GALFO SYTEMS

INTEGER BASIC COMPILER

FOR THE APPLE II MICROCOMPUTER

INSTRUCTION MANUAL

GALFO SYSTEMS´ Integer BASIC compiler (IBC) greatly enhances the computing power of the Apple II microcomputer since it is a true compiler for Apple´s interpreted language. This allows the user to take advantage of the fast execution speed offered by a compiled language, yet maintain the advantage of interactive program development and testing available only in an interpreted language. IBC extends the scope of Integer BASIC by removing many programming restrictions found in Apple´s version of the language. For example, the restriction of string length to 255 characters and the limitation of what characters can be stored in a string have been removed. The IBC system offers the advanced programmer a host of new features which make Integer BASIC a truely fine development tool for the Apple II.

IBC translates the source program into either of two forms of object code: pure GSL code or mixed 6502 and GSL code, the latter being generated to optimize execution speed. GSL is a computer code specially designed for the 6502 microprocessor and Integer BASIC. It is highly efficient and offers much higher execution speed on a 6502 processor than other coding schemes, such as Pascal´s P-code. Pure GSL code is very compact, with typical programs producing object codes 20 to 50 percent shorter than the source program code. The compiler´s generation of mixed 6502 and GSL code optimizes program execution for maximum speed, but increases the object code length by 60 to 300 percent over pure GSL code. Mixed code will execute typical BASIC programs 20 to 50 percent faster than pure GSL code, although certain simple statement types will execute several times faster. The choice of which code is produced by the compiler is left to the user. In either form, compiled programs will execute a factor of 7 to greater than 50 times faster than interpreted BASIC, making Integer BASIC applicable to many programming tasks which normally could be written only in machine code.

A partial list of specifications for the compiler and its run-time system is given on the back of this sheet. The compiler package is provided on two 16-sector disks (compatible with DOS 3.3). The system runs in either 32 or 48 K of memory. The language card is not required. Apple´s Integer BASIC (either in the ROM or in the language card) is necessary for the compiler, but not required at run-time. System documentation is provided in a 30 page instruction manual. Many user services are provided by GALFO SYSTEMS including a user news letter, a disk replacement service, system notes and patches, and a software performance reporting form to provide user feedback and suggestions.

Price: $ 149.50 (includes First Class postage, Calif. add 6.5 %
                  sales tax, Foreign add $ 5.00 air mail)

* Apple is a registered trademark of Apple Computer Co.

PROGRAM SPECIFICATIONS

PROGRAM STATEMENTS:	IBC supports all of Apple´s Integer BASIC syntax in a program except the LIST, TRACE, and DSP statements.
STRING LENGTH:	Up to 32767 characters, all 256 character codes allowed. Input of lower-case ASCII and long strings via the INPUT statement.
NUMERICAL OPERATIONS:	16-bit signed aritmetic, overflow allowed.
DISK SUPPORT:	Standard DOS 3.3 commands supported via the PRINT ctrl-D sequence. Disk I/O of long strings and lower-case via text (T) files.
ADDITIONAL FUNCTIONS:	CHR$(0,expr) numerical to string conversion. GET(0) single character input function. KEY(0) keyboard sampling function.
ADDITIONAL STATEMENTS:	HOME, CLEAR, INVERT, NRML, FLASH, FULL, MIXED, LO, HI, H2, POINT, LINE, and SHAPE.

HOME clears the screen and homes the cursor; CLEAR clears from cursor to end of line; INVERT, NRML, & FLASH set screen display mode; FULL & MIXED set graphics display mode; LO, HI, & H2 set the plotting mode for Lo-Res, Hi-Res or Page 2 Hi-Res graphics; POINT, LINE, & SHAPE determine the Hi-Res plotting mode for the PLOT statement.

VARIABLE EQUIVALENCE:	Compiler allows sets of variables to share the same memory locations, if desired.
OBJECT CODE:	Object code produced by the compiler is relocatable to any 6502 page bourdary by a simple memory move. Object code can be saved on disk as a BRUNable file with or without the run-time system.
RUN-TIME SYSTEM:	3.5 K of code, relocatable to any page boundry by a simple memory move. At run-time, only the monitor ROM is necessary; i.e. once compiled, Integer BASIC programs will run on any Apple II.
COMPILING SPEED:	Typically 50 to 200 BASIC lines per second. Compiler written in 6502 machine code.
EXECUTION SPEED:	Program BM7: 2.9 sec. (KILOBAUD Oct. 1977). FOR - NEXT loop to 1000: 0.16 sec. Variable assigned to a constant: 20 micro-sec.

Your IBC SYSTEM disk is provided with 3 different “HELLO” programs, each one designed to make the booting process as convenient as possible for different Apple II / II Plus systems. Follow one of the 3 procedures listed below to select the proper HELLO program for your Apple II. First, boot your SYSTEM DISK, then:

1) For a standard Apple II or APPLE II Plus with an Integer BASIC ROM (firmware) card:

```
>LOAD HELLO APPLE II
>SAVE HELLO
```

2) For a standard Apple II with a language card:

```
>LOAD HELLO APPLE II & RAM CARD
>SAVE HELLO
```

The following is optional, but recommended if you´d like
to have access to APPLESOFT in your language card.
Load your DOS 3.3 SYSTEM MASTER in drive 2 (see note below).

```
>BLOAD FPBASIC,A$1000,D2
>BSAVE FPBASIC,A$1000,L$3000,D1
```

3) For an Apple II Plus with a language card:

```
DELETE HELLO
LOAD APPLE II PLUS & RAM CARD
SAVE HELLO
```

Load your DOS 3.3 SYSTEM MASTER in drive 2 (see note below).

```
BLOAD INTBASIC,A$1000,D2
BSAVE INTBASIC,A$1000,L$3000,D1
```

After your have completed the appropiate instructions, re-boot your IBC SYSTEM DISK. HELLO is automatically run by DOS when you boot your SYSTEM disk. Running HELLO “configures” your system, setting memory pointers so that the compiler (IBC) will function properly. When booting from another disk, you must manually RUN HELLO before using the compiler. The compiler (IBC) issues the error message: IMPROPER SYSTEM CONFIGURATION if you attempt to compile a program without running HELLO. Please read chapter 3 of the manual and any update notes before you begin using the compiler.

NOTE: To copy either file FPBASIC or INTBASIC from the DOS 3.3 SYSTEM MASTER to your IBC SYSTEM DISK using a single disk drive, you will have to swap disks and drop the ´D2´ (drive 2) specification from the BLOAD command line.

Contents

PREFACE

Welcome to the Integer BASIC Compiler family of users! I´m sure that you´ll enjoy this fine software product and find many uses for it. It is our aim to bring you the highest quality software for your Apple II microcomputer. The development of this compiler and its associated run-time system required a massive programming effort to complete and is, to the best of our knowledge, free from defects. You, the end user, must realize that any program as complex as a compiler can´t be guaranteed to be bug-free since its operation depends on the source code that it compiles. For this reason, a Software Performance Report Form is included in Appendix D to bring to our attention any operating bugs so they can be corrected in later revisions of the compiler or run-time system.

The run-time system file, GSL.SYS, is necessary for operation of your compiled (object form) program, and thus would have to be included in any program developed under the compiler. If you plan to market such programs, you should contact us and request a distribution license for GSL.SYS. The cost of an application form for such a license is $5.00 to cover our processing costs.

The computer code provided on both disks and this manual are protected by Federal Copyright Law. Copying part or all of this package violates that law and could result in criminal prosecution.

A software registration form is included in this package. You should take the time to send it in so that you´ll be included on our user mailing list and get prompt response to your requests for information, updates, etc. A self-addressed, stamped envelope is provided for this purpose. I hope you enjoy your new Integer BASIC compiler!

Christopher Galfo

GALFO SYSTEMS

* Apple and Applesoft are registered trademarks of Apple Computer Co. Cupertino, CA.

- CHAPTER 1 -

INTRODUCTION

The Integer BASIC compiler (IBC) greatly enhances the computing power of the Apple II microcomputer since it is a true compiler for Apple´s interpreted language. This allows the user to take advantage of the fast execution speed offered by a compiled language, yet maintain the advantage of interactive program development and testing available only in an interpreted language. The IBC also extends the scope of Integer BASIC by removing many programming restrictions found in Apple´s version of the language. For example, the restriction of string length to 255 characters and the limitation of what characters can be stored in a string have been removed. The IBC system offers the advanced programmer a host of new features which make Integer BASIC a truely fine development tool for the Apple II.

The compiler´s operation is fairly simple. The source program is written with the aid of Apple´s interpreter, which contains a line oriented editor. Each line is checked for proper language syntax as it is typed in, and errors are flagged so they can be corrected immediately. When the program is complete, it can be run on the interpreter, and any execution problems can be easily resolved. When the program is ready to be compiled, IBC is run using the procedure described in chapter 3. Programs take only a few seconds to compile since much of the compiler is written in machine code utilizing efficient algorithms.

IBC translates the source program into either of two forms of object code: pure GSL code or mixed 6502 and GSL code, the latter being generated to optimize execution speed. GSL (Galfo Stack Language) is a computer code specially designed for the 6502 microprocessor and Integer BASIC. It is highly efficient and offers much higher execution speed on a 6502 processor than other coding schemes, such as Pascal´s P-code or XPL0´s I2L code. GSL gets its speed by data exchange and manipulation on the 6502 stack, its direct linking technique, the use of special op-codes to handle common BASIC statement sequences, and numerous programming innovations that improve character string manipulation and 16-bit arithmetic. Pure GSL code is very compact, with typical programs producing object codes 20 to 50 percent shorter than the source program code. The compiler´s generation of mixed 6502 and GSL code optimizes program execution for maximum speed, but increases the object code length by 60 to 300 percent over pure GSL code. Mixed code will execute typical BASIC programs 20 to 50 percent faster than pure GSL code, although certain simple statement types will execute several times faster. The choice of which code is produced by the compiler is left to the user, since many applications are concerned with memory usage rather than absolute maximum execution speed. In either form, compiled programs will execute a factor of 7 to greater than 50 times faster than interpreted BASIC, making Integer BASIC applicable to many programming tasks which normally could be written only in machine code. Appendix B gives a comparison of execution times for several standard benchmark programs run under the IBC/GSL system and Apple´s BASIC interpreters.

The run-time system provides the environment necessary for the execution of GSL code. It occupies a little over 3K of memory and is completely relocatable (to any 6502 page boundary) by a simple memory move. It is normally loaded below DOS in the high end of memory starting at location $8800 for a 48K system. The operation of the GSL run-time system is

completely transparent to the user. The reason for this is that each program developed under the compiler contains a built-in disk loading routine for the run-time system in case it is not found in memory when execution is begun. This feature allows object programs to be stored on the disk with or without the run-time system as part of the object file, saving
disk with or without the run-time system as part of the object file, saving
a lot of valuable disk space.

This manual provides the information needed to use the Integer BASIC compiler and GSL run-time system. A description of the language is provided in chapter 2, which should be read even if you very familar with Integer BASIC. The details of compiling and executing a program are given in chapter 3. Chapter 4 describes the advanced programming features available in the compiled Integer BASIC language that are not provided in Apple´s (interpreted) version.

The information contained in this manual is believed to be accurate. It is recommended that independent verification of the IBC/GSL system operation be used in applications that may not be adequately described in the manual. The Software Performance Report Form (provided in appendix D) should be used to report errors found in this documentation.

- CHAPTER 2 -

THE INTEGER BASIC LANGUAGE

This chapter describes the Integer BASIC programming language as implemented by the IBC/GSL system. It is assumed that you are familiar with BASIC and have read Apple Computer Inc.´s publication “APPLE II BASIC PROGRAMMING MANUAL” (#A2L0005X).

The IBC/GSL system is designed to be completely compatible with Apple´s version of the language. Numerous extensions have been added to the language, but these additions are all within the legal syntax bounds of the language. It is generally true that all programs written to be run by Apple´s interpreted BASIC can be compiled and executed by the IBC/GSL sytem. The reverse is also true, if the programmer chooses not to use the language extensions offered by the IBC/GSL system. The purpose of this chapter is to document each BASIC statement type and point out any differences that may exist between the interpreted and compiled implementation of the language.

2.1 NUMERICAL MANIPULATION.

Integer BASIC allows numerical values in the range -32768 to +32767 to be calculated by a numerical expression. An expression (henceforth referred to as “expr”) is allowed in almost all statement types. It consists of an arrangement of variables, intrinsic functions, and relation operators that result in a single value. Rules of precedence are used to determine the order in which an expression is evaluated. In order of highest to lowest priority, the evaluation order is as follows:

1. ( ) (expressions enclosed in parentheses).

2. = # (string relation operators).

3. NOT and - (the minus sign).

4. ^ (exponentiation operator).

5. * / MOD (multiplication, division and mod operators).

6. + - (addition and subtraction operators).

7. > < = >= <= # <> (relation operators).

8. AND (logical AND operator).

8. AND (logical AND operator).

9. OR (logical OR operator).

Expressions are evaluated from left to right when two or more opeations have equal priority of precedence. The relation and logical operators always return the value 0 for FALSE and 1 for TRUE. In evaluating NOT, AND, or OR, any value that is not zero is considered TRUE. These conventions follow Apple´s exactly.

The BASIC language contains several intrinsic functions which are allowed to appear in an expression. Here is a brief description of each of these functions:

ABS (expr)	Returns the absolute value of an expression.
ASC (string)	Returns the ASCII value of the first character of a string.
LEN (string)	Returns the length of a string.
PEEK (expr)	Returns the byte value located at the memory location specified by expr.
PDL (expr)	Returns the value, from 0 to 255, of the APPLE´s game paddle specified by expr. Expr should be in the range 0 to 3.
RND (expr)	Returns a random number between 0 and expr-1 if expr is positive and 0 to expr+1 if expr is negative.
SCRN (expr1,expr2)	Returns the lo-res screen color, from 0 to 15, of the screen location specified by (expr1,expr2)
SGN (expr)	Returns the value -1 if expr is less than zero, 0 if expr is zero, and +1 if expr is greater than zero.

In addition to these intrinsic functions the compiler also recognizes three additional functions, CHR$(0,expr), GET(0), and KEY(0) as described in chapter 4.

The GSL run-time system allows all numerical operations to “overflow” without issuing an error message. When an operation results in a value outside the 16-bit arithmetic limitation, only the last 16-bits of the value are retained. Also, division by zero or MOD zero are not flagged as an error. The result of N/0 or N MOD 0 is returned as N.

2.1.1 NUMERICAL ASSIGNMENT STATEMENT

```
    LET var = expr (the keyword “LET” is optional).
```

This statement assigns the value of an expression to a variable (var). The variable can be a simple variable or an array element. Examples:

```
    10 LET A = (X+Y)*NUM
    20 TIME(I+1) = A(I)+B(J)
```

In the second example (line 20), TIME, A, and B are integer arrays and the optional “LET” is omitted.

2.1.2 NUMERICAL OUTPUT

```
        PRINT expr1;expr2;...etc.
```

This statement prints the value(s) (in decimal for numerical expressions) of an expession or list of expressions on the output device. If a semicolon (;) separates two items in the list, then no space is printed between them. If a comma (,) separates two items, then an automatic 8 position tab is generated. If the list is terminated by a semicolon, then no carriage-return is generated at the end of the printed line. Examples:

```
    10 PRINT X,Y
    20 PRINT J+2,A(J/3);0;
```

Numerical expressions and strings can be mixed in the PRINT statement and described in section 2.2.2.

2.1.3 NUMERICAL INPUT

```
    INPUT var1,var2,...etc.

    INPUT “text”,var1,var2,...etc.
```

This statement reads numbers in decimal from the input device and stores each value in the variable(s) var1 to varN listed. A question-mark is printed as a prompt for numerical input. Input values should be separated by commas or carriage returns. If the second form of the statement is used, text is first printed (with no carriage-return). The GSL run-time system allows numbers in the range -65535 to 65535 to be converted on input. Numbers outside this range or non-numerical input will generate an error message with a “?” to cue the reentry of data. Examples:

```
    10 INPUT X,Y,Z
    20 INPUT “X=”,X
```

2.2 STRING MANIPULATION

The integer BASIC language allows extensive high-speed manipulation of character data in the form of strings. Under the GSL run-time system, string length is not limited to 255 characters as it is with Apple´s interpreted version of the language, but can extend to 32767 characters. (note: in some cases strings of even greater length can be manipulated). In addition, the run-time system allows any data byte (0 to 255) to be manipulated as part of a string, a feature which can be exploited by an advanced programmer.

In integer BASIC, string variables (hence refered to as “var$”) must be dimensioned to reserve memory space for character storage. This is
In integer BASIC, string variables (hence refered to as “var$”) must be dimensioned to reserve memory space for character storage. This is
covered in chapter 4, section 4.1.

A string can be represented by a var$, a var$(expr), a var$(expr1,expr2), or “text”. Here is an explanation of each representation:

var$	When a string variable is used without a subcript, all characters of the string are represented.
var$(N)	When used with a single subcript, all characters from the Nth. to the end of the string are represented.
var$(N,M)	When used with two subscripts, all characters from the Nth. to the Mth. are represented. (note: M must be >= N for proper operation).
“text”	The string is a constant, consisting of the characters enclosed in quotes.

Note: N and M can be expressions, but should be greater than zero and less than or equal to the string length for proper operation. No error checking is performed by the run-time system to assure that M or N are within their proper bounds.

2.2.1 STRING ASSIGNMENT STATEMENT

```
    LET var$ = string (the keyword “LET” is optional).
    LET var$(N) = string
```

This statement copies the characters of a string to a string variable (var$). If the second form of the statement is used, characters of the string will be copied, but they will be added to var$ starting at the Nth. character of var$. In this case, the length of var$ will be N-1 plus the length of the string on the right of the equal sign. Example:

```
    10 A$=”ABC”
    20 B$=”WXYZ”
    30 A$(LEN(A$)+1)=B$(2,4)
```

If these three statements were executed in sequence, A$ would be ABCXYZ. This is an example of string concatenation.

2.2.2 STRING OUTPUT

```
    PRINT string1;string2;...etc.
```

The PRINT statement, when used with strings, prints the characters of a string on the output device. The semicolon and comma separaters function the same with strings as with numerical output. Numerical exrpessions and strings can be freely mixed in the PRINT statement. Examples:

```
    10 PRINT “X= “;X;” Y= “;Y
    20 PRINT A$;B$(2,4)
```

2.2.3 STRING INPUT

```
    INPUT var1$,var2$,...etc.

    INPUT “text”,var1$,var2$,...etc.
```

This statement reads characters from the input device and stores them in the corresponding string variable(s) in the list. There is no prompt character printed for string input. The end of each string is indicated by a carriage-return on the input device. The text enclosed in quotes is printed prior to reading the first string variable if the second form of the INPUT statement is used. The GSL run-time system does not use the standard Apple II “GETLN” monitor routine when receiving characters for string input, but instead, it uses an improved version of this routine. This allows strings of length greater than 255 to be read directly. The maximum number of characters that can be read is determined by the space available to the string variable (i.e. its dimensioned size). Also, lower-case characters being read are not converted to upper-case as is normally done in Apple BASIC´s INPUT statement. The user will find this implementation of the INPUT statement very useful for disk input or when using a keyboard that generates lower-case ASCII characters. Examples:

```
    10 INPUT A$,B$
    20 INPUT “YOUR NAME, PLEASE? “,NAME$
```

As with the PRINT statement, numerical and string input lists can be mixed in a single INPUT statement. If input lists are mixed in this manner, the string and numerical data read must be separated by carriage-returns.

2.3 BRANCH, LOOP, AND OTHER STATEMENTS

This section covers BASIC´s branch and loop construction statements. Also, several miscellaneous BASIC statements: TAB, VTAB, PR#, IN# and POKE are briefly documented.

2.3.1 UNCONDITIONAL BRANCHES

```
GOTO ln# or GOTO expr

GOSUB ln# or GOSUB expr
```

The BASIC statement GOTO causes program execution to continue at the line number (ln#) specified. If the second form of the GOTO (i.e. GOTO expr) is used, program execution continues at the line number equal to the value of expr. GOSUB is similar to GOTO, except the address of the next statement following the GOSUB is saved on a stack. The two forms of the GOTO and GOSUB statements are compiled differently. In the case of GOTO ln# and GOSUB ln#, the compiler checks the destination line number of the branch and supplies the proper address to the object code. When the GOTO expr or GOSUB expr is used, the compiler can´t compute or check the destination line number and must supply a reference table of all program line numbers to the object code so that the branch address can be determined at run-time. All GOTO expr or GOSUB expr statements in the program share the same reference table. The compiler issues the warning message:

```
IBC WARNING: CAN´T OPTIMIZE GOTO, GOSUB
```

when it encounters the first GOTO expr or GOSUB expr in a program, since the reference table can add substantial length to the compiled code (4 bytes for each line number in the program).

2.3.2 RETURN and POP statements

The RETURN statement causes program execution to return to the statement following the most recently executed GOSUB statement. If there was no corresponding GOSUB executed, the run-time system issues the error message: “RET ERR”. The GSL system allows GOSUB levels to be stacked a maximum of 24 deep, in contrast to Apple´s interpreter, which allows 16 levels. If the limit of 24 stack levels is exceeded, a “SUB ERR” error message is generated. The POP statement, like RETURN, removes one level from the GOSUB return address stack. The effect is the same as executing a RETURN, except program execution continues at the statement following the POP statement.

2.3.3 THE CALL STATEMENT

```
CALL expr
```

The CALL statement causes program execution by the 6502 microprocessor to continue at the absolute memory address given
by the value of expr. Program execution (in GSL code) continues at the statement following the CALL statement after an RTS (return from subroutine) instruction is executed by the machine code subroutine called.

2.3.4 The END statement

The END statement terminates program execution and restores page zero of memory. The GSL run-time system terminates by a jump to the Apple soft-start address: $E003.

2.3.5 CONDITIONAL BRANCHES

```
IF expr THEN ln#

IF expr THEN statement
```

The IF expr THEN ln# construction tests the value of expr and causes program execution to continue at the line number specified if expr is TRUE (i.e. non-zero). In its alternate form, IF expr THEN statement, the statement following the THEN is executed if expr is TRUE; if FALSE (i.e. zero) then this statement is skipped and program execution continues at the next statement. Example:

```
10 IF A>0 THEN 40
20 IF NOT A THEN PRINT “A IS ZERO” : GOTO 50
30 PRINT “A IS LESS THAN ZERO” : GOTO 50
40 PRINT “A IS GREATER THAN ZERO”
50 ...
```

The IBC compiles IF expr THEN ln# the same as IF expr THEN GOTO ln#, so the programmer may prefer to use the later construction for clarity. The
The IBC compiles IF expr THEN ln# the same as IF expr THEN GOTO ln#, so the programmer may prefer to use the later construction for clarity. The
construction IF expr THEN GOSUB ln# is specially compiled for high speed execution since it is a commonly used BASIC statement sequence.

2.3.6 FOR and NEXT statements

```
FOR var=expr1 TO expr2 STEP expr3

NEXT var1,var2,...etc.
```

The FOR and NEXT statements are used together to cause repeated execution ofa sequence of program steps. The FOR statement initalizes var to the value of expr1 and will step var by the value of expr3 on each loop iteration until the value of expr2 is exceeded. If the “STEP expr3” part of the FOR construction is omitted, a step value of +1 is assumed. The NEXT var statement causes a loop back to the statement following the corresponding FOR statement if the loop is to be repeated (i.e. the value of var hasn´t surpassed expr2 after being steped by expr3). If the FOR loop is complete, then program execution continues at the statement following the NEXT statement. The construction: NEXT var1,var2,...etc. is an abbreviation for its equivalent meaning of separate NEXT statements: NEXT var1 : NEXT var2 : ... etc. It should be noted that the statements in a FOR - NEXT loop are always executed at least once regardless of the values of expr1 or expr2.

Examples:

```
     10 FOR I=1 TO 10
     20 PRINT I,I^2
     30 NEXT I

     50 FOR I=100 TO 10 STEP -10
     60 FOR J=1 TO 2
     70 PRINT “I IS “;I;” J IS “;J
     80 NEXT J,I
```

The GSL run-time system allows a maximum of 14 FOR loops to be active at any given time during program execution. If a NEXT statement is executed with no corresponding FOR, the run-time error message “NXT ERR” is issued.

2.3.7 SCREEN OUTPUT CONTROL STATEMENTS

```
     TAB expr

     VTAB expr
```

The TAB and VTAB statements are used set the position (on the Apple´s video screen) where the next character will be printed. These statements also function when using certain printers. TAB expr sets the horizontal position on a line, 1 to 40, whereas VTAB expr sets the line on the screen, 1 to 24, 1 being the top of the screen.

2.3.8 I/O DEVICE SELECTION STATEMENTS

```
     PR# expr

     IN# expr
```

The PR# expr statement selects the output device used for printing. The device selected must contain a software driver of its own and be located in a peripheral slot 1 to 7. The value of expr (should be 0 to 7) selects the device, with a value of zero selecting the Apple´s video screen. IN# is similar to PR#, except that the input device is selected, with IN# 0 being the Apple´s keyboard. The GSL run-time system does not disconnect DOS when excuting a PR# or IN# , as does Apple´s BASIC.

2.3.9 DIRECT MEMORY STORAGE STATEMENT

```
     POKE expr1,expr2
```

The POKE statement causes the byte value (0 to 255) of expr2 to be stored in the absolute memory location specified by the value of expr1. The GSL system stores only the least significant byte of expr2, thus, its value can be greater than 255 without generating an error.

2.4 LO-RES GRAPHICS

Apple Lo-Res (color) graphics is directly supported by BASIC. The Lo-Res graphics screen has 40 positions horizontally (numbered 0 to 39) and 48 vertical positions (numbered (0 to 47 - with 0 being the top of the screen). There are 16 colors (numbered 0 to 15) available. Many of the Lo-Res statements described in this section can be used in the Hi-Res graphics supported under GSL as is documented in chapter 4, section 4.5.

2.4.1 SETTING AND CLEARING COLOR GRAPHICS

```
GR

TEXT
```

The GR statement sets the Apple´s screen to the color graphics mode and clears the graphics screen. The mixed graphics mode is set (i.e. 40 by 40 screen - with the 4 bottom lines for text). After a GR is executed, COLOR is set to zero. The TEXT statement clears the graphics mode and sets the usual 24 line by 40 character full-screen text mode. After TEXT is executed, printed output will appear at the last line on the screen, as if a VTAB 24 were also executed.

2.4.2 CHANGING COLOR

```
COLOR= expr
```

The COLOR= expr changes the color used in plotting to the value of expr. The GSL system uses only the 4 least significant bits in setting the color, so no error message is issued if the value of expr falls outside the range of 0 to 15.

2.4.3 THE PLOT STATMENT

```
PLOT expr1, expr2
```

The PLOT statement is used to plot a single point on the graphics screen. The horizontal coordinate is specified by the value of expr1 and must be in the range 0 to 39 to assure proper program execution under GSL. The vertical coordinate is determined by the value of expr2 and must be in the range 0 to 47 for proper program execution. The color of the point plotted vertical coordinate is determined by the value of expr2 and must be in the range 0 to 47 for proper program execution. The color of the point plotted is determined by the current COLOR setting.

2.4.4 READING THE SCREEN COLOR

```
SCRN (expr1, expr2)
```

The SCRN function (mentioned in section 2.1) is used to read the graphics screen color at the horizontal coordinate specified by the value of expr1 and vertical coordinate specified by the value of expr2.

2.4.5 DRAWING LINES

```
HLIN expr1, expr2 AT expr3

VLIN expr1, expr2 AT expr3
```

HLIN draws a horizontal line from the point (expr1,expr3) to the point (expr2,expr3) using the current COLOR setting. The values of expr1 and expr2 must be in the range 0 to 39, and the value expr3 must be in the range of 0 to 47. VLIN draws a vertical line from the point (expr3,expr1) to the point (expr3,expr2). In the case of VLIN, expr1 and expr2 must be in the range of 0 to 47, and expr3 must be in the range of 0 to 39.

This completes chapter 2. You should carefully read chapter 4 if you plan to use the many additional language features offered by the IBC/GSL system.

- CHAPTER 3 -

USING THE COMPILER AND RUN-TIME SYSTEM

Your Integer BASIC Compiler and GSL run-time system are designed to be easy to use. The software in the IBC/GSL package is supplied to you on two disks: a “compiler” disk and a “system” disk. The compiler disk is a non-standard, non-DOS disk with its own self-contained high-speed disk driver routines to handle the compiler´s numerous memory overlays. Compiler operation is initiated by running a program called “IBC” which is supplied on the system disk. To compile a program, IBC requires access to the compiler disk after it is run. The compiler disk is write-protected and recorded on both sides for your convenience. If you are using a two drive system, you should place your system disk in drive 1 and your compiler disk in drive 2 and IBC will automatically access the drives as needed. On a single drive system, you will have to place the compiler disk in drive 1 after IBC is run. In either case, your disk drive controler must be in slot 6. The run-time system is contained in a file called “GSL.SYS” on the system disk. It is automatically loaded into memory as needed by your object program or the compiler. Here are the steps that you´d use to compile and execute a program under IBC:

1) Begin by booting the system disk or by RUNning HELLO. This only needs to be done once.

2) LOAD the integer BASIC source program from disk. If your program is already in memory, then make sure to SAVE the latest version of it before you compile it. This is necessary on very long programs since IBC may choose to write object code over the source program when compiling it.

3) BRUN IBC. This begins the compiling process. If you are using a single disk drive, it will be necessary to remove the system disk and replace it with the compiler disk when IBC requests: PLEASE LOAD COMPILER DISK.

4) IBC then asks three questions, which should be answered with a Y (for YES) or with an N (or a carriage-return) for NO:

   OPTIMIZE FOR SPEED? If answered NO, IBC will generate pure GSL code giving you the most compact object code. Answering YES will generate mixed GSL/6502 code which executes at maximum speed.

   DISASSEMBLE COMPILED CODE? Answering YES will give you a disassembled listing of your program in symbolic GSL (or GSL/6502) code. (see user notes for an explanation of the disassembled listing).

DISABLE VARIABLE CLEARING? You should normally answer NO so that your variables will be cleared (i.e. set to zero) when your program is run. If you answer YES, your program will be compiled so that its variables are not cleared when the program is executed. This feature would be useful if you wanted to have two (or more) programs pass data to eachother in “COMMON” memory locations. After this question is answered, IBC will begin to compile your program.

5) As your program is being compiled, you may stop output or resume it by typing a CTRL-S. Output can be suppressed by typing a CTRL-O.

6) When your program is compiled, IBC reports the object code length in hexadecimal, using the format: $llll. If you are using a single drive system, the compiler disk can be removed and the system disk replaced at this point. IBC then asks: OBJECT CODE STARTING ADDRESS? You can answer this either by supplying a hexadecimal address (omit the $) or by typing a carriage-return. If you answer with a carriage-return, IBC will move the object code to the lowest available memory location. IBC will never move the object below location $A00 or above $8000 to avoid writing over itself.

7) After the object code is moved to the desired address, IBC reports the actual location of the code in the following format: $ssss.eeee, where address, IBC reports the actual location of the code in the following format: $ssss.eeee, where ssss is the starting address (in hex) and eeee is the ending address. IBC then asks: EXECUTE (Y/N) ? which you should answer with a Y or YES if you´d like to run the compiled program or answer with an N or carriage-return if you´d like to save the object code before running it.

To save the object form of your program on disk, it is only necessary to note the starting address and length of the object code, as reported to you by IBC. To write the object code to disk, use the DOS command:

```
BSAVE file name, A$ssss, L$llll
```

When saved in this manner, all that is necessary to execute the object code is to use the DOS command:

```
BRUN file name
```

Here is an example of compiling, saving and then executing a program called “TEST”:

```
    >LOAD TEST
    >BRUN IBC

    *** INTEGER BASIC COMPILER - VS 2.2 ***

    COPYRIGHT (C) 1981, GALFO SYSTEMS

    IBC: PLEASE LOAD COMPILER DISK!
    IBC: OPTIMIZE FOR SPEED (Y/N) ? NO
    IBC: DISASSEMBLE COMPILED CODE (Y/N)? NO
    IBC: DISABLE VARIABLE CLEARING (Y/N)? NO

    VARIABLE NAME TYPE <$LOC> <$LEN>
    ---------------------------------------
    D$........... SHORT STR <0800> <0010>
    A............ INTEGER <0810> <0002>
    I............ INTEGER <0812> <0002>
    ARRAY1....... INT ARRAY <0814> <0018>
    IN$.......... STRING <082C> <0038>

    IBC: OBJ. CODE LENGTH: $05E8
    IBC: OBJ. CODE STARTING ADDRESS? C00
    IBC: OBJECT CODE LOCATED: $0C00.11E8
    IBC: EXECUTE (Y/N) ? NO

    >BSAVE C.TEST,A$C00,L$5E8
    >BRUN C.TEST
```

Object code in memory can also be executed by the monitor “G” command or by CALLing it from BASIC. Also, you can execute the last compiled program (assuming that it is still in memory) by a CALL 99 statement from BASIC.

The object code generated by IBC is relocatable to any 6502 page boundary by a simple memory move. A page boundary is an address with the lower-order byte of the 16-bit address being zero (same as having the last two
a simple memory move. A page boundary is an address with the lower-order byte of the 16-bit address being zero (same as having the last two
hexadecimal digits of the four digit address being zeros). Thus, you can load or run the object code at an address different from that which it was saved. For example, if you have an object file named “C.TEST” with a length of $5E8 and you want to have it loaded at address $4000, you can use the DOS command:

```
    BLOAD C.TEST,A$4000
```

and then save it (with the new starting address of $4000) using the DOS command:

```
    BSAVE C.TEST,A$4000,L$5E8
```

When object files are saved in the manner described above, they should be placed on a disk that contains the run-time system file GSL.SYS, so that they can be executed. It is possible to save the object program and the run-time system in one file, since both are relocatable. In the example above, after BLOADing C.TEST, GSL.SYS could be loaded at location $4600 (the next available page boundary above $45E8) using the DOS command:

```
    BLOAD GSL.SYS,A$4600
```

and the entire code saved using the DOS command:

```
    BSAVE C.TEST,A$4000,L$1400
```

Note that length of the saved file must be great enough to include both the object code and the run-time system (which has a length of $E00). In this example $1400 was chosen since $600 + $E00 is $1400.
There are two important points that you should be aware of if you plan to relocate the run-time system:

1) The run-time system code modifies its vector table the first time that GSL code is executed, thus it can´t be relocated after running an object program. The easiest way to avoid trying to relocate a ´used´ copy of GSL.SYS is to use the BLOAD command to relocate it.

2) If more than one copy of the run-time system code is present in memory (this would be the case if it were relocated), then the one with the lower starting address is used to execute GSL code. This can cause confusion if several programs are executed in sequence each having the run-time system in different locations. This can be avoided by either having all programs use the same copy of the run-time system (as is normally the case when using the compiler), or by “deleting” the run-time system after it is used at a nonstandard memory position. The run-time system can be “deleted” from memory by changing the first byte of its code to anything other than a $D8; for example, write a zero over the $D8.

Object programs that are executed under the GSL run-time system cannot be interrupted by a control-C (CTRL-C) except when doing a keyboard input
Object programs that are executed under the GSL run-time system cannot be interrupted by a control-C (CTRL-C) except when doing a keyboard input
operation via an INPUT statement (GET and KEY functions pass CTRL-C´s to your program). You can interrupt program execution by pressing RESET, which will return you to BASIC (autostart ROM) or to the monitor (monitor ROM). If you return to integer BASIC in this manner, then you should type the BASIC command: NOTRACE, since frequently you´ll find that BASIC is left in the TRACE mode. If you return to Applesoft via a RESET or a CTRL-C, type the DOS command FP to initialize Applesoft. To switch from Applesoft to Integer BASIC to compile a program, simply RUN HELLO. HELLO assures that HIMEM is set properly so that source programs will not overlay the run-time system. When in doubt, you can always re-boot your system disk.

- CHAPTER 4 -

ADVANCED PROGRAMMING FEATURES

The IBC/GSL system supports many andvanced programming features. This chapter describes these features and provides examples of how they are used in a porgram.

4.1 VARIABLE ALLOCATION

The Integer BASIC Complier allows to you control what memory locations are used by your program variables. Here is a brief explanation of how the IBC allocates memory:

The IBC first reads the entire source program (starting at the lowest numbered line) and makes a list (symbol table) of every variable found and its classification. There are four classifications: INTEGER, INT ARRAY, SHORT STR, and STRING. An INTEGER is a simple integer variable; an INT ARRAY is an integer variable that is dimensioned somewhere in the program; a SHORT STR is a string variable that is not dimensioned; a STRING is one that is dimensioned. The following list gives the number of bytes allocated to each variable type:

INTEGER	- Each simple integer is allocated 2 bytes.
INT ARRAY	- Each integer array is allocated 2n + 4 bytes of memory, where n is the size of the array specified in the DIM statement.
SHORT STR	- An undimensioned string is allocated 16 bytes of memory.
STRING	- A dimensioned string is allocated n + 16 bytes of memory, where n is the length of the string specified in the DIM statement.

Extra memory is allocated to an INT ARRAY to allow space for the zeroth subscript element permitted in BASIC and also lets the (n+1)th element act as spare element and buffer zone between the last element of the array and the next memory location. Strings are allocated a little more memory than necessary because the GSL run-time system stores the length of each string in addition to its character data. In manipulating strings, the GSL system also moves characters 4 bytes at a time for maximum execution speed, and thus, some extra space is needed in each string to permit this. Each string can contain about 12 characters more than dimensioned without causing an adjacent variable to be overwritten.

The absolute memory location where the compiler begins allocating memory is determined by the setting of LOMEM, which is normally 2048 (i.e. $800). Thus, you can type: LOMEN:nnnnn before compiling a program and set the address (nnnnn in decimal) where IBC begins its variable allocation. In

Thus, you can type: LOMEN:nnnnn before compiling a program and set the address (nnnnn in decimal) where IBC begins its variable allocation. In most cases, you´ll probably want LOMEM to be set to 2048, but it can be moved higher; say, for example, you have machine language subroutines that are to be located in the low end of memory.

The compiler lists its symbol table on the output device when it allocates memory. This list shows each variable name, its classification, its beinning memory location (in hex) and the number of bytes allocated (also in hex). Example symbol tables are shown in sections 4.1.1 and 4.1.2.

4.1.1 THE DIMENSION STATEMENT

```
    DIM var1(n1), var2(n2), var3$(n3), ... etc.
```

The DIM statement is treated as a non-executable statement by the compiler. It is used to allocate memory for integer array and string variables. The dimensioned size, n, must be a positive constant, i.e. a number from 0 to 32767, or the compiler will issue an error message. Although DIM statements can appear anywhere in a program, it is best (for purposes of human readability) to place your DIM statements at the beginning of a program. Examples:

```
    10 DIM ARRAY1(10)
    20 DIM Z(250), ANS$(40)
```

These starements, if compiled with LOMEM set to 2048 (i.e. $800 hex) produce the following variable allocation:

```
    VARIABLE NAME TYPE <$LOC> <$LEN>
    --------------------------------------
    ARRAY1....... INT ARRAY <0800> <0018>
    Z............ INT ARRAY <0818> <01F8>
    ANS$......... STRING <0A10> <0038>
```

You will find the symbol table very useful in many applications, since it tells you exactly where data is stored in memory.

4.1.2 SHARING MEMORY LOCATIONS

There is a clever trick that you can play on the compiler to force it allocate the same memory location to two or more variables in a program. Veteran FORTRAN programmers know of the usefulness of this through the EQUIVALENCE statement. The most important application of sharing memory is that of being able to reference the same data in two different ways. For example, if you have a small array, say 3 elements, that stores the time of day, called TIME (with TIME(1) being seconds, TIME(2) being minutes, and TIME(3) being hours), you may sometimes wish to refer to the first element as SEC instead of using TIME(1). By having TIME(1) and a simple integer variable SEC share the same memory location, you could use either to reference the same data. There are more exotic applications of this programming technique, such as having two arrays share overlaping memory locations but with their origins offset, or, performing data conversions by arranging for an integer array to share memory with that of a string.

The “trick” used to force IBC to allocate the same memory locations to two variables is to dimension a very large “dummy array” between the two variables that are to share common locations. When the compiler tries to allocate the next variable, it uses 16-bit arithmetic that ignores variables that are to share common locations. When the compiler tries to allocate the next variable, it uses 16-bit arithmetic that ignores overflows, and the result is that the two variables are assigned to the same memory location. The dimensioned size needed for the dummy array is 32765 when used between two integer variables, or 32763-n when used between two integer arrays (where n is the dimensioned size of the first array). In more complicated cases, you´ll probably need to experiment a little to make your variable allocation work out properly. Here is an example that should clarify this:

```
    10 VAR1=0: DIM DUMMY(32765): VAR2=0
```

When compiled, the following variable allocation is made by IBC:

```
    VAR1......... INTEGER <0800> <0002>
    DUMMY........ INT ARRAY <0802> <FFFE>
    VAR2......... INTEGER <0800> <0002>
```

As can be seen from the symbol table, varaibles VAR1 and VAR2 have been assigned to the same memory location, $0800. (note that $FFFE + $0802 is $0800 in 16-bit precision arithmetic). The array dimensioned between variables need not be a “dummy”, but could be an array that you wish to reference in your program. Consider the following example that allocated 4 arrays: A, B, C, and D each of size 10, with A & C sharing the same location, and B & D sharing the same location:

```
10 DIM A(10),B(32754),C(10),D(10)
```

which produces the following variable allocations:

```
    A............ INT ARRAY <0800> <0018>
    B............ INT ARRAY <0818> <FFE8>
    C............ INT ARRAY <0800> <0018>
    D............ INT ARRAY <0818> <0018>
```

In this example, you should note how only one large dimension size (the dimension of B) is necessary to make two sets of variables equivalent. This technique could be extended to large sets of equivalent variables.

4.2 THE CHR$(0,expr) STRING FUNCTION

Apple´s integer BASIC interpreter lacks a function that converts an integer into a character for use as a string, although it has the inverse function, i.e. the ASC function. The IBC/GSL system has a function available for this purpose. This character generation function is referenced as: CHR$(0,expr) and can be used anywhere a string is allowed. It generates a string of length 1 containing the ASCII character given by the value of expr. Since it operates like an inverse of the ASC function, expr must be greater than 127 if bit-8 of the ASCII character is to be set (Apple standard). Thus, the value of expr normally is in the range of 128 to 255 , although 0 to 127 can be used to produce character strings that print as inverted or flashing characters on the Apple´s screen or for use in strings that contain non-character data. Example:

```
     10 D$=CHR$(0,132)
     20 FOR I=0 TO 255 : PRINT CHR$(0,I);: NEXT I
```

In this example line 10 sets the string D$ to a control-D chartacter (D$ would be useful in PRINT statements to talk to DOS) and line 20 prints all the Apple ASCII character set.
would be useful in PRINT statements to talk to DOS) and line 20 prints all the Apple ASCII character set.

4.3 SINGLE CHARACTER INPUT: GET(0) FUNCTION

The normal INPUT statement in BASIC takes in all characters from the input device without returning control to your program until a carriage-return is received. The GET(0) function allows a single character to be read from the input device (keyboard, disk, etc.). When a GET(0) function is executed, it waits for one character to be received from the input device and returns the ASCII value (128 to 255) of the character read. The character received from the input device is also echoed to the output device as is done by the INPUT statement. GET(0) does not convert lower-case ASCII to upper-case.
Example:

```
     10 CHAR = GET(0)
     20 PRINT “ ASCII VALUE IS “;CHAR : GOTO 10
```

GET(0) can be combined with CHR$(0,expr) to convert an input character into a string:

```
     10 C$=CHR$(0,GET(0))
     20 TEXT$(LEN(TEXT$)+1)=C$
```

In this example, line 10 reads in one character and converts it to a one character string called C$. Line 20 then adds (concatenates) it to the end of the string TEXT$.

4.4 KEYBOARD INPUT: KEY(0) FUNCTION

The Apple II keyboard can be directly accessed from GSL via the KEY(0) input function. When a KEY(0) is executed, the Apple keyboard is checked to see whether or not a key has been pressed. If no key was pressed, then KEY(0) returns a value of zero (i.e. FALSE); if a key was pressed, its ASCII value (128 to 255) is returned and the keyboard is cleared to allow another key to be read. KEY(0) does not echo the character read from the keyboard.
Example:

```
    10 K = KEY(0) : IF NOT K THEN 10
    20 PRINT “ASCII VALUE IS “;K : GOTO 10
```

The KEY(0) function is most useful for occasional checking of the keyboard in a main program loop.

4.5 DSP EXTENSION STATEMENTS AND HI-RES GRAPHICS

The IBC/GSL system allows many statement keywords not implemented by the Apple interpreter. These statements give you direct access to the Apple´s screen display features and Hi-Res graphics routines. These extended features were added to the BASIC language by a special syntax construction using DSP as the first three letters of a statement:
DSP keyword
where keyword is one of the following 13 words:

```
    HOME CLEAR INVERT NRML FLASH

    FULL MIXED LO HI H2 POINT LINE SHAPE
```

The compiler only reads the first two letters of a keyword following DSP, so, for example, the statement DSP INVERT could be abbreviated to DSP IN or DSP INV.

4.5.1 DSP HOME and DSP CLEAR statements

The statement DSP HOME moves the cursor to the top left screen position and clears the screen. It is identical to CALL -936 or typing “esc @ return”. Normally the entire screen is cleared, but if a smaller scrolling text window has been defined (by modifying locations $20 to $23) only the window will be cleared, with the cursor being moved to the top left position of the text window.

DSP CLEAR clears the screen from the cursor position to the end of the current line. It is identical to CALL -868. The cursor is not moved.

4.5.2 DSP INVERT, DSP NRML, and DSP FLASH statements

The Apple´s screen can display characters in normal video (white on black), inverted video (black on white) and flashing video (alternating normal and inverted video). The three statements DSP INVERT, DSP NRML, and DSP FLASH are used to set the video display modes to inverted, normal or flashing. These statements are equivalent to POKE 50,63, POKE 50,255 or POKE 50,127.

4.5.3 DSP FULL and DSP MIXED graphics display statements.

The Apple allows a graphics display (in both lo-res and hi-res) in either a full screen format or a mixed graphics and 4 line text format. The two statements DSP FULL and DSP MIXED can be used to select these graphics display formats. DSP FULL is equivalent to a POKE -16302,0 , while DSP MIXED is the same as a POKE -16301,0.

4.5.4 HI-RES / LO-RES GRAPHICS SELECTION

```
    DSP LO DSP HI DSP H2
```

The GSL system can access either Lo-Res or Hi-Res graphics driver routines via three statements: GR, COLOR=expr, and PLOT expr1,expr2. In order to use these statements for Hi-Res graphics, three DSP extension statements are provided to switch between Lo-Res, Hi-Res and page 2 Hi-Res graphics. These statements are DSP LO, DSP HI and DSP H2. DSP LO sets the normal Lo-Res mode for GSL (see chapter 2 section 2.4), and enables the Lo-Res (page 1) display. When a program is run, GSL always assumes the Lo-Res graphics mode. DSP HI and DSP H2 set the Hi-Res mode for GSL and enable either page 1 ($2000 to $3FFF) or page 2 ($4000 to $5FFF) of the Hi-Res display. Executing a ny of these three statements does not clear the graphics screen, but only enables the graphics display selected by the statement. The screen is cleared when a GR statement is executed.
After a DSP HI or DSP H2 statement, the COLOR=expr statement can be used to set the Hi-Res plotting color (0 to 255). In Hi-Res the colors displayed are:

0 - black	128 - black2
42 - green	170 - orange
85 - violet	213 - blue
127 - white	255 - white2

On the original Apple II (those with Serial number below 6000), 170 will appear green and 213 will appear violet.

4.5.5 HI-RES PLOT MODE SELECTION

When in Hi-Res mode, the PLOT expr1,expr2 statement can be used to plot points, lines, or shapes on the 280 by 192 Hi-Res display. Executing a DSP POINT statement establishes the single point plotting mode for the PLOT statement. In this mode, a single point is plotted at the coordinates (expr1,expr2) each time a PLOT statement is executed. This mode is automatically restored when a DSP HI or DSP H2 statement is executed. The value of expr1 in the PLOT statement should be in the range of 0 to 279 and the value of expr2 should be limited to 0 to 191 (values greater than 159 are not displayed when in the mixed graphics mode.

The DSP LINE statement establishes a line plotting mode for the PLOT statement. After a DSP LINE is executed, a pair of PLOT statements can be used to draw a line from one point to another on the graphics screen. For example, the statement sequence:

```
    DSP LINE : PLOT 10,12 : PLOT 260,140
```

would plot a line from point (10,12) to point (260,140). If more than two plot statements are executed after a DSP LINE, line drawing continues be connecting each new point to the end-point of the previous line. For example:

```
    DSP LINE : PLOT X1,Y1 : PLOT X2,Y2 : PLOT X3,Y3
```

would draw a line from (X1,Y1) to (X2,Y2) and another line from (X2,Y2) to (X3,Y3).

The DSP SHAPE statement establishes a third type of plotting mode for the PLOT statement, that is, the plotting of a predefined graphics shape centered at the coordinates (expr1,expr2). In order to use this feature, you must define the shape to be plotted by a “shape table” in memory. (see the original Apple II reference manual pages 50-53 on how to set-up a shape table). The shape to be plotted is selected by passing the starting address of the shape table in locations 804 and 805 (decimal), with its scaling factor in location 806 and rotation factor (0 to 64) in location 807.

A sample Hi-Res program that uses all three plotting modes (point, line, and shape) is included on the IBC/GSL disk (file name: HIRES TEST).

IMPORTANT NOTE: The GSL system does not actually contain the graphics driver routines to do Hi-Res plotting, but only contains links to these routines which are located in the file: HIRES DRIVER. Thus, this file should be loaded before any Hi-Res plotting is attempted. These routines load in memory from $C00 to $FFF and are not relocatable (by a simple memory move). If you wish to use ROM based graphics driver routines, see appendix C on patching GSL.SYS to use the Programmers Aid #1 ROM.

This concludes the description of the advanced programming features of the IBC/GSL system. Happy Programming !

- APPENDIX A -

ERROR MESSAGES

The IBC/GSL system, like most other BASIC language systems, issues error messages only when absolutely necessary. Apple integer BASIC does a syntax check as each program line is entered into the computer, so most kinds of typing errors or misuses of language syntax are brought to your attention immediately after you finish typing the line. The compiler further checks each statement as it is being compiled. There are eight error messages that are reported by the compiler. Here is an explanation of each:

“MEMORY FULL - NO ROOM FOR SYMBOL TABLE”

This message is reported when the compiler runs out of free memory when building its symbol table. The program being compiled may be too large or HIMEM may be set too low.

“MEMORY FULL - NO ROOM FOR OBJECT CODE”

This message indicates that IBC ran out of free memory when it was trying to write the program object code to memory. The program being compiled may be too large for IBC or HIMEM may be set too low. If you tried to optimize your program for speed and got this error message, try recompiling it optimzed for space (i.e. answer the first compiler question with a “NO” or return). Optimizing for speed produces a larger object file.

“VARIABLE ARRAY DIMENSIONS ARE ILLEGAL”

This message is issued when IBC finds an array dimension (in a DIM statement) that is not a constant. Although it is rarely used, the BASIC interpreter allows array dimensions to be computed by an expression. IBC demands array dimensions be specified by a positive number (0 to 32767), since memory is allocated at compile-time, not run-time. The solution is simple, just change your array dimension(s) to a constant.

“GOTO OR GOSUB TO A NONEXISTENT LINE”

This error is detected by the compiler when a GOTO ln# or GOSUB ln# or an IF expr THEN ln# statement specifies a line number (ln#) that can´t be found in the program. To correct this kind of error, simply change to ln# to the proper line so the compiler knows what to do. Note: When the compiler lists the line where it found this error, it sometimes is one line off, so please don´t think that it went bananas on you!

“STATEMENT TYPE IS ILLEGAL FOR COMPILER”

You´ll get this error message if your program contains certain statements types such as LIST that can´t be implemented by GSL at run-time.

"PARENTHESES NOT MATCHED IN THIS LINE"

This message indicates IBC has found a statement containing unmatched parentheses. This condition is normally detected by BASIC when the line is entered.

"SYNTAX ERROR OF SOME SORT - CODE XX"

This error condition is reported by the compiler when it finds a syntax error in a statement being compiled. Normally you should not get this error message since BASIC checks each line for proper syntax. If you do get this message, try retyping the line in question (or the line before it) and recompiling your program. If the error persists, please see the instructions in appendix D.

"INTERNAL COMPILER ERROR"

The compiler issues this message when it finds an inconsistency in its own operation. Please see Appendix D.

In addition to these error messages, the compiler issues two warning messages:

"CAN´T OPTIMIZE GOTO, GOSUB"

Please see chapter 2 section 2.3.1 for an explanation of this message.

"OBJ. CODE OVERLAYS SOURCE"

This message is issued when IBC runs short of free memory and writes the object code over the source program. After compiling the program, the beginning of the source code in memory may be destroyed. Thus, long source programs should be saved to disk before compiling them as described in chapter 3.

The GSL run-time system also reports five types of errors that occur when the object program is run. These are:

BR ERR - This means that a GOTO or GOSUB tried to branch to a nonexistent line number.

RET ERR - A subroutine return error - See 2.3.2.

SUB ERR - Greater than 24 GOSUBs - See 2.3.2.

NXT ERR - A NEXT without a FOR - See 2.3.6.

OP ERR - An attempt to execute an illegal GSL op code. This indicates a fault with the object code, memory, or the compiler. See appendix D.

- APPENDIX B -

BENCHMARK PROGRAM EXECUTION TIMES

The following is a table showing execution times for 7 standard benchmark programs designed to test BASIC on small computers. The benchmark listings were first published in the June and the Oct. 1977 issues of KILOBAUD magazine by T. Rugg and P. Feldman. These programs are short and represent a “worst case” basis for comparing a compiler to an interpreter, since an interpreter´s performance is optimum with small programs whereas compiled code execution speeds are independent of program length. Nonetheless, the performance of the IBC/GSL system with these benchmarks is impressive, indicating why we believe that it is the fastest higher level language system availiable for a 6502 processor. The execution times listed in this table are in seconds.

Program #	IBC/GSL (Opt. speed)	IBC/GSL (Opt. space)	APPLE Integer BASIC	APPLE Applesoft BASIC
BM1	0.16	0.16	1.4	1.3
BM2	0.33	0.46	3.2	8.0
BM3	1.5	1.8	8.0	16
BM4	1.0	1.2	7.0	17
BM5	1.2	1.3	9.0	19
BM6	2.1	2.3	18	28
BM7	2.9	3.4	28	45

Here is a listing of BM7:

```
    300 PRINT “START”
    400 K=0
    430 DIM M(5)
    500 K=K+1
    510 A=K/2*3+4-5
    520 GOSUB 820
    530 FOR L=1 TO 5
    535 M(L)=A
    540 NEXT L
    600 IF K<1000 THEN 500
    700 PRINT “END”
    800 END
    820 RETURN
```

BM2 through BM6 are subsets of this program.

- APPENDIX C -

NOTES ON GSL.SYS

The following patches can be made to the run-time system at the user´s discretion. GALFO SYSTEMS assumes no responsibility for these changes, however, they should function properly. Before making any of these patches to GSL.SYS, make sure that you have a back-up copy of it. To patch the run-time system, first BLOAD GSL.SYS, then enter the Apple II monitor via a CALL -151. Type the desired changes, and then BSAVE GSL.SYS,A$8800,L$DFB. (on a 32K system, use A$4800 instead of A$8800).

1) HI-RES use of PROGRAMMER´S AID #1 ROM.

This patch lets GSL.SYS use ROM based Hi-Res graphics driver routines instead of the RAM based routines provided in the file: HIRES DRIVER. This patch does not work for the SHAPE plotting mode.

```
48 K system:      32 K system:

*9434:24          *5434:24
*9446:00 D0       *5446:00 D0
*9486:7A D0       *5486:7A D0
*9490:2E D0       *5490:2E D0
*94A3:64 D1       *54A3:64 D1
```

2) To disable the CTRL-C trap (on the INPUT statement):

```
48 K system:      32 K system:

*92D0:0           *52D0:0
*936C:0           *536C:0
```

3) To change the END statement so that it returns to an address other than $E003 (BASIC in ROM):

```
48 K system:      32 K system:

*9111:xx          *5111:xx
*9114:yy          *5114:yy
```

xx is the higher-order address byte and yy the lower-order address byte of the address that the END statement will return

USER NOTES - IBC / GSL SYSTEM - Feb. 1982
IBC 2.3 - GSL 1.6

The following notes are being provided as a supplement to our instruction manual.

1) DOS 3.3 can be relocated in the language (RAM) card so that all programs (including the compiler) have 10 K more memory available. The operation of the DOS moving routine is described in the July / Aug. 1981 issue of CALL A.P.P.L.E. magazine. (note: your Apple must have Integer BASIC in ROM on the Apple II mother-board, in order to use the relocated DOS with IBC). When DOS is moved to the RAM card, CONFIG.SYS (BRUN by your HELLO program) will load GSL.SYS
relocated DOS with IBC). When DOS is moved to the RAM card, CONFIG.SYS (BRUN by your HELLO program) will load GSL.SYS
at $B100 instead of $8800. You can move DOS to the language card by doing a BRUN DOS -> RAM CARD. Your system disk is always re-booted after this routine is run. The relocated DOS contains an additional command ´PADDR´ which allows you to obtain the starting address and length of the last file BLOADed or BRUN.

2) The largest program that can be compiled by IBC is about 104 sectors (about 25K) or about 148 sectors (about 36K) with DOS in the laguage card assuming that pure GSL code is being generated.
3) The GSL run-time system uses only section $90 to $DF of page zero. This section of page zero is restored when the END statement is executed.

4) IBC contains a symbolic disassembler for GSL code and mixed 6502 / GSL code. The purpose of providing the disassembler is to allow the user to see exactly what code is being generated by the compiler in a format similar to an assembly listing. The listing is broken down by source line number and each line of the listing follows the format: address, op-code, operand. The addresses listed are relative; that is, the actual memory location (when a particular instruction will be located) is this address plus the starting address. The op-codes listed are either a mnemonic representation of GSL instrustions or standard 6502 codes (indented 3 spaces for clarity). The operand field contains the user´s variable name or data referenced by the instruction. CTRL-S can be used to stop or resume a disassembler listing; CTRL-O causes output to be suppressed or resumed.

5) A run-time error message “FOR ERR @xxxx” was added to GSL.SYS. It is issued when more than 16 FOR statements are active at one time. The hex address xxxx printed in all run-time error messages are the absolute address where the error occured. To trace this back to a BASIC source line, subtract the starting address from this address (obtaining the relative address) and consult a disassembler listing of the program to locate the exact source statement.

6) GSL.SYS allows you to recover from DOS errors that occur at run-time. Here´s how it works. When a DOS error

occurs, GSL.SYS places the error code in location 222 (decimal) and reenters your program at the first line (no variables are cleared, GOSUBs and FORs are still active). This allows you to test location 222 and branch to your error handling routine if a non-zero code is returned. You can enable/disable error trapping by setting location 223 to 0 (enable) or 255 (disable). The defult value is 255, i.e. disabled. For example:

```
10 ERR = PEEK(222) : IF ERR THEN 999
20 POKE 223,0 : REM Enable error trap
... ...
999 REM  Error handling routine; variable “ERR” contains
         the DOS error code (1-15) upon entry.
```

The error codes issued by DOS are explained in the DOS manual, pages 114-122. If the error code is zero (i.e. “FALSE”) then no error occured.

7) IBC can read the LOMEM:nnnn statement from within a program. Since this statement can´t normally be entered into a program (it is illegal syntax when typed), it must be manually placed into memory using token code $11. The starting point (origin) of variable allocation is determined by setting LOMEM as explained in section 4.1.

8) The system disk contains the following files:

```
HELLO - See Important
HELLO APPLE II - User Note on Hello
HELLO APPLE II & RAM CARD - Programs.
HELLO APPLE II PLUS & RAM CARD -
IBC - Compiler (disk in slot 6)
AUTO COMPILE - See note 9
GSL.SYS - Run-time system A$8800, L$DFB
CONFIG.SYS - See manual Chapter 3
AUTOFILE.SYS - See note 9
HALT.SYS - Used by AUTO COMPILE exec files
TO INT BASIC - EXEC to return to INTEGER
DOS -> RAM CARD - DOS relocation routine - see note 1
HIRES DRIVER - See manual section 4.5.5
HIRES TEST - Sample Hi-Res using IBC extensions
HIRES SHAPE TABLE - Used by HIRES TEST
COLOR DEMOS - Lo-Res Graphics demo
GSL.SYS (BACKUP) - Extra copy of run-time system
IBC (BACKUP) - Extra copy of IBC for slot 6
IBC (SLOT 4) - See note 10
IBC (SLOT 5) - See note 10
```

9) AUTO COMPILE is a utility program developed to allow you to build an EXEC file that will automatically LOAD, compile, and BSAVE (the object file) for one or more source programs. AUTO COMPILE is conversational and contains built-in instructions. AUTO COMPILE uses a subprogram called AUTOFILE.SYS to allow general purpose, interactive EXEC files to be built. Two disk drives are needed to use the EXEC file created by AUTO COMPILE.

To use the program, simply type: BRUN AUTO COMPILE.

10) Two additional copies of IBC are provided so that the compiler disk to be located in disk drives with slot 4 or slot 5 controlers. (file names: IBC (SLOT 4) and IBC (SLOT 5). These versions allow you to use the compiler and system disk with different media, such as using a Corvis hard-disk to hold the files on the system disk.

11) If you´re having trouble compiling a lengthy program which you are not familar with, here are a few suggestions:

    11.1) A program that LISTs improperly may contain both BASIC and machine language. This type of program must be separated into its original parts before the BASIC segment can be compiled.

    11.2) In general, any program that makes extensive use of machine language subroutines or PEEKing and POKEing into the normal BASIC symbol table will have to be modified before it can be run in its compiled form.

    11.3) Many published programs contain ´hidden´ code or normally ´illegal´ syntax that has been placed manually in the program (by poking data directly into memory). These types of modifications do not always show up in a LISTing but may affect the operation of a program when it is run. When compiled, such a program may not function properly, or may cause the compiler or run-time system to crash.

    11.4) One easy way of removing all improper syntax and hidden code from a program is to simply retype the program. A lot of work you say? Not if you let DOS do the typing! The procedure to be followed is outlined on pages 76 and 77 of the DOS manual. By turning the BASIC listing into a text file and EXECing that file, Integer BASIC will check all syntax as it rebuilds the program. You´ll always get a good copy of the program if you do this. Here are a few things to watch for: 1) Remember to type NEW before EXECing the text file listing. 2) Doing a MON I,O,C is useful to observe what is happening as DOS retypes the program. 3) If the retyped program failes to run as before, it had some form of hidden code in it. After this procedure, the program should be able to be compiled and executed successfully.

12) CALL 99 will execute the last program compiled by IBC. (A description of CALL 99 is missing from some IBC manuals).

- END OF USER NOTES -

IBC/GSL USER UPDATE - May 5, 1982 GSL.SYS VS 1.7 & HIRES.SYS

HIRES.SYS is a new HI-RES graphics driver being supplied with the system. This routine has many features that the old HIRES DRIVER did not support, and it is RELOCATABLE ! Here are a list of the new driver´s additional features:

- Works in all 8 colors (the old driver did not)
- It is page relocatable (communication with GSL.SYS VS 1.7 or later version is done through fixed page-3 vectors)
- Shares page-zero memory locations with GSL.SYS ($90 to $DF)
- Is compatable with SHAPE TABLES having an index (Modern Apple format for shape tables)
- Supports XOR drawing of shapes (´undrawing´) for animation.
- A “collision” count is supported.
- Allows clearing the srceen using colors other than black.

USING HIRES.SYS :

HIRES.SYS operates exactly like HIRES DRIVER (as documented in Chapter 4 of the manual) for the following commands:

- DSP HI or DSP H2
- DSP POINT mode for PLOT
- DSP LINE mode for PLOT
- COLOR= expr (all 8 colors work)

HIRES.SYS differs in the use of the GR command. The GR command will now set the entire screen to the color specified by the last COLOR= expr executed. Thus to clear the HIRES screen:

```
10 COLOR=0 : GR : REM Clear screen (set to BLACK)
```

The DSP SHAPE drawing mode for the PLOT statement has been changed:

1) SHAPE TABLES must have an index.
2) The page-3 locations for the shape table address, rotation and scale factors have been changed.
3) A SHAPE number must now be specified.
4) A plotting MODE must be specified.

The SHAPE TABLE starting address must be passed through address 808 (decimal) and 809. The SCALE factor is passed throught location 807 and the ROTATION factor through 812. The SHAPE number to be plotted is specified by poking into 811. The plotting MODE used to plot the shape must be POKEd into 813. There are four modes:

```
POKE 813,0 - Shapes will be drawn normally
POKE 813,1 - Shapes will be drawn by exclusive ORing the shape
             data with the screen. (i.e. undrawing)

POKE 813,2 - Normal drawing at the last point plotted or drawn.
POKE 813,3 - XOR drawing at the last point plotted or drawn.
```

In modes 2 and 3, the x,y coordinates passed by the PLOT statement are ignored. A “collision count” is returned in location 810 after a shape is plotted. A change in this count indicates that the shape has been plotted over another feature on the screen.

HIRES.SYS does some checking to make sure that the SHAPE number and plotting MODE that you´ve specified are legal. If either is illegal, a Run-time error: SHAPE ERR @xxxx is issued.

If you intend to use HIRES graphics in your program, HIRES.SYS must be loaded into memory. To do this, it must be BRUN from your program. It must be BRUN, not just BLOADed (as was HIRES DRIVER). You can choose any free section of memory to BRUN it as long as the address that you sfecify is a page boundry. When the file is BRUN, it sets up page-3 verctors for GSL.SYS, it relocates itself, and returns to your program (it can also be CALLed if already in memory). The length of HIRES.SYS is $4FB. Example:

```
     10 D$=CHR$(0,132): REM CTRL-D
     20 PRINT D$;”BRUN HIRES.SYS,A$4000”: REM Load HIRES routines
```

The file HIRES TEST provids an example of the HIRES graphics extensions offered by the IBC/GSL system and HIRES.SYS.