MEMO

To: Harry Stewart

From: Robert C. Fruth

Subject: "Deep Blue" C Compiler

Attached is a copy of my final report on the "Deep Blue" C compiler. Deep Blue C is due to be released by APX in December, and will be included in their next catalog. Please feel free to report your questions and comments to me.

If you need additional copies of the report, please see me for them. If you want a copy of the Deep Blue C compiler, then contact Gene Plagge (the product manager for APX) at 745-2124, as I cannot distribute copies of Deep Blue C.

AN EVALUATION

OF THE

"DEEP BLUE" C COMPILER

ROBERT C. FRUTH

9-82

REV. 0.1

AN EVALUATION OF THE "DEEP BLUE" C COMPILER

I. INTRODUCTION

The C programming language was originally developed at Bell Labs in conjunction with the development of the UNIX operating system, which is written in C. As a programming language, C combines the structured code and data organization of a high-level language with the machine-level interface that is possible with assembly language. Because of its flexibility, C is preferred by many systems programmers and others who must be able to directly access the computer hardware. The C Programming Language by Kernighan and Ritchie is the standard reference for the C language.

"Deep Blue C" is a proper subset of version 7 of the full C programming language. The author is John Howard Palevich, who also authored the "Chameleon CRT Terminal Emulator". Palevich wrote the "Deep Blue C" compiler by making extensive modifications to the "Small-C" compiler written by Ron Cain and Brian Smith. In his documentation, Palevich noted that although Small-C is in the public domain, Deep Blue C is copyright protected.

Writing and subsequently executing a program using Deep Blue C is not difficult. First, the programmer enters his source code into a file (default extension .C). Next, the source code is compiled into a tokenized form (default file extension .CCC), and then linked with various other files to form the executable file (extension .COM). These other files may be files containing compiled C source code, compiled library files, and/or files (default extension .OBJ) containing assembly language code that has been assembled using the macro-assembler. The file DBC.OBJ, which contains the Deep Blue C run-time module, must be one of the files that is linked.

This evaluation of Deep Blue C was carried out using versions 0.0 and 0.1 of the compiler, linker, and library files. Some of the features and bugs described in this paper may be changed in later revisions of Deep Blue C. Of course, I hope that the bugs will be changed.

II. SUPPORTED FEATURES OF STANDARD C

Deep Blue C supports most of the program structures of the standard C language. This syntax includes assignment statements, single branch (if-else) and multiple branch (switch-case) decision constructs, and do-until, while, and for loops. The "break", "continue", "default", and "return" keywords are also supported. Deep Blue C supports almost all of the operators of standard C, including standard arithmetic, relational, logical, pointer, assignment, bitwise, conditional, and shift operators. These operators can be applied to three types of data: integers, characters, and pointers. Variables in Deep Blue C may be any of these three types, and may be either internal or external (using the "extern" keyword). In addition, Deep Blue C supports one-dimensional arrays. The "#define" and "#include" compiler directives are also supported by Deep Blue C, thus allowing the user to define his own constants and to include a file (usually containing the user's standard declarations for a program that is divided into several modules) in many different programs. However, Deep Blue C does not support all of the standard syntactical constructions that are possible using the #define directive, as will be discussed later.

III. UNSUPPORTED FEATURES OF STANDARD C

A. Floating Point Numbers

Floating point numbers are not supported by Deep Blue C. Functions may not return floating point values, variables of the standard C types "float" and "double" are not supported, and floating point constants are not allowed. As a result, Deep Blue C does not support any trigonometric routines or other functions that return real values. For many applications, this absence of floating point numbers is not significant. However, for applications that require a substantial amount of mathematical processing and/or the use of real values, the lack of floating point numbers may be crucial. A user who requires floating point numbers for an application might be able to write software routines to overcome the lack of floating point numbers. The addition of a floating point package to a future revision of Deep Blue C could be very beneficial.

B. Structures and Unions

Deep Blue C does not support structures (similar to records in Pascal) or unions (comparable to variant records). The absence of structures is unfortunate, as structures allow the user to represent almost any type and amount of data without too much effort. Using structures and pointers, a user can easily build complex organizations of readily accessible data. The use of structures make programs that handle large amounts of data easy to design and code. However, the absence of structures does not prevent the using of large amounts of data, as structures are only one of several methods that can be utilized to represent and handle data. Nevertheless, structures are useful because they simplify the design and coding of complex data representations. Programs written using structures are usually easier to read and understand. In my opinion, structures are one of the best reasons for using C as a language for programming.

III. UNSUPPORTED FEATURES OF STANDARD C

B. Structures and Unions - (Cont')

The lack of unions in Deep Blue C is not nearly as critical as the lack of structures. Unions permit the user to combine two or more similar structures into one structure with one or more variant fields. The supporting of unions would be convenient for the user, at a cost of necessitating some additional code in the object files for the user's programs. However, the question of unions is meaningless, since structures are not supported.

C. Functions Returning Non-integers

In the current version of Deep Blue C, functions may only return integers. When first considered, this convention seems unworkable, but with the exception of floating point numbers, it can be worked with fairly easily. In order to return pointers and arrays, the address of the pointer or array should be returned instead of the actual pointer or array. Instead of returning characters, the numerical values of the characters should be returned. Returning structures, if they were supported, would be accomplished by returning the address of the appropriate structure rather than the structure itself. The calling routine would have to know what type of structure is at the address being returned. If floating point numbers were implemented in a future revision of Deep Blue C, then functions would have to be able to return floating point values, otherwise much of the usefulness of floating point numbers would be wasted, and trigonometric functions and other routines that return real values could not be supported.

III. UNSUPPORTED FEATURES OF STANDARD C - (Cont')

D. Multi-dimensional Arrays

The lack of multi-dimensional arrays is not as significant as it first seems. In the standard C language, a two-dimensional array is by definition a one-dimensional array composed of one-dimensional arrays. Thus, a set of one-dimensional arrays can readily be used in lieu of a single multi-dimensional array. The addition of multi-dimenstional arrays to a future revision of Deep Blue C should be given relatively low priority.

E. Unsupported Keywords

In addition to those keywords already mentioned, some others that Deep Blue C does not support deserve comment.

goto: The goto statement in C is similar to the goto statement in Pascal, both in its syntax and in the desirability of not using it. I do not consider the absence of the goto statement to be a great loss.

typedef: When used in a declaration, the typedef keyword adds a new name for an existing type without creating a new type. The primary reasons for using the typedef keyword are to enhance program portability and the writing of machine-independent code, and to help make programs more readable. In addition, when teamed with explicit type casting (described in section III, part F), this keyword provides an excellent way of improving program security, especially in multi-user or multi-process environments. Supporting the typedef keyword would enhance Deep Blue C, but the absence of it is not particularily critical, unless explicit type casting is supported.

III. UNSUPPORTED FEATURES OF STANDARD C

E. Unsupported Keywords - (Cont')

unsigned: When used in an declaration for an integer variable, the keyword unsigned makes that variable an unsigned integer quantity. The sign bit in this variable is then treated the same as any other bit. Positive integers of a larger magnitude than is possible when using signed variables could be employed if this keyword was supported.
register: The register keyword is also used in variable declarations. It is employed to advise the compiler that this particular variable will be used often and thus should be kept in a register as much as possible. In terms of the Atari 800 and the 6502 microprocessor, the absence of the register keyword probably saves more memory space than using it would save, due to the limited number of registers available.
static: Static variables may be separated into two classes. The first class are variables defined in the main program outside of any subroutine (i.e. function). These global variable are known to the main program and all of its subroutines, and are "static" in that they are active during the entire period of program execution. They do not require the use of the static keyword in their declarations. The other class of static variables do have the static keyword used in their declarations, and remain in existence rather than "coming and going" each time the block of code where they are declared is executed. When used in a section of code,, a static variable of this class provides permanent storage within that section of code, and is not known to any other section of code. Because Deep Blue C does not support the static keyword, only the first class of static variables may be employed with the present version. The addition of the static keyword to a future revision of Deep Blue C would allow the useful second class of static variables to be utilized.

III. UNSUPPORTED FEATURES OF STANDARD C - (Cont')

F. Unsupported Operators

As described in section II, Deep Blue C supports almost all of the operators of standard C. Two of the four unsupported operators are used with structures. These are the -> (right arrow) and the . (dot) operators. The absence of these operators is insignificant, since structures are also unsupported. If structures were implemented in a later revision of Deep Blue C, then these two operators should be implemented as well. The third unsupported operator is the "sizeof" operator. In standard C, the expression "sizeof (object)" returns the size of "object" in bytes, where "object" can be either a simple variable, an array, or a structure. Supporting the sizeof operator would be convenient for the user, but the absence of this operator can be eased through user-written software. The fourth and last unsupported operator is explicit type casting, which makes it possible to force an expression to be of a specific type. When used in tandem with the typedef keyword, this operation is useful both as a way of disguising data structures and as a means of increasing program security, especially in multi-user and multi-process environments. Explicit type casting can also be used for these two purposes without the use of the typedef keyword, and probably should be given medium priority in a revision of Deep Blue C.

G. Other Unsupported Features

Several other features of standard C are unsupported by Deep Blue C. One of these is the standard C library function "exit". When called and executed, exit terminates program execution. Exit would be a useful feature to have and probably should be supported.

III. UNSUPPORTED FEATURES OF STANDARD C

G. Other Unsupported Features - (Cont')

As was stated earlier, the "#define" compiler directive is not fully supported. The macro substitution governed by the use of this directive is supported as defined in standard C, but the passing of arguments to the resulting macro is not supported. Defining macros using this directive allows the user replace a section of C source code with one word. This usage is very beneficial in cases where the section in question is used several times and putting it into a subroutine is not feasible, and when space for the source file is limited. The means to pass arguments to macros increases the usefulness of the macros, and not having this ability is a hindrance. Fully supporting the #define compiler directive in a future revision of Deep Blue C should be accomplished if possible.

In addition to the #define and #include compiler directives already discussed, standard C defines a set of directives which control conditional compilation. These directives include "#if", "#ifdef", "#ifndef", "#else", and "#endif", and are not currently supported by Deep Blue C. By making use of these directives, the programmer may compile selected sections of his source code. This feature is particularily useful for setting apart and selectively compiling code that is to be used as a debugging aid, and for writing code that is to be run on different machines or under different operating systems. The employing of conditional compilation as an aid for debugging is probably the more valuable of these two uses. Conditional compilation should be added to a future revision of Deep Blue C if the resources involved would allow it to be implemented.

III. UNSUPPORTED FEATURES OF STANDARD C

G. Other Unsupported Features - (Cont')

Another unsupported feature is the capacity to use command-line arguments in order to pass parameters to a program when it first begins executing. For a user oriented machine like the 800, the absence of command-line arguments is insignificant, since programs can easily prompt the user for input.

The last unsupported feature of standard C which I will discuss is the explicit initialization of variables when they are declared. This feature enhances program readibility and is convenient for the programmer, and so would be nice to have supported. However, the initialization of variables when they are declared probably does not decrease the amount of generated object code, and thus probably should not be given high priority in future revisions of Deep Blue C.

IV. ATARI SPECIFIC FEATURES AND FUNCTIONS

In several ways, the features and functions of Deep Blue C that are specific to the hardware of the 800 are similar to the corresponding features and functions of Atari Basic. For example, both Basic and Deep Blue C have a machine-level interface provided by the execution of a function called "usr". The execution of these two "usr" functions is approximately equivalent. Each accepts a mandatory argument, which gives the address of a machine language subroutine, followed by one or more optional arguments, which are parameters to be passed on to that subroutine. Both functions employ a similar stack structure, and the machine language subroutines in both have to pull the input arguments off the stack (using PLA's) before returning to the calling program, so that the top two bytes of the stack contain the return address. However, Deep Blue C has an additional interface with machine language that Basic does not have. By defining a function written in Deep Blue C using the statement "asm number;" instead of the standard function definition, "$(<statements> $)", the machine language routine located at address "number" is executed. This routine is assembled at address "number" using the assembler of the user's choice, and the resultant object file (mandatory extension .OBJ) must be linked with the other required files by the linker. (In the current version, this means that the name of the object file, with the extension .OBJ, must be included in the appropriate link file.) The parameters that are passed to the Deep Blue C function are pushed onto the stack when the asm statement is executed, and thus are passed on to the machine language routine. By using this construction, a user may easily include machine language routines in a program written in Deep Blue C. In particular, machine language routines that are executed more than once should especially be included using this mehtod, as these routines are readily executed repeatedly by calling the Deep Blue C function that contains the asm statement.

IV. ATARI SPECIFIC FEATURES AND FUNCTIONS - (Cont')

Deep Blue C and Atari Basic are also similar in the way input, output, and graphics are handled. In fact, the I/O and graphics routines for Deep Blue C have been modeled after those of Atari Basic, and access the hardware of the 800 on about the same level as the Basic routines. John Palevich's movivation for this similarity is straightforward. The C language differs from most other programming languages in that it has no standard built-in I/O functions. As a result, every version of C has its own specific I/O functions. Most, if not all, Atari 800 user are familiar with the I/O and graphics functions of Atari Basic, and so Palevich used these familiar and easily understood routines as a base for developing the I/O and graphics functions for Deep Blue C. In some respects, this decision is unfortunate, as the Basic graphics functions interact with the hardware on a fairly abstract and distant level. However, programmers employing Deep Blue C can define their own I/O and graphics functions using the asm keyword described earlier. Although there are some differences between the I/O and graphics routines of Atari Basic and those of Deep Blue C, a programmer who already knows the Basic functions should have no problems with the corresponding routines of Deep Blue C.

The graphics and I/O routines for Deep Blue C are contained in four library files: AIO, Printf, Graphics, and PMG. These files are presented in two forms of the Deep Blue C distribution disk, both as Deep Blue C source code (extension .C), and as compiled source code (extension .CCC). In order to use the routines contained in these library files in his program, the user must link the compiled form of the desired library file with the other required files, also in compiled firm, to form the executable file (extension .COM). Thus, the user need only link those library files which contain routines used in his source program. Unfortunately, when a library file is included in a link, the entire file is used, regardless of how many of the routines contained in that library file are used in the

IV. ATARI SPECIFIC FEATURES AND FUNCTIONS - (Cont')

program. As a result, the library file ultimately occupies the same amount of space in the executable file irrespective of whether the user's program uses zero, one, two, or all of the routines contained in that file. For example, the minimal program shown below occupied 35 sectors on the disk when linked with the library file AIO.CCC, and only 17 sectors when AIO.CCC was not included in the link. Both the large and the small versions of the program executed correctly.

```
main ()
$(
$)
```

Since AIO.CCC occupies 19 disk sectors, it is obvious that all of AIO.CCC was included in the executable (.COM) file, even though the minimal program doesn't use any of the routines contained in AIO.CCC. (The loss of one sector (17 + 19 = 36 = 35 + 1) can be attributed to the two files actually occupying a number of disk sectors and a fractional part of one additional sector.) If the linker were modified so that it searched the named library files and only extracted the routines used in the user's source code, then the user's programs in their executable form would require less disk space. This modification would result in a substantial reduction in the number of disk sectors occupied by the user's executable files. In the example above, the executable file for the program would occupy only 17 sectors regardless of whether or not the program was linked with the library file AIO.CCC, or with any other library file for that matter. Implementing searchable library files would not be too difficult, and would primarily involve modifying the linker so that it treated each library file as a collection of separately extractable functions, rather than as a whole file. The cost of implementing this feature would be very small when compared with the amount of disk space its implementation would save.

IV. ATARI SPECIFIC FEATURES AND FUNCTIONS - (Cont')

The Deep Blue C graphics and I/O routines are distributed among the four library files according to the similar purposes and uses of the routines. The first library file, AIO, contains functions that allow the user to open, close, read from, and write to files. These functions are all very similar to the analogous Basic functions. In addition, the contents of AIO include routines that perform string functions (copy, length, finding a substring within a string, and returning the decimal or hexadecimal value of a string), filename normalizing, uppercase to lowercase and lowercase to uppercase conversions, and memory accessing. The memory access routines include functions for peeking and poking both bytes and words, clearing a block of memory, and moving a memory block from one location to another. AIO also contains the "usr" function described earlier.

Printf, the second library file, contains the "standard" C formatted output function, "printf", and a related output routine. Printf is not a part of the standard C language, which has no defined input or output, but rather is a part of the standard library of routines that are accessible from C. Using printf, the user can easily specify the format for the output of program data. The standard library of routines also includes the formatted input function "scanf", which Deep Blue C does not support at this time. Both scanf and printf are discussed in detail in The C Programming Language, by Kernighan and Ritchie.

The third library file, Graphics, contains functions that control the graphics and sound hardware of the Atari 800. These routines are modeled after the analogous functions of Atari Basic, and include "graphics", "color", "setcolor", "plot", "drawto", "position", "sound", "locate", and "fill" (similar to Basic XIO). Each of these functions operates in almost the same way as its Basic counterpart. In addition, this library file

IV. ATARI SPECIFIC FEATURES AND FUNCTIONS - (Cont')

contains functions for reading the values returned by game controllers such as joysticks and paddles. These functions are "paddle", "ptrig", "stick", "strig", "vstick", and "hstick". Vstick and hstick return the vertical and horizontal components of the joysticks. All of the other functions operate like the equivalent Basic functions. Graphics also includes the function "rnd", which accepts an argument "n", and returns a random integer between 0 and n-1.

Deep Blue C also supports a set of functions for manipulating player/missile and character set graphics. The library file PMG contains these functions, which are described more completely in the Deep Blue C documentation. Briefly, the character set routines permit the user to fetch both the current and the original fonts for ATASCII characters, and to store new fonts for characters. The functions for the player/missile graphics allow the user to initialize the player graphics, set the color, width, and resolution of the players, fetch the address of the players' memory area, clear this memory area, move the players within player memory (and thus on the screen), and flush the player memory when the players are no longer needed. In addition, PMG contains a pair of functions which can be used to detect player to player and player to playfield collisions. All of these functions help make player/missile graphics seem less mysterious to the user, at the expense of "direct" contact with the controlling hardware registers and player memory itself. Unfortunately, PMG does not seem to include any routines for initializing and using the missiles (as opposed to the players). Unless I am in error, the routines currently existing in PMG only access the registers and memory locations that control the players and not those that control the missiles. In addition, because these functions hide the specific registers from the user, the only way to enable the fifth player seems to be by directly accessing memory. Thus, the user who employs these functions to use players in his program has to

IV. ATARI SPECIFIC FEATURES AND FUNCTIONS - (Cont')

include separate code for using the missiles. This omission of functions that handle the missile graphics is unfortunate, for it defeats the entire purpose of having and using the player/missile graphics functions. The intent behind the player/missile graphics functions is good, but unless some modifications are made so that the functions can control the missiles, they will be of limited use. If this analysis of the player/missle functions is incorrect, then the Deep Blue C documentation must be changed to correctly describe the operation and effect of these functions.

In addition to the four library files, the Deep Blue C disk includes an extremely important file, DBC.OBJ. This file contains the Deep Blue C runtime package, comprising the runtime routines and the "C-code" interpreter that every program needs in order to execute properly. The runtime routines include the 6502 code for the functions in the four library files which use the "asm" keyword, and subroutines written in 6502 that perform the multiply and divide operations. The C-code interpreter is needed because the linker produces pseudo-code rather than 6502 code. If the linker produced 6502 code, then the resulting object code would exceed the amount of available memory. Hence, the decision was made to have the linker produce pseudo-code, which requires the presence of an interpreter when it is executed. Thus, the file DBC.OBJ must be included in every link of a Deep Blue C program, or else the program in question will not execute.

V. ADDITIONAL DESIRABLE FEATURES AND SOME OBSERVATIONS

In addition to the unsupported attributes of standard C that I have already identified as beneficial, there are several other desirable features that are not implemented. One of these is some method by which a program may transfer control from itself to another program. Such a "chaining" facility is currently implemented in both Atari Basic and Atari Pascal. To transfer control from one Basic program to another, a programmer simply inserts a "run" statement in the first program. Using the Pascal chaining facility is not difficult either, and basically consists of assigning the name of the appropriate external file to an internally declared file variable, resetting this file, and then calling the chain function to transfer control over to the program within this second file. Atari Pascal also has two methods for the two chained programs to communicate with each other: absolute variables and shared global variables. To perform the operation I have just described, Atari Pascal utilizes two keywords, "global", and "absolute", and three functions, "assign", "reset", and "chain". Global and absolute are both qualifiers used in variable declarations. Global variables are declared using the global keyword, and require the use of a linker option switch to insure that the variables are each placed in the same memory location in all of the programs that are to communicate with each other. The memory space for absolute variables is allocated at compile time, with the address of the variable being a part of the syntax of its declaration. Assign is a function that takes two arguments, one an internal file variable and the other the name of an external disk file, and associates the two arguments with each other. Internal file variables are a part of the syntax of standard Pascal, and are declared as identifiers of type "file". The reset function is a standard Pascal procedure which opens the file named in its parameter (a file variable) for reading by resetting the file pointer to the beginning of the file. Finally, the chain function performs the actual transfer of control from one program to the next. The chaining facility is useful

V. ADDITIONAL DESIRABLE FEATURES AND SOME OBSERVATIONS - (Cont')

both for executing large programs which require more memory than is available, and for segmenting programs for purposes of modularity and maintenance.

In order to implement a chaining facility in Deep Blue C, several features are needed. First, some method for two or more programs to communicate with each other is required. Either of the two methods used by Atari Pascal would be suitable, but each would necessitate major modifications to both the linker and the compiler. If shared global variables were used, then the keyword "global" would have to be supported by the compiler, and the linker would have to place the shared global variables in the same memory locations in each of the communicating programs. Using absolute variables would require that the compiler support a special syntax for the declaration of absolute variables, including the keyword "absolute", and that the compiler allocate memory space for absolute variables during compile time. The linker would have to recognize this memory allocation. Standard C does not support file variables, but these are not necessary, as the names of files can be contained in arrays of characters, preferably declared as pointers to characters (they are treated equivalently, but pointers to characters are a little faster). An assign function would not be required if the names of files were contained in pointers to characters. The file containing the program to which control is to be transferred could be opened and reset for reading by calling either the "copen" or "open" functions, both of which are currently supported by Deep Blue C. Thus, an explicit reset function would not be needed. Finally, a function to perform the actual transfer of control would have to be implemented. This function could be very similar to the chain function of Atari Pascal, and would accept the name of a file as a parameter, transfer control to

V. ADDITIONAL DESIRABLE FEATURES AND SOME OBSERVATIONS - (Cont')

the program contained in that file, and then execute that program. Thus, the implementation of program chaining in Deep Blue C would require complex modifications to be made to both the compiler and the linker. However, the benefits of having a chaining feature are greater than the cost of implementing it. Using the current version of Deep Blue C, programs may not be larger than the amount of available memory, which in the 800 is not a large quantity. After the implementation of chaining, however, program size would be limited only by the amount of available disk space. As a result, many large programs and lengthy applications could be implemented using Deep Blue C. The use of chaining would probably permit the compiler itself to be split into two or more separate files, resulting in more space for new features to be added to Deep Blue C. The implementation of a chaining facility in Deep Blue C would be difficult, but the benefits of having it would far outweigh the costs of its implementation.

Another desirable feature is included in many other implementations of the C language, especially those implementations of C running on the UNIX operating system. This feature is a program called "lint" which performs more rigorous type checking than the C compiler does. Lint got its name from its ability to pick "bits of fluff" from submitted C programs. In addition to performing strong type checking, lint checks for such programming errors as variables that are either unused or uninitialized, arguments that are used inconsistently, and the using of a value returned by a function that does not actually return a value (but rather returns garbage). Given that most C compilers do not perform this kind of rigorous error checking, if is easily seen that having a program like lint available is very convenient for the programmer. The addition of a lint-like utility to the Deep Blue C compiler would greatly aid program development and result in a more complete programming system.

V. ADDITIONAL DESIRABLE FEATURES AND SOME OBSERVATIONS - (Cont')

Although the compiler has several helpful user-oriented features, an improvement in user interface is desirable. In the current version, the compiler displays the name of the function currently being parsed, prints rather cryptic error messages in response to the syntax errors that it finds, and rings a bell to alert the user that either the compilation is complete, or that an error has been found. The bell is only rung for the first error that the compiler finds. Unfortunately, the easiest way to determine that a compilation is proceeding smoothly is by listening to the numberous disk timeouts that occur during compilation. In order to keep the user aware of the progres of compilation, the compiler should display more than just the name of the function currently being parsed. The names of the functions that have been parsed previously should also remain on the screen. In addition, the compiler could print a special character on the screen for each line of source code as it is parsed. Atari Pascal displays a dot for each line of source code, and every 32 lines prints a count of the number of lines that have been scanned so far. As a result, the user is able to see that the compiler is continuing to proceed through the source code. Keeping the user informed of the progress of compilation is especially crucial with Deep Blue C, as the compiler is quite slow when compiling a program of any significant length. Reducing the time required to compile large programs would be major improvement, and should be given high priority in future revisions.

The Compiler's error messages are obviously the error messages used in standard C. Even so, they are fairly cryptic, and could be improved. For example, for an undeclared variable, the message "must be lvalue" is displayed. This message does not tell the user that the variable is undeclared, but only that the identifier in question has been misused.

V. ADDITIONAL DESIRABLE FEATURES AND SOME OBSERVATIONS - (Cont')

Several additional features would make the compiler more convenient for the user. First, it would be very convenient if a user could go directly from the compiler to the linker without first having to go back to DOS. Currently, when a compilation is completed, the user is given two choices: either entering the name of another file to compile, or returning to DOS. In order to use the linker, the user must return to DOS, and then perform a time consuming binary load of the file containing the linker. Unfortunately, going directly from the compiler to the linker probably will not be realized until after a chaining facility has been implemented. Second, it would be convenient for the user if the compiler would let him either continue or abort a compilation after an error has been found. Often, one syntax error in the source code will cause the compiler to print several error messages. Giving the user the option of either continuing or aborting could save him a considerable amount of time. The capability to abort a compilation could be easily implemented using the exit function, which is currently unsupported. The goto statement could also be employed for this purpose, but I do not recommend using it. Finally, it would greatly speed up program debugging, especially the debugging of large programs, if the user had the option of listing his source code and compiler's error messages to the printer. In order to list this information, the compiler would first have to ask the user if he wanted his source code to be printed, then confirm that the printer (preferably an 80 column printer) was active, and finally send the source code and error messages to the printer during compilation. Adding this capability to the compiler would not be difficult, and would greatly aid the user.

V. ADDITIONAL DESIRABLE FEATURES AND SOME OBSERVATIONS - (Cont')

As a last comment about the compiler itself, I would like to note that the compiler (executable file CC.COM) is a very large program and that it takes quite a while to load. If a chaining facility similar to the one I have described were implemented, then the compiler could be divided into two or more files, and the user would only need to load the first of these files. This would speed up the process of program development, especially when the time spent debugging is considered.

In general, I found that the Deep Blue C linker has better user interface than the compiler. The linker displays the names of all the files that are being linked together, printing each file name when that file is used. When the link is complete, the linker alerts the user by sounding a buzzer. If the link was successful, then the message "no errors" is printed; otherwise, a message explaining the unsuccessful link is printed. Usually, an unsuccessful link is caused by the existence of one or more undeclared variables or unknown functions resident in one or more of the linked files. The linker displayed enough information to allow me to follow the progress of the link. In addition, the linker has a convenient duplicate command that lets the user make copies of small files without going back to DOS.

Like the compiler, the linker also could have some additional features added to it to help make it more convenient for the user. For example, it would save time if the user could execute compiled and linked programs without first having to return from the linker to DOS. Also, it would be convenient if the linker would print the contents of the user-specified link file (extension .LNK) on the screen before beginning the actual linking process and then ask the user if he wished to continue or abort the link. The user could thus examine the link file and perhaps determine that not all of the names of required files were included in the link

V. ADDITIONAL DESIRABLE FEATURES AND SOME OBSERVATIONS - (Cont')

file. If this were the case, the user could choose to abort the link. The ability to abort links could be added easily if the exit function were supported.

The mandatory use of link files for linking programs is a restrictive requirement. Why not have the linker prompt the user for a file name or file names and allow the user to respond by either entering the name of a link file or by entering the names of the files to be linked together? The user would also enter some sort of toggle (similar to the linker switches in Atari Pascal), so that the linker could determine whether the name of a link file or the names of files to be linked together had been entered. Giving the user this choice would permit him greater flexibility in linking his programs. If he wished, the user could use the editor to create a link file and then only have to enter one filename in response to the linker's prompt. Alternately, in response to the linker's prompt, the user could choose to enter the names of all the files that he wished to have linked. The first method saves time, whereas the second method saves a small amount of disk space (approximately one sector per link file) and allows the user to easily try different links. Although the first method would probably be used most of the time, the user should be able to employ either method.

Having been accustomed to the disk swapping that is necessary with Atari Pascal, I was pleasantly surprised when I found that no such disk changing is necessary when using Deep Blue C with two disk drives. I received an additional surprise when I discovered that it is possible to compile, link, and run Deep Blue C programs using only one disk drive. Of course, using two drives is much more convenient, but it is possible to use Deep Blue C with a one drive system. In order to do so without a considerable amount of disk swapping, it is necessary to have all of the following

V. ADDITIONAL DESIRABLE FEATURES AND SOME OBSERVATIONS - (Cont')

files present on the same disk: DOS.SYS, DUP.SYS, CC.COM, CLINK.COM, MEDIT, DBC.OBJ, AIO.CCC, GRAPHICS.CCC, PMG.CCC, and PRINTF.CCC. These files contain the disk operating system, compiler, linker, editor, linkable run-time interpreter, and the four library files. Unfortunately, these ten files occupy 494 sectors on the disk, leaving slightly more than 200 sectors for the user to quickly fill with his source files (.C), link files (.LNK), token files (.CCC), and executable files (.COM). A user employing a one drive system will have to spend a considerable amount of time copying files from one disk to another. In addition, the size of the user's programs will be restricted, and the process of debugging programs will be even more difficult than it already is. Thus, although it is possible to use Deep Blue C with a one drive system, I do not recommend doing so. Using a two drive system is easier, more convenient, and saves much programmer time and energy.

VI. KNOWN BUGS AND ERRORS

In the course of examining Version 1.1 of the Deep Blue C compiler, I have discovered several bugs. The most serious bug is resident in the file handler of both the compiler and linker. Each of these programs correctly produces the appropriate output file if there are no errors or other problems. However, if some trouble is encountered when an attempt is made to read the appropriate input file, both the compiler and the linker open the correct output file, but then fail to close and delete it when the problems are encountered with the input file. For example, during an attempted link, the linker successfully opened the appropriate .LNK file, but was unable to open the .CCC file that was named within the .LNK file. The link was aborted, and the user was prompted for a new task. Unfortunately, the linker had already opened the appropriate .COM file, and did not close it. This file was not listed in the DOS disk directory, but did occupy an entry in the complete disk directory which was examined with the "fixdmp" utility. In addition, using this utility to perform a sector trace on the directory entry showed that it had several disk sectors linked to it. Thus, not only was the supposedly deleted file taking up a directory entry, it was also monopolizing an unknown number of sectors on the disk. Eventually, after several aborted links and compiles, the disk became filled with supposedly nonexistent files, and a disk utility such as fixdmp had to be utilized to remove this garbage. For a user who does not have fixdmp available, this bug would prevent effective use of the compiler.

The three other bugs that I came across are not nearly as serious, but they all should be corrected before the compiler is released, as they prevent the compiler from fully using the hardware of the 800. First, the sound routine contained in the library file "Graphics" does not properly access voices 2 and 3, so that the user is limited to using only two of

VI. KNOWN BUGS AND ERRORS - (Cont')

the four voices. Second, the routine called "pmgraphics", which is included in the library file "PMG", does not work if the parameter value is 2. This bug limits the uer of the pre-written player/missile graphics routines to using single-line resolution player/missile graphics only. In the present version, this bug can be neutralized by poking the required values directly into the appropriate memory locations. The last bug is resident in the syntax analysis section of the compiler, which will not accept subroutines with the following general structure:

```
$(
  if (x<3) x++;  /*x is an integer variable declared*/
                 /*in the calling program.*/
$)
```

This bug is not very serious, but it might be a indicator of additional errors in the compiler's syntax analysis routines.

The documentation supplied with Deep Blue C has at least one error within it. For the "plmove" routine, the character array "shape" should contain octal numbers rather than decimal numbers as might be supposed, since the documentation does not say anything about the type of integers required. In addition, some of the documentation needs to be rewritten, with a greater amount of attention paid to details.

There are probably additional errors and bugs that I have not yet found. I will report on any new ones that I find.

VII. TIME AND MEMORY CONSIDERATIONS

Deep Blue C compares favorably with both Atari Basic and Atari Pascal when time and memory requirements are considered. I made my comparision using a simple player/missile graphics demonstration program translated into each language. Rather than using quantitative methods for my analysis, I just performed visual comparisions. The player/missile program was quite slow when written in Basic, but was significantly faster when translated into both Pascal and C, with the C version seeming to be slightly faster than the Pascal version. The Basic program occupied the smallest amount of space on the floppy disk. Both the Pascal and the C programs were significant larger (approximately 30 additional disk sectors), with the Pascal file requiring 9 fewer disk sectors than the C file. However, the C file contained both the compiled and linked C source code, as well as the Deep Blue C interpreter. Both the Basic and Pascal programs required the presence of interpreters, which were loaded into memory separately and were not a part of the program files. Because of the size of these interpreters (8K for Basic, 10K for Pascal), of the three high level languages, Deep Blue C required the smallest total amount of memory to execute the simple player/missile graphics program. In summary, these results show that Deep Blue C is faster than Basic, and at least as fast as Pascal (in this application). Deep Blue C also requires a smaller amount of memory (due to the relative sizes of the interpreters), at a cost of not supporting the full standard C language, and not including nearly as many features as are included in Atari Pascal.

I did not perform any direct comparisions between Deep Blue C and 6502 assembly language, but there is little doubt that using assembly language for development results in faster, more compact programs. However, using a high level language such as Deep Blue C for development has several beneficial effects. These include a shorter development cycle, portability from one machine to another that is easier to accomplish than it is with assembly language programs (when using two different processors), simpler program maintenance, and more convenient debugging.

VIII. CONCLUSION

Deep Blue C should be a solid addition to the product line of the Atari Program Exchange (APX). Although it only supports a proper subset of the full C programming language, Deep Blue C supports that subset extrememly well. Atari users who are familiar with Basic should be able to learn Deep Blue C with a minimum of confusion if they use a reference such as The C Programming Language by Kernighan and Ritchie. The similarity between the input/output and graphics functions of Atari Basic and those of Deep Blue C should be an additional aid to learning.

Several elements of the user interface of Deep Blue C need to be improved. The compiler needs to display additional information so that the user can follow the progress of compilation more readily. This aspect of the compiler's user interface is especially crucial because of the slow speed of the current version of the Deep Blue C compiler. At present, the easiest way to follow the progress of compilation is to listen to the numerous disk drive timeouts. In future revisions, the speed of compilation should be improved, if possible. The linker currently displays sufficient information for the process of linking to be followed, so that improving the user interface of the linker is not as imperative. However, the linker should be modified so that the use of link files (extension .LNK) is not mandatory. In addition, the documentation for Deep Blue C probably should be re-organized and an index or table of contents added. Several sections could also be more detailed in their descriptions and explanations. Of course, the bugs that I have described and any others that are present should be fixed.

The subset of the C language that Deep Blue C supports should satisfy those users who learn C as a result of Deep Blue C being offered as an APX product. However, the limitations of the subset will frustrate some users, especially those programmers who already know C. In particular,

VIII. CONCLUSION - (Cont')

the lack of such features as floating point numbers and structures will limit some users' creative energies. Several currently unsupported features should definitely be implemented in future revisions of Deep Blue C. In addition to floating point numbers and structures, these include the two operators that operate specifically on structures (-> and dot), searchable library files, and allowing functions to return non-integer values. Some users might also find an implemented chaining facility useful, especially if one of the two methods I have described to pass variables from program to program was also supported. In spite of the absence of these particular features and the other drawbacks I've discussed, Deep Blue C is a well-thought out product that will probably perform better than expected in the marketplace.

As I have discussed, Deep Blue C has several advantages over the other high-level languages that are currently available for the Atari 800. These advantages are primarily in the area of time and memory requirements. Deep Blue C is comparable with Atari Pascal when speed of execution is considered, and is significantly faster than Atari Basic. The interpreter for Deep Blue C is much smaller than both the Pascal and Basic interpreters, and thus programs written in Deep Blue C generally require less memory. In addition, Deep Blue C has two methods for including assembly language routines in a program, and is a more structured language than Basic. However, Deep Blue C supports only a subset of the full C programming language, and the limitations if the subset will prohibit the use of Deep Blue C for some applications. In particular, applications which require the use of floating point numbers, easily structured data (using structures), chaining from one program to another, or any of the other unsupported features which I have described cannot be implemented using Deep Blue C. However, Deep Blue C is a good language to use for applications which do not require the compact code and fast execution of assembly language and thus can be implemented in a high level language, and only require features that are currently supported.

VIII. CONCLUSION - (Cont')

In order for Deep Blue C to be used as an internal development language, several improvements should be made. First, unless the compiler can be modified to perform significantly faster compilations, Deep Blue C should be present on the Data General, with the capability for full uploading from and downloading to the 800. A utility, such as the "lint" utility which has been discussed, should also be available in order to ease the process of debugging Deep Blue C programs. Second, searchable library files must be implemented, so that only those routines that are used in a program will be extracted for the libraries. The implementation of this feature will result in a decrease in the memory required to execute compiled and linked Deep Blue C programs. Finally, Deep Blue C should be revised so that it supports some of the more desirable unsupported features. These include a floating point package, complete with a full library of trigonometric and related routines, and the support of floating point variables and constants and user-written functions that return floating point values. Other practical features include a chaining facility such as the one discussed, global or absolute variables so that two programs can communicate with each other, and the supporting of structures and the two operators that specifically operate on them. Additional features that are not supported at present could be implemented as the need or desire of them arises. In spite of the limitations caused by the absence of some very important features, many applications could be written in Deep Blue C, and thus Deep Blue C should be given careful consideration as a language to be used for development.

IX. ADDITIONAL INFORMATION

As I mentioned in the introduction, the standard reference for the C language is The C Programming Language, by Brain W. Kernghan and Dennis M. Ritchie (published by Prentice-Hall, copyright 1978 by Bell Labs). There are also a number of other books available. In addition, the documentation supplied with the Deep Blue C compiler is a good source of information, and the source code for Deep Blue C is available and may be consulted in desparate cases.