Color Computer SMALL C Compiler

1982

DUGGER'S GROWING SYSTEMS

Duggers C Ver 1.2
TRS|DOS Version

Helpful Hints:

A. Syntax

1. Use '@' instead of '{'
2. Use '$' instead of '}'
3. Functions pass 16 bytes of arguments (max)

B. Variable storage

1. Global variables are stored PC relative
2. Local variables, temporary variables, and arguments are stored on stack U.
3. Function arguments are pushed "in-order" - when all args are pushed, U points to the last argument.
4. The # of bytes of arguments passed to a function can be determined using the following code:

```
func (argc, *argv)
int argc;
char *)argv;
@
    #asm
(cont...)
```

```
        LDA  C,S7
        ANDB #15
        [illegible]
        CLR A
        PSH U  D          ; Set Arg C
        ADD B  #2
        TFR  U,X
        LEAX B,X
        PSH U  X          ; set *Argv -
# end asm                   points to last byte
```

Program body goes here

```
# asm
        LEAU 4,U          ; De-allocate argc,argv
# end asm
 return ();
$
```

Note: The LEAU 4,U must proceed every return or exit from any routine that uses this technique.

5. The statement <var name>; leaves the address of the named variable in X

C. Output Code
1. Code is re-locatable
2. Code is not "ROMABLE"
3. Run-time library is not re-entrant.

COLOR COMPUTER C COMPILER NOTES

I. Diskette contents.

DCSCC.BIN	C compiler - standard (lower case commands)
UPC.BIN	C compiler - (upper case commands)
CLIBR.LIB	Runtime source library
INCLIB.TXT	Runtime source - to be used with LIB.INC (see procedure below)
LIB.INC	Special input file to be added during compiling.
PRIME.TXT	Example C program - finding all the prime numbers between 1 and 4096. Set for 10 iterations.
BPRIME.TXT	BASIC program same as PRIME.TXT - 1 iteration
LISTIT.TXT	Example C program - list a file on the screen.
LISTIT.BIN	Execute program for the list program.
PRIME.BIN	Execute program for the prime number routine.
FPOINT.TXT	Floating point functions.
FPPRT.TXT	Floating point print function in C.
UPPRIME.TXT	Prime program in upper case commands.
ULIB.INC	Special input file to be added during compiling. (upper case usage)

II. Corrections to the manual.

- On page 6-4 under fopen example line
  char fildat[12] change to: *fildat
  char *fp; change to: int fp;
- On page 6-4 under c=getc(fp) example
  c=getc(c,fp) change to: c=getc(fp)

III. Special usage of the runtime source.

If your assembler has a LIBS directive, then you can use the LIB.INC file instead of the CLIBR.LIB during compiling. This will shorten your compiling time, plus minimize the amount of storage on the diskette. This uses the INCLIB.TXT file.

Computerware's assembler has this feature. Use ULIB.INC with UPC.

IV. Prime number example.

To give you an example of the speed of using C vis Basic, run both PRIME.BIN and BPRIME.TXT. BPRIME.TXT will take about 4 mins. for **one** iteration and PRIME.BIN for **10** interations will take (**try it!**).

V. Upper case C compiler.

Based on many requests, an upper case version of the C compiler has been included. UPC is the upper case version. This will handle all commands (WHILE, CHAR, IF, etc.) in the upper case.

VI. Error reporting.

The errors encountered during compiling will be displayed on the screen and the compiling will stop. To continue depress the ENTER key or to abort depress the X key followed by the ENTER key.

VII. Printing.

To print during the compiling, keyin PRINT when the compiler asks for OUTPUT FILENAME. This will direct the output from the compile to the printer and the screen.

FLOATING POINT FUNCTIONS

This version of the compiler does not support floating point operations. A later version will support the "float" type. In the meantime a set of functions has been developed to use the floating point features of the TRS Dos ROM subroutines. The following are the functions available and a brief explanation of their usage.

I. Floating point multiply function - FPMUL(R,X,Y)

where: R,X,Y are character arrays, 5 characters per variable.
R gets the results of multiplying X by Y as R=X*Y.

II. Floating point divide function - FPDIV(R,X,Y);

where: R,X,Y are character arrays, 5 characters per variable.
R gets the results of dividing X by Y as R=X/Y.

III. Floating point add function - FPADD(R,X,Y)
where: R,X,Y are character arrays, 5 characters per variable.
R gets the results of adding X and Y as R=X+Y.

IV. Floating point subtract function - FPSUB(R,X,Y)
where: R,X,Y are character arrays, 5 characters per variable.
R gets the results of subtracting Y from X as R=X-Y.

V. Integer to floating point - ITOF(I,X);
where: I is a integer (int) and X is a character array 5 in size.
I is converted to floating point number and stored in X.

VI. Floating point to integer - FTOI(X,&I);
where: X is a character array and I is a integer.
X is converted to a integer and stored in I.
&I refers to the address of I.
NOTE! The integer range is 32767 to -32767.

VII. Floating point print function - FPPRT(X);
where: X is a floating point value.
This output the value of the number to the screen, both integer and factional portions. This routine is written in C language so that it can be used also as an example of the usage of the floating point functions.

VIII. Example usage

```
        char R[5],X[5],Y[5],Z[50];
        int I,J,K;
        I=50; /* set X to 50 */
        ITOF(I,X);
     or ITOF(50,X);
        ITOF(20,Y); /* set Y to 20 */
        FPMUL(R,X,Y); /* multiply X*Y and store in R */
        FPPRT(R); /* print results of X*Y */
        FTOI(R,&I); /* convert results to a integer */
        PRINTF("I=%d",I);
        I=0;
        while(I<10) /* build a FLT PT table 0,10,20,30,---90 */
         @
           K=I*5;
           J=I*10;
           ITOF(J,&Z[K]);
           ++I;
         $
```

DGS C COMPILER FOR THE TRS-80 COLOR COMPUTER USERS MANUAL VERSION 1.0

DUGGERS GROWING SYSTEMS
P.O. BOX 305
SOLANA BEACH, CALIFORNIA 92075
(619) 755-4373

DGS C COMPILER FOR THE COLOR COMPUTER VERSION 1.0

PUBLISHED BY DUGGER'S GROWING SYSTEMS
P.O. BOX 305
SOLANA BEACH, CALIFORNIA 92075

BY BRUCE W. DUGGER & JAMES L. WAGGONER

C USERS MANUAL

TABLE OF CONTENTS

SECTION 1	1-1
Introduction	1-1
SECTION 2	2-1
Scope	2-1
Abbreviations	2-1
Color Computer Characters	2-2
SECTION 3	3-1
How to use the C Compiler Features	3-1
Source	3-1
Loading Compiler	3-1
Including Source in Output	3-1
Defining Globals	3-1
Label Numbers	3-1
Output File Name	3-2
Input File Name	3-2
SECTION 4	4-1
C Program Source Structure	4-1
#Define	4-2
Externals	4-3
Main()	4-3
#Asm, #Endasm	4-3
SECTION 5	5-1
Reserved Words	5-1
Types, Operators and Expressions	5-1
Variable Names	5-1
Data Types and sizes	5-2
Constants	5-2
Comments	5-3
Declarations	5-3

Type Modifiers 5-4
Arithmetic Operators 5-5
Relational/Logical Operators 5-6
Comparison Operators 5-6
Binary Logical Operators 5-6
Unary Expression Operators 5-7
Bitwise AND, inclusive OR, Exclusive OR 5-7
Left Shift-Right Shift 5-8
Control Flow 5-10
Statements and Blocks 5-10
If-Else 5-10
Else-If 5-11
While 5-11
Break and Continue 5-12
Return 5-12
(;) Semicolon 5-12
Functions and Program Structure 5-13

SECTION 6 6-1
DGS 6809 C Runtime Library 6-1
getchar() 6-2
putchar() 6-2
printf() 6-2
scanf() 6-3
fopen() 6-4
fclose() 6-4
getc() 6-4
putc() 6-4
Utility Functions 6-5

SECTION 7 7-1
The Differences Between DGS C For The Color Computer And Standard C 7-1

APPENDIX A A-1
Sample C Program A-1

APPENDIX B B-1
DGS C Error Codes B-1

APPENDIX C C-1
DGS C Compiler Trouble Report C-1

INDEX I-1

DGS C COMPILER FOR THE COLOR COMPUTER VERSION 1.0

Section 1. INTRODUCTION

C was originally designed for and implemented on the UNIX operating system on the DEC PDP-11, by Dennis Ritchie. The operating system, the C compiler, and essentially all UNIX applications programs are written in C.

What is C?

C is a general purpose programming language which features economy of language, modern control flow and data structures, and a rich set of operators. C allows you to write your programs clearly and simply. Linkage conventions encourage modularity and good program organization, making changes and debugging easier. C is not tied to any particular area of application. Absence of restrictions and its generality make it more convenient and effective for many tasks than supposedly more powerful languages.

What is the difference between a BASIC Interpreter and the C Compiler ?

A BASIC Interpreter (or any interpreter) reads a source file written in BASIC (from disk, tape, or RAM) and "interprets" each line or statement every time the line is executed. Take for example the BASIC statement " B = A * C + 32/7 ", every time this statement is executed (it may be inside a loop of 100 iterations) the BASIC interpreter must take the number 32 convert it to hexadecimal, take the number 7 and convert it to hex, divide them, find the memory location of C, determine its type (integer, single precision, double precision, etc.), convert the quotient to the same type, add the quotient and C, find the memory location of A, determine its type, convert the sum to the same type, multiply the sum times A, find the memory location of B, determine its type, store the results of the multiplication in B. Needless to say this takes time.

The actual execution of A * C + 32/7 is but a fraction of the time expended looking up variables, determining their type, converting types, etc. The C Compiler (or any compiler) reads a source file and "compiles" the file into object (machine language) code. This means that all the variable locations are determined one time and when the code is executed the computer knows exactly where the variables and constants are located. This plus the fact that the C compiler subroutines are generally more efficient (because a compiler is less flexible than an interpreter) allows tremendous gains in execution speed.

As an example a program to determine all the prime numbers between 1 and 10,000 ran for 4 minutes and 50 seconds (on a 6809 system with a 2 megahertz clock) when written in BASIC interpreter mode. The same program written in C for the same system ran in 3.2 seconds, or 90 times faster.

The other major advantage of a compiled program over a BASIC interpreter program is memory savings. In a BASIC program every line of the program must be in memory while it is executing. Each of the lines must be read before BASIC determines the processing required. This includes comments and every unused space in the program. When you see a BASIC program that is all bunched up without spaces or comments, the programmer was probably trying to reduce memory requirements and speed up execution. With a compiler there are actually three versions of the program. The first is the program source written in C, FORTRAN, COBOL, or even BASIC. The compiler "compiles" this program and produces an "assembly language" version of the program (assembly language looks like " LD HL,(40H), ADD HL,DE, LD (44H), HL , ETC."). This version of the program is then "assembled" into machine language (machine language looks like "2A400019224400" for The example above). The machine language version is all that is required in memory to execute the program. The programmer can comment his programs, spread them out with spaces and tabs and practically ignore the memory limitations while at the same time improving his execution speed by a factor of almost 100.

C is a high level language that generates code close to the machine upon which it is intended to run. C is designed in such a way that it generates efficient code, very close to the efficiency that could be obtained by writing in assembly language (without all the problems that assembly language programs present in documentation and maintenance). In fact most C compilers generate assembly language code as an output that is then run through an assembler for the target machine. This provides the opportunity for the programmer who just cannot resist "bit fiddling" to "optimize" the assembly code for his special routines while maintaining the "mundane" code in the original C output version.

What is DGS 6809 C?

DGS 6809 C is a compiler which compiles a subset of the C language. The aim is not to support the full C language, but rather to support enough of a subset to be able to create C programs which would be compatible with standard C. Then, as the compiler expands, more and more features could be added to bring it close to its full capabilities.

Section 2. SCOPE

This manual has been designed to provide the user with the details on the features of C available in this implementation and how to use them. To obtain a detailed tutorial on the usage of the C language, consult The C Programming Language by B. W. Kernighan and D. M. Ritchie: Prentice-Hall.

The rest of this manual is presented as follows;

Section 3	How to use the Compiler
Section 4	C Program Source Structure
Section 5	C Program Statements and Use
Section 6	DGS 6809 C Runtime Library
Section 7	TRS-80 Color Computer C Compiler VS Standard C Compiler

Abbreviations:

The following list of terms, symbols, and abbreviations is used throughout this manual.

arg1,arg2 Abbreviation for argument used as input to a function

char charater data type (reserved word)

cr Carriage return or line feed or just return depending upon the terminal used.

func Abbreviation for function

int integer data type (reserved word)

{} Braces used to define beginning and ending of a group of statements within function or main program

NOTE: Braces ({}) are not available on the Color Computer. The "at sign" (@) will be used for the left brace. The "dollar sign" ($) will be used for the right brace. The braces will be shown in parentheses when the "@" and "$" are used to remind the user.

@ $ Color Computer substitute for braces ({}).

/* */ Delimiters of a comment. Anything between "/*" and "*/" is ignored by C.

" " Delimiters of a string of characters. Defines a string.

(Abbreviations continued)

' '	Delimiters of a character constant.
[ ]	Contains size of an array or indicates pointer to an array (if "[]").
;	Statement terminator.

COLOR COMPUTER CHARACTERS:

The Color Computer has a limited character set with which to implement the C Compiler. The following list indicates the Color Computer characters that should be substituted for "standard" C characters that are not available;

Standard C Characters	Color Computer C
Upper Case A-Z	Upper Case A-Z
Numerals 0-9	Numerals 0-9
Lower Case a-z	Lower Case a-z
	(Shift 0 (zero))
Special Characters	(Shift 0)
=,!,",#,%,&,',(,),*	Same
+,-,/,$,@	
[	(Shift DOWN ARROW)
]	(Shift RIGHT ARROW)
|	#
^	(UP ARROW)
{	@
}	$
<-	(Shift UP ARROW)

Section 3. HOW TO USE THE DGS 6809 C COMPILER

The compiler source can be created using any standard Color Computer editor. The source lines must contain a 'cr'(carriage return) for end of line and the characters must be standard ASCII. **Do not** use any form of compression on the source file when saving it. The compiler only recognizes **ASCII** code.

To start the compiler perform the following operations:

load the **C** compiler into the machine:

type **LOADM "DGSCC" cr**
type **EXEC cr**

compiler replies:

```
* * * COLOR COMPUTER C COMPILER * * *
          by B. W. Dugger
(c) 1982 DUGGER'S GROWING SYSTEMS

     INCLUDE C-TEXT ON OUTPUT?
```

type **YES cr** (or Y cr) if you want the input source lines to be output as assembly comment lines. Type NO cr (or N cr) to suppress this option. Including the source lines will increase the length of your output file and the listing but will provide a source of documentation for the assembly listing.

compiler replies DEFINE GLOBALS?

type **YES cr** (or Y cr) if you want the external or global data types to be output. If not, type NO cr (or N cr). Globals are variables that are defined externally. If the function or module you are going to compile refers to variables that are defined elsewhere, then answer the question with NO cr (or N cr). When you are ready to compile and link all your functions and subroutines, collect the GLOBAL definitions in a single file or module and compile it, answering yes to the question "DEFINE GLOBALS?". This will cause the compiler to allocate storage for the GLOBAL variables. When you subsequently compile the other functions and modules with this external file, they will access these global definitions . During this compilation process remember to: compile the file with the global definitions first, answering YES to the "DEFINE GLOBALS?" question; compile the subsequent files answering NO to the "DEFINE GLOBALS?" question.

compiler replies STARTING LABEL NUMBER/

type: a number (**1-9999**) cr or cr.

This is the starting label number (ccxxxx) generated by the compiler to handle program flow. If you are going to merge this with another C compiled program, then this will allow you to change the label sequence numbering to avoid duplicate labels. When merging programs make sure you allow enough sequence numbers for all labels within the program. The default is 1.

compiler replies: OUTPUT FILENAME?

type: **filename specification* cr** or cr.

This is the file where the output from the compile will be directed, or if just cr is typed, it is directed to the user's terminal.

compiler replies: INPUT FILENAME?

type: **filename specification * cr** or cr.

C source input file name to be compiled. At this point, the compiler will start analyzing the source input and at completion the compiler will again reply INPUT FILENAME? If there are additional source files to be added to this compile, repeat the above. If not, type **CLIBR/LIB** to include the runtime library or type cr to complete the compile.

* filename specification = filename plus extension . The '"' (double quotes) are not required.

EXAMPLE:

```
          PRIME/TXT
or        PRIME.TXT
```

Section 4. C PROGRAM SOURCE STRUCTURE

The structure of a C program is as follows;

```
(definitions)
#define  AMASK      0377  /* bit mask */
#define  ABUF       132   /* a line of input*/
(externals)
int  count,numb;          /* integer externals*/
char  alph,bet;           /* character externals*/
(main program)
main()                    /* main program*/
{                         /* left brace-groups statement*/
(statements)
count=0;
while( count< 10 & numb < 100 & alph == 'aa')
.
.
.
     bet = funcl(count,numb);
#asm
( assembler code )
#endasm
}                         /* right brace to end group*/

(functions )
funcl(argl,arg2,...argn)   /* function definition */
(argl, arg2, data types)
int  argl,arg2;
char arg3,argn;

{                         /* left brace to start group
                          of statements */
(local data types)
int  count,buf,num;
char aline[132];          /* 132 character array*/
char bca;                 /* character variable */
( statements)
count=0;
while(count<10........
}                         /* right brace to end group
                          of function statements */
```

(Program structure continued)

```
func2(arg1, arg2, ...)  /* next function def */
arg1, arg2 data types
{                       /* start group of statements*/
local data types
.............
}       /* end of program */
```

NOTE: Remember to substitute an @ for left braces ({) and a $ for right braces (}).

Define

The define statement is used to define program constants. A constant (number, string, mask, etc.) following a "#define " will be substituted in the source at each place the identifier appears in the source. **Do not** use semicolons in a define statement since these will also be substituted into the source. This version does not support macro definitions in the "define".

This version does not support the "#undef " so any defines will be active throughout the source file. If an identifier is encountered in the source that has been associated with a define the substitution will take place unless the identifier is enclosed in double quotes.

EXAMPLES:

```
#define  LOOPCNT    100        /* loop counter*/
#define   STEP      3          /* loop step */
#define   START     @ ({)      /* start code */
#define   END       $ (})      /* end code */

main()
     while( count< LOOPCNT )
          START                 /* replaces @ ({) */
          count = count + STEP
          END                   /* replaces $ (}) */
```

Externals

Externals are similar to variable definitions and may have all the same attributes. The difference is that external definitions must appear before any function or statement references them and therefore must be placed ahead of the "main()" statement in a source file. Once an external is defined it is "known " to all subsequent statements and functions in the program and need not be re-defined before use. This version does not support the "extern" statement.

Main

The "main()" statement informs the compiler that this is the beginning of the program. When the program is executed, the statements following "main()" will be executed first. The main statement is also used to delimit externals since by definition any variables defined ahead of the "main()" statement are "externals". Since control will be transferred to the statement following "main()" when the program is executed, a source file should contain only one "main()" statement. When other files are "appended", they must be in function format without any "main()" statement.

In-line Assembly Code

Assembly level code may be included in the source file(s). The compiler treats all statements between the "#asm" and "#endasm" as assembly level code. No syntax or error checking is done on these statements. When including low level code in your source file, you should make certain that the code conforms to the rules of the assembler for your machine. For instance the symbology defining comments (/*, */ in C) may be an asterisk in your assembler.

EXAMPLES:

```
        #asm
        SECT4     LDA 0,X              Note ASCII Equiv
                  ANDA  #$7F
                  CMPA  #$1F
                  BHI   SECT5
                  ...
        #endasm
```

Section 5. C LANGUAGE STATEMENT REFERENCES

The following keywords are reserved by the compiler and should not be used as labels, variable names, array names ,etc.

RESERVED WORDS

asm
break
char
continue
define
else
endasm
if
int
return
static
while

TYPES, OPERATORS AND EXPRESSIONS

Variable Names

Names are made up of letters and digits; the first character must be a letter. Upper and lower case are recognized as different characters (A-Z= 65-90 decimal in ASCII, a-z = 97-122 decimal). The traditional C practice is to use lower case for variable names, and all upper case for symbolic constants.

Only the first eight characters of an internal name are significant, although more may be used. For external names such as function names and external variables, the assembler will recognize only six significant characters. Certain keywords are reserved (if,while,return,etc.) and are illegal to use for variable names.

EXAMPLES:

legal:
MAXLINE
A2345
ABCDEF
abcdef

illegal:	2adata	(starts with number)
	?scanfz	assembler must have a-z 1st character
	datatypeb	(not illegal but duplicates
	datatypea	following label)
	define	(reserved word)

Data Types and Sizes

DGS C contains the data types int and char. Int is integer data type and char is character data type. Do not confuse these with the BASIC "INT" and CHR$" which actually convert a value to integer and character.

Data Types

char	a single byte, capable of holding one character of data
int	an integer, a two byte signed value with range of + or - 32767.

EXAMPLES:

```
int xray;           /* integer variable */
char papa;          /* character variable */
```

Constants

DGS C for the Color Computer allows the following constant definitions;

- a decimal number (1,2,344,677, 24980,etc)
- a single ASCII character enclosed in single quotes ('A', 'a', 'z', 'f', etc)
- a string enclosed in double quotes, such as; "this is a string".
- The value constant yielded is a pointer to the first character of the string (*buf, *char, *s, etc.)

Constants are used to define a value to the program. They permit the programmer to assign a meaningful name to a value such as "pi" defined as "3.14159". By using a constant rather than the actual value in the program you also improve the readability of your programs and facilitate updating or correcting.

The following are examples of legal and illegal constant definitions.

EXAMPLES:

```
legal:    name = "datafile";
          data = 'a';
          MAXLINE = 100;
          sign = 'D';
          loc = *buf;
          strng = " DATE                TIME";
          mile  = 5280;

illegal:  code = 'cd';    (to many characters for char
                               variable- C will use d )
          value = 65666;   ( out of range for integer
                               must be -32767 to +32767)
          val   = 32500_  (no semicolon terminator )
```

Comments

Any characters between /* and */ are ignored by the compiler; they may be used freely to make a program easier to understand. Comments may appear anywhere a blank or new line can. As in the following example.

```
Main()
/* this a multi line comment.  As long as the
end comment is not used the compiler will ignore
the character stream. */
```

NOTE: If the end "*/" is omitted, all statements that follow will be considered as a continuation of the comment!!

Declarations

All variables must be declared before use. A declaration specifies a type (int or char), and is followed by a list of one or more variables of that type.

EXAMPLES:

```
int lower, upper, down;    /* lower, upper, down all
                              integer variables */

char k, line[200];         /* k = char variable, line
                              = 200 byte char array */

int array[23];              /* array= 46 byte
                           integer array  */
```

Type Modifiers

Allowable modifiers of the basic types are:

type *name - declares name to be a pointer to an element of the specified type.
type name[] - syntactically identical to the above.

EXAMPLES:

```
int *knum;
int knum[];
char kchr[];
char *kchr;
```

type name[constant] - declares an array of "constant" size where each array element is of the specified type. Examples:

```
int narray[100];     /* narray = 100 value integer
                           array */
char alpha[132]      /* alpha  = 132 character
                           character array */
```

"int narray[100]" is the BASIC equivalent of
DIM NARRAY(100)

"char alpha[132]" is the BASIC equivalent of
DIM ALPHA$ (132)

Arithmetic Operators

The binary arithmetic operators are +,-,*,/, and the modulus operator %. There is a unary -, but no unary +. Integer division **truncates** any fractional part. The expression x%y produces the remainder when x is divided by y, and thus is zero when y divides x exactly.

```
EXAMPLES: (Assume x=8, y=2, z=3, and w=7)
          w = x+y;   ( w is now = 10)
          w = x-y;   ( w is now = 6 )
          w = x*y;   ( w is now = 16)
          w = x/y;   ( w is now = 4 )
          w = x%y;   ( w is now = 0 )
          w = x/z;   ( w is now = 2 - remember integer division
                       truncates fractional part)
          w = y/z;   ( w is = 0 )
          w = x%z;   ( w is = 2 - in this case 2 is the remainder
                       not the quotient )
          w = (x*y)- (x*z);   ( w is = -12 )
```

EXAMPLE:

A year is a leap year if it is divisible by 4 but not by 100, except that years divisible by 400 are leap years. Therefore:

```
 if(year % 4 == 0 & year % 100 !=0 | year % 400 == 0)
   ( Translation: If variable "year" modulo 4 equals 0
     and variable "year" modulo 100 is not equal to 0
     or variable "year" modulo 400 equals 0 )

          /* it's a leap year */
 else
          /* it's not */
```

```
     BASIC equivalent;
100  IF YEAR/4 = INT(YEAR/4) AND YEAR/ 100 <> INT(YEAR/100)
     THEN GOTO 500
200  IF YEAR/400 = INT (YEAR/400) THEN GOTO 500
300  REM ITS NOT A LEAP YEAR
400  .....
500  REM IT'S A LEAP YEAR
510 .....
```

Relational and Logical Operators

Comparison Operators

Comparison operators compare two expressions and yield either zero or a one depending whether the result of the compare is false or true, respectively.

Comparison operators are:

```
"==" - test for equality
"!=" - test for inequality
"<"  - test for less than
">"  - test for greater than
"<=" - test for less than or equal to
">=" - test for greater than or equal to
```

```
EXAMPLES:( Assume; x = -3, y = 2, z = 5, w = 4 )

          if( x+y == 0 )     (will yield 0, or false)
          if( y+z != 0 )     (will yield 1, or true )
          if( x != y   )     (will yield 1, or true )
          if( y+z >= 7 )     (will yield 1, or true )
          if( x+y >= 0 )     (will yield 0, or false)
          if (x > y )        (will yield 0, or false)
          if (x != (y-z))    (will yield 0, or false)
```

Binary logical operators

Binary logical operators (Boolean operators) compare two binary values on a bit by bit basis and set or clear bits in the result (true = set, false = clear).

```
"|" - yields the logical inclusive "or"
      of the expressions (use "#" for color computer).

"&" - yields the logical "and".

"=" - assigns the value of the expression
      on the right to the one on the left.
      Since evaluation is done right to left
      syntaxes like:
          x = y = z = 0;
      are legal.
```

```
EXAMPLES: (Assume; x=-3, y=2, z=7, w=9 )

          if( x>0 | y>0 )       (will yield 1, or true since y=2)
          if( x<=0 & y>= 0)     (will yield 1, or true )
          if( x+y>=0 & y+z>5) (will yield 0, or false )
```

Unary expression operators are:

"-" - forms the two's complement of the expression (minus).
"*" - refers to the element pointed to by the expression (providing the expression is a pointer).
"&" - evaluates the address of the given expression, providing it has one. Hence, "&count" yields the address of the element "count". &1000 is an error.
"++"- increments the expression by one. If this appears before the expression, it increments before using it. If it appears after it, it will increment it after. If this operator is applied to an integer pointer, it will increment by 2.
"--"- decrements the expression by one. This works just like "++" but subtracts one rather than adding one.

"!" - this operator produces a 0 (false) if the value of the expression is non-zero (true), and a 1 if the value is zero (false).

```
EXAMPLES: ( Assume; x = -3, y = 2, z = 5, w = 4 )
     w = -y;              (w = -2 in twos complement form)
     *w = &x;             (w = a pointer variable containing
                          the address of the variable x )
     w = ++y;             (w = 3, y = 3 )
     w = y++;             (w = 2, y = 3 )
     w = --x;             (w = -4)
     ++y;                 (y = 3)
     --y;                 (y = 1)
     w = x = y = z;       (w,x,y, and z all = 5 )
```

Bitwise AND, inclusive OR, exclusive OR

AND (&)

The bitwise "and" operator is used to compare two **integer** variables or a variable and a "mask" (a bit pattern of ones and zeros). For each bit set (=1) in the mask the corresponding bit in the other variable remains the same. For each bit not set (=0) the corresponding bit in the other variable is cleared.

```
EXAMPLES: ( Assume x= 255, y= 15, and mask= 00001111 in binary)
          c= x & mask;        (c is now =  15 decimal)
          c= y & mask;        (c is now =  15 decimal)
          c= x & 0360;        (c is now =  240 decimal)
```

Bitwise inclusive OR

The bitwise inclusive or is used to compare two **integer** variables or an integer variable and a mask. For each bit set (=1) in the mask the corresponding bit in the other variable is set. For each bit not set in the mask the corresponding bit in the other variable is ignored.

Examples: (Assume x= 0, y= 15, and mask = 01010101 binary)

```
c= x | mask        (c is now =  85 decimal )
c= y | mask        (c is now =  95 decimal)
c= x | 0177        (c is now =  15 decimal)
```

NOTE: # is used instead of the vertical bar.
(c= x # mask, etc.)

Bitwise exclusive OR "^" (XOR)

The "exclusive or" is used to compare two **integer** variables or an integer variable and a mask. The comparison is made on a bit by bit basis. If the bits are the same (both zero or both one) the result is a zero. If the bit in the mask is different (mask bit is zero and variable bit is one or vice versa), the result is a one.

Examples: (Assume x= 227 , y= 187, and mask= 133)

```
c = x ^ mask       ( c is now = 102)
c = y ^ mask       ( c is now = 62)
c = x ^ 133        ( c is now = 98)
```

NOTE: The difference between the "inclusive or" and the "exclusive or" is as follows. In a compound "inclusive or" expression; for example
"if(a==2 | b==3)"
if either expression is true, the value is true. In a compound "exclusive or" expression; for example
"if(a==2 ^ b==3)"
if both expressions are false or both expressions are true, the value is false! Only if one expression (not both) is true, is the value equal to true.

Left Shift-Right Shift "<<",">>"

An unsigned shift capability is available by using "<<" for left shift and ">>" for right shift.

EXAMPLE: (Assume X= 4)

```
C = X << 1         (C is now equal to 8)
C = X >> 2         (C is now equal to 1)
```

EXAMPLE:

Example of a function to convert an input to an integer:

```
atoi (s)      /* convert s to integer */
char s[];
{
   int i, n;    /*define i, n as integers */
   n = i = 0;   /* set i, n to 0 */

/* While the character is greater than or equal to zero
  ( 48 decimal in ASCII) and less than or equal to nine
  ( 57 decimal in ASCII) */

    while( s[i] >= '0' & s[i] <= '9')

    /* Set n to ten times n plus the value of the
    character (48 to 57 decimal) minus the value of
    an ASCII zero (48 decimal) */

        {  n = 10 * n + s[i] - '0';

           ++i;     /* increment i */
        }           /* end of statements*/
   return;          /* return to calling program */
}                   /* end of function */
```

NOTE: Don't forget "{" and "}" are "@" and "$" respectively

The same program in BASIC for comparison:

```
1000      REM CONVERT AN INPUT TO INTEGER
1010      DEFINT I,N                 'DEFINE I,N AS INTEGER
1015      DIM C$(10)                 ' INPUT IN C$, OUTPUT N
1020      N=0:I=0                    'CLEAR N AND I
1040      REM PERFORM THE FOLLOWING WHILE THE CHARACTER
1060      REM IS GREATER THAN OR EQUAL TO 0
1080      REM AND LESS THAN OR EQUAL TO 9
1100      REM (ASSUME 10 CHARACTER STRING INPUT-LSB IN
          C$(10))
1120      FOR I = 1 TO 10
1140      IF C$(I) >= "0" AND C$(I) <= "9"  THEN
          N= 10 * N + VAL(C$(I))   ELSE RETURN
1160      NEXT I
1200      RETURN
```

CONTROL FLOW

Statements and Blocks

An expression such as x = 0 or i++ or scanf(...) becomes a statement when it is followed by a semicolon, as in

```
x = 0;
i++;
scanf(...);
```

In C, the semicolon ";" is a statement terminator.The brace "{" (@) and "}" ($) are used to group declarations and statements together into a compound statement or block, syntactically equivalent to a single statement. Braces surround statements of a function; multiple statements after an if, else, while, etc.

If - Else

The if-else statement is used to make decisions. Formally, the syntax is;

```
if( expression )
        statement-1
else
        statement-2
```

where the else part is optional. The expression is evaluated; if it is "true" (if expression has a non-zero value), statement-1 is performed. If it is false (expression is zero) and if there is an else part, expression-2 is performed.

IN C

```
if (n > 0)
   if (a > b)
     z = a;
   else
     z = b;
else .....
if(n > 0) {
     if(a > b)
        z = a;
     }
     else
        z = b;
```

IN BASIC

```
100 IF N<=0 THEN GOTO 500
150 IF A <= B THEN GOTO 300
200 Z=A : GOTO 500
300 Z=B

500 .....
```

NOTE: Don't forget "{" = "@" and "}" = "$"

Else-if

The construction;

```
if( expression)
     statement
else if(expression)
     statement
else if( expression )
        statement
else
        statement
```

is the most general way of writing a multi-way decision. The expressions are evaluated in order; if any expression is true, the statement associated with it is executed, and this terminates the whole chain. The optional else handles the default case where none of the other conditions were satisfied.

EXAMPLE: This function will search an array "v" and locate the value "x". The value "n" is the highest location in "v" to search. The example uses the "else if " construct.

```
binary(x, v, n)          /* find x in v[0]...v[n-1] */
     int x, v[], n;
{
        int low, high, mid;      /* declare integers */
        low = 0;                 /* set low to 0 */
        high = n-1;              /* set high to n-1 */
        while(low <= high)  {
                mid = (low+high)/2;
                if(x < v[mid])
                        high = mid-1;
                else if(x > v[mid])
                        low = mid+1;
                else   /* if x not less and not greater
                     found match */
                        return(mid);
        }
        return(-1);
}
```

While statement

While(expression) statement

The statement is performed until the expression becomes zero or false. Since the test is made before the statement is executed the first time, it need not be executed at all.

```
while((c = getchar()) == ' ' | c== 'n' | c=='t');
```

EXAMPLES:

```
while (1);    /* do forever */
while (n < 100); /* 100 iteration loop */
```

Break and Continue statements

break;

This statement will cause control to be transferred out of the innermost "while" loop. The next statement to be executed will be the statement immediately following that loop.

continue;

This statement, used within a "while" loop, will transfer control back to the top of the loop.

Return <expression> statement

return; and return (expression);

The " return; " statement does an immediate return from the current function to the calling function. The " return (expression); " statement allows a function to return a value explicitly. In the example below the function " getchar() " ends with a " return (arg); " where arg contains the character input at the terminal.
A function normally ends with a return statement, one is performed regardless. A function may contain several return statements to indicate various conditions encountered. An error encountered in a function may cause a " return (-1) " which would indicate the error condition to the calling function. The location returned to will be the first executable statement following the function call. Function calls imbedded in complex statements will return to the processing of the complex statement.

Example:

```
if (( c=getchar()) != "a")
```

In this statement the function "getchar()" will be called, the input character returned, and processing will continue by assigning the value of the input character to "c". Notice the parentheses around " c= getchar() " to ensure the proper order of execution of the statement.

(;) semicolon statement

The semicolon is used as a statement terminator. A semicolon by itself is considered a null statement which does nothing but take the place of a statement.

FUNCTIONS AND PROGRAM STRUCTURE

Functions break large computing tasks into smaller ones, and enables modular and efficient coding. C programs generally consist of numerous small functions rather than a few big ones. A program may reside on one or more source files in any convenient way; the source files may be compiled separately and assembled together, along with compile functions of the libraries or combined together during the compile.

Each function has the form;

```
name( argument list, if any )
argument declaration, if any
{
        declaration and statements, if any
}
```

As suggested, the various parts may be absent; a minimal function is;

```
dummy() {}
```

which does nothing.

A Program is just a set of individual function definitions. Communication between the functions is (in this case) by argument and values returned by the functions; it can also be via external variables. The function can occur in any order in the source file as long as the first function is a main function, preceded by data definitions. These data definitions will be treated as external definitions (accessible by all functions).

EXAMPLE:

MAIN:

```
main()          /* Compute factorial of input */
char str[40], str1;
int c,i,k;
i=k=1;
str1 = "  number you wish factorial of ";
     while(1)  /* do forever */
     {         printf( " enter %s", str1 )
               scanf(%d,&c)
     while( c >= i ) {
     k *= i;              /* k = k * i */
     i ++ }
     printf(" Factorial = %d",k)
     exit(): }
```

The functions are "printf","scanf", and "exit" which are all generated by the compiler.

Section 6. DGS 6809 C RUNTIME LIBRARY

The DGS 6809 C RUNTIME LIBRARY contains source code of functions which provide compiler generated functions, input/output functions, and utility functions. The compiler functions are the functions which handle statements and procedures which require excessive amounts of coding to perform.

Compiler Generated Functions:

CCOR - logical OR of two 16 bit values

CCAND - logical AND of two 16 bit values

CCXOR - logical XOR of two 16 bit values

CCASL - arithmetic shift left of a 16 bit value n places

CCASR - arithmetic shift right of a 16 bit value n places

CCCOM - complement a 16 bit value

CCNEG - negate or 2's complement of a 16 bit value

CCMULT - multiply two 16 bit integer values return 16 bit value

CCDIV - divide two 16 bit integer values return 16 bit value

CCMOD - modulo divide two 16 bit values return bit remainder

The next group of functions are input/ouput functions. I/O facilities are not part of the C language. In order to communicate externally from a C program the following routines have been provided. These functions allow communication between a terminal and the running program, and also the handling of I/O to a disk file.

Input Output Funtions:

c=getchar() - returns the next input character from the user's terminal and places it in variable c. This allows the users to control the input from his terminal.

putchar(c) - puts the character c out to the user's terminal. This allows the user to control the output of each character to his terminal.

printf(control, argl, arg2, ...) - formatted output. Printf converts, formats, and prints the arguments on the user's terminal. The control string contains two types of objects; ordinary characters, which are simply copied to the output stream, and conversion specifications, each of which causes conversion and printing of the next successive argument to printf. Each conversion specification is introduced by the character % and ended by a conversion character. The conversion characters are:

d - The argument is converted to decimal notation.
c - The argument is taken to be a single character.
s - The argument is a string; characters from the string are printed until a null character is reached.

EXAMPLE: printf("%d people %s her%c",number,str,'e')

where: number= integer containing 10
str= char array containing "are"

results output to the terminal:
10 people are here

scanf(control, arg1, arg2, ...) - read characters from the user's terminal, interprets them according to the format specified in control, and stores the results in the remaining arguments. The control argument is described below; the other arguments, each of which must be a pointer, indicate where the corresponding converted input should be stored. Conversion specifications, consist of the character "%" and the conversion character. An input field is defined as a string of non-space characters; it extends either to the next space or carriage return. The conversion character indicates the interpretation of the input field; the corresponding argument must be a pointer as required by the C language. The following are the conversion characters:

d - a decimal integer is expected in the input; the corresponding argument should be an integer pointer.

c - a single character is expected; the corresponding argument should be a character pointer.

h - a short integer 1 byte is expected; the corresponding argument should be a pointer to a char.

s - a character string is expected; the corresponding argument should be a character array pointer.

EXAMPLE:

```
scanf("%d%h%c%s",*number,*csd,*cin,strin)
      where :
           int number;
           char csd,cin,strin[];

      then typing 100 3 w hello

      results in:
                 number = 100
                 csd    = 3
                 cin    = w
                 strin[] = hello
```

fp = fopen(name, mode, file #)-

Open a disk file. The 1st argument is the name of the file, as a character string. The second argument is the mode, also a character string, which indicates how to use the file. Allowable modes are read ("r") or write ("w"). The third argument is an integer (1-4) which indicates the file number.

NOTE: Opening an existing file for writing ("w") or opening a non existant file for reading ("r") will cause an error.

EXAMPLE:

```
fp=fopen(fildat,"w",filnn)

 where: char fildat[12]
        int filnn;
        fildat="filename.specification";
        char *fp; /* pointer */

 resulting in:
        return's status in fp
       0        = bad
       positive = good
```

This opens the named file for writing

fclose(fp) -

Closes the file specified by the fp. The fp is a pointer to the file's control block.

EXAMPLE:

```
   fclose(fp);
```

c= getc(fp) -

Read the next character from the file referred to by fp, and place in c. EOF is indicated by a -1.

EXAMPLE:

c = getc(c, fp) - read character from the file referred to by fp and place in c..

putc(c,fp) -

Write the next character to the file referred to by fp.

EXAMPLE:

```
char inname[12];
char c;
int i;
i=0;
while((c=getchar())!=13) /* input to a cr */
        {inname[i++]);putc(c,fp);}
```

the above statements will read from the terminal into character array inname, and write to the file referred to by fp.

Utility Functions

The following functions are utility functions which assist the user in programming his routines. It should be noted again that the user can add his own functions to this runtime library or make a separate file of his functions and include them during compiling (INPUT FROM?).

exit(): - exits from the running program back to the operating system.

isalpha(c) - non-zero if c is alphabetic, 0 if not

isupper(c) - non-zero if c is upper case, 0 if not

islower(c) - non-zero if c is lower case, 0 if not

isdigit(c) - non-zero if c is digit, 0 if not

isspace(c) - non-zero if c is blank, tab or newline, 0 if not

toupper(c) - convert c to upper case

strcmp(str1, str2) - non-zero string compare, 0 if not

strcpl(str1, str2, length) - non-zero string compare, 0 if not

The users are encouraged to send to DGS functions which they have implemented and DGS will in turn provide listings of these functions to other users.

Section 7. THE DIFFERENCES BETWEEN DGS C AND STANDARD C

As stated in the introduction, DGS C is a subset of the standard UNIX C. All of the basic operators have been provided in DGS C. The operators that are missing are operators which provide enhancements to the basic language. With the operators provided, a user can write a program which will perform the same as standard C. DGS plans on providing additional operators in later versions of the compiler.

The following table lists all of the operators which are available in standard C. The table also indicates those operators provided by DGS C or the version in which they will be available and provides an alternate operator to perform the same operation for C operators not included in this version.

STANDARD C	DGS avail.	available version	alternate
int	yes	version 1	
char	yes	version 1	
float	no	version 2	
double	no	version 3	
struct	no	version 3	
union	no	version 3	
long	no	version 2	
short	no	version 2	
auto	no	version 2	
register	no	version 3	
typedef	no	version 3	
static	no	version 3	
goto - label	no	version 2	
return	yes	version 1	
sizeof	no	version 3	
break	yes	version 1	
continue	yes	version 1	
if	yes	version 1	
else else if	yes	version 1	
for	no	version 2	while
do - while	no	version 2	while
while	yes	version 1	
switch - case	no	version 3	if-else-if
default	no	version 3	

Appendix A. SAMPLE DGS 6809 PROGRAM

```
#define MAXLINE 1000
#define EOF -1
#define EOL 13
#define NULL 0
char line[maxline];
/* find all lines matching a pattern */
main()
{
while(GETLINE(LINE, MAXLINE) > 0)
                  if(INDEX(LINE, "THE") >= 0)
                        printf("%S", LINE);
}
GETLINE(S, lim)         /* get line into s, return length */
char S[];
int lim;
{
int C, I;

I = 0;
        while(--lim > 0 & (C=GETCHAR()) != EOF & C != EOL)
                S[I++] = C;
        if(c == EOL)
                S[I++] = C;
        S[I] = NULL;
        return I;
}
INDEX(S, T)           /* return index of t in s, -1 if none *?
char S[], T[];
{
init I, J, K;
        I = 0;
        while(S[I] != NULL) {
                J = I;
                K = 0;
                while()t[K] != NULL & S[J] == T[K])
                          {  J++; K++; }
                if(T[K] == NULL) return I;
                I++;
        }
        return(-1);
}


SHELL(V,N) /* SORT V[0]....V[N-1] INTO INCREASING ORDER */
int V[], N;
{
int GAP, I, J, TEMP;
```

```
GAP=N/2;
while(GAP>0)
{I=GAP;
        while(I<N)
                {J=I-GAP;
                while((J>=0 & (V[J]>V[J+GAP]))
                   {TEMP=V[J];
                   V[J]=V[J+GAP];
                   V[J+GAP] = TEMP;
                   J+J-GAP;
                   }
                 I++;
                 }
       GAP=GAP/2;
       }
}


#define SIZE 8190
/* Prime number program in C */
char PRIME[8191];
int COUNT,I,J,K,L;
MAIN()
{
L=0;
while(L,10)
{
COUNT = I = 0;
       while (I<=SIZE){PRIME[I++]=1;}
I=0;
while(I<=SIZE)
       { if (PRIME[I]]!=0)
            { J=I+I+3; K = I+J;
              while(K<=SIZE)
                 {PRIME[K]=0;
                  K=K+J; }
              COUNT=COUNT+1;
             }
       ++I;
       }
printf("%D"S/N",COUNT," PRIMES");
++L;
}
}
```

Appendix B. DGS 6809 C COMPILER ERROR CODES

The DGS 6809 C compiler indicates syntax errors by pointing to the offending position within the statement. There is an attempt to correctly report this, but at times, the error is really generated in the previous statement. Many error reports can be encountered if the offending error occurs at the beginning of a statement. The errors have the following form.

```
if a>b) data=b;
  ^ *  ******MISSING OPEN PAREN******
```

The following is a list of errors which can be encountered.

MISSING CLOSING BRACKET

OPEN FAILURE - encountered during initial input or output file request.

INCLUDE OPEN FAILURE - error in opening an #include request.

MUST BE CONSTANT - error in type declaration of an array.

NEGATIVE SIZE ILLEGAL - array declaration with a negative size.

ILLEGAL FUNC OR DECLAR - illegal function call or declaration.

MISSING OPEN PAREN - syntax required an open parenthesis.

ILLEGAL ARG NAME - an illegal character was used argument naming.

EXPECTED COMMA - argument not separated by a comma.

WRONG #ARGS - wrong number of arguments.

ILLEGAL NAME - use of an illegal character in a name.

MISSING SEMICOLON - expected a semicolon in this position.

ILLEGAL SYMBOL NAME - use of different name than was declared.

ALREADY DEFINE - symbol was defined previously.

MISSING BRACKET - missing open or closing bracket.

GLOBAL TABLE OVERFLOW - too many global symbols, resize global table.

LOCAL TABLE OVERFLOW - too many locals defined in one function. Resize.

TOO MANY ACTIVE WHILES - resize the WQ table.

MISSING APOSTROPHE - missing closing apostrophe character declaration.

LINE TOO LONG - more than 80 characters in a line.

OUTPUT FILE ERROR - encountered an error on output.

CAN NOT SUBSCRIPT - using a variable as an array.

ILLEGAL ADDRESS - defining an absolute address incorrectly.

INVALID EXPRESSION - syntax not correct.

STRING SPACE EXHAUSTED - literal table overflow resize.

xx ERRORS IN COMPILATION - indicate how many errors encountered.

Appendix C. DGS 6809 C COMPILER TROUBLE REPORT

VERSION NUMBER

NAME_____

ADDRESS_____

CITY_____

STATE_____

ZIP_____

DESCRIPTION OF THE PROBLEM;

DESCRIBE CODE USED TO CAUSE PROBLEM

SEND TO: DUGGER'S GROWING SYSTEMS
P.O. BOX 305
SOLANA BEACH, CALIF 92075

I N D E X

!= 5-6,5-12,6-5,A-1
% 5-5
& (&var) 5-6,5-7
' ' 2-2,5-3,5-9,5-11,6-2
* 5-5,5-7,5-9
* (*var) 5-7
+ 5-5,5-6,5-7
- 5-5,5-7,5-8
/ 5-5
/* */ 5-3,2-1,4-1,4-2,5-2,
5-4,5-5,A-1,A-2
; 5-12,2-2,4-1,5-2,5-3,5-4
5-5
< (less than) 5-6,5-11,5-12
<< (left shift) 5-8
<= (lt or eq) 5-6,5-11,A-2
== (identity) 5-6,5-11,A-1
> (gt) 5-6,5-10,5-11,5-12,A-1
>= (gt eq) 5-6,5-9,A-2
>> (right shift) 5-8
@ 2-1,2-2,4-2,5-10,
Argl 2-1,4-1,4-2,6-2
ASCII 3-1
And (&) 5-7,5-6,A-2
Arithemetic Operators 5-5,5-6,5-7,5-8,5-2
Asm (#asm) 4-3,5-1
Binary Operators 5-6,5-13,A-1,A-2
Break 5-12
Char 5-2,4-1,5-1,5-3,5-4,5-13,6-5
Comments 5-3,2-1,4-1,4-2,5-2
Comparison Operators 5-6,5-9,5-10,A-1,A-2
Constants 5-2,5-3,4-2,A-1,A-2
Continue 5-12
Cr 2-1,3-1,3-2
DEC PDP-11 1-1
Data Types 5-2,5-4,5-13,A-1

Declarations 5-3,2-1,4-1,4-2,4-3
Define 4-2,A-1,A-2,B-1
Double Quotes (" ") 5-2,2-1,4-1,5-3,5-13
Dummy () 5-13,5-12
Else-if 5-11
Endasm (#endasm) 4-3,5-1
Errors b-1,4-3,6-4,7-1
Examples 2-1,3-1,3-2,4-1,4-2,4-3,5-1,5-2,5-3,5-4
Exclusive Or 5-8
Externals 4-3,A-1,A-2,B-1
Fclose 6-4,A-1
Fopen 6-4,A-1
Functions 2-1,4-1,4-2,5-13
Getchar 6-2,6-5
Getc(fp) 6-4,A-1
If-else 5-10,5-11,8-2
In-line Assy Code 4-3,4-1
Include 3-1
Inclusive Or 5-7,5-8
Int 5-3,2-1,4-1,5-1,5-9,5-11,6-5,7-1
Integer Constant 2-1,5-2
Main () 4-3,5-13,A-1,A-2
Printf 6-2,A-1
Putchar 6-2,6-5
Putc() 6-4
Reserved Words 5-1,5-4
Return 5-12
Right Shift (>>) 5-8
Scanf 6-3,A-1
Statements 5-10,5-11,5-12
Type Modifiers 5-4,5-1,5-2
Unary Operators 5-7,5-13
Undef (#undef) 4-2
Utility Functions 6-5

```
Variable names 5-1,5-4,5-13,A-1,A-2
While      5-11,4-1,5-1,8-2,5-13,A-1
[ ]        5-4,2-2,4-1,5-3,5-5,5-11,5-15,6-4,6-5,7-1,A-2
{ }        2-1,2-2,4-1,4-2,5-10,5-11,5-13,A-1,A-2
           (REMEMBER: "{" = @ and "}" = $)

|        (use "#" for XOR)  5-6,5-8,5-11
```

DUGGER'S GROWING SYSTEMS

POST OFFICE BOX 305 / SOLANA BEACH, CALIFORNIA 92075 (619) 755-4373