat AVERAGE and somewhat TALL. According to Figure 33, if you were to draw a vertical line at 5’ 10” you would intersect lines belonging to fuzzy sets AVERAGE and TALL at around 0.4 and 0.6, respectively. Such an individual (according to our fuzzy sets) belongs to the set of people described by AVERAGE and TALL. This is the way in which all inputs and outputs of a Fuzzy Control block are handled. The inputs/outputs are identified, divided into subgroups (fuzzy sets), and given names. One thing to note is that the triangular shapes of the fuzzy sets in Figure 33 may be replaced by other common shapes such as a bell and trapezoid. However, throughout this document all fuzzy sets will be addressed as triangles because that is the shape available when using the *FUZZY* statement to be described in detail later.



Membership Functions for Input Height

Process Logic

The next step in Fuzzy Logic processing is to operate on the fuzzified input with process logic. This logic is the rules which guide the operation of the control system and are entered by you, the expert of the system. All process logic is in the form of *IF ... THEN* statements. An excellent, real world example is the way in which we operate the accelerator of a car. Consider that our desire is to maintain the speed of the car at 55 MPH. A couple of typical rules which might govern the way we think are:

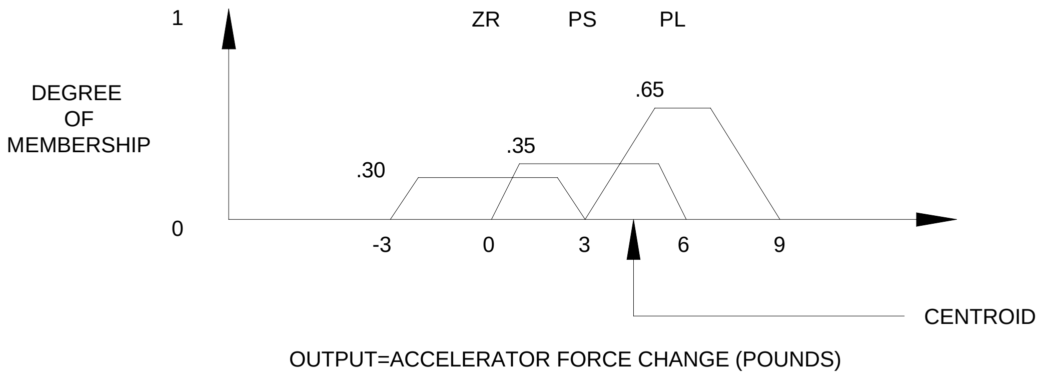
IF OUR SPEED ERROR IS POSITIVE SMALL AND OUR RATE OF CHANGE IS NEGATIVE SMALL *THEN* CHANGE THE ACCELERATOR ZERO.

IF OUR SPEED ERROR IS NEGATIVE LARGE AND OUR RATE OF CHANGE IS ZERO *THEN* CHANGE THE ACCELERATOR POSITIVE LARGE.

In other words, Rule 1 is saying that the car is going a little faster than we want, but we are decreasing in speed so don’t change anything. Rule 2 suggests that we are going much slower than we would like and not currently increasing or decreasing in speed so punch on the accelerator. This simple example clearly shows that a system to control the accelerator of a car would require a minimum of two (2) inputs and one (1) output. The inputs would be the current speed error and the rate of change of speed and the output would be the change to the accelerator. The inputs and output would be divided into subgroups, labeled such things as NEGATIVE LARGE, NEGATIVE SMALL, ZERO, POSITIVE SMALL, and POSITIVE LARGE, and the process logic which guides the control system is simply the way in which you, as an expert of operating a car accelerator, would react.

Defuzzification

Defuzzification is the process by which the applicable rules for a given situation are transformed into an actual output suitable for control of the plant. In the example described above for controlling the accelerator of a car this would be change in force on the pedal. Because in Fuzzy Logic it is not unusual for several output fuzzy sets to be affected by the input data, there must be a method for determining the actual output. Although there are several methods, the most common technique is centroid defuzzification. Centroid defuzzification takes the center of mass of all applicable outputs as the result. Consider that both inputs were operated on by the process logic and three output rules were “fired”; ZERO, POSITIVE SMALL, and POSITIVE LARGE. Through some standard calculations of Fuzzy Logic it was determined that ZERO, POSITIVE SMALL, and POSITIVE LARGE had degree’s of membership of 0.30, 0.35, and 0.65, respectively. The actual output would be the centroid of all of the fired outputs. This process is shown graphically in Figure 34.



Centroid Defuzzication

Thus, using defuzzification a plant (system) can be controlled when used in conjunction with your expertise and experience encoded in the input / output fuzzy sets and process logic.

Selecting Inputs and Outputs

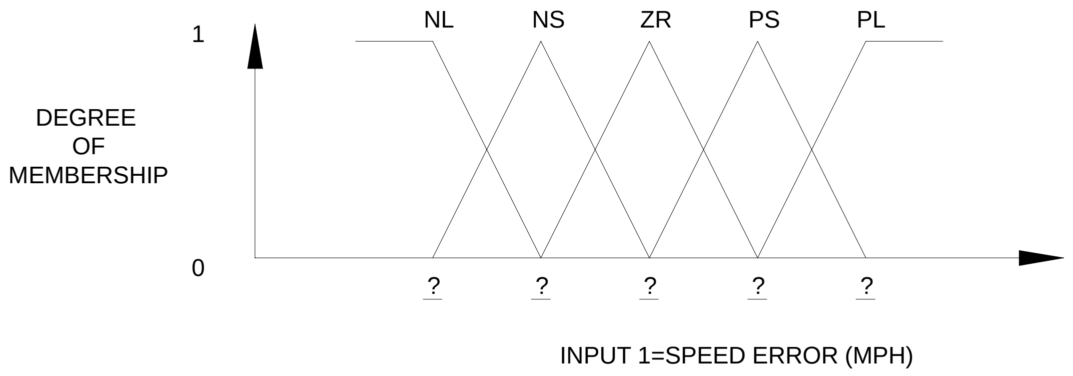
A logical concern in a Fuzzy Logic application is what inputs are essential to the successful implementation of the system. To put it simply, the designer must have a thorough understanding of the system and Fuzzy Logic fundamentals. A good design practice might be to first ensure complete system understanding, ask yourself what are the variables you would use to control the system, and then implement the system. As an example consider our Fuzzy Logic control of the car accelerator. Common sense dictates that a minimum of two input variables are required, speed error and rate of change of speed. Without these two pieces of information our mind could not even control a car’s accelerator. Thus, as a rule of thumb, always start with what appears to be the minimum number of inputs and then add additional inputs and process rules as increased performance dictates.

Selecting Fuzzy Set Values

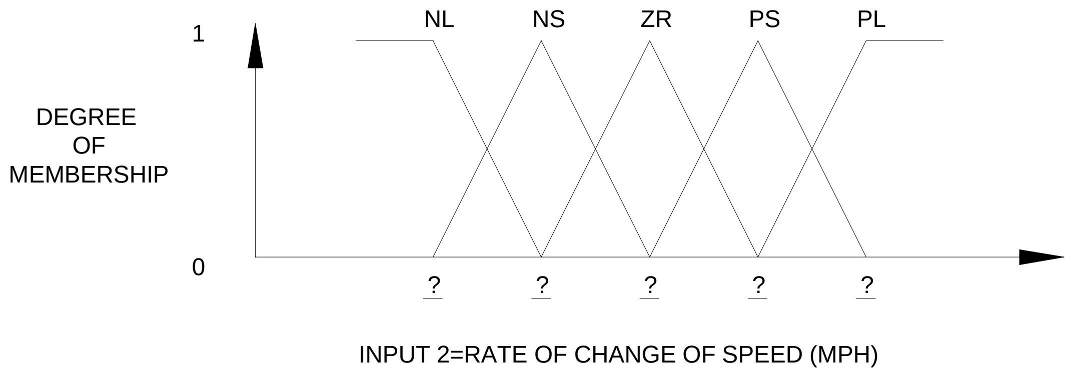
After inputs and outputs have been selected, the next step is to determine exactly what names and values are to be placed on the fuzzy sets. A general overview can be obtained by carefully considering Figure 33 and Figure 34. However, specifics can only be obtained with an example. Therefore, as a trial problem let us try to determine the appropriate fuzzy sets for a simple speed control problem. Just as the problem in controlling the car accelerator, the logical inputs to the Fuzzy Controller are the speed error and the rate of change of speed. An appropriate system output might be the change in the speed. Appropriate fuzzy sets for all three of these might be labeled {NEGATIVE LARGE, NEGATIVE SMALL, ZERO, POSITIVE SMALL, POSITIVE LARGE} for a system with five (5) fuzzy sets per input or output. The next problem is to determine where are the starting and ending points for the fuzzy sets. Review Figure 35, Figure 36, and Figure 37 for schematic representations of these inputs and outputs to the Fuzzy Controller.

The objective in a speed control is to reach the desired speed (setpoint) as smooth as possible, in the fastest time, and without going over (overshoot) or oscillating around the setpoint. To avoid oscillations and to increase the chances of smooth control, it is generally good practice to cover the entire range of the input with the membership functions. Thus, in this case a good choice might have POSITIVE SMALL centered at 40mph and POSITIVE LARGE centered at 80mph. You may be asking yourself the obvious question; How can a 40mph speed error be considered small!! By reviewing Figure 35 you can see that with POSITIVE SMALL centered at 40, the entire range actually spans from 0 to 80mph and POSITIVE LARGE from 40mph to indefinitely. Thus, a value of 40mph actually covers the entire range better than a small value. Additionally, in the actual implementation of this Fuzzy Controller one would find that the larger this value, the smoother the control. This is true because the transition from one fuzzy set to the next is quite large which prevents rapid, and sometimes uncontrolled, changes in the rules which are affected by your process logic. Note that the negative half of the membership functions are the same as the positive half with minus signs.

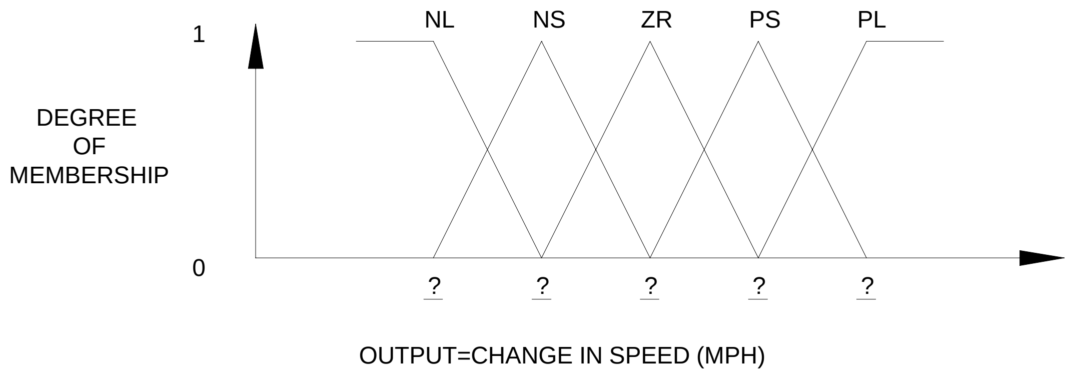
With input 1 complete, the next step is to determine appropriate values for fuzzy sets of input 2. Change in speed actually determines how much change between samples is considered small, large, and so on. Thus, the values of the fuzzy sets for input 2 actually depend on how often you sample inputs 1 and 2. Thus, a thorough understanding of the overall system design is critical to appropriately pick fuzzy set values. Assume that we sample the inputs every .1 seconds. A concept which agrees with common sense is that if we went from 0 to 60mph in 5 seconds this would be quite fast.



Input 1: Speed Error Membership Function



Input 2: Rate of Change of Speed of Membership Function



Output 1: Change in Speed of Membership Function

With some simple mathematics this works out to 1.2mph for every .1 second sample period (60 mph / (5 seconds * .1 seconds)). Thus, a reasonable center value for POSITIVE SMALL for input 2 might be 1.2mph. Once again, just as with input 1, if this value is too small, this will cause sudden and numerous changes in the output rules (which are affected by your process logic) and thus erratic operation. If this number is too big, the controller might never perceive its rate as being large (positive or negative) and thus will not behave correctly. Thus, common sense, knowledge of the system, and a basic understanding of fuzzy logic is required to appropriately select these values.

The last values to determine are those for the only system output, change in speed. The values assigned to the fuzzy sets for this output are also affected by the sampling time of the system. Additionally, a big consideration is the inherently slow response time of most cars to changes in speed. Thus, if the values for these fuzzy sets are too high, the system will be jerky and most likely unstable because every time a new output rule is affected by your process logic, the transition will not be smooth. Additionally, if the input is too low, the system will get to the setpoint slowly. Once again conservative numbers are in order. An appropriate value might be 5-10mph for the center of POSITIVE SMALL.

The primary point to be remembered in the selection of fuzzy set values is to use common sense and moderation. In all cases, values which are extreme can cause unstable operation of the system. Especially before tuning a system, be conservative with the values to avoid damage, especially when the cost of erratic operation can be high.

Constructing Process Logic (Rule Tables)

Process logic is at the heart of why Fuzzy Logic provides an undisputed amount of flexibility in control systems design. Unlike conventional control technology, Fuzzy Logic control systems have the ability to take the expertise and human intuition of the user into account. This expertise is encoded into the controller in terms of process logic. Process logic are the rules which guide the operation of the system and are in the form of *IF...THEN* statements. Process logic is usually constructed in the form of rule tables as is described in Figure 38.

		NL	NS	ZR	PS	PL
	NL		PL	PL	PS	
	NS		PS	PS	ZR	
RATE OF CHANGE	ZR	PL	PS	ZR	NS	NL
	PS		ZR	NS	NS	
	PL		NS	NL	NL	

Example Rule Table for Speed Control

IF SPEED ERROR IS POSITIVE SMALL AND RATE OF CHANGE IS NEGATIVE SMALL *THEN* CHANGE SPEED ZERO.

or

IF SPEED ERROR IS POSITIVE LARGE *THEN* CHANGE SPEED NEGATIVE LARGE.

Such rules are based on expertise and intuition of how the controller should operate. The exact calculations depend on the input and output fuzzy sets, process logic, and some standard calculations required by Fuzzy Logic. Note that in some cases rules may not make sense. It is always in the best interest of the design to remove unnecessary rules to avoid excess processing. However, one should be quite certain when doing this because the rules represent knowledge, and a lack of knowledge can cause unpredictable results.

Tuning a Fuzzy Logic Controller

Once the system operation has been understood, inputs and outputs determined, and fuzzy sets generated, the next step is to simulate or implement the controller for verification. It is recommended that simulations of the system be performed first if such tools are available. Computer modelling of the process will increase the probability of creating a system with acceptable results during the first implementation. However, in some cases this recommended design approach may not be possible and the only way to ensure correct operation is to implement the controller in a real system and then tune the controller. Tuning of a Fuzzy

Logic controller is required just as with conventional control systems. However, unlike tuning controllers such as PID (Proportional, Integral, Derivative), tuning a Fuzzy Logic controller is more intuitive because of the reliance on human experience. Typically, a Fuzzy Logic controller is not successful for the following reasons:

- Incorrect process logic
- Incorrect inputs and/or outputs
- Incorrect fuzzy set values
- An incorrect number of fuzzy sets

Tuning of a Fuzzy Logic controller involves modifying any or all of the four listed items above until acceptable system behavior is obtained. The first two items being incorrect typically results from a lack of understanding of the overall system prior to the implementation of the controller. Without the correct process logic the system cannot respond as desired. Having incorrect inputs and/or outputs sometimes results because other factors are more important in the system than they appear at first. Generally speaking, it is easier to change the process logic than the inputs and outputs because the latter may involve changing the overall system design (including hardware). Thus, understanding what needs to flow into and out of a Fuzzy Logic controller is critical from the outset of a design.

The last two items being incorrect could result because of a lack of understanding of Fuzzy Logic fundamentals. As described in section 11.1.1.3, Selecting Fuzzy Set Values, it is not uncommon to be overly aggressive in selecting fuzzy set values. Being too aggressive with these values from the outset can result in unstable or erratic operation of the control system. Thus, it is generally a good practice to be very conservative in the initial implementation of the controller and tune the system up for more aggressive response after the initial implementation. This involves increasing the values for what might be considered SMALL, etc. for the inputs and decreasing the values for what might be considered SMALL in the output. The previous statement is generally true for such applications as speed, position, and temperature control, just to name a few. With regards to the number of fuzzy sets, a good rule of thumb in many process control applications is to have between five (5) and nine (9) fuzzy sets for each input and/or output. As described in section 11.2, the *FUZZY* statement contains two (2) inputs and two (2) outputs with seven (7) fuzzy sets, which is sufficient for most applications.

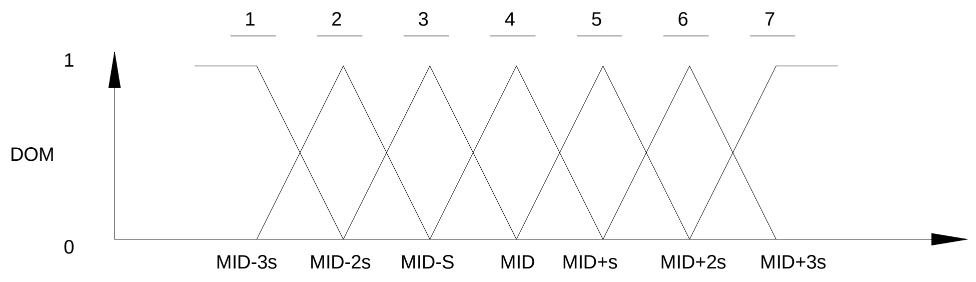
Using the *FUZZY* Statement

The Fuzzy Logic capabilities using the Universal Control Panel are entirely contained in one easy to use statement, *FUZZY*. *FUZZY* has been optimized for speed and simplicity over a broad range of applications. The *FUZZY* statement supports two (2) inputs and outputs, seven (7) fuzzy sets with symmetric triangles, user-definable fuzzy set values, and a return status. Execution of the *FUZZY* statement requires less than 1.0mS, providing sufficient speed for most single-loop applications. The format of the *FUZZY* statement using the Divelbiss 32-bit BASIC programming language is as follows:

```
FUZZY ADR(DATA VARIABLE 1), INPUT1, INPUT2, OUTPUT1, OUTPUT2, STATUS
```

where ADR(DATA VARIABLE 1) is the address of the first variable in the Fuzzy data list, INPUT1 and INPUT2 are integer variables which contain the input values passed to the statement within your Divelbiss 32-bit BASIC routine, OUTPUT1 and OUTPUT2 are integer variables which will contain the return values from the *FUZZY* statement and STATUS is an integer variable which contains the status of the Fuzzy processing.

The first item to be addressed is the Fuzzy variable list. In all there are 106 integer values which are required for the Fuzzy Logic processing using the *FUZZY* statement. The 106 integers consist of the midpoint and sensitivity values for INPUT1, INPUT2, OUTPUT1, and OUTPUT2, and the 49 rules for each output. For clarification on the midpoint and sensitivity values see Figure 39 which shows the shape and numbering scheme for the membership functions for inputs and outputs. The seven (7) fuzzy sets will always be labeled 1 through 7 with 1 being left-most and 7 right-most. In the division of the inputs and outputs these may be labeled by you as NEGATIVE LARGE, NEGATIVE MEDIUM, etc. However, for processing purposes these are given numerical values. Additionally, as described in Figure 39, all fuzzy sets are triangular in shape and symmetric about the mid-point. Membership functions of this shape give the additional advantage of only requiring the mid-point and the distance between triangles (sensitivity) to fully describe the function.



Shape and Numbering Scheme for Inputs and Outputs using the FUZZY statement

The *FUZZY* statement can be utilized to solve your control problems once you have identified the problem, determined the inputs and outputs, divided these into 7 regions, and generated process logic. The following explanations will assume a working knowledge of the BEARBASIC programming language and thus only the portions of the code relating to the *FUZZY* statement will be addressed.

As stated, their are 106 integer data values which are required to successfully implement the *FUZZY* statement, and the variable declarations for these variables must be in successive order at the beginning of your BEARBASIC software. Because of this the simplest way to implement the data list is to declare all of the Fuzzy data as one large array, use the BEARBASIC DATA statement to enter the data, and then use the READ statement to enter the data into the declared variables. This minimizes the length of the software and makes the program easier to read and debug. Note: The Fuzzy data must be declared as one array for the *FUZZY* statement to operate correctly. Consider a problem requiring two (2) inputs and one (1) output with the following membership function shapes:

INPUT1	midpoint = 0
INPUT1	sensitivity = 15
INPUT2	midpoint = 0
INPUT2	sensitivity = 30
OUTPUT1	midpoint = 0
OUTPUT1	sensitivity = 4
OUTPUT2	midpoint = 0
OUTPUT2	sensitivity = 0

Additionally, the rule tables for OUTPUT1 and OUTPUT2 are described in Figure 40 and Figure 41. Note that since OUTPUT2 is not required that the rule table contains all zero's along with having zero midpoints and sensitivities. As an example, the rule table for OUTPUT1 states that if INPUT1 is in division 3 and INPUT2 is in division 6 then OUTPUT1 responds within division 3.

To implement this controller using the BEARBASIC language and the *FUZZY* statement the first step is the variable declarations.

Rule Table for Output 1

	INPUT1						
INPUT2	1	2	3	4	5	6	7
1	7	7	6	5	6	5	4
2	7	6	6	5	5	4	3
3	7	6	5	5	4	3	2
4	7	6	5	4	3	2	1
5	6	5	4	3	3	2	1
6	5	4	3	3	2	2	1
7	4	3	2	3	2	1	1

Rule Table for Output 2

	INPUT1						
INPUT2	1	2	3	4	5	6	7
1	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0

Thus, the portion of a Divelbiss 32-bit BASIC program related to the *FUZZY* statement for this example might be as follows:

```
100 INTEGER IN1, IN2, OUT1, OUT2, ST
110 INTEGER FUZLOOP
120 INTEGER FUZDAT(106)

140 DATA 0,15
160 DATA 0,30
180 DATA 0,4
200 DATA 0,0
```

```
200 DATA 7,7,7,7,6,5,4
220 DATA 7,6,6,6,5,4,3
240 DATA 6,6,5,5,4,3,2
260 DATA 5,5,5,4,3,3,3
280 DATA 6,5,4,3,3,2,2
300 DATA 5,4,3,2,2,2,1
320 DATA 4,3,2,1,1,1,1

340 DATA 0,0,0,0,0,0,0
360 DATA 0,0,0,0,0,0,0
380 DATA 0,0,0,0,0,0,0
400 DATA 0,0,0,0,0,0,0
420 DATA 0,0,0,0,0,0,0
440 DATA 0,0,0,0,0,0,0
460 DATA 0,0,0,0,0,0,0

FOR FUZLOOP = 0 TO 105
READ FUZDAT(FUZLOOP)
NEXT FUZLOOP
IN1 = 20
IN2 = 5
FUZZY ADR(FUZDAT(0)),IN1,IN2,OUT1,OUT2,ST
```

The return values from the statement are contained in variables OUT1 and OUT2 which can then be manipulated as required in the remaining portion of your Divelbiss 32-bit BASIC software. The status variable, ST, will be; 0 if the processing occurred correctly, 1 if OUT2 was beyond the range of an integer, 2 if OUT1 was beyond the range of an integer, and 3 if both outputs were beyond the range of an integer. When an output is beyond the range of an integer the return value from the *FUZZY* statement will be clamped at +32767 or -32767 depending on the actual sign of the output. The ST variable will contain a 0 in most circumstances and should only be checked in extreme cases.

A couple of things to note are that the data (all 0's) for OUTPUT2 are required even though this output is not being utilized and that the first value read into the data array is FUZDAT(0). Thus, the first value being passed to the *FUZZY* statement is the address of this variable (ADR(FUZDAT(0)).

The Fuzzy data to be read in must be in the order described above in the example routine. A summary of this order is provided below:

INPUT1	midpoint, sensitivity
INPUT2	midpoint, sensitivity
OUTPUT1	midpoint, sensitivity
OUTPUT2	midpoint, sensitivity

```
RULE OUTPUT1[IN1=1,IN2=1]   ...   RULE OUTPUT1[IN1=1,IN2=7]
RULE OUTPUT1[IN2=1,IN2=1]   ...   RULE OUTPUT1[IN1=2,IN2=7]
RULE OUTPUT1[IN3=1,IN2=1]   ...   RULE OUTPUT1[IN1=3,IN2=7]
RULE OUTPUT1[IN4=1,IN2=1]   ...   RULE OUTPUT1[IN1=4,IN2=7]
RULE OUTPUT1[IN5=1,IN2=1]   ...   RULE OUTPUT1[IN1=5,IN2=7]
RULE OUTPUT1[IN6=1,IN2=1]   ...   RULE OUTPUT1[IN1=6,IN2=7]
RULE OUTPUT1[IN7=1,IN2=1]   ...   RULE OUTPUT1[IN1=7,IN2=7]
RULE OUTPUT2[IN1=1,IN2=1]   ...   RULE OUTPUT1[IN1=1,IN2=7]
RULE OUTPUT2[IN2=1,IN2=1]   ...   RULE OUTPUT1[IN1=2,IN2=7]
RULE OUTPUT2[IN3=1,IN2=1]   ...   RULE OUTPUT1[IN1=3,IN2=7]
RULE OUTPUT2[IN4=1,IN2=1]   ...   RULE OUTPUT1[IN1=4,IN2=7]
RULE OUTPUT2[IN5=1,IN2=1]   ...   RULE OUTPUT1[IN1=5,IN2=7]
RULE OUTPUT2[IN6=1,IN2=1]   ...   RULE OUTPUT1[IN1=6,IN2=7]
RULE OUTPUT2[IN7=1,IN2=1]   ...   RULE OUTPUT1[IN1=7,IN2=7]
```

for a total of 106 data points (8 for midpoints and sensitivities, 49 for the rules for OUTPUT1, and 49 for the rules for OUTPUT2).

In summary, the *FUZZY* statement provides the simplest approach to the implementation of Fuzzy Logic controllers. Sophisticated control systems for a variety of applications can be implemented with very little programming using the Divelbiss 32-bit Controller.

A summary of the specifications for the *FUZZY* statement are provided below:

Number of Inputs:	2
Number of Outputs:	2
Number of Fuzzy Sets:	7
Fuzzy Set Shape:	Symmetric Triangular
Rules per Output:	49
Defuzzification Method:	Centroid
Approximate Execution Time:	0.5mS - 1.0mS
Number of Fuzzy Control Loops:	1

PLEASE NOTE THAT THESE SPECIFICATIONS ARE NOT FOR THE UNIVERSAL CONTROL PANEL BUT ONLY FOR THE *FUZZY* STATEMENT AVAILABLE IN THE DIVELBISS 32-BIT BASIC PROGRAMMING LANGUAGE. THE UNIVERSAL CONTROL PANEL IS A GENERAL PURPOSE CONTROLLER OF WHICH FUZZY LOGIC CONTROL IS ONE CAPABILITY. PLEASE CONSULT THE FACTORY FOR INFORMATION REGARDING THE IMPLEMENTATION OF MULTIPLE CONTROL LOOPS USING THE *FUZZY* STATEMENT.

Applications and Examples of Fuzzy Logic

Is *FUZZY* Right for Your Application?

The majority of areas in which the *FUZZY* statement would be best suited occur in single loop process control applications. Some examples might be temperature, climate, speed, position, pressure, and fluid depth, just to name a few. All of these applications require at most two (2) inputs and usually one (1) output and thus would be well suited for Fuzzy Logic control using the Divelbiss 32-bit Controller. Additionally, because the Divelbiss 32-bit Controller is a general purpose controller, other applications can be handled simultaneously along with your real-time control requirements, providing maximum flexibility in system design.

One necessity when designing a control system is understanding the performance requirements of the loop. If conventional control techniques such as PID (Proportional, Integral, Derivative) have resulted in undesired behavior of the system such as slow response, excessive overshoot, and/or poor disturbance response, the Fuzzy Logic control capabilities using the Boss 32 might be the solution. Additional considerations are the ease of tuning and the long-term reliability of your control solution. As described earlier, Fuzzy Logic provides an intuitive approach to controller tuning which is not available with other techniques such as PID. Finally, it has been proven that a tuned PID control loop is only good if the plant is reasonably stable over time. Any change in the plant such as weight, volume, etc. could result in erratic operation of the PID loop. However, Fuzzy Logic controllers are quite immune to variations in the plant dynamics, thus providing long-term reliability of the system without the need for extensive changes over the life of the system.

Fuzzy Logic Control Examples Using *FUZZY*

Motor Speed Control

An excellent, real-world example of using Fuzzy Logic is motor speed control. Speed control is a traditional problem of closed-loop control systems and is well understood by the majority of system designers. Fuzzy Logic control lends itself to this problem because this expertise can be mapped into the control system design. This is something which is not typically possible with traditional control design methodologies and demonstrates some of the exciting possibilities of this emerging technology in industry.

Problem Statement

The speed control problem is best defined as the tracking of the set-point speed of a motor with robust recovery from extreme set-point changes, external disturbances, and changes in plant dynamics.

Input / Output Definition

According to the problem statement the primary goal of the control system is to track the set-point speed of the motor. In order to accomplish this one must know how far from the set-point the system is and thus, a required input to the Fuzzy Logic controller is the current speed error defined as follows:

INPUT1: SPEED ERROR = (CURRENT SPEED) - (SET POINT)

Additionally, previous information about speed is critical. For instance, suppose at the current sample the speed turns out to be exactly at the set-point, but at the previous sample the speed was significantly below the set-point. Thus, in a short amount of time the speed increased significantly. Because no real system can stop instantaneously, it is quite probable that at the next sample the system will be well above the set-point. Therefore, the rate of change of speed between samples is necessary in addition to the speed error:

INPUT2: RATE OF CHANGE = (CURRENT SPEED) - (LAST MEASURED SPEED)

A logical control output for the system is the change in the drive signal to the motor. This is logical because at all moments the system speed must be increased or decreased from its present value in order to track the set-point speed:

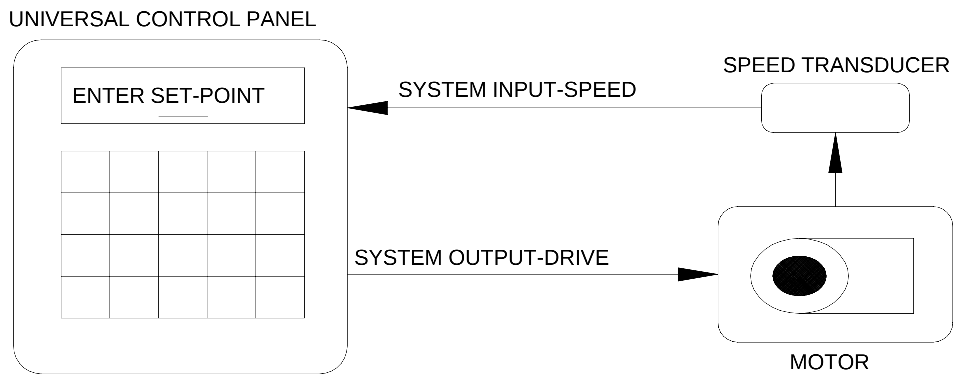
OUTPUT1: DRIVE SIGNAL CHANGE

Hardware Interface

Figure 42 describes the hardware design of this system. Notice that only one analog input, current speed, is required into the unit and not two as was explained previously in

Input/Output Definition. This is because the input/output definition for the Fuzzy Control is not necessarily the same as the input/output definition for the system. With some BEARBASIC programming, the set-point can be entered from the keypad. This setpoint can be compared with the current speed received from the transducer to arrive at INPUT1 required by the Fuzzy Control system. INPUT2 to the Fuzzy Control block can also be obtained through software programming from the current speed. Once the current speed is obtained, it can be compared to the previous sample to arrive at the rate of change. Once this calculation is made, the current speed may be saved as the old speed for use during the next sample.

Another important point is that the Fuzzy Logic control block has an output which is the change in the drive signal while the actual system output is simply the drive signal. With programming, it is possible to store the current drive signal and then add the Fuzzy Logic output to obtain the signal to transmit to the motor. This once again demonstrates that the input/output requirements of the Fuzzy Logic control block are not the input/output requirements of the system.



Hardware Interface ofr Motor Speed Control

Membership Functions

The next step is to determine the membership functions for both inputs and the single output. See Figure 39 for a review of the membership functions utilized with the *FUZZY* statement. The two values to determine for each input and output are the sensitivity (s = distance between triangle centers) and the midpoint. For all of the inputs and outputs of this system, it is clear that the midpoint value is zero (0). This is because the goal for INPUT1 (speed error) is to have a value of zero (0) with the actual value being above or below the set-point. INPUT2 (speed rate of change) can also be zero with the actual value being above or below this. This argument is also true for OUTPUT1. Thus, the only values left to decide are the sensitivities for each input and output.

Sensitivity selection is dependant upon your expertise of the problem/process and some understanding of Fuzzy Logic fundamentals. As described in Section 11.1.3, as the sensitivity for both inputs is decreased, the Fuzzy Logic controller becomes more sensitive to the error and rate of change, respectively. Therefore conservative values should be utilized in the initial attempt at selecting these values. A reasonable approach is to determine what would be considered a “large” value for each of the inputs/outputs and divide this number by three (3) to determine the sensitivity. Once again, in the initial design attempt it is best to enter larger values for both of these input sensitivities and a smaller output sensitivity to avoid erratic behavior. Remember, the controller can always be tuned for more aggressive behavior following the initial implementation. For purposes of demonstration, assume the set-point will be around 200rpms. Thus, to evenly divide the space, let a large error be 200, which makes the sensitivity for INPUT1 67. The sensitivity for INPUT2 depends on the sample time as described in Section 11.1.3. For purposes of demonstration assume this sensitivity is 20. OUTPUT1 is the change in the output drive signal, and in order to cover the whole space a reasonable sensitivity might be 67 just as INPUT1. However, to be conservative the first value to be implemented should be around 33. Once implemented, these values should be tuned in order to optimize the response of the system.

Process Logic

Figure 31 describes reasonable process logic for a speed control problem. If the speed error, INPUT1, is negative large, then regardless of the rate of change, INPUT2, change the output drive signal positive large. There are two important things to notice about the rule table: the closer to the zero (0) the speed error becomes, the more conservative the change in the output and the table is symmetric about INPUT1, only in the reverse direction.

Conservative behavior about the set-point is quite typical for most of the applications listed in Section 11.3.1 Is *FUZZY* Right for Your Application?. This is because abrupt changes in the output around the desired value will most likely cause oscillations. Thus, the further away from the set-point, the more aggressively the output control action and visa-versa as the input approaches the set-point.

		INPUT1 SPEED ERROR						
		1	2	3	4	5	6	7
	1	7	7	6	6	6	4	1
	2	7	7	6	5	5	3	1
	3	7	6	5	5	4	3	1
INPUT2 RATE OF CHANGE	4	7	6	5	4	3	2	1
	5	7	5	4	3	3	2	1
	6	7	5	3	3	2	1	1
	7	7	4	2	2	2	1	1

Rule Table for Speed Control Example

With regards to the symmetry of the rule table this is also quite typical for process control applications. As an example, if INPUT1 is negative small (3) and INPUT2 is positive medium (6) then the rule table says change the output negative small (3). The opposite situation is also true; if INPUT1 is positive small (5) and INPUT2 is negative medium then change the output positive small (5). Thus, as the table is being constructed, look for exactly opposite situations which will simplify the design.

Software Design

With the inputs and outputs, hardware interface, membership functions, and process logic determined, the only remaining step is to implement your controller using the Divelbiss 32-bit BASIC programming language. As described in Section 11.3.2.1.3, the only input to the controller is from the transducer which measures the input speed. This speed is accessible using the analog capabilities of the Boss 32. Specifically, with an analog board and the ADC statement the speed is available for manipulation in your program to generate INPUT1 and INPUT2 as shown in Section 11.3.2.1.2. With these inputs available and software similar to that described in Section 11.2, your controller can be implemented with all the advantages of Fuzzy Logic with minimal setup.

On-Off Control

On-Off control can be accomplished just as any process control application (See Section 11.3.2.1 Motor Speed Control - for more details) with a slight modification in the system output. Remember that a modification in the system output is not the same as a modification in the Fuzzy Logic Control output. The output of the Fuzzy Logic controller (OUTPUT from the *FUZZY* statement) can be compared with a previously determined value set by you in order to decide if the control should be On or Off. Thus, On-Off control may require use of a Divelbiss Corporation HDIO (High Density I/O) board in addition to the analog board in order to switch the output on and off.

Conclusion

Although not the solution for every application, Fuzzy Logic should be considered as a possibility in the design of every control system. To make this possible, Divelbiss Corporation has made an easy to use Fuzzy Logic statement available with its Divelbiss 32-bit Controller. The *FUZZY* statement provides the simplest approach to the implementation of Fuzzy Logic controllers. Additionally, because the Divelbiss 32-bit Controller is a general purpose programmable controller, other applications can be handled simultaneously along with your real-time control requirements, providing maximum flexibility in system design.

Chapter Six
Error Handling

.

There are several types of programming errors that can occur during the development cycle:

● Syntax errors. These occur when the programmer enters code which doesn't follow the required format of the language. The compiler catches these and displays an error message and the line number, making it simple to correct the problem. Examples of this include misspelled keywords (ie. PRNT instead of PRINT), incorrect number of arguments to a statement or function (ie. SETDATE 9,4,91 instead of SETDATE 9,4,91,4), and incorrect program format (ie. NEXT without corresponding FOR).

● Runtime errors caused by program implementation errors. These are mistakes made by the programmer in the coding of the program, that may or may not get caught by the Bear BASIC system while the program is running. An example of one that would get caught would be to declare an array of 10 elements and then try to access the 12th element, generating a "Subscript Out of Range" error; error checking would have to be enabled for this to get caught (see ERROR and NOERR). These errors can be extremely difficult to find and correct, especially if they occur in a section of code that is seldom executed.

● Runtime errors caused when an I/O device detects an error condition. An I/O error may or may not be important, depending upon the individual system.

● Logic errors in the design of the system. These can be the most difficult errors to correct, since they result from insufficient planning and incomplete understanding of the system operation. A fundamental logic error may not be detected until the system is in operation, but it may require that large portions of a program be rewritten.

In addition to detecting syntax errors during the program entry and compile phases, Bear BASIC provides facilities that allow the BASIC program to handle two types of errors at runtime: I/O errors and program errors. These are handled in different ways, so they are discussed separately.

The ERR function returns the latest error that was detected by Bear BASIC. Each time a new error occurs, it overwrites the previous error register value with the new error number. Each time that ERR is referenced, the error register is reset back to 0. The ERR function returns the following error numbers at runtime:

0	$0	No error
2	$2	Failure programming EEPROM
3	$3	EEPROM address out of range
10	$A	COM1 serial transmission timeout
11	$B	COM2 serial transmission timeout
12	$C	COM3 serial transmission timeout
13	$D	COM4 serial transmission timeout
16	$10	COM1 serial receive error (overrun, parity, framing)
17	$11	COM2 serial receive error (overrun, parity, framing)
18	$12	COM3 serial receive error (overrun, parity, framing)
19	$13	COM4 serial receive error (overrun, parity, framing)
256	$100	Undefined error
263	$107	Expression Error
264	$108	String Variable Error
266	$10A	Line Number Does Not Exist
267	$10B	Task Error

271	$10F	RETURN Without GOSUB
272	$110	Subscript out of Range
273	$111	Overflow
275	$113	Task Mismatch
276	$114	Function Error
277	$115	String Length Exceeded
280	$118	Illegal Print/Input Format
281	$119	String Space Exceeded
282	$11A	Illegal File Number
283	$11B	Improper Data to INPUT Statement
285	$11D	Data Statement Does Not Match Read
286	$11E	Failure Programming EPROM or EEPROM

I/O Errors

Some I/O devices detect errors periodically during their normal operation. On the Boss 32, the serial ports are the prime example of this; in an electrically noisy environment, they may detect parity and framing errors frequently. These are not fatal errors which warrant stopping the program's execution, but they are important enough that the program will need to process the error appropriately. I/O errors can be handled by the programmer in two ways: using the ON ERROR statement and checking the ERR function periodically. There are nine I/O errors that are detected with the Divelbiss 32-bit Controller system software:

2	$2	Failure programming EEPROM
10	$A	COM1 serial transmission timeout
11	$B	COM2 serial transmission timeout
12	$C	COM3 serial transmission timeout
13	$D	COM4 serial transmission timeout
16	$10	COM1 serial receive error (overrun, parity, framing)
17	$11	COM2 serial receive error (overrun, parity, framing)
18	$12	COM3 serial receive error (overrun, parity, framing)
19	$13	COM4 serial receive error (overrun, parity, framing)

The ON ERROR statement causes the program execution to proceed at a specified line number when an error is detected by one of the I/O functions. The following example demonstrates this:

```
100  INTEGER K
150  ONERROR 300                   ‘ Set up the error handler
200  FILE 0: K=GET                 ‘ Get a character from the console
210  FILE 6: PRINT CHR$(K);        ‘ Print it onto the onboard display
220  GOTO 200
300  FILE 6: PRINT “*** “; ERR     ‘ Error detected
310  GOTO 200
```

Line 150 sets up the error handler at line 300. If an error is detected when the GET function is called, then the program will jump to line 300. Line 300 will display the error number (returned by the ERR function), an jump back to line 200. Note that the error is not detected until an I/O operation is attempted; in this case, the GET function.

The ON ERROR statement has one important limitation: everything having to do with it must be in task 0. This includes the line containing the ON ERROR statement, the line referenced by the ON ERROR statement, and the I/O operations that may cause the error to be detected. If an I/O operation occurs in a task other than 0 and detects an error, then the program execution will continue without jumping to the line referenced by the ON ERROR statement; it will be as if no ON ERROR had occurred. If all of the serial I/O is performed within task 0, however, then ON ERROR can be quite useful. The ON ERROR approach to error handling generally works best in programs that perform simple serial I/O operations.

I/O errors can also be handled by checking the ERR value periodically while performing I/O. This allows the programmer more control over the error detection, it doesn't change the program flow like ON ERROR does, and it isn't limited to just task 0. This approach works well in more complex serial applications, such as the packet-based communications protocols used by many intelligent sensors. In this type of communications, it may only be necessary to check the error status at the end of each packet, instead of after each character.

```
100 INTEGER K
200  FILE 0: K=GET                  ‘ Get a character from the console
210  FILE 6                         ‘ Set up to print to onboard display
220  IF ERR=16 THEN PRINT “ Error “  ‘ Check for COM1 receive error
230  PRINT CHR$(K);                 ‘ Print the character
240  GOTO 200
```

This example just checks the value of ERR after every character is received, in line 220. If a receive error has occurred on COM1, then ERR will return a 16 ($10), causing line 220 to print “Error”.

Program Errors

Divelbiss 32-bit BASIC performs error checking while the program is running, if error checking is enabled (see ERROR and NOERR). If a program error occurs, Divelbiss 32-bit BASIC will normally print an error message to the console (COM1 serial port) and return to the command prompt. During the initial debugging this is an acceptable response. During system testing, or especially after the system is installed, this can be very inconvenient, for two reasons. First, if there isn't a terminal attached to COM1 then the error message will be lost. Second, stopping the program at the instant that the error is detected may leave the system in an unacceptable state.

To solve these problems, the Divelbiss 32-bit BASIC programmer can get access to the program error handler by using the INTERRUPT statement. By using INTERRUPT, the program can be made to jump to a task when a program error is detected; this task should be treated as a hardware interrupt task. This task can put the system into a safe state and store the error number in a variable (or perhaps in EEPROM). The task must either stop the program (with the STOP statement) or restart the program (by using CHAIN); a program error indicates that an unrecoverable problem has occurred, so it isn't possible to continue running the program.

```
100  INTEGER K
110  STRING A$(25)
150  INTERRUPT 2,1          ‘ Set up error handler task
200  A$=””
```

```
210  FOR K=1 TO 26
220  A$=CONCAT$(A$,CHR$(K+64))     ‘ Fill string with ASCII letters
230  NEXT K
240  PRINT “Done.”: STOP
500  TASK 1
510  FILE 6                        ‘ Display error message on onboard display
520  PRINT “Error “; ERR           ‘ Display error number
530  STOP                          ‘ Stop program execution
```

This example causes a program error to occur when, in line 220, it accesses the 26th character in A$, which is only declared to be 25 characters long. Line 150 has set up task 1 to handle program errors, so when the error occurs, task 1 prints the error number. When this program is run, it will display “Error 277”, which is the “String Length Exceeded” error. In general, the PRINT and FPRINT statements shouldn’t be used inside of an interrupt task (which includes this error handler task), because they re-enable the interrupts and let the multitasking to continue, either of which could cause the system to lock up, depending upon what the other tasks are doing. In this example, however, the rest of the program isn’t doing anything, so this won’t cause a problem.

Appendix A
Error Messages

The following error messages may be displayed by UMC BASIC on the console terminal. If the error is associated with a line number in the BASIC program, then a line number will be displayed before the error message. When an error occurs, UMC BASIC returns to the compiler prompt after displaying the error message.

`*** BAD INPUT. PLEASE RE-ENTER ***`
Displayed at runtime if the data entered in response to an INPUT statement doesn't match the arguments given. Just retype the input data properly. This error can be avoided by using the FINPUT statement.

`Data Statement Does Not Match Read`
Occurs when a READ or RESTORE is encountered, but no DATA statements have been found. All DATA statements must be before the first READ.

`Expression Error`
Occurs when a mathematical expression has been formatted incorrectly.

`Failure Programming EPROM or EEPROM`
Indicates that a byte didn't verify correctly after being programmed in the FLASH.

`File not Found`
FLASH file not found.

`File Verify Failure`
An error was detected after writing a file to the FLASH.

`Function Error`
Occurs when a program is compiled if the function had an argument in the wrong mode, or if the wrong number of arguments were given for the function. At runtime, a FUNCTION ERROR may appear if the argument to a function is out of range, for example, if the square root of a negative number is taken.

`Illegal Direct Command`
Occurs if an unknown direct command is entered.

`Illegal File Number`
On EPROM LOAD [*filenum*] command, this error occurs when the specified [*filenum*] is not present on the EPROM.

`Illegal Print/Input Format`
Is displayed at runtime if the format given in an FPRINT or FINPUT statement does not match the number of arguments in the FPRINT/FINPUT statement (if there are more things to be printed/input than there are formats), or if the format specified is not correct.

`Improper Data to INPUT Statement`
Occurs when data is being read from a file and the data doesn't match the arguments of the INPUT statement.

`Insufficient EPROM Space`
The user's EPROM is full.

`Line Number Does Not Exist`
Occurs if a line number is referenced in a GOTO, GOSUB, or IF statement and that line cannot be found.

`Line Number Error`
Occurs when an illegal line number is used. Line numbers must be integers from 1 to 32,767.

`Misuse of String Expression`
Occurs when a string expression is used in a place where a real or integer expression is needed, or if a real or integer expression is used in lieu of a string.

`No Compiled Code`
Occurs when the GO command is issued but there is no compiled code to execute. If any change is made to the program, it must be recompiled.

`Not Enough Memory for Network Register`
This occurs when there is not enough variable memory to allocate for the network register. See the SETOPTION VARSIZE direct command for more information.

`Not Enough Memory to Compile Program`
This can be issued under two circumstances. First, the compiled code and variable space may be too large; try shrinking arrays, compiling with NOERR, and using subroutines to eliminate duplicate code. Second, the source code may be too large, which doesn't leave enough room for the compiler's temporary storage; try removing comments to save space.

`Not Enough Variable Memory to Compile Program`
This occurs when the variable memory needed exceeds the allocated variable memory. See the SETOPTION VARSIZE direct command for more information.

`Overflow`
Indicates that a math overflow occurred, such as divide by 0, LOG of a negative number, etc.

`PROM is incompatible with this Compiler Version`
This occurs when a program on the user's EPROM is executed on power up or from a CHAIN statement. The compiled code on the user's FLASH was created with a previous version of the UMC BASIC software. Clear the FLASH, load the source code, recompile the program, and save it.

`Quote or Parenthesis Mismatch`
Occurs when a program is being entered and a line with an odd number of parenthesis or double quotes is found.

`RETURN Without GOSUB`

Occurs during program execution if a RETURN is encountered when no GOSUB is active. All GOSUBs must have a corresponding RETURN.

`Statement Formed Poorly`

Occurs when a statement is entered that does not conform to the syntax rules for UMC BASIC.

`Statement Ordering Error`

Occurs if an INTEGER, REAL, or STRING statement appears after executable statements in the program.

`String Length Exceeded`

Occurs when a string exceeds 127 characters or a string variable exceeds the maximum size assigned to it in the STRING statement.

`String Space Exceeded`

Occurs if one line of the program requires too many string temporaries to evaluate. Try splitting the line into several simpler ones.

`String Variable Error`

Displayed when a variable is used incorrectly, for example, if a string is used as a real or integer.

`Subscript out of Range`

Occurs if the subscript of a dimensioned variable exceeds the range assigned in an INTEGER or REAL statement.

`Task Error`

can be caused by any of the following:
receiving a hardware interrupt which was vectored to a non-existent task.
- trying to RUN a non-existent task.
- trying to RUN TASK 0.
- trying to use more than 31 TASK statements.

`Task Mismatch`

Internal task error.

`Too many FOR statements`

More than 255 FOR statements found.

`Too Many Variables`

More than 128 variable declarations found.

`Undefined error`

Internal compiler error.

`Undefined Variable`

Occurs during compilation if a variable is encountered which was not defined in an INTEGER, REAL or STRING statement.

Appendix B
ASCII Character Table

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	NUL	32	20		64	40	@	96	60	`
1	01	SOH	33	21	!	65	41	A	97	61	a
2	02	STX	34	22	“	66	42	B	98	62	b
3	03	ETX	35	23	#	67	43	C	99	63	c
4	04	EOT	36	24	$	68	44	D	100	64	d
5	05	ENQ	37	25	%	69	45	E	101	65	e
6	06	ACK	38	26	&	70	46	F	102	66	f
7	07	BEL	39	27	‘	71	47	G	103	67	g
8	08	BS	40	28	(	72	48	H	104	68	h
9	09	HT	41	29	)	73	49	I	105	69	i
10	0A	LF	42	2A	*	74	4A	J	106	6A	j
11	0B	VT	43	2B	+	75	4B	K	107	6B	k
12	0C	FF	44	2C	,	76	4C	L	108	6C	l
13	0D	CR	45	2D	-	77	4D	M	109	6D	m
14	0E	SO	46	2E	.	78	4E	N	110	6E	n
15	0F	SI	47	2F	/	79	4F	O	111	6F	o
16	10	DLE	48	30	0	80	50	P	112	70	p
17	11	DC1	49	31	1	81	51	Q	113	71	q
18	12	DC2	50	32	2	82	52	R	114	72	r
19	13	DC3	51	33	3	83	53	S	115	73	s
20	14	DC4	52	34	4	84	54	T	116	74	t
21	15	NAK	53	35	5	85	55	U	117	75	u
22	16	SYN	54	36	6	86	56	V	118	76	v
23	17	ETB	55	37	7	87	57	W	119	77	w
24	18	CAN	56	38	8	88	58	X	120	78	x
25	19	EM	57	39	9	89	59	Y	121	79	y
26	1A	SUB	58	3A	:	90	5A	Z	122	7A	z
27	1B	ESC	59	3B	;	91	5B	[	123	7B	{
28	1C	FS	60	3C	<	92	5C	\	124	7C	|
29	1D	GS	61	3D	=	93	5D	]	125	7D	}
30	1E	RS	62	3E	>	94	5E	^	126	7E	~
31	1F	US	63	3F	?	95	5F	_	127	7F	DEL

ASCII, which is an acronym for American Standard Code for Information Interchange, is one of the main ways in which computers process text data. Each printable character is represented as a number between 32 and 126 ($20 and $7E). The numbers between 0 and 31 ($00 and $1F) form the “control codes”, such as carriage return (“CR”), backspace (“BS”), and horizontal tab (“HT”).