FIRST AND FINEST

C/65

C/65
C/65
C/65
C/65

Systems Software for
Apple and Atari Computers

Optimized Systems Software, Inc.

a reference manual for

C / 6 5

a small C language compiler for use with
Atari 400, Atari 800, and Apple II Computers

PREFACE

We realize that C/65 is not the most sophisticated, most complete, language on the market today, but we believe that the inherent power and flexibility that it exhibits within its compact size are a good match for the size and features of the machines it is intended for.

C/65 was authored by Sam Dillon and John Lowry, under the company name of LightSpeed Software, based on the Small C Compiler published in Dr. Dobb's Journal. C/65 is a hand-coded translation from C code to 6502 assembly language and is, as a result, a fast and easy to use compiler.

TRADEMARKS

The following trademarked names are used in various places within this manual, and credit is hereby given:

OS/A+, BASIC A+, MAC/65, and C/65 are trademarks of Optimized Systems Software, Inc.

Apple, Apple II, and Apple Computer(s) are trademarks of Apple Computer, Inc., Cupertino, CA

Atari, Atari 400, Atari 800, Atari Home Computers, and Atari 850 Interface Module are trademarks of Atari, Inc., Sunnyvale, CA.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

C/65 is a subset of the C programming language as defined by Kernighan and Ritchie in the book, "The C Programming Language", published by Prentice-Hall.

With a few clearly noted exceptions, programs written in C/65 are compilable without modification under standard C.

The C/65 package comes with a runtime library, which includes standard-looking character input and output functions, all of which are described later in this document. The output of the C/65 compiler is MAC/65 assembly language, which must be run through the MAC/65 assembler to produce a runnable object module. It is possible for this to be done automatically. Since the output is assembly language, it is easy to write your own assembly languauge routines that are compatible with the code generated by the compiler.

SECTION 1.1 WHAT'S IN IT

Very briefly, C/65 supports

- the basic data types CHAR and INT
- pointers to the basic types (*)
- one dimensional arrays of the basic types ([])
- the basic arithmetic, logical, and bit operations familiar to C programmers
- simple source level character substitution (#DEFINE)
- file inclusion of C source code
- file inclusion of assembly language source code (not compatible with standard C)
- functions with parameters and local variables
- an if statement, with an optional else clause
- a while statement

- break and continue statements

- a return statement, with an optional return value

- compound statements grouped by braces (although '$(' and '$)' must be used instead of '{' and '}').

- separate compilation

- limited external declarations

SECTION 1.2 WHAT'S NOT IN IT

For experienced C programmers, use of the following from standard C will get you in trouble:

- long ints

- unsigned ints (but note that pointers to char will do most of what you want here)

- floats and doubles

- structures, unions and bit fields

- multi-dimensional arrays

- parameters to #DEFINE macros

- += and his brothers -=, *=, etc.

- for statement

- do-while statement

- switch statement

- &&,||, unary !

There are other restrictions not listed here, but these seem to be the major ones. Despite this, C/65 is complete enough that one could write C/65 in itself, space considerations aside. For various reasons, C/65 is written in assembly language, which makes it extremely fast and quite small.

CHAPTER 2: USING C/65

Are you a C hacker already? Did the introduction (sections 1.1 and 1.2 tell enough? Anxious to get started? Here you go!

After using your favorite text editor to create a C source file, enter the command to OS/A+:

```
C65 filename1 filename2 [-T]
```

where

filename1 is the name of the source file,
filename2 is the name of the output file,
-T is an optional flag which tells C/65 to include the C source text as comments in the assembler output file.

Special Note:

E: is a valid filename for either source or output files (or both). I.e., compiler output can go directly to the screen, or you can even type in your C program on the fly.

Assemble the output file using MAC/65. Consult your MAC/65 reference manual for details of this operation. A complete, start to finish, compilation and assembly is shown in the next section.

NOTE: MAC/65 may be used to edit C source files if TEXTMODE is selected (via the TEXT command).

2.1 SIMPLE EXAMPLE

The following is an example of a complete, start to finish, C/65 program edit, compile, assembly, and "run".

NOTE: This example assumes you are working with an unprotected version of the master disk which has had MAC65.COM COPYed to it. PLEASE don't do this on your master system disk! We purposely do not protect our system disks so that you can keep safe, backup copies.

```
{D1:} MAC65
{EDIT}
TEXT
{TEXTMODE}
10 MAIN ( )
20     $(
30       PUTS("\N THIS IS A C/65 PROGRAM\N");
40     )$
50 #ASM D:IO.LIB
LIST #D:CTEST.C
{TEXTMODE}
DOS
{D1:} C65 TEST.C TEST.A -T
{D1:} MAC65 TEST.A D:TEST.COM -A
{D1:} TEST
```

NOTE: The characters in brackets {thusly} are intended to show you what the computer has put on the screen. For example, the computer has "D1:" on the screen, and you type "MAC65". MAC/65 loads from disk and prompts you with "EDIT"; you respond with "TEXT", and so on.

2.2 SOURCE FORM

In general, please note that the line numbers are optional and that line boundaries are ignored except for those at the end of compiler control statements (#ASM, #INCLUDE, #DEFINE).

CHAPTER 3: LANGUAGE DEFINITION

This chapter will begin an informal, top-down discussion of C/65. In general, C is a simplistic looking language; it achieves its popularity and power from its modular approach to programming. By its very nature, C encourages the user to build his/her own library of capabilities (i.e., functions).

3.1 RESERVED WORDS

Unlike many contemporary languages, C has very few "built in" statements and no predefined I/O capability at all. In fact, the complete list of C/65 keywords is as follows:

```
BREAK
CHAR
CONTINUE
ELSE
EXTERN
IF
INT
RETURN
WHILE
```

These keywords are reserved for compiler use and may NOT be used for any other purpose. Additionally, we would like to recommend that the C/65 user avoid the following keywords, which constitute the rest of the list used by standard C, if compatibility with other C compilers is desired.

auto	case	default	do	double
entry	float	for	goto	long
register	short	sizeof	static	struct
switch	typedef		union	unsigned

SPECIAL NOTE: The current version of C/65 recognizes keywords in UPPER CASE ONLY and is sensitive to case in all words. We anticipate that future versions will recognize keywords in both upper and lower case (or perhaps even mixed case). In the meantime, those experienced in C who prefer the lower case keywords may use #DEFINE, if desired, to redefine lower case versions (e.g., #DEFINE int INT). See section 3.3 for more information on the #DEFINE compiler control directive.

3.2 USER COMMENTS

C/65 conforms to the C standard for inclusion of user comments in C programs. Comments begin with the character pair '/*' and continue, ignoring line boundaries, until the character pair '*/' is found.

Comments may be used anywhere in C, even in the middle of an expression or statement, so they will not be further discussed hereafter.

CAUTION: Comments are NOT nested by C.

EXAMPLE

```
/*  this begins a comment
/*  this does nothing!
    then some more comments
    which end with the
*/
    this is NOT a comment
    and will cause compilation
    errors!
*/
    and that was too late...this
    generates more errors.

/*  a comment again...on one line */
```

3.3 GENERAL FORM OF A C/65 PROGRAM

The outermost level of a C/65 program may be thought of as consisting of just THREE distinct types of elements, which may be mixed and repeated in any order.

The elements of a C/65 program are:

- Compiler Controls
- Function Definitions
- Global Data Declarations

Each of these elements will be separately discussed in the sections which follow, but a simple (non-functional) example of each, used in the order above, might be as follows:

```
#DEFINE  char  CHAR
square ( num )  INT num ;
        $(  return (num*num);
        $)
EXTERN char  c ;
```

3.4 COMPILER CONTROLS

The C/65 compiler recognizes certain compiler control directives which begin with a "#" in the first column. Compiler controls do not DIRECTLY generate or affect the compiled code and need not be considered part of the formal language. Nevertheless, the specifications of C do include the compiler controls. While C/65 does not implement all the specified controls, it implements three very useful controls, including one which is not specified in standard C.

3.4.1 Source File Inclusion: #INCLUDE

form: #INCLUDE filename

purpose: requests inclusion of the source code of the specified file in the current compilation.

example: #INCLUDE D:STDIO.H

The #INCLUDE statement is most commonly used to include "header files" which define and/or implement various standard functions, variables, etc.

NOTE: The filename should not be enclosed by or preceded by any special characters (in contrast to standard C, where it would be enclosed by "..." or <...>).

CAUTION: #INCLUDE statements are NOT nestable. A file which has been #INCLUDEd may not itself contain a #INCLUDE compiler control directive.

3.4.2 Text Substitution: #DEFINE

form: #DEFINE anycharacters anyothercharacters

purpose: allows substitution of one character string for another, throughout a compile

example:
```
#DEFINE BEGIN $(
#DEFINE END   $)
```

These examples allow the user to redefine the C/65 block delmiters $(and $) to a form possibly more familiar looking.

The #DEFINE compiler control will cause the compiler to change all occurences of the first given string to the second given string.

NOTE: Macro arguments are NOT allowed as in standard C. C/65 simply performs a text substitution.

3.4.3 Assembly Language Inclusion: #ASM

form: #ASM filename

purpose: includes the named assembly language file (in place of the current line).

caveat: #ASM is not a standard C directive.

example: #ASM D:IO.LIB

Since the OSS products C/65 and MAC/65 do not yet produce relocatable, linkable object code, some means of including various library routines needs to be provided. #ASM is the means by which this is done in C/65.

NOTE: C/65 implements the #ASM directive by writing the line " .INCLUDE #filename" to the assembly language output file, in a form compatible with MAC/65. Because of this, the assembly language file cannot itself contain a .INCLUDE directive. ALSO, if the assembler used is indeed MAC/65, the included file MUST be a file SAVEd under MAC/65 and may NOT be an ASCII format file.

SPECIAL NOTE: The libraries to be included via the #ASM directive need NOT have been originally written in assembly language. In fact, a common way of performing multiple module compiles with C/65 is to compile one module (or several), go to MAC/65 and ENTER the C/65 assembly language output, SAVE the assembly language to another file, and then #ASM the SAVEd code.

If doing compiles of very large files, in fact, the only way to assemble the entire result might be to break the C/65 source into modules which may then be #ASMed by a master module. The critical restriction here is that any one assembly language file output by C/65 must be capable of fitting into MAC/65's editor memory space so that it can then be SAVEd.

3.5 FUNCTION DEFINITIONS

Functions are the largest building blocks of C. In fact, the language supports no other form of callable module. A program written in C is not in and of itself callable!

There is a convention, however, that the function named MAIN will receive control when the program is loaded and run by the operating system. This MAIN function must then setup and control the flow to the rest of the program by, in turn, making function calls.

In any case, we first need a format for function definitions:

Function Definition

```
Function Declaration
        $(
           Local Data Definition(s)
           Statement(s)
        $)
```

The character pairs $(and $) are the block delimiters of C/65, since the keyboard of Apple and Atari computers cannot usually generate the { and } characters which are used by standard C (but see section 3.4.1 if you don't like those characters).

Local data definitions will be discussed in section 3.6, along with the global data definitions. Statements will be introduced in section 3.7, but the subject is complex enough to require its own chapter (chapter 5, because before we can seriously discuss statements we must understand expressions, chapter 4).

The function declaration, however, needs explanation now. In many ways, the C/65 function declarations are significantly simpler and more restricted than those of standard C. In fact the general form is simply as follows:

Function Declaration

```
function_name ( opt_param1 , opt_param2 , ... )
        declaration_of opt_param1 ;
        declaration_of opt_param2 ;
        ...
```

Functions in C/65 are presumed to return INTs. If you need to return something else (e.g., a pointer), simply assign its returned value to a variable of the proper type (see the example below).

There may be any number of parameters (including zero), each of which is presumed to be an INT unless otherwise declared. The form of a parameter declaration is the same as that of a local variable declaration, to be discussed in section 3.6, but briefly we may state here that a parameter may be of any standard C/65 variable type.

As promised, then, here is an example of a function. This routine will search a string of characters for a digit and return the address of (or a pointer to...same thing) the first digit found:

```
looky ( here )   CHAR *here ;
        $(  WHILE ( *here > '9' | *here < 'Ø'  )
                ++here ;
            RETURN ( here ) ;
        $)
```

Several parts of this function need the explanations which will follow in subsequent chapters, but the points to be made here are:

The variable 'here' is a character pointer (or character string) passed into 'looky' from the calling function.

The function returns the updated value of 'here', the address of the digit. (Of course, this function is flawed, in that it keeps looking forever for that digit, which it might not find.)

Since C/65 believes that functions always return INT, the calling program could play it safe thusly:

```
CHAR *foundit ;
foundit = looky ("find a digit 1 2 3");
```

3.6 DATA DECLARATIONS

There are four places in C/65 where one or more data declarations are legal, three of those places have already been noted (the fourth will be discussed in Section 5.2). The legal places are:

GLOBAL VARIABLES
Outside of any functions (at the global level): any number at any place.

FUNCTION PARAMETERS
In a function declaration, after the right parenthesis and before the left brace (and matching name(s) with parameter names listed between the parentheses).

LOCAL VARIABLES
In a function definition, after the left brace and before the first statement(s).

LOCAL VARIABLES
In any compound statement, after the left brace and before the first statement(s).

Although there are fundamental differences in the implementation of the various types of variables, the user will see little if any difference in usage or form. However, one important difference to be noted is the scope of the various types of variables.

3.6.1 SCOPE OF VARIABLES

Global variables are known throughout a program (NOT just a program module, in the case of separately compiled modules). Function parameters are known throughout the function in which they are declared. Local variables are known within the block (delimited by braces, whether or not function delimiting braces) in which they are defined.

Given two variables of the same name, which must be declared at different "levels" (as levels are given below), the "inner" variable will be known while the outer one is temporarily forgotten.

EXAMPLE:

```
CHAR a,b,c ;  /* global variables */

afunction ( a,d )
        INT a,d ;  /* parameters */
    $(
    CHAR *b ; /* local to this function */
    a ;   /* refers to INT parameter */
    b ;   /* refers to local *CHAR */
    c ;   /* refers to global CHAR */
    d ;   /* refers to INT parameter */
        $(
        INT *c, *d ; /* local to this
                        block */
        c ; /* the local *INT */
        d ; /* the local *INT */
        b ; /* still the local *CHAR */
        $)
    c ;   /* back to the global CHAR */
    $)
```

3.6.2 C/65 DATA TYPES

C/65 has two primary data types: INT and CHAR.

INTs are signed 16 bit quantities; CHARs are signed 8 bit quantities.

CHARs are widened to 16 bits with sign extension prior to being passed as parameters or being used in expressions. Be aware that when using characters with the high bit set, you are dealing with negative numbers. This can have amusing (?) side effects when using characters to index into an array, for example.

In addition to the two primary data types, C/65 allows the user to declare pointers to the primary types AND singly dimensioned arrays of the primary types.

Rather than try to make a complex single form which shows the possible data declarations, we present here a short table of the allowed declarations:

```
CHAR name ;
INT  name ;      /* the simplest forms        */
CHAR *name ;     /* a pointer to a CHARacter  */
INT  *name ;     /* a pointer to an INTeger   */
CHAR name[] ;    /* an array of CHARacters    */
```

```
CHAR name[constant] ; /* array of CHAR with
                         size of array defined*/
INT  name[] ;   /* an array of INTegers       */
INT  name[constant] ; /* array of INT  with
                         size of array defined*/
```

ALSO, all of the above declarations may be prefaced by the keyword EXTERN (but see the next section for restrictions).

SPECIAL NOTE: All declared names in C/65 must begin with an alphabetic character and may contain any number of alphanumeric characters. However, only the first 8 characters are significant. Thus C/65 sees the names "lengthofline" and "lengthofrecord" as being identical. (But "LeNgTHofthis" is different -- remember that case is significant).

3.6.3 GLOBAL DECLARATIONS

In C, any variable declared outside of any function is perforce a global variable. Presuming we are restricted to a single module compile, the only real differences between a global and local variable in C/65 are as follows: (i) a global variable's space is defined at compile time while local variables are defined on a system stack at run time; (ii) a global value retains its value at all times while a local variable's value is forgotten each time the function (or block) defining it is exited; and (iii) global variables and their references may be found in the assembly language listing by name while local variable names are known only to the compiler (thus global variables may be easier to debug with).

However, if we consider separately compiled modules (or, for that matter, assembly language modules and libraries, since they are really the same thing), one aspect of global variables becomes important: they are known by name to all modules of the (assembled) program.

Consider, though, what would happen if two separate modules tried to declare the same variable name. The assembler would receive two separate definitions (e.g., .WORD or .BYTE) of the same name and would give a "Duplicate Definition" error. Therefore was the keyword EXTERN invented and reserved.

Any data declaration which is, at the global level, prefaced by the keyword EXTERN is presumed to refer to a name which will be defined IN ANOTHER MODULE of the same (assembled) program. This means that, in any complete program, each variable name should be declared WITHOUT the keyword EXTERN one time and one time only!

Finally, the other important point to be noted is that array declarations have a similar problem: the size of an array must be defined once and only once. Thus, it is good practice to avoid putting an array size (a constant) between the brackets when the EXTERN keyword is used.

SPECIAL NOTE: The global name "A" is illegal in C/65, to avoid conflict with 6502 mnemonics which use "A" to designate the accumulator.

3.6.4 LOCAL DATA DECLARATIONS

All occurrences of data declarations within a pair of braces (recall that C/65 uses $(and $) in lieu of { and }) are presumed to be local declarations.

In C/65, local variables are allocated space on the C/65 stack and "live" only as long as the function defining them lives (i.e., until the function exits or RETURNs). Access to local variables is thus somewhat more complicated and slower than access to global variables; and yet, through a quirk in the necessary 65Ø2 implementation of the language, an access to a local variable actually requires less memory than a similar global access.

Since all local variables can only be defined within the enclosing block, there is no need for an ambiguous array reference (that is, one which does not declare the constant size of the array). The program SHOULD provide a size for each local array.

Incidentally, C/65 generates less code for local variables which are contained within the first 127 bytes of local space (also known as AUTOmatic space in standard C). It is therefore a good idea to place all local array declarations AFTER the non-array declarations unless the array names are used considerably more than the non-array names.

NOTE: The keyword EXTERN is ILLEGAL inside the body of a function. A local variable may NOT be declared EXTERN.

3.6.5 PARAMETER DATA DECLARATIONS

Function parameter variables, as described above, are also allocated space within the C/65 stack (and see chapter 7 for a description of exactly what part of the stack is used). In most respects, then, function parameters are identical to local variables.

However, there is one important difference, having to do with how C defines and uses pointers. Briefly, any expression involving a pointer may be converted by the C compiler to one involving an array reference (or vice versa, as desired by the implementer). Section 4.4 will present more details on this concept, and generally the substitution will be invisible to the user.

Nowhere, though, is this subtle point more strongly felt than when function parameters are involved. To illustrate:

```
anyfunction ( buffer )  CHAR *buffer ;
        is  EXACTLY the same as
anyfunction ( buffer )  CHAR buffer[] ;
```

And within 'anyfunction', the programmer could code

```
*(buffer+i) ;     *buffer;
        or, EXACTLY equivalently,
buffer[ i ] ;     buffer[ 0 ];
```

NOTE THE IMPLICATIONS: the calling function will presumably pass the function a CHARacter array (which might be a literal string, as in 3.5 above). What is actually passed, though, is the ADDRESS of the array. Thus, the use of the pointer ('*buffer', etc., above) is a better visualization of what actually occurs. BUT the user who prefers to think in terms of arrays is encouraged to do so: the compiler literally cannot see the difference.

Finally, note that parameter arrays should not have a size defined (there should be no constant between the brackets), since no array is actually allocated. (A not uncommon practice, incidentally, is to pass a function not only the array--via its address--but also, as a separate parameter, the array's size.)

SIDELIGHT: Though not immediately obvious, all the above taken as a whole suggests that a usage of 'arrayname' is equivalent to a usage of '& arrayname[Ø]' (that is, 'the address of the zeroeth element of arrayname). Indeed, this is true, and it is perfectly legal in C to use either of the following forms:

```
given:
  CHAR buffer[5ØØ] ;

then:
  callfunction( buffer ) ;
is the same as:
  callfunction( & buffer [Ø] ) ;
```

Note: '&' is the 'address-of' operator. See section 4.5.1 for clarification.

3.7 Introduction to Statements

Just as functions are the building blocks of C programs, so are statements the building blocks of functions. If you are new to C and/or other block structured languages (e.g., if you are only familiar with BASIC or PILOT or similar simple languages), statements may be the most familiar looking part of C/65. After all, most languages provide for statements similar to this:

```
total = total + newamount ;
```

And perhaps the semicolon looks foreign to you, but at least it looks "right". So it is with most C/65 statements: they "look right" (well...maybe almost right?) to most programmers.

EXCEPT. There always has to be a catch. The catch in C is that there are so few statement types. The C novice almost always asks, "But how do I do Input/Output?" And the answer is, simply, "With functions." The LANGUAGE DEFINITION of C does not actually include a specification of ANY input/output capabilities whatsoever. And yet, if you examine chapter 6, you will find a rich array of I/O functions defined (with definitions virtually identical to those used on UNIX). BUT...the real beauty of C is that, if you don't like what we give you, you can write your own functions.

And this applies to all aspects of the language: if you don't find a statement to do what you want, write a function which will (using the statements which are provided, of course). Then, anytime you need such a statement, use your function.

Chapter 5 presents a fairly complete view of the various types of statements, but let us finish this section by simply noting that any expression (including an assignment, of course) may be used as a statement, any function may be used as a statement, and any group of statements may be combined into a single statement. This is ALL in addition to the keyword statements (IF, WHILE, etc.) which are native to C/65.

Chapter 4: EXPRESSIONS

Expressions are the building blocks of C statements. In point of fact, an expression is a valid statement in C, whether it be an assignment statement or not. This is not surprising, since an expression may contain one or more function calls and/or may perform variable incrementing or decrementing, all of which may alter the values of one or more C variables.

When used with the various C keywords, expressions are built into all the statement types recognized by C/65.

Expressions are built "from the inside out". Rather than give a formal definition (e.g., a Backus-Naur listing) of the various expression forms, we will present the components of expressions in a "bottom up" order.

4.1 Local or global variable names:

Any name previously declared as a local variable, parameter, or global variable may be used, by itself, as an expression.

Remember, names are significant to 8 characters, must begin with an alphabetic character, and case is preserved.

EXAMPLE:

```
EXTERN CHAR *name ;
INT globalint ;

anyfunction( thisisa )
        INT thisisa ;
    $(  INT localint ;
        /* after the above declarations,
           all the following are valid
           expressions: */
        localint;
        name    ;
        thisisa ;
        globalinfallible ;
    $)
        /* note that this last is the
        same as 'globalint' since only
        eight characters are used */
```

4.2 Constants

Recognized constant forms are as follows:

- decimal numbers, in the range allowed by C/65

- one or two ascii characters enclosed in single quotes (').
  E.g., 'z', 'ab'.

  Standard C escape sequences are also recognized.

  They are:
  '\n' -- newline
  '\b' -- backspace
  '\t' -- tab char
  '\nnn' -- three octal digits
  (e.g., \ØØ4 is control-D)
  '\Øxhh' -- two hex digits
  (e.g., \Øx04 is also control-D)

- a string of ascii characters inside of double quotes (").
  E.g., "this is a string".

  As in standard C, the value of a string constant is the address of the first character. Succeeding characters are stored sequentially and are terminated with an ascii nul (zero byte).

  The escape sequences defined above for character constants also work in string constants.
  e.g., "\nlinel\nline2"

  NOTE: Of course, an expression consisting of only a single contstant or variable name doesn't "do" anything -- it just sits there and evaluates its navel.

4.3 Functions

Properly, a function usage (remember, we are here talking about elements of expressions) consists of a function name, followed by a set of zero or more parameter expressions enclosed in parentheses. In standard C, a pointer to a function may be used in place of the function name, and the call is then made to the address contained in the pointer.

Since C/65, at this time, has no means to declare that something (e.g., a variable) is indeed a pointer to a function, the C/65 definition is simpler:

> Functions are ANY expression followed by an open parenthesis.
>
> (And, of course, a name qualifies as an expression, so the simplest standard C form is satisfied by this definition.)

While this is far from standard C, if a program limits itself to name(s) followed by the open parenthesis, it will remain upward compatible with standard C. However, the looser definition allows such crudities (or niceties, depending upon your viewpoint) as:

```
1000();     /* calls location 1000 decimal */

array[2]() /* calls routine whose address is
              in the 3rd element of array
              ( remember, C zero-indexes arrays)
           */
```

The parameters to functions are simply listed between the open parenthesis and a closing parenthesis, separated by commas, and are themselves expressions! (See how cleverly and easily we begin to build up to more complicated expressions.)

EXAMPLE:

```
int i ;          /* declare i an integer */

foobar( i ) ;    /* a valid expression,
                    although it does assume
                    the existence of the
                    function 'foobar'.
                 */
```

Parameters are pushed onto the C system stack in the order listed (not important unless you are trying to interface to C/65 from assembly language, in which case see Chapter 7).

A value is ALWAYS returned from a function call, but it need not be used and may be junk (if the called function neglects to return a specific value).

4.4 Subscripted Variables

Arrays and pointers may be followed by an expression enclosed in square brackets to access the elements of the given array (or elements of the presumed array pointed to by a pointer).

A convention in standard C which is carried over to C/65 is that any subscripted variable may also be represented by its pointer expression equivalent. That is, the form

```
name [ element ]
```

is functionally and properly equivalent to

```
name + element  .
```

The subtle implication here is that the "element" number is "sized". In C/65, this means that if "name" is a character pointer or character array, the value of "element" is added to the address of name (for arrays) or the contents of name (for pointers) to achieve the address of the element asked for.

For integer pointers or integer arrays, though, the value of "element" must be doubled before the addition takes place, since integers occupy two bytes each.

If this point seems esoteric and unnecessary at this time, we apologize. But the concept needs explanation, since otherwise integer pointers can and will cause problems. (And see also section 3.6.5 for related discussion).

```
char vector[30] ;
int *pointer ;

vector[0];    /* first element of array vector */
vector[j+10]; /* 10 is added to j, with the result
                 being used as the index */
pointer[9];   /* the number stored in pointer is added
                 to 18 and the value at that location
                 is fetched */
pointer+9;    /* exactly the same as the line above! */
```

As mentioned, only single-subscript arrays are allowed.

4.5 Introduction to Operators

More complex expressions may be constructed from the primary expressions by using three kinds of operators: unary, binary, and comparison.

4.5.1 Unary Expression Operators:

Given an expression x,

-x negates x

*x Standard C definition: if x is a pointer, refer to the object pointed to by x. However, C/65 allows a looser definition: If x is an expression, refer to the (assumed) INT pointed to by the expression. "Pointed to by x" means that the value of x is a memory address and the program is to operate on the contents of that address (rather than the address itself).

EXAMPLE:

```
CHAR *pc ; /* pc is a CHARacter pointer*/
INT  *pi ; /* pi is an INTeger pointer */
INT  i   ; /* i is a simple integer    */

pc = pi = 1000 ;  /* both now point to
                     location 1000 */

*pc = 5 ; /* loc'n 1000 now contains 5 */
*(pc+1)=8 ; /* and 1001 contains 8 */

i = *pi ; /* i gets the INT at location
             1000, which is 5 + 256*8
             (standard 6502 order) so
             i now equals 2053 */
i = *1000 ; /* same effect as above!! */

*pi = 257 ; /* stores the INT 257 into
               locations 1000-1001    */
*1000=257 ; /* ditto...in C/65 only   */
*pc = 'A' ; /* stores the CHAR value of
               65 into location 1000...
               does not affect 1001.  */
*1000='A' ; /* NOT the same!  Stores the
               INT value 65 into loc'ns
               1000 and 1001...careful*/
```

REMEMBER: unless otherwise explicitly declared, pointer expressions are always assumed to point to INTegers!!!

&x evaluates to the address of x. Generally, this operator may only be applied to variables and array elements. Since most expressions have a value only (since they "exist" only on the system stack), trying to take their addresses is illegal.

EXAMPLE: &1000 ; /* illegal! */

After all, just what memory location contains the constant 1000? Perhaps none, perhaps several? C says that it can't know and won't try to tell you.

EXAMPLE:

```
CHAR *p,c;

If you do:    p = &c;
Then:         *p is equivalent to c

But:          &p is not equal to &c, and
              &p is not equal to c, etc.
```

This example shows that there is no relationship between the address of p and the address of c. Here we let p equal the address of c. Then we can say that the object pointed to by p is equivalent to c. But the address of p does not equal the address of c nor does the address of p equal to c.

x++ ++x The two forms of this operator refer to post increment and pre increment respectively. Post increment means that the storage location x will be incremented AFTER it is used. Pre increment means that the storage location x will be incremented BEFORE it is used. The value that x will be incremented by (whether it be post or pre) depends on what x was declared as. If x was declared as anything other than a pointer to INT then x++ and ++x will increment the storage location of x by one. If x was declared as a pointer to INT then x++ and ++x will increment the storage location of x by two.

x-- The two forms of this operator refer to post
--x decrement and pre decrement respectfully. Post decrement means that the location x will be decremented after it is used. Pre decrement means that the storage location x will be decremented before it is used. The value that x will be decremented by (whether it be post or pre) depends on what x was declared as. If x was declared as anything other than a pointer to INT then x-- will decrement the storage location of x by one. If x was declared as a pointer to INT then x-- will decrement the storage location of x by two.

NOTE: Usages of ++x and --x generate less code than usages of x++ and x--. So use the former versions when no order of operation is needed.

4.5.2 Binary Operators:

Binary operators take two expressions, operate on them, and result in another expression.

Given expressions a and b,

a+b adds a to b.

a-b subtracts b from a.

a*b multiplies the signed a value to the signed b value producing a signed result.

a/b divides the signed a value by the signed b value producing a signed result.

a%b The value returned for this operation is the remainder of a divided by b (or a modulo b).

Example:

(5 % 2)

would produce the value 1. The division performed is signed.

CAUTION: The above 5 operators do not recognize overflows and underflows.

a|b gives the bitwise inclusive or of a and b. "Inclusive or" can be defined as: Given the binary value of a and the binary value of b, if either of the corresponding bits are a 1 then the resulting bit is a 1, otherwise the resulting bit is a Ø. As shown below:

Example:

```
(5 | 12) == 13

 where  ØØØØØ1Ø1  =  5
 and    ØØØØ11ØØ  = 12
        --------
 result ØØØØ11Ø1  = 13
```

a^b gives the bitwise exclusive or of a and b. "Exclusive or" can be defined as: Given the binary value of a and the binary value of b, if both of the corresponding bits are the same then the resulting bit is a Ø, otherwise the resulting bit is a 1. As shown below:

Example:

```
(5 ^ 12) == 9

where   ØØØØØ1Ø1  = 5
and     ØØØØ11ØØ  =12
        --------
result  ØØØØ1ØØ1  = 9
```

a&b gives the bitwise and of a and b. A "bitwise and" can be defined as: Given the binary value of a and the binary value of b, if both of the corresponding bits are a 1 then the resulting bit is a 1, otherwise the resulting bit is a Ø. As shown below:

Example:

```
(5 & 12) == 4

where   ØØØØØ1Ø1  = 5
and     ØØØØ11ØØ  =12
        --------
result  ØØØØØ1ØØ  = 4
```

a<<b shifts a arithmetically left b bits

Example:

(7 << 3) evaluates to 56

(8 << 3) evaluates to 64

a>>b shifts a arithmetically right b bits

Example:

(7 >> 3) evaluates to Ø

(8 >> 3) evaluates to 1

NOTE: In the 2 shift operators above, any bit or bits shifted too far left or right, out of the CHAR or INT, will be lost as C does not recognize the concept of a "carry bit".

a=b The assignment operator ' = ' can be used anywhere a binary operator can be used.

Example:

x[k=k+3] = a-(b=c/d)

This example performs 3 assignments. b is set to the value of c/d. k is set to k+3. The array element x (new value of k) is set to a minus new value of b.

Example:

a = b = c = Ø

c is set to Ø. Then b is set to c, i.e., to Ø. Then a is set to b, also Ø.

4.5.3 Comparision Operators:

Comparison operators return a 1 or Ø based on the result of a comparison of two expressions. A 1 is returned if the expression resulting from the comparison is true, a Ø is returned if it is false.

Given the expressions a and b:

a==b Tests if a is equal to b

a!=b Tests for inequality

a<b Tests for a less than b

a>b Tests for a greater than b

a<=b Tests for a less than or equal to b

a>=b Tests for a greater than or equal to b

If a comparison involves a pointer, an unsigned comparison is performed. Otherwise, a signed comparison results.

Example:

```
INT  i,j, *pi,*pj ;
pi = i = -1000 ;  /* but C/65 'thinks' of
                     pi as containing an
                     address of 55535 !! */
pj = j =  1000 ;

i  <  j  ; /* is true...returns 1 */
pi <  j  ; /* is false...unsigned compare
              looks like 55535 < 1000
              so returns Ø  */
pj < pi  ; /* is true...returns 1 */
```

4.5.4 Operator Precedence:

The table below summarizes the rules of precedence of all operators. Operators on the same line have the same precedence; rows are in order of decreasing precedence, so for example, *, /, and % all have the same precedence, which is higher than that of + and -.

	Operator
	() []
@	++ -- - * {pointer} & {address}
	* / %
	+ -
	<< >>
	< <= > >=
	== !=
	&
	^
	|
@	=

@ NOTE: operators on these two lines associate right to left; all other operators associate left to right.

EXAMPLE:

```
* p ++  is equivalent to
* ( p++ )

* & q   is equivalent to
* ( &q )
```

4.6 Building Complex Expressions and Statements

The various primary operators and operands presented above may be combined in some very complex and exotic ways to provide some sophisticated power to the C/65 user. And, since any expression may be turned into a C/65 statement by simply appending a semicolon, we can easily expand the built in structures of the language.

Some of the more obviously useful statement/expressions may be grouped as follows:

Expressions which call functions.
Since calling a function invokes all the code of that function and all the code of any function it in turn calls.
Expressions which perform assignments.
Since we are changing program and system variables in hopefully meaningful ways.
Expressions which perform incrementing or decrementing.
Again, since we are changing a system or program location in a meaningful way.

EXAMPLES:

```
PUTC( c ) ; /* call a function to
               perform I/O */
i = j * k ; /* calculate a new value
               for a variable */
++counter ; /* count how many times
               something happens */
```

But the real power of C becomes apparent when we start combining all these capabilities into single statements and sequences of statements.

EXAMPLES:

```
CHAR *to, *from;
* to ++  =  * from ++ ;
          /* moves a character from the location
             pointed to by 'from' to the
             location pointed to by 'to'; also
             increments both 'to' and 'from'
             AFTER using each !!  */
```

```
flag = ( (c = GETC(channel) ) >= 'Ø') & (c <= '9');
          /* gets  a character from the I/O file
             specified by channel and assigns it
             to the variable c.  Checks  to  see
             if the character is numeric (in the
             range   of   ASCII   'Ø'   to   '9'
             inclusive).   If  it  is  numeric,
             assigns  1  to  flag.  If it is not
             numeric, assigns Ø to flag.  */

spaces = spaces + ( ( *buf++ = GETC(channel))==32 );
          /* gets a character from the file  and
             stores   it  in  a  buffer  at  the
             location pointed to by  'buf'.   If
             the  character is a space, then the
             counter  'spaces'  is  incremented.
             In  any  case, 'buf' is incremented
             to point to the next loc'n */
```

NOTE FOR BASIC USERS ONLY:
Just to give you an idea of the power implicit here, we present the BASIC A+ equivalents of the above examples:

```
1.  poke from,peek(to) :
        from=from+1 :
        to = to+1
2.  get #channel,c :
        flag = (c >= ASC("Ø")) AND
               (c <= ASC("9"))
3.  get #channel,c :
        poke buf,c :
        buf = buf + 1 :
        if c=32 then spaces = spaces+1
```

CHAPTER 5: STATEMENTS

There are 3 kinds of C statements: simple statements, compound statements, and keyword statements.

5.1 SIMPLE STATEMENTS

A simple statement is merely an expression followed by a semicolon. That is, a simple statement has the form:

```
expression;
```

Some examples of a simple statement follow.

```
1) c = 0;

2) ++buffer pointer;

3) PUTS ("a message");

4) a = a + doit(3,doocheck(7,do(7)),do(3));
```

NOTE: An expression may or may not involve an assignment operation, as shown.

By definition, any place a simple statement is legal and/or needed in C, a compound statement is equally and equivalently legal and/or necessary.

5.2 COMPOUND STATEMENTS

Compound statements can be defined as any number of statements (of any kind, including other compound statements) inclosed in braces to form a single statement.

(Remember, braces cannot be generated by the keyboard, so C/65 uses '$(' for the '{' and '$)' for the '}' of standard C.)

Compound statements have the form:

```
$( statement1;
   statement2;
      ...
   statementN;
$)
```

An example of a compound statement:

```
$( INT a,b,c;

   a = 1;
   b = 2;
   c = a + b;
$)
```

NOTE: Variables may be declared at the beginning of any compound statement as shown above. See also section 3.6

Of course other statements can be used in compound statements and some examples follow in the keyword statement definition.

5.3 KEYWORD STATEMENTS

C is by nature a recursive language; hence it is not surprising that the definition of the language involves recursive definitions. Compound statements (last section) are a perfect example of this: a compound statement consists of a collection of statements any of which might in turn be a compound statement, etc.

The keyword statements of C/65 build on this same concept: some of the keyword definitions require the use of a statement to complete their definition. And what kind of statement can be used thusly? Any statement, of course, including a simple statement, a compound statement (which consists of any number of statements, etc.), or a keyword statement (which can be of the same type as the original statement, thus requiring yet another statement, ad nauseum). Perhaps section 5.3.3 gives the best example of this logic, in the example of an ELSE IF structure.

5.3.1 IF statement:

The IF statement is used in decision making. It has the form:

```
IF (expression) statement;
```

Here the expression is evaluated. if it is non-zero then the statement is executed, otherwise it is not.

EXAMPLE:

```
...
INT c;
c = GETCHAR();
IF (c == 'a') PUTCHAR(c);
...
```

This example will get one character from the keyboard. If the character is the letter "a" then it will put the letter back out on the screen, otherwise it will do nothing.

5.3.2 IF-ELSE statement:

The IF-ELSE statement group is also used for decision making. It has the form:

```
IF (expression)
   stetement1;
ELSE
   statement2;
```

Here the expression is evaluated. If it is non-zero then statement1 is executed and control passes to after statement2. If the expression evaluates to zero then statement2 is executed and control continues sequentially.

EXAMPLE:

```
#DEFINE Alfnum '1'

$(
   INT c;
   c = GETCHAR();
   IF (c == 'a')
        PUTCHAR(Alfnum);
   ELSE
        PUTCHAR(c);
$)
```

The word Alfnum gets defined as a constant, the character '1'. The variable c will be equal to the letter typed at the keyboard. If the letter typed in was the letter "a" then the statement PUTCHAR(alfnum), will be executed putting the character '1' back onto the screen; otherwise the letter typed in will be repeated on the screen, by the execution of the statement following the ELSE.

5.3.2 (continued)

IF-ELSE statements can also be nested as shown below.

EXAMPLE:

```
$(
   INT c;
   c = GETCHAR();
   IF (c >= 'A')
   $( PUTCHAR('1');

      IF (c == 'B')
         PUTCHAR('2');
   $)
   ELSE
      PUTCHAR(c);
$)
```

Here c will equal a character typed in from the keyboard. If the letter is greater than the letter "A" the number "1" will be printed on the screen. At the same time if the letter is a "B" then the number "1" and the number "2" will be printed on the screen. Otherwise the character input will be repeated on the screen.

5.3.3 ELSE IF statement:

Even though "ELSE IF" is not properly a C/65 statement type, the construction occurs so often that it is worth a brief separate discussion. It has the form:

```
IF (expression)
   statement1;
ELSE IF (expression)
   statement2;
ELSE IF (expression)
   statementM;
ELSE
   statementN;
```

This sequence of IF's 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 code for each statement is either a single statement or a compound statement.

The last ELSE part handles the "none of the above" or default case where none of the other conditions were satisfied. Sometimes there is no explicit action for the default; is that case the trailing

```
ELSE
   statement;
```

can be omitted, or it may be used for error checking to catch an "impossible" condition.

EXAMPLE:

```
INT c;

c = GETCHAR();

IF (c == 'A')
   PUTS("ALWAYS");
ELSE IF (c == 'B')
   PUTS("BE);
ELSE IF (c == 'C')
   PUTS("CAREFUL");
ELSE
   PUTS("!!!");
```

5.3.3 (continued)

Because of the definition of the IF ELSE statement, this is equivalent to:

```
INT c;

c = GETCHAR();

IF (c == 'A')
   $( PUTS("ALWAYS"); $)
ELSE
   $( IF (c == 'B')
         PUTS ("BE");
      ELSE
         $( IF (c == 'C')
               PUTS("CAREFUL");
            ELSE
               PUTS("!!!");
         $)
   $)
```

If the character from the keyboard is the letter A then the message ALWAYS gets printed on the screen and control passes to after the last ELSE. If the character is the letter B then the message BE will get printed on the screen, and again control passes to after the last ELSE. If the character was the letter C then the message CAREFUL will be printed on the screen and control passes to after the last ELSE. If the character from the keyboard is not the letter A, B, or C then the message !!! will be printed on the screen.

Notice, even though these two examples execute the same, when example 1 is compiled C/65 visualizes the program as if the braces were there as they are in example 2.

NOTE: Any number of ELSE IF's may be used in these constructions.

5.3.4 WHILE statement:

The WHILE statement is used for program flow and has the form:

```
WHILE (expression) statement;
```

The statement can either be simple or compound. With the WHILE statement, at execution time the expression is evaluated. If it is non-zero the statement is executed and the expression is evaluated again. This flow will continue until the expression evaluates to a zero. When this happens, control is passed to the next statement following the statement.

EXAMPLE:

```
INT c;

WHILE ( (c = GETCHAR(c)) != 'Z')
PUTCHAR(c);
```

This example will get characters from the keyboard and put them to the screen. This process will continue until the letter "Z" is encountered which will cause the expression to evaluate to zero, ending the WHILE loop.

EXAMPLE:

```
INT c,alfnum;
alfnum = '1';

WHILE ((c = GETCHAR() != 'Z')
$(
        IF (c == 'A')
        PUTCHAR(alfnum);
        ELSE
        PUTCHAR(c);
$)
```

This is an example of a WHILE with a compound statement. Here the expression is evaluated. If the letter from the keyboard is not the letter "Z" then we will execute the compound statement which will check to see if the letter input is the letter "A". If it is then we will put the number "1" to the screen, otherwise we will just put the letter to the screen. If the character is a "Z", the compound statement is skipped over.

5.3.5 BREAK statement:

The BREAK statement is used to provide control loop exits other than by testing at the top or bottom. In C/65, it provides an early exit from the WHILE statement. A BREAK statement will cause the innermost enclosing loop to be exited immediately. The BREAK statement has the form:

```
BREAK;
```

EXAMPLE:

```
INT c,d;

d = '9';

WHILE ((c = GETCHAR()) != 'A')
   $(
        IF (c == 'Z')
           BREAK;
        ELSE
          PUTCHAR(c);

PUTCHAR(d);
   $)
```

This example shows how a BREAK statement can work. Here the WHILE loop will end if the letter "A" is typed in from the keyboard (this is controled by the same line the WHILE statement is on). But inside the WHILE loop, if the letter typed was a "Z" then the BREAK statement is executed, which causes the WHILE loop to end and the number "9" to be printed on the screen.

5.3.6 CONTINUE statement:

The CONTINUE statement is related to the BREAK statement, but less often used. CONTINUE causes the next interation of the enclosing WHILE loop to begin. This means the test part of the WHILE loop will be executed immediately. CONTINUE has the form:

```
CONTINUE;
```

EXAMPLE:

```
INT c;

WHILE ((c = GETCHAR()) != 'Z')
$(
   IF (c == 'A')
      CONTINUE;
   PUTCHAR(c);
$)
```

Here the WHILE loop will get characters from the keyboard and write them back to the screen as long as the letter "Z" is not typed in. The CONTINUE statement comes into play only when the letter "A" is typed in. When the letter "A" is typed in the CONTINUE statement causes the control of the WHILE loop to go back and get another character from the keyboard without printing the letter "A" on the screen.

5.3.7 RETURN statement:

The return statement is used to return control back to the caller. It can also pass back values to the caller if they are needed. The RETURN statement has the form:

```
RETURN;
  or
RETURN expression;
```

EXAMPLE:

```
INT c,d;
c = GETCHAR();

IF (c >= 'Ø') | (c <= '9')
   d = 1;
ELSE
   d = Ø;
RETURN(d);
```

In this example c is equal to the character typed in from the keyboard. If the character is in the range Ø-9 then the variable d will be returned with a one in it, otherwise d will be returned with a zero.

This example could be used when only numeric input is allowed from the keyboard. The program that called this function would look at what was returned and if it was a zero an error message could be printed on the screen reminding the user that only numeric entries are allowed.

5.3.8 Null statement:

The NULL statement does nothing. It can sometimes be useful as a place holder in WHILE and IF statements. It has the form:

```
;
```

EXAMPLE:

```
INT c;

WHILE ((c = GETCHAR()) != 'Z')
   IF (c == 'A')
      PUTS("ALLRIGHT");
   ELSE
      ;
```

This example illustrates how the NULL statement is used as a place holder. Here as long as the letter "Z" is not input from the keyboard the WHILE loop will continue. If the letter "A" is typed, the message ALLRIGHT gets printed on the screen. If the letter is not an "A" then nothing happens, but by putting the NULL statement in we have made it easier to change the program if later we would like it to do something after the ELSE.

EXAMPLE:

```
WHILE (( *buf++ = GETCHAR() ) >= Ø );
```

In this example because all our data checking and movement is done within the control part of the WHILE loop, a NULL statement must be used because the rules of the WHILE statement specify it.

CHAPTER 6: C/65 LIBRARY FUNCTIONS

C/65 comes with four libraries:

- a runtime library to provide the routines called directly by the compiler to do arithmetic and logic.
- an I/O library to provide low level input and output functions.
- a simple graphics library, allowing only the most fundamental graphics capabilities.
- a storage allocation library to provide a dynamic storage allocation capability.

The runtime library is always necessary and is automatically included by the compiler. It also contains the routines GETCHAR and PUTCHAR, so it may be all that you need, including I/O.

The I/O library is only necessary if you will perform I/O involving the standard C/65 functions listed and described in section 6.3. Similarly, the graphics library functions are optional and are listed and described in section 6.4.

The storage allocation library is only necessary if calls will be made to ALLOC and FREE, as they are described in section 6.5.

If you use any of the routines of the C/65 I/O, graphics, and/or allocation libraries, it is necessary to include one or more of the following lines (as appropriate) at the END of your C source code:

```
#ASM  D:IO.LIB
#ASM  D:GRAPHICS.LIB
#ASM  D:ALLOC.LIB
```

NOTE: The I/O library has been coded to take advantage of MAC/65's ".if .ref" feature. Thus it is necessary to put the #ASM statement at the end of your code so that MAC/65 can tell what routines have been used so as to assemble only those routines.

ALSO NOTE: The previous #ASM statements assume that the libraries are on D1:. Use Dn: as appropriate with your system.

6.1 STANDARD ERROR CODES

Most of the I/O fuctions in each library return an INT value. This value is used to determine if the function called has executed properly. If the value returned was Ø or a positive value the function called has executed properly. If the value returned was negative, add 256 to the value and use this number to determine the OS/A+ operating system error. An explanation of each can be found in your OS/A+ manual.

All references in the following sections to "Standard Errors" or "Standard Error Codes" imply the above convention.

6.2 RUNTIME LIBRARY FUNCTIONS

The Runtime Library supplied with C/65 has the basic building blocks needed by C/65 to create the assembly language for your C programs. It is mostly invisible to the user and performs all the operations used in C/65, such as multiply, divide, stacking and many more. The two functions that are visible to the user, PUTCHAR and GETCHAR are described below.

6.2.1 Runtime Function: PUTCHAR

form:

```
PUTCHAR (c)
        CHAR c ;
```

purpose:

PUTCHAR takes its argument and writes it on the standard output.

arguments:

A single character. If passed an INT or other non-CHAR value, only the least significant byte of the argument is used.

returns:

INT: The value returned will either be positive, indicating proper execution, or negative indicating a standard error code. See section 6.1 for information about standard errors.

discussion:

Currently standard output is the screen and cannot be redirected. If file independent I/O is desired, we recommend that the function PUTC be used.

6.2.2 Runtime Function: GETCHAR

form:

GETCHAR ()

purpose:

GETCHAR returns a character from the standard input.

arguments:

none

returns:

INT: The value returned is normally the character form the standard input. If -1 is returned, an End of File has been requested (via CONTROL-3 on Atari keyboard; see OS/A+ manual for Apple II EOF character, usually CONTROL-Z). Other negative values are standard C/65 error codes.

discussions:

Currently standard input is the keyboard and cannot be redirected. If file independent I/O is desired, we recommend that the function GETC be used.

CAUTION: Due to the peculiarities of screen I/O on the Atari, if you use GETCHAR, do NOT output characters to the screen (e.g., via PUTCHAR) unless the last character recieved from GETCHAR was a RETURN ($9B, 155 decimal).

6.3 I/O LIBRARY FUNCTIONS

You will notice that as the definition of C/65 is being explained in chapters 3, 4 and 5 there has been no mention of any input or output statements (except for some of the examples maybe). The reason for this is that input-output facilities are not part of the C language. As in standard C, the I/O functions ("statements", if you wish) are contained in the I/O library. These functions are designed to provide a standard I/O system for C programs. Also these routines are meant to be portable, in the sense that they will exist in (or be adaptable to) a compatible form on any system where C exists. So, described below are the I/O functions that C/65 supports.

EXCEPTIONS: The functions NOTE, POINT, and XIO, as described in the final subsections hereof, do NOT always have exact counterparts on all C systems, since they are dependent upon the foibles of OS/A+ for their operation.

6.3.1 I/O Function: FOPEN

form:

```
FOPEN (filename, options)
        CHAR *filename;
        CHAR *options;
```

purpose:

FOPEN opens the named file in the specified mode.

arguments:

filename, a character string specifying a standard OS/A+ device or file name.

options, a character string specifying the mode in which the file is to be opened.

returns:

A positive INT (a channel number, usually referred to in subsequent sections as "iochan") is returned upon a successful FOPEN; errors are indicated by the standard error code return.

discussion:

CAUTION: the INTeger "iochan" returned by FOPEN must be retained and used as an argument to subsequent I/O operations. Severe errors and/or strange and wondrous things can occur if the various I/O operations are not passed a channel number obtained via a successful FOPEN or OPEN.

OPTIONS must be a string containing one or more of the following characters:

- R -- read access
- W -- write access
- A -- append mode
- D -- read access to the directory of the specified device.

The semantics of combining modes is tricky and not recommended, but consult your OS/A+ documentation if you want more complete information.

Section 6.3.1 (FOPEN) continued:

There are two files that do not have to be opened: standard input and standard output. They refer to the keyboard and screen, respectively and currently cannot be redirected. The IOCHAN for both of these is Ø.

If closer control over the type of OPEN to be performed is needed, consult Section 6.3.2 for usage of the OPEN function.

6.3.2 I/O Function: OPEN

form:

```
OPEN (filename, mode)
        CHAR *filename ;
        INT   mode ;
```

purpose:

Opens a file with given name for access according to given mode. Allows greater control over mode of opening than FOPEN.

arguments:

filename, a character string specifying a standard OS/A+ device or file name.

mode is an INTeger. The low byte of mode goes into AUX1 and the high byte into AUX2 of the IOCB associated with the OPEN'ed file. (See your OS/A+ reference manual for more details of the IOCB.)

returns:

A positive INT (a channel number, usually referred to in subsequent sections as "iochan") is returned upon a successful OPEN; errors are indicated by the standard error code return.

discussion:

CAUTION: the INTeger "iochan" returned by FOPEN must be retained and used as an argument to subsequent I/O operations. Severe errors and/or strange and wondrous things can occur if the various I/O operations are not passed a channel number obtained via a successful FOPEN or OPEN.

There are two files that do not have to be opened: standard input and standard output. They refer to the keyboard and screen, respectively and currently cannot be redirected. The IOCHAN for both of these is Ø.

MODES Ø, 1, and 2 (read, write, and update) are converted to 4, 8, and 12, for convenience and to provide conformance with standard C.

6.3.3 I/O Function: FGETC

form:

```
FGETC (iochan)
        INT iochan;
```

purpose:

FGETC returns the next byte from the specified I/O channel.

arguments:

iochan MUST be an INTeger channel number of a file opened for read access (or read/write access) obtained as the result of a previously successful call to FOPEN.

returns:

INT: the next byte from the specified channel. A -1 is returned on end of file; other negative values are standard error codes.

discussion:

Note that FGETC returns an INTeger character, NOT a CHAR extended to INT. This implies that successful returned values will always be in the range of Ø to 255 decimal. However, if the character returned is assigned to a CHAR and then used in a signed comparison, a negative value (indicating an error) will result if the character's value is actually 128 to 255, since the CHAR will then be sign extended.

EXAMPLE:

```
CHAR c;

IF  ( (c=fgetc(Ø)) < Ø )
        PUTS ("I/O ERROR") ;
```

In this example, the user will see an apparent I/O error anytime the byte fetched from the file has a value from 128 to 255. A better approach would have been to declare "c" to be INT.

As with all I/O operations, "iochan" may be specified as zero (Ø), indicating input from the standard input (the keyboard).

6.3.4 I/O Functions: GETC

form:

```
GETC (iochan)
      INT iochan;
```

purpose:

GETC is exactly the same as FGETC. The second entry name is for consistency and convenience only.

arguments:

iochan is an INTeger channel number of a file previously opened for read access (or read/write access).

returns:

INT: Same as FGETC

discussion:

see FGETC for cautions and hints

6.3.5 I/O Functions: FPUTC

form:

```
FPUTC (c, iochan)
        CHAR c ;
        INT  iochan ;
```

purpose:

FPUTC writes the CHARacter c to the specified channel, iochan.

arguments:

c is a single character. If passed an INT or other non-CHAR value, only the least significant byte of the argument is used.

iochan is an INTeger channel number of a file previously opened for write access (or read/write access).

returns:

INT: The value returned will either be positive, indicating proper execution, or negative indicating a standard error code. See section 6.1 for information about standard errors.

discussion:

As with all I/O operations, "iochan" may be specified as zero (Ø), indicating input from standard input (the keyboard).

6.3.6 I/O Fuctions: PUTC

form:

```
PUTC (c, iochan)
     CHAR c;
     INT iochan:
```

purpose:

PUTC is exactly the same as FPUTC. The second entry name is for consistency and convenience only.

arguments:

c is a single character. If passed an INT or other non-CHAR value, only the least significant byte of the argument is used.

iochan is an INTeger channel number of a file previously opened for write access (or read/write access).

returns:

INT: Return codes are exactly the same as FPUTC

discussion:

See FPUTC for discussion.

6.3.7 I/O Function: READ

form:

```
READ (iochan, buffer, count)
        INT    iochan;
        CHAR  *buffer;
        INT    count ;
```

purpose:

READ reads a binary record of up to COUNT characters from the file specified by IOCHAN into BUFFER.

arguments:

iochan is an INTeger channel number of a file previously opened for read access (or read/write access).

buffer is a pointer to an array of characters. The array must have been declared large enough (at least of size count) to contain the requested record.

count is an INTeger which specifies the size of the record to be read.

returns:

INT: The return value will either be the number of characters read, a zero indicating an end-of-file occured, or a negative number indicating an error. See section 6.1 for details on standard error codes.

discussion:

Under OS/A+ version 2, READ will always return count unless an end of file was encountered while trying to read the specified record, in which case the actual number of characters read is returned. If this "short count" is non-zero, then the next and all subsequent reads will return zero.

Under OS/A+ version 4, the above rules apply except that, if the program reads a record in a random access file which has a "hole" in it, it is possible that a short read will result. The next read will then result in either zero bytes read or an error code. However, if the file pointer is moved past the hole (via POINT), further reads might be successful.

6.3.8 I/O Function: WRITE

form:

```
WRITE (IOCHAN, BUFFER, COUNT)
        INT    iochan ;
        CHAR * buffer;
        INT    count  ;
```

purpose:

WRITE writes a binary record of length COUNT from BUFFER to the file specified by IOCHAN.

arguments:

iochan is an INTeger channel number of a file previously opened for write access (or read/write access).

buffer is a pointer to an array of characters. The array should have been declared large enough (at least of size count) to contain the requested record.

count is an INTeger which specifies the size of the record to be written.

returns:

INT: The return value will either be the number of characters transfered or a negative number indicating an error occured. See section 6.1 for details on standard error codes.

6.3.8 (continued)

discussion:

Generally, the returned INTeger will always be equal to count unless some fatal error (e.g., disk write protected or disk full) occurred.

Since C is "stupid" about "buffers", the user might consider setting up some record I/O like this:

EXAMPLE:

```
CHAR name[25] ;
CHAR address[25] ;
CHAR city[15] ;
CHAR state[2] ;
CHAR zipcode[5] ;
#DEFINE record name
#DEFINE recordsize 72
...
MAIN ( )
$(
...
    WRITE ( iochan,record,recordsize );
```

CAUTION: This trick only works if the character arrays defining the record are globals. Order of allocation of local (auto) variables on the system stack is not so neatly predictable.

6.3.9 I/O Function: FGETS

form:

```
FGETS (buffer, count, iochan)
        CHAR  *buffer ;
        INT    count  ;
        INT    iochan ;
```

purpose:

FGETS reads up to count characters from iochan into buffer. Input is terminated early if a carriage return is encountered. A zero is appended after the last character read.

arguments:

buffer is a pointer to a character array which will contain the characters read. Because of the appended zero byte, buffer should be declared as containing at least count+1 bytes. It is the user's responsibility to ensure this, as no checking of this is done.

count is an INTeger which specifies the maximum number of bytes (characters) to be read into buffer. The read will terminate upon reaching a carriage return character or upon reading count bytes, whichever occurs first.

iochan is an INTeger channel number of a file previously opened for read access (or read/write access).

returns:

INT: The value will either be the number of characters gotten or a negative value indicating an error. See section 6.1 for more information on standard errors.

discussion:

The same discussion noted for the READ function, section 6.3.7, re end of file and/or "holes" in files applies here as well.

6.3.10 I/O Fuctions: GETS

form:

```
GETS (BUFFER)
      char * buffer;
```

purpose:

GETS reads the standard input until a carriage return is seen, putting the characters in BUFFER. The carriage return is overwritten by a zero.

arguments:

buffer is a pointer to a character array which will contain the characters read. Because of the appended zero byte, buffer should be declared as containing at least count+1 bytes. It is the user's responsibility to ensure this, as no checking of this is done.

count is an INTeger which specifies the maximum number of bytes (characters) to be read into buffer. The read will terminate upon reaching a carriage return character or upon reading count bytes, whichever occurs first.

returns:

INT: The value will either be the number of characters gotten or a negative value indicating an error. See section 6.1 for more information on standard errors.

discussion:

See discussion of FGETS, section 6.3.9. GETS is equivalent to FGETS, excepting that channel 0 is assumed and the terminating RETURN code is handled differently.

6.3.11 I/O Function: FPUTS

form:

```
FPUTS (buffer, iochan)
        CHAR * buffer;
        INT  iochan;
```

purpose:

FPUTS writes the null-terminated buffer on the indicated IOCHAN. No newline is appended.

arguments:

buffer is a pointer to an array of characters. The array should have been declared large enough to contain the character string which is to be written.

iochan is an INTeger channel number of a file previously opened for write access (or read/write access).

returns:

INT: The value returned will either be positive (number of characters written) or negative, indicating an error. See section 6.1 for details on standard error codes.

discussion:

FPUTS is designed to be used with line oriented and character string oriented output, since the record (or line) to be written to the file is nul terminated, just as is a normal C character string.

Remember, the nul byte is not written, and a Return character is not appended. If a Return character is desired in a literal string, use the standard escape convention, thusly:

EXAMPLE:

```
FPUTS("\0x0CPrinter Page Heading\n",pr);
```

The 0x0C is a standard ASCII form feed character. The \n is a newline character, specifying the appropriate byte value for the machine on which it is used.

6.3.12 I/O Function: PUTS

form:

```
PUTS (buffer)
        CHAR * buffer ;
```

purpose:

Writes the null-terminated string BUFFER on the standard output. A newline IS appended.

arguments:

buffer is a pointer to an array of characters. The array should have been declared large enough to contain the nul terminated string.

returns: INT: The value returned will either be positive, indicating proper execution,or negative indicating a standard error.

discussion:

PUTS is designed to be used with line oriented and character string oriented output, since the record (or line) to be written to the file is nul terminated, just as is a normal C character string.

CAUTION: note the difference between FPUTS and PUTS! PUTS does indeed automatically append a Return character to the output line while FPUTS does not. Should you need to output a line to the screen (standard output) without the appended Return, simply use FPUTS(buffer, Ø), since channel Ø is always standard output.

6.3.13 I/O Function: FERROR

form:

```
FERROR (iochan)
        INT iochan;
```

purpose:

FERROR returns the last return code generated by the operating system for the specified I/O channel.

arguments:

iochan MUST be an INTeger channel number of an accessible filed obtained as the result of a previously successful call to FOPEN (or OPEN).

returns:

INT: Always returns an error code, as specified in section 6.1; but the code returned may also be 1, meaning no errors.

discussion:

The main purpose for FERROR is that it allows the user to "trap" certain errors that may require some sort of special attention. Such as:

```
IF (FERROR(channel) == Dfull)
          PUTS("\nDisk Full");
```

6.3.14 I/O Function: FEOF

form:

```
FEOF (iochan)
        INT iochan;
```

purpose:

FEOF returns non-zero (TRUE) if end of file has been reached on the specified I/O channel.

arguments:

iochan MUST be an INTeger channel number of an accessible filed obtained as the result of a previously successful call to FOPEN (or OPEN).

returns: INT: If the value returned is non zero, then an end-of-file has been reached on the specified channel; otherwise it has not.

discussions:

The advantage of FEOF is that it allows the user to control the reading of a file with only one statement, such as in:

```
WHILE (FEOF(1) == Ø)
$(
    ...
$)
```

6.3.15 I/O Fuction: FCLOSE

form:

```
FCLOSE (iochan)
       INT iochan;
```

purpose:

FCLOSE closes the specified channel.

arguments:

iochan MUST be an INTeger channel number of an accessible filed obtained as the result of a previously successful call to FOPEN (or OPEN).

returns:

INT: The value returned will either be positive indicating proper execution or negative indicating an error. See section 6.1 for details on error codes.

discussion:

The function FCLOSE is the reverse of FOPEN; it breaks the connection between the file descripter and the external name that was established by FOPEN.

NOTE: When control is returned to OS/A+ all open files are closed automatically.

6.3.16 I/O Fuction: CLOSE

form:

```
CLOSE (IOCHAN)
        int iochan;
```

purpose:

CLOSE is identical to FCLOSE.

arguments:

See description of FCLOSE.

returns:

INT: Return value same as FCLOSE

discussion:

Same as FCLOSE.

6.3.17 I/O Fuction: EXIT

form:

```
EXIT (error)
        INT error;
```

purpose:

EXIT returns control to the operating system.

arguments:

error is an INTeger value, intended to designate the degree of failure (or success) of the C/65 program.

returns: INT: The value returned is ignored by the operating system at the present time.

discussion:

It is expected that if a returned error code system is implemented in OS/A+, it shall be a one byte error code and the following convention will be used:

Ø,1	Normal error free return
2-127	Warnings...non-fatal errors
128-255	Fatal errors

6.3.18 I/O Functions: NOTE and POINT

forms:

```
NOTE (iochan, type)
        INT iochan ;
        INT type   ;
POINT (iochan, pointerØ, pointer1)
        INT iochan   ;
        INT pointerØ ;
        INT pointer1 ;
```

purpose:

Used for random access to disk files. NOTE reports the current position in an opened file. POINT changes the current position in an opened file.

arguments:

iochan MUST be the INTeger channel number of an accessible file obtained as the result of a previously successful call to FOPEN (or OPEN).

type (NOTE only) is a flag which determines which file position pointer is to be returned.

pointerØ and pointer1 (POINT only) are the sector and byte (or page number and byte, see below) of the to-be-made-current position in the file.

returns:

NOTE returns INT: either pointer value Ø (sector or page number) or pointer value 1 (byte number) of the current position within the open file.

POINT returns INT: a standard error code.

discussion:

NOTE and POINT are grouped together here because, in Version 2 of OS/A+ (and, naturally, Atari DOS 2.Øs), they are a tightly linked pair used in building and using random access files.

(Section 6.3.18 continued)

Specifically, since true random access files are not supported by Version 2 OS/A+, one must build a sequential file (opened for write) and NOTE the disk sector and byte numbers at the beginning of each record (perhaps saving the NOTEd numbers in yet another sequential file). Then, when one wishes to read or update a record in that same file, one MUST use a set of the NOTEd values to POINT to an absolute sector and byte number on the disk.

Under Version 4 of OS/A+ (the only version available for Apple II users, an optional double density diskette version for Atari owners), proper and true random access is supported. So NOTE becomes a convenience function rather than a necessity, and POINT may be used to seek to any position in any open file (including positions not yet written...caution).

An examination of section 5.4.3 of the OS/A+ manual will show that NOTE returns an integer (in AUX3 and AUX4 of the IOCB) and a byte (in AUX5 of the IOCB). The "type" parameter to the C/65 NOTE function determines which will be returned: if type is zero, the sector number (page number under version 4) will be returned (and is known as pointerØ when used with POINT); if type is non-zero, the byte number within the current sector (page) will be returned (and is known as pointer1 when used with POINT).

EXAMPLE:

```
sector = NOTE ( file, Ø ) ;
byte   = NOTE ( file, 1 ) ;
...
/* miscellaneous operations...
   presumably including file I/O
   on channel 'File'  */
...
minusiferror = POINT ( sector, byte ) ;
/* the file pointer is repositioned
   to the same place it was when the
   NOTE function calls were made */
```

(Section 6.3.18 continued)

FINAL NOTE for Version 4 of OS/A+ ONLY:

POINT may be used in an approximation of the standard (Unix-oriented) C function "lseek", which usually has the form:

```
lseek( iochan, byteposition )
        int iochan; long byteposition ;
```

Unfortunately, C/65 doesn't (yet?) support the type "long" (traditionally a 32 bit integer), so a similar function would allow random file positioning only within the first 64K bytes of a file. Thus we borrowed a chapter from pre-version 7 Unix and provided POINT, which may be thought of as

```
POINT( iochan, pageposition, byteinpage )
        int iochan, pageposition, byteinpage ;
```

Remember, the "pages" are always 256 bytes long, regardless of the sector or block size in use with version 4 OS/A+. Therefor, if you need to port a C/65 program to a system supporting the "lseek" function, you could easily rewrite POINT as follows:

```
POINT( io, page, byte ) int io,page,byte ;
    { return lseek( io, (page<<8)+byte ), Ø ; }
```

Or, if the new system's C compiler supports #define macros with parameters, one could simply code

```
#define POINT(i,p,b)  lseek( i, p*256+b, Ø )
```

For more information on these possibilities and others, we recommend a thorough study of chapters 7 and 8 of "The C Programming Language".

6.3.19 I/O Function: XIO

form:

```
XIO ( command, iochan, aux1, aux2, filename )
        INT   command ;
        INT   iochan  ;
        INT   aux1    ;
        INT   aux2    ;
        CHAR *filename;
```

purpose:

XIO provides a maximum level of access to the various file manager functions of OS/A+.

arguments:

command is the equivalent of the OS/A+ COMMAND byte (ICCOM in the IOCB).

iochan must be an INTeger channel number. Depending on the XIO function desired, the channel may or may not be one associated with an OPENed file.

aux1 and aux2 are the equivalent of the ICAUX1 and ICAUX2 bytes of the OS/A+ IOCB.

filename is a character string specifying a standard OS/A+ device or file name. Generally, if "iochan" refers to a previously opened file, filename will be ignored. If "iochan" refers to an available (CLOSEd) channel, then filename will be significant.

returns:

INT: a standard error code

discussion:

This function is a generally non-transportable system call designed to provide properly compatible access to OS/A+. Those of you familiar with Atari BASIC and/or BASIC A+ will recognize XIO as a direct translation of BASIC's XIO statement.

(Section 6.3.19 continued)

Rather than give a complete list of all the possible uses of XIO here, we will refer you to Chapter 5 of the OS/A+ manual. The C/65 XIO function can perform all the system commands listed therein other than NOTE, POINT, and the various data transfer operations--all of which are available via other C/65 standard functions previously described in this chapter.

XIO can even be used to open a file on a specific channel, rather than letting C/65 choose the channel for you:

```
EXAMPLE:
minusiferror = XIO( 3,7,6,Ø,"D:*.*") ;
     /* will perform an open (command 3) on
        channel 7 for directory read (aux1=6)
        of all files ("*.*") on drive 1 ("D:")*/
```

And, of course, XIO can be used for such functions as renaming, erasing, protecting, and unprotecting files, as well as much more. As a final example, we show here the implementation of an ERASE (file from disk directory) function:

```
ERASE ( file )
        CHAR *file   ;
$(
RETURN  XIO( 33,7,Ø,Ø,file ) ;
$)
```

CAUTION: This example assumes that channel 7 is available for use by the XIO function. Generally, since the C/65 FOPEN and OPEN functions allocate channels in increasing order starting from channel 1, channel 7 will be the last one used. Still, if you wanted to write a truly safe function, you should perhaps examine the ICHID field of channel 7's IOCB (and, again, see your OS/A+ manual for the specific location of the field and the IOCBs). Of course, you can avoid the problem by also using XIO to perform your file opens to specific channels, but this will make your program less portable to other C systems.

6.4 GRAPHICS LIBRARY FUNCTIONS

The graphics library of C/65 gives you limited access to some of the graphics features of the Atari and Apple microcomputer. These functions are not supported by standard C and they probably will make your C programs non portable. They do however make life a little easier when trying to use your computer's graphics.

6.4.1 Graphics Function: GRAPHICS

form:

```
GRAPHICS(mode)
   INT mode;
```

purpose:

The GRAPHICS function allows the user to set his/her system to a particular mode, such as mode 7 for high resolution, four color graphics.

arguments:

mode is an INTeger value, the legal values for mode are Ø-11 and 17-24. Remember that not all of these values are legal on the Apple II.

returns:

The value returned is the standard error code, see section 6.1 for details on error codes.

discussion:

The modes selected are simply those modes available via the systems graphics driver. C/65 knows nothing about GRAPHICS per se but instead performs an operating system call to execute the requested function.

6.4.2 Graphics function: SETCOLOR

form:

```
SETCOLOR(reg,hue,lum)
   INT reg,hue,lum;
```

purpose:

The SETCOLOR function allows the user to access the color registers of the Atari microcomputer.

users:

ATARI ONLY

arguments:

reg, hue and lum are all INTeger values. The legal limits for these arguments are:

reg	Ø-4
hue	Ø-15
lum	Ø-14 - even numbers only!

returns:

undefined

discussion:

The values for SETCOLOR's arguments are the same as if you were programming in BASIC. For a description of what each value does refer to your Atari BASIC or BASIC A+ manual.

6.4.3 Graphics Function: COLOR

form:

```
COLOR(c)
   INT c;
```

purpose:

To select a particular color for plotting points on the screen.

arguments:

The argument c is an INTeger value. The legal limits for the argument c are normally Ø-3 but other values can be used depending on what graphics mode is in use.

returns:

The value COLOR returns is undefined.

discussion:

The COLOR function can be executed many times in a program, the result being that points plotted will be the color selected by the last COLOR function to execute.

6.4.4 Graphics function: PLOT

form:

```
PLOT(x,y)
  INT x,y;
```

purpose:

PLOT allows the user to plot a point anywhere on the screen.

arguments:

x and y are INTeger values. The value for each depends on the particular graphics mode you are in and represent the requested horizontal and vertical position of the point to be plotted. Consult your operating system, technical, or BASIC manual to be sure you are using legal values for the graphics mode you've selected.

returns:

PLOT returns the standard error code. Refer to section 6.1 for details on error codes.

discussion:

The PLOT function works the same way that BASIC's does, with the x value corresponding to the horizonal axis and the y value corresponding to the vertical axis.

6.4.5 Graphics Function: DRAWTO

form:

```
DRAWTO(x,y)
   INT x,y;
```

purpose:

DRAWTO will draw a line from the last point plotted to the point x,y.

arguments:

x and y are INTeger values representing the horizontal and vertical position of the end point of a line to be drawn. Legal values for x and y depend on the graphics mode selected.

returns:

DRAWTO returns the standard error codes. See section 6.1 for details on error codes.

discussion:

DRAWTO causes a line to be drawn from the last point PLOTted to the specified x,y coordinate. Again, we suggest you consult the appropriate operating system, technical, or BASIC manual for details and legal values for x and y.

6.4.6 Graphics Function: POSITION

form:

```
POSITION(x,y)
  INT x,y;
```

purpose:

Positions the horizonal and vertical pointer to the x y value selected.

arguments:

x and y are INTeger values. Their limits depend on the particular graphics mode selected.

returns:

The value POSTION returns is undefined.

discussion:

Although the POSITION function can be used in all graphics modes, its best use is in text mode(s) where the cursor will be positioned at the point x,y.

6.5 STORAGE ALLOCATOR LIBRARY FUNCTIONS

The storage allocator functions provide a way of obtaining and releasing variable-sized blocks of memory. Freed blocks are coalesced if possible. The memory allocated is obtained from the "free memory above the end of your C/65 program and below HIMEM. The user should refrain from calling operating system routines that change the value of HIMEM after the storage allocator (ALLOC) has been called the first time.

6.5.1 Allocation Function: ALLOC

form:

```
ALLOC  (SIZE)
        INT size;
```

purpose:

ALLOC returns a pointer to an area of memory SIZE bytes long--if such an area is available.

arguments:

size is an INTeger value and represents the area in bytes that you want to allocate.

returns:

CHAR: The value returned will be a pointer to the area of memory size bytes long. If a zero is returned then there was no large enough available block of memory.

discussion:

The ONLY area the can be allocated by ALLOC is the memory space between the end of your C program and himem.

---this page intentionally left blank---

6.5.2 Allocation function: FREE

form:

FREE (STORAGE) char *storage;

purpose:

FREE returns previously allocated memory to the available pool.

arguments:

storage is a pointer to the block of memory to be freed.

returns:

Undefined.

discussion:

The FREE function requires special attention by the user. If the pointer passed to FREE is not the result of a successful call to ALLOC, the consequences could be disasterous.

CHAPTER 7: Interfacing to Assembly Language

Although programs written in C/65 can run up to 10 times faster than BASIC programs, sometimes it is desirable to use Assembly Language routines for even greater speed and compactness. For example, the I/O library provided with C/65 is written entirely in MAC/65 Macro Assembly Language.

Since there is (currently) no linking loader available for OSS and MAC/65, the easiest way to use assembly code is via the #ASM directive. This directive simply causes a .INCLUDE directive to be placed in the assembly language output file generated by C/65. (See MAC/65 manual for a full description of the .INCLUDE directive, but the form is generally .INCLUDE #<filespec>.)

Typically, #ASM directives are placed outside of C functions to define entire functions, but they could also be used inside of C functions for optimization. At this writing, we have made little, if any, use of this latter capability.

A little theory about how C/65 generates code may help.

7.1 C/65 Zero Page and System Stack Usage

First, C/65 defines several locations in zero page. A 16 bit primary register refered to as RL is where C/65 does most of its work. The high byte of this register may be referred to as RH in addition to RL+1. A 16 bit secondary register called RE (high byte known as RD) is also heavily used.

Binary operators have their operands placed in RL and RE before the operation is executed, the result going back into RL.

There is also a 16 bit tertiary register used for internal operations called RC, whose high byte may be referenced by the name RB, and there is an 8 bit temporary register called RA that is only used for temporary storage.

These registers are used by a series of routines that the compiler calls directly (over and over again). Collectively these routines are known as the "runtime library".

There is one other 16 bit register used by the runtime library, a stack pointer known as RSPL(whose high byte can be addressed as RSPH if necessary). The C system stack (not to be confused with the 6502 stack located from $100 to $1FF) is initialized to what is assumed to be the bottom of the user program upon program execution and grows DOWNWARD. By default, a C program's base address is $4000, with the system stack residing between the contents of LOMEM and $3FFF (CAUTION: C makes no check for a "crash" of the system stack with LOMEM).

Note that although the standard stack operations, "push" and "pop" must be done with more than one instruction, at least the C/65 stack can be more than 256 bytes deep and reside anywhere in memory.

SPECIAL NOTE: The inital value of the stack pointer may be changed by editing the runtime library source code. (Change the equate of the string "STARTSTACK"). The executable code file grows up from the intial stack pointer value.

7.2 Accessing Function Parameters

Parameters are passed to called functions, whether written in C or assembly language, via C/65's system stack. The rule for placing parameters on the stack is: Decrement first, and then store, for each parameter in the order which they are defined.

Parameters are stored in the standard low byte/high byte format (i.e, the high byte of a 16 bit parameter is stored in the higher address). All parameters, even character parameters, are sign-extended, and arrays and strings are passed as pointers to the actual data.

For example, suppose that the stack pointer's value is $3500, and that there is a function named "FOOBAR" which expects 3 parameters: an integer, a character, and a string (or, more properly, a "pointer to character"). Then, assume a call of the form:

```
FOOBAR (3, 'c', "abc")
```

If we assume that the compiler has allocated space for the string "abc" starting at location $5000, then upon entry to FOOBAR, the stack looks like this:

```
         ---------
  $34FF  |  $00
         ---------
  $34FE  |  $03          /* the constant 3 */
         ---------
  $34FD  |  $00
         ---------
  $34FC  |  $63          /* hex equivalent of 'c' */
         ---------
  $34FB  |  $50
         ---------
  $34FA  |  $00       <-------RSPL points here
         ---------
```

Now, let us assume that FOOBAR is an assembly language routine which we are writing. Let us further assume that we want access to the third parameter, the character pointer (or string address, or ...). A function compiled by C/65 will use code similar to the following:

```
        LDY #0
        LDA (RSPL),Y
        STA RL
        INY
        LDA (RSPL),Y
        STA RH
```

And that code loads the address of "abc" in the primary register. Of course, an assembly language routine might wish to place the parameter it has retrieved somewhere else, but the principal is the same. The second parameter to the function is accessed in the same way by simply replacing the "LDY #0" with "LDY #2". And, of course the first parameter is accessed via "LDY #4". Remember: the receiving function sees the parameters on the stack in reverse order compared to the way they are written in the function call.

CAUTION: The compiler allocates space for local variables on the stack BELOW the system stack pointer. Thus the above code will not work INSIDE of a compiled routine unless it is placed directly after the function's opening left brace and before any local declarations. (Of course, if the function defines no local variables, the code given might be valid.)

7.3 Passing Values Via Global Variables

Any variable declared at the "global" level in a C/65 program is known, by its label, to the assembler. Therefore, any assembly language program called by a C/65 program may refer to these variables by name.

Also, if the C/65 declares a variable to be "EXTERN", that variable may be defined in the assembly language routine (so that it is, indeed, EXTERN to the C/65 module).

Remember, INT variables are equivalent to ".WORD" assembly language locations, with the LSB before the MSB.

7.4 Returning Values to the C Expression

Any C/65 function (and that includes function subroutines written in assembly language) may pass back one and only one value to the function which called it. In the current version of C/65, the returned value is always take to be an INTeger. If some other usage is desired, it is the caller's and callee's responsibility to coordinate the meaning of the returned value.

To return a value to a caller, simply place the 16 bit return value in location RL (which is the LSB, RH is the MSB). Care should be taken to zero or sign extend the MSB if a one byte value is being returned.

7.5 A Simple Example

The following example shows a C/65 program and a C-callable assembly language routine which demonstrate nearly all of the points made in sections 7.1 through 7.4. The C/65 function MAIN() uses both entry and EXTERN global variables and local variables and expects the assembly language routine to return a proper value.

The assembly language, NEWROUTINE, adds what is in a global location named GORP to a passed parameter and returns the result. This routine illustrates three principles: (i) passing values in global locations, (ii) passing values via the C system stack, and (iii) returning values to the caller via the C expression evaluation mechanism. In addition, though not part of the code of the routine per se, the assembly code defines a variable (an initialized array, no less) to be referenced by the C routine.

Also, please note that since the C program defines the global GORP, the assembly language routine need not do so. And, contrariwise, since the assembly language code defines the variable SQRTABLE, the C program needs only make an EXTERN reference to it.

(The program example follows on the next page.)

The C Calling Program:

```
INT GORP; /* a global...defined here */
EXTERN INT SQRTABLE[] ; /* an externally defined
                           array of integers */


MAIN () $(
    INT RES ;              /* a local variable */
    GORP = 5;
    RES = NEWROUTINE (7); /* RES should be 12 */
    RES = SQRTABLE[ RES ] ; /* and now RES =
                               the square of
                               itself...144 */
$)
```

The MAC/65 Assembly Language Routine:

```
NEWROUTINE
    LDY #Ø          ;to fetch parameter
    CLC             ;get ready to add
    LDA (RSPL),Y    ;fetch low byte of parameter
    ADC GORP        ;add to low byte of global
    STA RL          ;store low byte of result
    INY             ;point at high byte of parameter
    LDA (RSPL),Y    ;fetch it
    ADC GORP+1      ;add to high byte of global
    STA RH          ;store high byte of result
    RTS             ;return to C/65

SQRTABLE = *        ;a table of squares
    .WORD  Ø*Ø, 1*1, 2*2, 3*3
    .WORD  4*4, 5*5, 6*6, 7*7
    .WORD  8*8, 9*9, 1Ø*1Ø, 11*11
    .WORD  12*12, 13*13, 14*14, 15*15
; note how we let the assembler do the work
; for us ... and it's faster than letting C/65
; do the work at runtime
```