Deep Blue C Compiler Version 1.1

INTRODUCTION

Overview

The Deep Blue C Compiler helps you create large programs -- ones that take more than a day to write and contain more than a hundred lines of code -- for your ATARI Home Computer. It lets you write your programs in a subset of the popular programming language "C". C is a general-purpose programming language designed to fill the "Software Gap" between BASIC and Assembly Language. C is more powerful and faster than BASIC, yet clearer and less error-prone than Assembly Language. Pointers, recursive functions, and high-level control structures make complex software systems easy to design, implement, and maintain.

C was created by system programmers as a viable high level alternative to assembly language. While slower running than assembly language, C code is much easier to write and understand. Furthermore, C is the "defacto" systems programming language of the new generation of "workstation computers". Unlike assembly language, you'll be able to transport your valuable C programs to other (especially non-6502-based) computers, with only trivial modifications.

Deep Blue C is a proper subset of version 7 C, which means that programs written for it will run almost without change on computers supporting the full language. The Deep Blue C Compiler is an extensively modified version of Ron Cain's & Brian Smith's public domain Small-C Compiler. It took the author about three months to convert the Small-C compiler into Deep Blue C. While the original Small-C compiler was in the public domain, this version is protected by copyright.

Minimum Ram and accessories:

48K RAM
810 Disk Drive
PROGRAM TEXT EDITOR
(or other no-line-number text editor)

Optional accessories:

ATARI Macro Assembler
Deep Blue C Supports the following

1) char, int, and pointer data types
2) single dimension arrays
3) Unary operators: +,-,*,&,++,--,!,$- (tilde)
4) Binary operators: +,-,*,/,%,|,^,&,==,!=,<,<=,>,>=,<<,>>
.. <op>=,&&,||,?:,comma
5) Statements: if,else,while,break,continue,return,
.. for,do,switch,case,default
6) #define and #include compiler directives
7) Relocating linker

Features of C not supported

1) structures, unions
2) multidimension arrays
3) floating point numbers
4) functions returning anything but int
5) Unary operators: sizeof
6) Binary operators: type casting

Special Syntax

C uses several ASCII characters not available on the ATARI's keyboard -- in particular the curly braces have been replaced by the two-letter combinations $(and $), and the tilde has been replaced by $-. The $ character is not used in C, so your editor's find & replace command can be used to convert standard C programs into a format acceptable to Deep Blue C.

References to related publications

This manual will not teach you C. If you do not know C, you should obtain a copy of "The C Programming Language", by Brian W. Kernighan & Dennis M Ritchie, (C) 1978 Bell Telephone Laboratories, Inc., which is published by Prentice Hall, Inc., Englewood Cliffs, NJ Ø7632. Note that many of the examples in the book use the Unix I/O functions, which are slightly different from the ones supplied with Deep Blue C.

GETTING STARTED

The first thing you must do is make working copies of your Deep Blue C disks. Keep the originals in a safe place, in case you should lose the working copies. Here is an explanation of the files on your disks:

Distribution Diskette

DOS.SYS -- Standard DOS II FMS file
DUP.SYS -- Standard DOS II DUP file

CC.COM -- Deep Blue C Compiler
CLINK.COM -- Deep Blue C Linker
DBC.OBJ -- C run time module.

AIO.C -- source for I/O functions
AIO.CCC -- object for AIO.C
GRAPHICS.C -- source for graphic & game i/o
GRAPHICS.CCC -- object for GRAPHICS.C
PMG.C -- source for player/missile & character set graphics
PMG.CCC -- object for PMG.C
PRINTF.C -- source for formatted output
PRINTF.CCC -- object for PRINTF.C

X.C -- source for demo program
X.CCC -- object for X.C
X.LNK -- link file for X.C
X.COM -- executable version of X.C
BOUNCE.* -- source, etc. for graphics demo

MEDITC.ECF -- PROGAM/TEXT EDITOR
"".C" customization file

Code Diskette

```
CC*.C -- source files for the compiler
CC.LNK -- link file for the compiler

CLINK*.C -- source files for the linker
CLINK.LNK -- link file for the linker

DBC*.MAC -- Atari MACRO ASSEMBLER files
.           for DBC.OBJ

MEDITMAC.ECF -- PROGRAM/TEXT EDITOR
.               ".MAC" customiztion file
```

USING DEEP BLUE C

There are four steps between a C program on paper and an executable machine language file on the ATARI:

1) The program must be entered as one or more source files using a text editor

2) The source files must be compiled into object files by the Deep Blue C Compiler

3) A link file must be created. The link file contains the names of all the object files that are part of the program, and is used by the linker (in step 4) to gather all the parts of the program together.

4) The individual object files that make up the whole program must be linked together into an executable file by the Deep Blue C Linker

Each kind of file has its own extension. Here is a list of the extensions used by Deep Blue C:

```
Source file     -- .C
Object file     -- .CCC
Link file       -- .LNK
Executable file -- .COM
```

Editing a C Source File

Deep Blue C source files contain the text representation of a C program, the comments associated with that program, and the compiler directives needed to compile the program. C is a modern high-level language best edited with a screen oriented text editor. In particular, the Atari Program Exchange's PROGRAM TEXT EDITOR is excellent for editing C programs. If you have this editor you'll find that the file MEDITC.ECF contains the apropriate tab settings for editing C text.

If you have another text editor, you can also use it to edit your C programs. The only requirement is that your editor must not insert line numbers at the beginning of each line. This means you can't use the BASIC or ASSEMBLER/EDITOR editors to edit your C text, unless you write a utility program to strip off the line numbers before compilation.

All C text files should have the extension ".C", as in AIO.C, PRINTF.C and X.C, The ".C" extension is traditional, and is also the default extension assumed by the compiler.

C source text programs may contain all ATASCII characters. The two formating characters TAB (decimal 127) and EOL (decimal 155) are treated as if they were spaces, which means that they can be used to indent the C text in a pleasing manner.

COMPILING A C PROGRAM

Once entered, the C program must be translated into a special code (called object code) before it can be executed. The program that does this translation is called the Deep Blue C compiler. The compiler reads a program from a C file, translates it into object code, then writes the object code into a file with the extension CCC. For example, to compile the program X.C you would do the following:

1) Remove all cartridges from your Atari, turn on the disk drive, insert the distribution diskette, and power on your Atari.

2) When DOS II prints its menu you type L (for Load File), the RETURN key, CC.COM, the return key, and wait for the Deep Blue C Compiler to load.

3) The Deep Blue C Compiler clears the screen and prints its header message:

```
Deep Blue C Compiler version 1.Ø
(C)1982 John Howard Palevich

File to compile (or RETURN to exit)
_
```

Figure 1. Deep Blue C Compiler Display

4) Type in the name of the C text file you want to compile -- where the full name might be D:X.C, you need only type the main part of the file name -- the X, and the rest of the name will default to the D: disk and the .C file. So you need only type X, then the RETURN key.

5) The compiler prints "D:X.C->D:X.CCC", which means that the input file D:X.C is being read in, translated to objectc code, and written out to a file called D:X.CCC. In general, the file Dn:<name>.C will be translated into the file Dn:<name>.CCC

6) The compilation may take several minutes, depending upon the length and complexity of the source program. To give you an idea what it is doing, the compiler prints the name of the current function it's parsing.

7) If you have any syntax errors the compiler will print out the line where it detected the error, an arrow pointing to the point in the line where it detected the error, and a line of text describing the error. An example would be:

```
main()$(ps("Hello, World");
.                          ^
Missing close $)
```

Figure 2. Compiler error message

8) If you have no syntax errors the compiler will print out the reassuring message "No Errors.". In either case, you will again be prompted "File to Compile (or RETURN to exit)". If you have more than one file to compile at a time, you can type the next file name now, followed by RETURN.

9) When the compier has finished compiling your files, press RETURN to go back to DOS II.

LINKING A C PROGRAM

Once the individual files making up the C program have been compiled without error, the whole program can be linked together into an executable file. To link together a C program one must construct a text file, called a "link" file, containing the names of all the files that have to be linked together to produce the complete program.

A typical small program, such as X.C, needs two files in addition to itself: AIO.CCC (the compiled C code of the I/O functions) and DBC.OBJ (the run time package). If you were to print out the file X.LNK you would see that it contains the following:

```
X
AIO
DBC.OBJ
```

Figure 3. X.LNK

The files must have the Dn: prefix if they are on a drive other than drive 1. If no extension is given a ".CCC" extension is assumed. The LNK file cannot contain any blank lines, not even at the end of the file.

Two types of files make up an executable C program -- .CCC files, produced by the compiler, and .OBJ files, produced by the ATARI MACRO ASSEMBLER (or other assembler), which contain machine language. .CCC files are linked together into a C program, while .OBJ files are copied verbatim into the output file.

All C programs MUST include the file name DBC.OBJ in their link file. DBC.OBJ contains the run time routines & the C-code interpreter needed to execute properly. If you use the "asm" keyword (described later) and want to have your own machine language file loaded automatically, then you would list it in the link file too.

Once you've written the link file for your program, you can link it by running the Deep Blue C linker. Put the distribution diskette into the drive, close the door, boot DOS II, and type L, then RETURN, CLINK.COM, then RETURN.

The Deep Blue C Linker will load into RAM and display its message:

```
Deep Blue C linker version 1.0
(C) 1982 John Howard Palevich
Link program, Duplicate file or Quit

_
```

Figure 4. Deep Blue C linker display

Type the first letter (L,D, or Q) of a command, then hit RETURN. Link will construct a working C program out of its parts. Duplicate will let you move small files from one disk to another without resorting to DOS II's O command. Quit will, of course, return you to DOS II.

Duplicating a file

Typing the letter D, then a space, then the name of the file you want to duplicate. The linker will prompt you to insert the source disk, after which you should press RETURN. The linker will read in the file, then prompt you to insert the destination disk, after which you should press RETURN. The linker will write out a duplicate copy of the file onto the destination disk. You can use this command to copy CCC files from the disk where they were compiled onto the disks they are to be linked upon. Except for the limited file size (about five thousand characters) this command acts like the DOS II O command.

Linking a file

Once you have compiled all the files that make up your C program, you must link them together. The L command of the linker is used to do this. To link the separate parts of your program together, type L, space, link file name, then RETURN. An example would be "L X", RETURN, which would instruct the linker to link together the program X.COM according to the directions in X.LNK.

The linker will fail to link if the files you specify do not exist. In addition, if it cannot find a function or external variable declaration it will complain "undefined label: ", and the missing variable's name. If you mistyped a variable name (such as "alhpa" instead of "alpha") the mis-spelling will be reported here.

If there are no errors the linker will print "No errors" before re-printing the "Duplicate, Link, or Quit" prompt.

Exiting the linker

When you've finished duplicating files and linking programs, you can exit the linker by typing "Q" followed by a RETURN.

Running a C program

A compiled and linked C program can be treated like any other executable file -- it can even be renamed AUTORUN.SYS in order to have it boot in when you turn on the disk drive. Like other object code files it should be loaded via the "L" command of DOS II.

Run-time Errors

There are only four errors that can occur at run-time (while the C program is executing). Of these, only the first is common. Should any of them occur, your program will stop and the following message will print out on the screen:

```
dbc 1 run-time-error <letter>
Type a key to return to DOS.
```

figure 5. Run Time Error Message

The <letter> will be one of the following:

A - stack overflowed RAMTOP -- either you are recursing endlessly, or you have defined too many variables.

B - Illegal op-code -- your program has messed up its code area, and tried to execute garbage.

C - version error -- you have versions of CC.COM, CLINK.COM and DBC.OBJ that do not have the same version number.

D - divide by zero -- you've tried to divide a number (or take its remainder) by zero.

Constants

Deep Blue C supports the following types of constants:

decimal numbers like -12, 134, 4500
octal numbers like 017, 045, 017777
hexidecimal numbers like 0xd400, 0xff, 0x2fc
character constants like 'a', 'ee', '0'
string constants like "foo", "bar blatz", "spam"

Any control or inverse video character that can be embedded in a BASIC string (like control-A thru control-Z or the arrows) can also be embedded in a deep blue C string. There is one exception: control-comma (the heart) is used to signal the end of the string & thus should not be used in a string constant.

In addition, the backslash character ('\') is used to generate certain useful characters:

```
\f -- clear screen.  \g -- ring bell    \h -- back space
\n -- EOL (new line) \r -- delete line  \\ -- backslash
\' -- apostrophy     \" -- quote        \t -- tab
```

\###, where ### is a one to three diget octal constant, produces the ATASCII character with that ATASCII code.

Differences from Standard C

The Deep Blue C language has the following non-standard features:

The last clause of a "switch" statement, either "case" or "default", must be terminated with a "break", a "continue", or a "return" statement.

The ancient =<op> construct has been removed. Use <op>= instead.

Characters are unsigned -- chars range in value from Ø to 255.

Strings cannot be continued on the next logical line.

C source code lines can be a maximum of seventy nine characters long.

Functions can have a maximum of 126 arguments.

The Deep Blue C Library

Unlike most other languages, C has no built-in I/O statements. Instead of Basic's PRINT or Pascal's WRITE, C uses functions for it's I/O. While extremely useful, this means that each version of the C language has its own version of the basic input/output functions. Deep Blue C is, alas, no exception, but if you find its mix of pre-defined functions lacking in one way or another, you are welcome to define new i/o functions to fit your needs!

The functions defined in the files AIO.C, GRAPHICS.C, PMG.C and PRINTF.C give you access to the Atari's hardware at about the same level as BASIC. C library functions with familiar names (like plot(), drawto(), and poke() act, on the whole, like their BASIC counterparts.

While the most accurate definition of each function is its C code, here is a description of each function, starting with the functions in the file AIO.C:

```
clear(s,len)
char *s;
int len;
```

clear() puts zero bytes in s[Ø..len-1], which makes it useful for initializing large arrays. For initializing integer arrays the length arguement should be multiplied by two, so that the length is in bytes rather than in words.

```
copen(fn,mode)
char *fn,mode;
```

The copen function opens file the file named in the string 'fn' for reading, writing, or appending, depending upon the value of the character 'mode':

'r' -- read file (like OPEN #n,4,Ø,fn$)
'w' -- write file (like OPEN #n,8,Ø,fn$)
'a' -- append file (like OPEN #n,12,Ø,fn$)

If the file is opened successfully, the IOCB number used by that file (Ø to 7) is returned as the value of the function. You must save this value in a variable in order to be able to actually use the file.

If the file does not open successfully, the function will return a negative number, where the number is the negative of the CIO error code. For example, if you typed the BREAK key while copen was trying to open a file, then copen would return a -128.

```
open(iocb,ax1,ax2,fname)
char iocb,ax1,ax2,*fname;
```

This is the familiar OPEN statement form BASIC. open() returns 1 if there was no problem, otherwise it returns the negative of the CIO error code.

```
close(i)
char i;
```

This is the familiar CLOSE statement from BASIC -- it closes the IOCB & returns 1 or the negative of the CIO error code.

```
cclose(i)
int i;
```

When you want to close a particular file, you should call cclose() with the number returned by copen(). cclose() will return either a 1 (if everything turned out OK) or a negative number (the negative of the CIO error code) if the file failed to close.

Actually, close() and cclose() do exactly the same thing and can be used interchangably. The number returned by copen() is the number of the IOCB that was opened for the file, so files opened with EITHER open() OR with copen() can use all of the rest of the i/o functions.

```
cgetc(iocb)
int iocb;
```

cgetc() is very much like the BASIC GET statement -- you pass is the iocb number and it returns either the next character in the file (which will be between Ø and 255) or a negative number that's the CIO error code.

```
cputc(c,iocb)
char c;
int iocb;
```

cputc() is very much like the BASIC PUT statement -- you give it the character you wish to print and the iocb number and it will print that character into that file. If there is no error, cputc() will return 1, otherwise it will return the negative of the CIO error code.

```
getchar()
```

getchar() will get one character from the screen (iocb Ø) and return it to you (or the negative of the CIO error code).

```
putchar(c)
char c;
```

putchar() will print the character you give it onto the screen (iocb Ø), and return 1 or the negative of the CIO error code.

```
gets(string)
char *string;
```

gets() is like the BASIC INPUT statement -- it will get an entire "logical line" of text from the user and place that line in the character array you give as an arguement. Make sure your character array is at least 12Ø characters long -- otherwise the user could overflow your array by typing in a very long line. If there are no errors, gets() will return the number of characters in the line of input (Ø to 12Ø). If there is an error, gets() will return the negative of the CIO error code.

```
cprints(string)
char *string;
```

cprints() is like the BASIC PRINT statement -- it will print the string you give it out onto the screen. It will NOT print a RETURN, but you can use the statement "putchar(155);" to cause a carriage return.

```
cputs(string,iocb)
char *string;
int iocb;
```

cputs() is like the BASIC PRINT# statement -- it will print the string you give it out to the file you specify. You should use the iocb number that copen() returned. If there are no errors, cputs() will return a 1, otherwise it will return the negative of the CIO error code.

```
ciov(iocb,com,bad,blen,ax1,ax2)
int iocb,com,blen,ax1,ax2;
char *bad;
```

ciov() is like the BASIC XIO call -- you can set up the iocb of your choice, then call the CIO via this function. The arguement iocb should be between Ø and 7, and specify which i/o control block you are using. COM is the ICCOM command code, bad is the ICBAD buffer address, blen is the ICBLEN buffer length, ax1 is the ICAX1 auxiliary byte, and ax2 is the ICAX2 auxiliary byte. If you do not want to change the current value of any of the last four arguements (buf, blen, ax1, or ac2) use the value -1. Thus, to read another line into the current buffer, you would use:
ciov(1,5,-1,-1,-1,-1);

Note that most of the i/o functions are implemented using calls to ciov(). The two exceptions, cgetc() and cputc(), are coded in assembly to speed them up slightly. If the CIO returns a result less than 128, ciov() returns it as-is, but if the CIO result code is greater than or equal to 128 (which means that an error has occured), ciov() returns the negative of that code. This is in keeping with "standard usage" in C, which has error codes less than zero.

```
normalize(fname,fext)
char *fname,*fext;
```

normalize() is a handy utility function used to convert free-form file names into CIO and FMS standard file names. First the file name is converted into upper case, then, if there is no device prefix, D: is added to the front of the name. If there is no extension, a period and the extension in the string fext is appended onto the file name. A typical use, "char fname[2Ø]; gets(fname); normalize(fname,"BAS") would ensure that the file name in fname is acceptable to the CIO system. If the user had input "prog", after normalize(fname,"BAS") the string fname would contain "D:PROG.BAS".

```
toupper(c)
char c;
```

If c is lower case, toupper() returns the upper case equivilant, or else toupper() returns c.

```
tolower(c)
char c;
```

If c is upper case, returns the lower case equivalent, or else returns c.

```
strcpy(a,b)
char *a,*b;
```

strcpy() copies a string from character array b to character array a. strcpy() returns the length of the string it copied, not counting the trailing zero byte.

```
move(a,b,len)
char *a,*b;
int len;
```

move() moves len bytes from a to b, starting with the byte at a[Ø] and finishing with the byte at a[len-1]. Funny things will happen if a <= b <= a+len.

```
usr(addr,. . . .)
int addr;
```

usr() is like the BASIC USR(X) function -- the first argument is the address of the machine language subroutine and the rest of the arguments are passed on to that subroutine. The result is passed in the A (low) and X (high) registers. When the user's routine is called, the stack looks (in the order items would be PLA'd off the stack) like this:

<number of arguments (zero to 120) besides the address>
<high byte first argument>
<low byte first argument>
<high byte second argument>
<low byte second argument, etc.>
<return address (two bytes)>

Zero page variables $F6 to $FF are free for use with usr() subroutines.

```
find(addr,len,ch)
char *addr,ch;
int len;
```

find() searches memory from addr to addr+len-1 for the first occurence of ch. If it doesn't find ch, it returns -1, otherwise it returns the number of characters past addr that it found ch (range of 0 to len-1).

```
peek(i)
char *i;
```

peek() returns the bvte at memory address i.

```
poke(i,d)
char *i,d;
```

poke() pokes byte d into address i, then returns the OLD byte at i.

```
dpeek(i)
char *i;
```

dpeek() returns the word at i (least signifigant byte) to i+1 (most signifigant byte).

```
dpoke(i,w)
char *i;
int w;
```

dpoke() pokes the word w into address i to i+1, then returns the old word at that address.

```
val(s)
char *s;
```

val(), like BASIC's VAL function, takes a string as input and returns its numeric value.

```
hval(s)
char *s;
```

hval() takes a string as input and returns its hexidecimal value.

Functions Defined in GRAPHICS.C

```
graphics(n)
char n;
```

graphics() will change the screen's graphics mode just like the BASIC GRAPHICS statement. returns same status as open().

```
color(c)
char c;
```

color() will set the color to plot() or drawto(), just like the BASIC COLOR statement. returns garbage.

```
drawto(x,y)
int x,y;
```

draws a line from last plotted point to (x,y), just like BASIC's DRAWTO. Returns 1 if ok, else CIO error code.

```
locate(x,y)
int x,y;
```

locates the graphics cursor at the position (x,y) and returns the value of that pixel, or the CIO error code. Exactly like BASIC's LOCATE statement.

```
plot(x,y)
int x,y;
```

plots a point at (X,Y) just like BASIC's PLOT statement. Returns 1 if OK, else the CIO error code.

```
position(x,y)
int x,y;
```

positions the graphics cursor at new (X,Y). Not actually moved until next output.

```
setcolor(reg,hue,lum)
char reg,hue,lum;
```

Sets color # reg to the color combination hue,lum. Just like the BASIC SETCOLOR statement.

```
fill(x,y,c)
int x,y;
char c;
```

Fill implements the FILL command of the S: device. It draws a line from the last point plotted to (x,y), filling the backround to the right of the line with the color provided. Somewhat useful for filling in large trapazoidal regions of the screen with color. See page 54 of the BASIC REFERENCE MANUAL for more details.

```
paddle(n)
char n;
```

paddle() returns the value of the numbered paddle, just like BASIC's PADDLE function.

```
ptrig(n)
char n;
```

ptrig() returns the value of the numbered paddle trigger, just like BASIC's PTRIG function.

```
stick(n)
char n;
```

stick() returns the value of the numbered joystick, just like BASIC's STICK function.

```
strig(n)
char n;
```

strig() returns the value of the numbered joystick's trigger button, just like BASIC's STRIG function.

```
vstick(n)
char n;
```

vstick() returns the vertical component of joystick n. If the joystick is pointed forward (up) vstick() returns 1. If back (down) vstick() returns -1. If centered (vertically) vstick() returns Ø.

```
hstick(n)
char n;
```

hstick() returns the horizontal component of joystick n. If the joystick is pointed left hstick() returns -1. If pointed right, hstick() returns 1. If centered (horizontally) hstick() returns Ø.

Functions Defined in PMG.C

```
pmcinit()
```

pmcinit() initializes player/missile and character set graphics. pmcinit() must be called exactly once, and should be used BEFORE any calls to graphics().

```
pmcflush()
```

pmcflush() flushes player/missile and character set graphics buffers out of RAM, returning the 4K of RAM that they use. pmcflush() should be called exactly once, just before returning to DOS.

```
pmgraphics(i)
int i;
```

pmgraphics() should be called AFTER each call to graphics() to set up the resolution of the player/missile graphics. pmgraphics(1) produces single line resolution, pmgraphics(2) produces double line resolution, and pmgraphics(Ø) inhibits player missile graphics all together.

```
hitclear()
```

hitclear() clears the collision registers.

```
hitp2pf(from,to)
char from,to;
```

hitp2pf() returns one if player # "from" hit playfield color "to", otherwise it returns zero.

```
hitp2pl(from,to)
char from,to;
```

hitp2pl() returns one if player "from" hit player # "to". If "from" is equal to "to", then one is returned.

```
pmclear(n)
char n;
```

pmclear() clears player number "n".

```
pmcolor(n,c,i)
char n,c,i;
```

pmcolor() sets the color of player/missile "n" to hue "c" and intensity "i". Similar to color().

```
pmwidth(n,w)
char n,w;
```

pmwidth() sets the width of player "n" to "w":

```
w == Ø means normal size
w == 1 means two times normal size
w == 3 means four times normal size
```

```
pladdr(n)
char n;
```

pladdr() returns the address of the buffer containing player "n".

```
plmove(n,x,y,shape)
char n,x,y,*shape;
```

plmove() moves player "n" to position "x","y" (in the current pmgraphics() mode's coordinates) and draws it's shape from the character array "shape". shape[Ø] is the size of the player's shape, and shape[1 .. size] is the byte pattern for the player itself. Be sure to put several zero bytes before and after the actual graphic so that previous images will be erased properly.

```
chget(c,s)
char c,*s;
```

chget() fills the s[Ø..7] with the character font for internal atascii character c. ATASCII blank-space's internal representation is Ø, so it's current font could be obtained by chget(Ø,s).

```
choget(c,s)
char c,*s;
```

choget() fills s[Ø..7] with the ORIGINAL character font for internal atascii character c.

```
chput(c,s)
char c,*s;
```

chput() makes s[Ø..7] the font for internal-atascii character c. To put a dot in the middle of the space, for instance, one would say
chput(Ø,"\Ø\Ø\Ø\6Ø\6Ø\Ø\Ø\Ø");

REMEMBER that one must use pmcinit() before any other function in PMG.C will work.

```
sound(voice,pitch,distortion,volume)
char voice,pitch,distortion,volume);
```

sound() makes sound effects just like BASIC's SOUND statement.

```
rnd(n)
int n;
```

rnd() returns a random number between Ø and n-1 (inclusive), so to generate a random number between 1 and 1Ø you would use the expression: 1+rnd(1Ø). If n is less than 2 then rnd() will return Ø.

In addition to the functions in AIO, there are two more library functions in PRINTF:

```
printf(s,. . . .)
char *s;
```

printf() is the standard C formatted output function. It takes a variable number of arguments. The first one is a format string containing the message to be printed, along with characters specifying where to insert the rest of the arguments.

The % character is special when it appears in the format string. The characters following the % tell how to print one of the arguments -- the first % matches the first argument after the format string, the second % matches the second argument, and so on. If you specify too few arguments (or too many %s) your output string will be garbled.

After a % you may type one of the following letters:

d -- to print a decimal number
x -- to print a hexidecimal number
c -- to print a character
s -- to print a string
or
% -- to print a %

If you want the argument to take at least a certain number of characters, type a number between the % and the format character. The value will be right justified. If you want it left justified, then insert a minus sign before the number. Here are some examples to clarify things:

```
printf("abcd"); produces
abcd
printf("=%s=","abcd"); produces
=abcd=
printf("=%5d=",99); produces
=   99=
printf("=%-5d=",99); produces
=99   =
and printf("%c %d %x",65,65,65); produces
A 65 41
```

```
fprintf(iocb,s,.....)
int iocb;
char *s;
```

fprintf() is just like printf() except that it takes an additional argument, iocb, and outputs to that iocb. printf() is esentially fprintf(Ø,...).

Adding machine language functions to Deep Blue C.

If you look at the AIO.C file you will note that the "primitive" functions, like ciov() are defined in a peculiar way, using the asm statement:

```
ciov(iocb,com,bad,blen,ax1,ax2)
int iocb,com,blen,ax1,ax2;
char *bad;
asm 12291;
```

This kind of function definition, using asm rather than $(<statements> $), creates a "hook" into machine language. When an "asm" function is called, the arguments are pushed onto the 6502 machine language stack just like the usr() function, then a jump is made to the address that follows the "asm" keyword. If you want to add "asm" function to your DBC programs, at memory location $600 (page six), you would simply write:

```
foo()
asm 0x600;
```

In addition, you've got to write the assembly language routine (using the assembler of your choice) and include the name of the object file (which must have the extension .OBJ) in your .LNK file.

Don't forget that the number of arguments you actually get may vary depending upon how many the user supplies. You should use the byte on the top of the stack to tell you how many arguments to pop off the stack before returning.

RAM usage: The Deep Blue C run-time package uses RAM from $3000 to $3FFF, and the user's program starts at $4000 and continues towards the top of memory. You can use page six and any RAM free between the top of the OS and $3000 for your own purposes. Although the compiler needs 48K, most Deep Blue C programs will run in much less space -- it is certainly possible to create useful programs that run in as little as 24K of RAM.

Modifying the Compiler

The compiler is contained in the 12 files cc0.c to cc9.c, ccv.c, and ccg.c. To modify the compiler, compile each of these modules except for ccg.c, which is an "include" file, and link them together using CC.LNK.

The run time package (the compiled-c-code interpreter) is contained in the files DBC.MAC, DBCX.MAC, and DBC2.MAC. These files can be re-assembled via the ATARI MACRO ASSEMBLER with the command line:

```
D:DBC.MAC
```

The source for the linker is in the files clink.c and clink2.c. When these files are compiled you can link them together using the CLINK.LNK file.

If you make changes to the compiler or the linker, rename the old version to OCC or OCLINK rather than deleteing it. This way you'll be able to backtrack in the event of bugs in your new code!

If you find any bugs (and especially if you have written a well tested fix for a bug) in either the compiler or the linker, please fill out the REVIEW FORM in the back of this manual and send it to APX. The compiler is known to compile itself, so it's pretty much bug-free. The author does NOT appreciate hearing about obscure bugs at three a.m., so please use the REVIEW FORM rather than the telephone!

Compiler Notes:

These notes are intended for a VERY experienced user who wishes to modify the compiler. In all probability this does NOT mean YOU. Mere mortals can safely ignore this section!

Ron Cain's original Small C compiler translated the C program into an assembly language source file. The user would then use an assembler to convert this file into executable code. There are to reasons why this could not be done for the Atari: a) The assembly language file would be about 180K bytes long, much larger than the Atari disk drives could handle, and b) The object code would be more than 30K, much larger than available RAM.

The first problem, that of gigantic intermediate files, was solved by having the compiler emit a compressed pseudo-assembly language and constructing a linker. The second problem, caused by the byte-oriented nature of the 6502, was overcome by having the compiler/linker emit pseudo-code rather than machine code. This pseudo-code is interpreted at run time by the machine code in the DBC.OBJ file, which is inserted into the COM file by the linker. (This insertion happens because of the DBC.OBJ line of the LNK file.)

If you want the COM file to reside in a different portion of memory (say, for instance, that you want your program to run in a cassette-based environment where DOS is not in RAM) then you should re-assemble DBC.MAC and CLINK.C to expect and emit the code at the new location.

Somebody should add structures to this compiler! (It won't be me!)