C
Programmer's Implementation Reference Manual

188-370-302 A

March 1986

• Software •
Publications

WICATsystems

first printing March 1986

Trademarks Used in this Publication

Information about this Manual

Review the following items before you read this publication.

The subject of this manual

This manual is designed to give information required to efficiently use WICAT's C compiler.

The audience for whom this publication was written

This manual is written for the advanced programmer using WICAT's C compiler. The information herein presupposes a general knowledge of C, and experience in programming.

An introductory C text, The C Primer, and a general overview of C, C: A Reference Manual, are available from WICAT Systems, Inc. This manual presupposses familiarity with the information in those books. This publication is not tied to a specific C release.

Related publications

The C Primer, part number 188-370 101 A, is an excellent introduction to C for the user who has little or no experience with C. This manual presupposes a knowledge of the information in The C Primer, or equivalent experience with C.

C: A Reference Manual, part number 188-370-301 A, is an overview of C. It is for those with some knowledge of C. The information in it is not specific to a particular C implementation.

Typographical Conventions Used in this Publication

Bold facing indicates what you should type.

Square brackets, [], indicate a function key, the name of which appears in uppercase within the brackets. For example, [RETRN], [CTRL], etc. Braces, {}, indicate a key in the keypad.

Underlining is used for emphasis.

Table of Contents

Table of Contents

Chapter 1

Introduction

This manual is intended to help the user of WICAT's C compiler. It should be used in conjunction with The C Primer, part number 188-370-101 A, and C: A Reference Manual, part number 188-370-301 A.

The information contained in this manual covers the implementation of WICAT's C under WMCS and under UniPlus+ System V. Where appropriate, topics are discussed in terms of both WMCS and UniPlus+ System V.

However, because information on C under UniPlus+ System V is available in the UniPlus+ System V documentation, this manual emphasizes C under WMCS.

For example, chapter 2 covers the compilation process under WMCS and UniPlus+ System V.

Chapter 3 gives hints for successful implementation of WICAT's C compiler for WMCS and UniPlus+ System V.

Chapter 4 is a brief introduction to the C and math libraries.

Chapter 5 describes WICAT's implementation of floating-point arithmetic in relation to C.

Chapter 6 gives information about optimization under WICAT's C compiler.

Chapter 7 is a dictionary of C libraries not available as part of the WMCS documentation. Because these libraries are documented in the UniPlus+ System V manuals, they are provided here specifically for the user of C under WMCS.

Chapter 8 contains the four WMCS C compilation commands. They are formatted in the WMCS command-description style. If you are using C under WMCS, read these command descriptions.

If you are using C under UniPlus+ System V, see the cc command description in the UniPlus+ System V User's Reference Manual (Section 1).

Chapter 9 contains information to help the user interface C with other languages.

Chapter 10 offers guidelines for debugging when using the C compiler.

This manual is written for programmers familiar with C. It presupposes an acquaintance with the other two books that are part of the WICAT C documentation set, and experience with C.

Chapter 2

The Compilation Process

Under UniPlus+ System V, C files are compiled with cc. Documentation for cc is in the <u>UniPlus+ System V User's Reference Manual (Section 1).</u>

Under WMCS, C files are compiled with compile. The compile command description is in chapter 8 of this manual.

C Compiler Passes

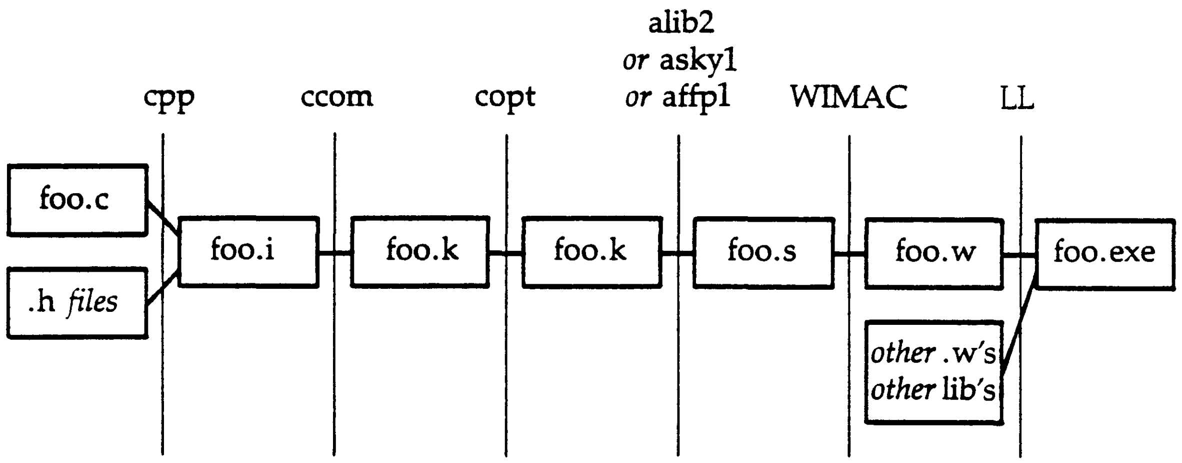
Compile under WMCS, and cc under UniPlus+ System V, are the interface to the Software Generation System (SGS). The SGS takes command line options and C source files, and invokes the passes required to translate a C source program to an executable object module.

Files with a .c extension are assumed to contain C source and follow the execution path shown. When more than one source file is specified on the command line, each file is run through the execution sequence separately (up to the LINKER pass). At this point, the LINKER combines the created object files into a single executable.

The following diagram shows the compile process under WMCS and UniPlus+ System V:

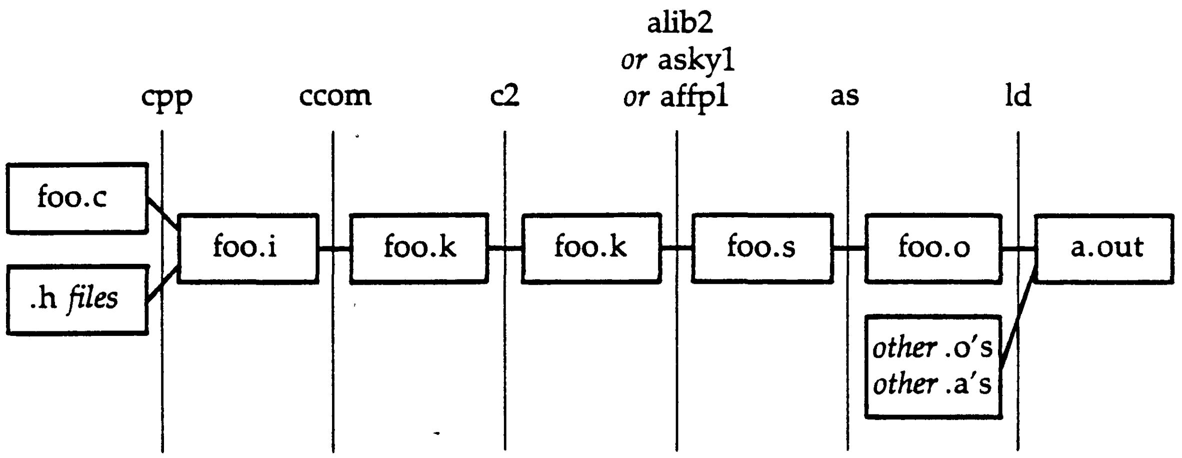
WMCS

filename.c ⟶ *filename.exe*



UniPlus+ System V

filename.c ⟶ *a.out*



.i's, .k's and .s's are temporary files with names derived from the process identification. They are hidden in the WMCS dictionary sys$disk/systmp/ or the UniPlus+ System V directory /usr/tmp.

Limitations on WICAT's C Compiler

The size of a C program is limited by the size of internal data structures in the compiler (which cannot be changed by the user). If the program requirements exceed these limits, the compiler can fail.

Following are limitations of the WICAT C compiler:

Symbol table	1300 symbols
Expression tree	500 nodes
parse stack	150 parameters
switch	500 cases
block nesting	30 levels

A program can also be limited by hardware. Some hardware limitations are apparent at compile time, others are apparent only when the program crashes.

These are the hardware limitations for a C program:

1. The total process size is limited to 2Mbytes (or the size of available memory if 2 Mbytes is not available).

   The process size includes the program's text, data, and run-time stack. Because of this, the following declaration of an array could fail:

   ```
   char bigsucker[100][100][100];
   ```

   A recursive function with a large amount of local storage could also fail because it causes the stack to grow quickly.

2. The register-indirect-with-offset addressing mode is used by the compiler. It is used for local variable and parameter references. It is limited to 16 bits (-32768 to 32767).

   However, this addressing mode cannot be used if the number of local variables, or the number of bytes of parameters, exceeds 32767.

3. Indirection from a NULL pointer can cause subtle problems. Though location 0 is technically part of a program's address space, it is a location in ROM.

Following are ways of misusing a NULL pointer:

Reading: Reading location Ø returns whatever is at the lowest ROM address (a jump instruction). When printed the result is garbage.

For example, the following program prints "cp is Ny."

```
char *cp = Ø;
printf("cp is %s\n", cp);
```

Writing: Although location Ø is not writable, an error is not produced. The write is ignored. Therefore, the following program segment does not produce a core dump, but it does not print "true" either.

```
char *cp = Ø;
*cp = 'x';
if (*cp == 'x')
        printf("true\n");
```

Executing: Location Ø contains a jump instruction to a destination in the system address space (Øx2ØØØØØ or greater). Executing the instruction at Ø causes a jump to system space. However, this is a memory violation. The program crashes with the program counter in the Øx2ØØØØØ+ range.

Chapter 3

WICAT C Implementation

This chapter contains information that will be helpful in the implementation of WICAT's C.

Storage

There are four kinds of storage:

1. Auto

   These are local variables. They are allocated on the stack at run time when a function is called. Auto variables in inner blocks are allocated when the lexically enclosing function is entered, not when the block is entered. However, optional initialization occurs when the block is entered.

   Names appear in the compiler output as offsets from the stack pointer as follows:

   ```
   -<offset>(sp)
   ```

2. External

   These are global variables. The compiler allocates space for these variables and assigns them the names given by the user, preceded by an underscore.

   If initialized, the storage is allocated and the value is assigned at compile time. If not initialized, space is not allocated until load time. Then the initial value is zero.

3. Static

   This is another form of global storage. It is global only to the defined function or file. Storage is allocated at compile time whether the variable is initialized or not. Names are not exported above the local level and are not visible to the user or other SGS utilities.

4. Register

   Register variables are contained in hardware machine registers. They are limited to scalar types.

   There can be up to six data variables (char/short/int/long), four pointer variables (e.g., char *), and four floating-point registers. The number of floating-point registers is hardware dependent.

   Where the user declares more register variables than there are registers, the register modifier is ignored. The mapping between the name declared by the user and the register containing it is often difficult to compute.

Storage Sizes

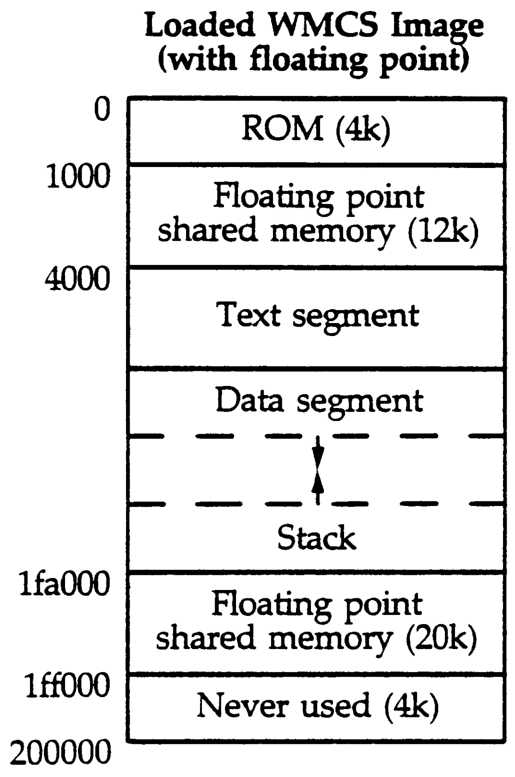
Following is a list of storage sizes for data types under WICAT's C compiler:

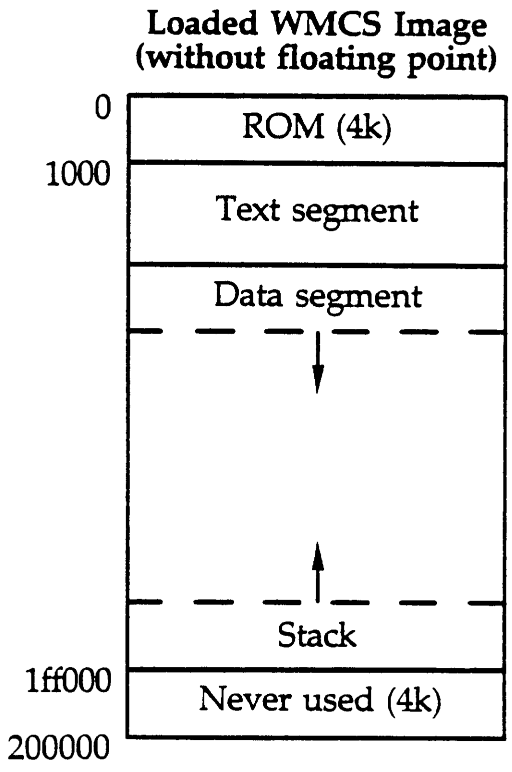
char	8 bits
short	16 bits
int	32 bits
long	32 bits
float	32 bits
double	64 bits

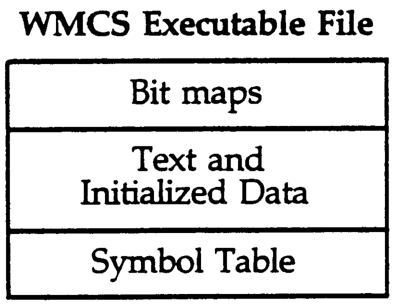
Storage Allocation and Access

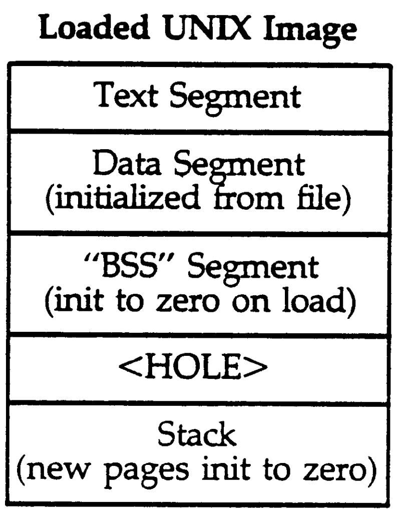
To understand how various classes are allocated and accessed, the format of C object files and images must be understood.

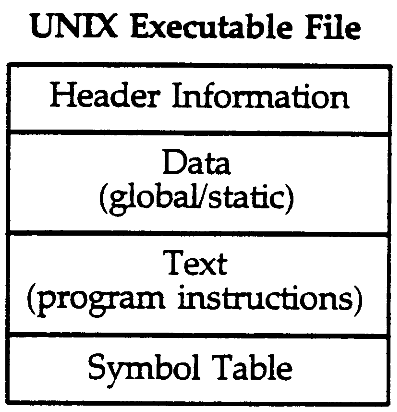
The following diagrams represent the format of executable files and images:











The UNIX executable file format is described in more detail in a.out(4) of the UniPlus+ System V User's Manual Sections (2-6).

Initialized external and static variables are contained in the data segment. Names and locations of these variables are in the symbol table. The symbol table also contains the names of uninitialized external variables. Locations of the variables are added at link time.

Storage for auto and register variables is not allocated until runtime. The names are only in the code that references them.

When loaded, the data segment has been expanded to include storage for the previously uninitialized data variables. This storage has been initialized to zero by the operating system. Auto variables are created on the stack when the lexically enclosing function is entered. Register variables are loaded into machine registers when the lexically enclosing function is entered.

Register Allocation

The C compiler uses the 68000's eight data registers and eight address registers as follows:

a0=1,d0,d1

These registers are used to compute and store temporary values during evaluation of expressions. They are not saved when a function is called, so their value is not guaranteed across functional calls.

a2-a5,d2-d7

These registers are available as user-defined register variables. If they are not user-defined, the compiler can use them as temporary storage. They are saved at function entry and restored at exit so their value is maintained across function calls.

a6

This is the function call stack frame pointer. It is used as a base for locating function parameters and local variables. It also serves as a backward link to previous function call stack frames, i.e. function calls.

a7 (sp)

This is the stack pointer.

Using the Stack

The C compiler builds and maintains a stack frame for each active function.

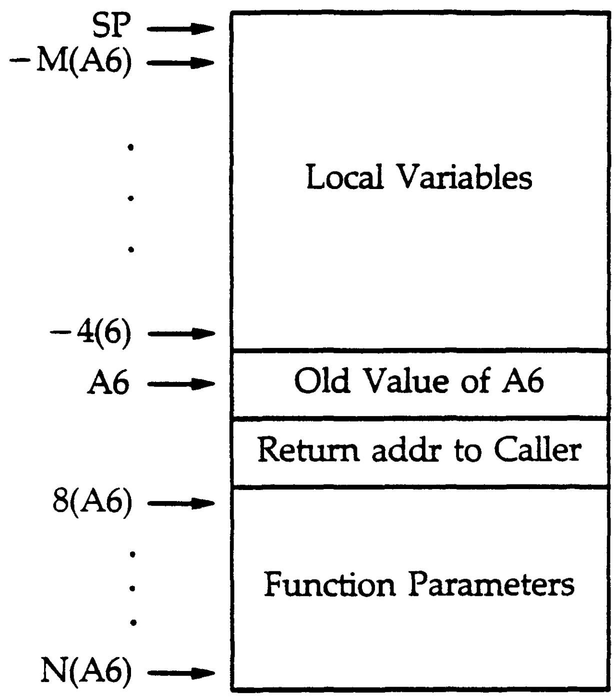
Address register 7 (A7, SP) and address register 6 (A6) have special meaning.

Address register 7 (A7) is the current lowest active address (top) of the stack.

Address register 6 (A6) is a frame pointer that defines a base for function parameters and local variables. It also provides a a pointer to the previous stack frame.

Stack frames are manipulated with LINK and UNLK 68000 instructions.

Following is a diagram of a function call stack frame:



In the foregoing diagram, local variables are accessed at negative offsets from A6, function parameters are at positive offsets, and the previous frame pointer value is at the location pointed to by A6.

Register variables in functions have no associated stack storage except a function parameter.

If a function parameter is declared as register, it has storage in the stack frame allocated by the calling function. Code is generated by the compiler to copy the contents of that location to a register when the function is entered.

The following function is a storage allocation example:

```
foo(p1, p2, p3)                                   /* Locations:          */
 char p1; register int p2; short p3;
{                                                 /* p1:   11(a6)         */
        register char *ap1 = 0;                   /* p2:   12(a6) and d7  */
        char a1[11];                              /* p3:   18(a6)         */
        static int s1 = 1;                        /* ap1:  a5             */
        int a2 = 2;                               /* a1:   -11(a6) to -1(a6)*/
                                                  /* s1:   not on stack   */
                 .                                /* a2:   -16(a6)        */
                 .
                 .
}
```

In the foregoing example:

p2 occupies space on the stack. The compiler generates code to copy from the stack location: 12(a6) to the appropriate register, D7.

apl does not occupy space on the stack because it is a local register variable. The compiler generates code to initialize it.

The calling function pushes all parameters on the stack in reverse order. Therefore, they appear as first parameter (low address) to last parameter (high address). Scalar parameter types shorter than a longword are padded to a longword boundary.

The local array al is arranged from lowest address (a[Ø]) to the highest address (a[1Ø]) even though local variables are allocated from highest to lowest (e.g., a2 occupies a lower address than al).

The static variable sl occupies no stack storage even though it is declared inside the function. It is allocated at compile time in the initialized data section of the object file.

Register variables are allocated starting with d7 and a5 to d2 and a2. Therefore, parameter p2 is in d7 and local variable apl is in a5.

NOTE: The C optimizer removes LINK and UNLK instructions in functions with no local variables. In other words, function parameters are accessed as positive offsets from the stack pointer instead of offsets from the now-unacceptable A6. The C optimizer can also remap register variables into scratch registers (dØ,dl,aØ,al). Read chapter 6 of this manual for more information on the C optimizer.

Signed and Unsigned Scalar Types

The scalar data types char, short, int, and long can be explicitly declared as signed or unsigned. Without a specification, they are signed.

Unsigned modifiers affect unary and binary operators as follows:

Arithmetic operators use unsigned arithmetic (e.g., DIVU rather than divs).

Logical operators use unsigned comparison (e.g., BLS rather than BLE).

The right shift operation zero-fills instead of sign-extends.

Implicit or explicit cast operations zero-fill rather than sign-extend from shorter types to longer types.

WICAT C Features

The WICAT C compiler implements extended operations on structures and unions. Structures and unions can be assigned, passed as parameters, and returned as the result of functions.

In addition, the WICAT C compiler supports enumeration types (enum), the void storage class, unsigned char and unsigned short data types, and the asm assembler escape.

Also, the keyword list for the C compiler conforms to the proposed ANSI standard. The keywords "const" and "volatile" are reserved for implementation of the ANSI standard. Attempts to use these keywords generates a syntax error from the compiler. Entry has been dropped from the keyword list.

A list of keywords for the C compiler appears in appendix c of this manual.

Temporary Files

The C compiler creates a temporary file for internal use. This file is used to collect string constants from throughout the source file so they can be emitted in a single section. When the preprocessor, compiler, optimizer, and floating-point postprocessor are run under cc (UniPlus+ System V) or compile (WMCS), temporary files are used to pass information from pass to pass.

These files are named CCCP<unique ID>. In the foregoing name, unique ID represents a number that is unique to the process executing cc or compile.

Under UniPlus+ System V, temporary files are created in the following standard, system-wide, temporary directory:

```
/usr/tmp
```

Under WMCS these files are created in the following standard, system-wide, temporary directory:

```
sys$disk/systmp
```

All temporary files are deleted when the compile process terminates, or when the compile process is interrupted.

Chapter 4

Optimization

The optimized code produced by the Software Generation System (SGS) is not really optimimum code; a better term is improved code. Even though the C optimizer doesn't really optimize code (it improves it), the term "optimization" will be used throughout this manual since that is the term traditionally used to describe the "improving" process.

Optimization occurs in three parts of the SGS: the compiler, the optimizer, and the assembler. The optimization performed by the compiler and the assembler is done automatically, i.e., it is performed on every program and the user cannot turn it off. However, the optimizer is turned off by default, and you must specify that the optimizer be used if you want those optimizations to be performed.

Compiler (ccom) Optimizations

The C compiler performs optimizations that the optimizer cannot do or that are very hard to do. Primarily, these optimizations require some source context that is not availabe in the generated assembly code (e.g., types of variables). They are performed whether or not the optimizer is turned on.

The input for these optimizations is the internal expression trees representing pieces of the program. The output is the assembly code for the program's statements. (The fundamental unit for compiler optimizations is a C statement.)

The following sections describe optimizations performed by the C compiler.

Constant folding

If the operands of an operator are constants (their value is known at compile time), the operation is performed by the compiler. For example, the following two assigment statements generate the same code:

```
i = 3 * 5;

i = 15;
```

Strength reduction

If one of the operands of a binary operator is a constant, the compiler must generate code to perform the operation. However, it can use an instruction that takes less time.

The following examples show instructions the compiler may use:

multiply by power-of-2	uses	left shift
unsigned divide by power-of-2	uses	right shift
some long-by-long multiples	uses	shifts and adds

Type reductions

Short integer types (i.e., char and short) that occur in expressions should undergo unary/binary conversion (to longer integer types) before the expression is evaluated. These conversions cause shorter types to be promoted to int.

For example, consider the following declarations and statement:

```
int i; short s; char cl, c2;     /*declarations*/
i = s * ( cl - c2 );             /*statement*/
```

The preceding statement should be evaluated as if it were this:

```
i = (int)s * ( (int)cl - (int)c2 );
```

Since the most efficient data size for the 68000 is 16 bits (a short) while the size of an int is 32 bits, code can be produced that is less efficient than it should be. In an effort to produce more efficient code, the compiler uses shorter types whenever doing so does not change the value of the expression.

Therefore, the statement **i = s * (c1 - c2);** would be evaluated as the following:

```
i = (int) ( s * ( (short)c1 - (short)c2 ) );
```

This type of shortening is done for bitwise, logical, and arithmetic operators. While the rules for type shortening are complicated and the shortening may not always be done, the result will not differ from the the result obtained had the usual conversions been applied.

Optimizer (c2/copt) Optimizations

Most optimizations are performed by the optimizer. Processor-independent (common tail merging), processor-dependent (register mapping), and WICAT-specific (suppressing NOPs following stack probes) optimizations are performed by the optimizer.

The input for the optimizer is the assembly-language program produced by the compiler. The output from the optimizer is another assembly-language program that should be functionally equivalent to the input program.

The optimizer uses three fundamental units of optimization:

A module, the largest unit, corresponds to an entire C function.

A module contains one or more basic blocks. Each basic block is a sequence of assembler instructions that start immediately after a label or branch and end with the next branch or label. (The next branch or label refers to the next one listed in the program, not the next one to be executed.)

Within each basic block, a moving window of one to three instructions is used for peephole optimizations.

This discussion of optimizer optimizations is divided into the techniques used to collect data for the optimization and the optimizations themselves.

Data collection techniques

The optimizer uses the following data collection techniques:

Flow analysis The basic blocks of a module are linked internally to form a flow graph. This graph provides information about the flow of execution in a module. One basic block is linked to another if control from one falls into the other, or if the second basic block is the target of a branch from the first.

Live/dead analysis For each instruction in a module, it is determined which registers are referenced and which are set. This information is transmitted throughout the module so that at every instruction it is known which registers contain values that are referenced again after the instruction (live registers) and which are not referenced again (dead registers).

For data registers, the live/dead information is kept for each addressable piece of the register, i.e., byte, word, or long. The live/dead information is not computed for condition codes or memory locations.

Optimization techniques

These are the optimizations the compiler performs:

Unreachable code elimination The flow graph is used to detect basic blocks that cannot be reached by any execution sequence. The optimizer eliminates these blocks.

In the following example, the basic block that consists of the assembly statements for 1 is removed:

```
register int i, j, k;
for (i = 0; < 10; i++) {
        if (i > 5) {
                k = j; continue;
        } else break;
        j = i * 3; k += 4;      /* 1 */
```

NOTE: This is a trivial example that even the compiler would detect.

Dead code elimination The live/dead information is used to determine whether the value computed by each statement is used. If it is not used and the statement has no side effects (such as setting condition codes prior to a test instruction), it is removed. The removal of a statement can produce dead code, so the procedure is repeated until no more dead code is found.

In the following example, the entire loop body will be eliminated because the value of register j is not used so 2 is eliminated, and then value of register k is not used so 1 is eliminated:

```
register int i, j, k;
for (i = 0; i < 1000000; i++) {
        k = (1 << i);            /* 1 */
        j = (i + k) & 0xFF;      /* 2 */
}
```

The loop itself is not eliminated because register i is used each time through the loop (even though the loop now does nothing).

Redundant branch elimination The flow graph is also used to detect and eliminate unconditional branches that are the target of other branches and branches to the following statement. In the following example, for the code on the left, the unconditional branch to .L2 at .L1 is removed and the label .L1 is moved down to the same location as .L2 as shown in the code on the right:

```
        bne .L1                 bne .L1
            .                       .
        <stuff>                 <stuff>
            .                       .
        bra .L4                 bra .L4
.L1:                    .L3:
        bra .L2                     .
.L3:                          <more stuff>
            .                       .
     <more stuff>       .L1:
            .           .L2:
        beq .L2
.L2:
```

The second branch to is also removed since .L2 is the label for the following statement.

Common tail merging Pairs of basic blocks are compared from the end back to the beginning to determine if the tails of the blocks are the same. (Essentially, blocks are the same if they are lexically equivalent.)

If common tails are found, the common portion is broken out to form a new block, and branches to the new block are added at the end of the old ones. For more efficient use of space and time, a common tail must contain a minimum number of statements before it is broken out.

Loop rotation The optimizer attempts to limit the number of branch instructions required in loops because branch instructions use a lot of CPU time. For example, the following simple while loop generates code with a test at the beginning, the body of the loop, and a branch back to the test:

```
while (i != Ø)
        f(i);
```

The test at the beginning includes a branch that skips the body if the condition (i != Ø) evaluates false. Therefore, every pass through the loop executes two branch instructions.

Loop rotation moves the test to the bottom of the loop and reverses the sense of the test so that a branch is taken to the beginning of the loop if the condition evaluates true. Now, only one branch is executed each time through the loop. A second branch before the loop jumps to the test, but it is only executed once.

In the following example, the original loop is shown on the left and the rotated loop is shown on the right:

```
.L1:                                    bra      .L1
        tst.l    _i         .L1:
        beq      .L2                    move.l   _i,-(sp)
        move.l   _i,-(sp)               jsr      _f
        jsr      _f                     addq.l   #4,sp
        addq.l   #4,sp      .L1:
        bra      .L1                    tst.l    _i
.L2:                                    bne      .L2
```

NOTE: The foregoing C code fragment is an example only. The C compiler does not actually generate the code shown on the left since the compiler itself rotates simple loops.

Register remapping Register variable declarations within functions cause an association to be formed between the variable and a hardware register. When the association is formed, the previous contents of the register must be saved so old value can be restored when the function is exited. The optimizer avoids these saves and restores register values by remapping register variables into scratch registers that do not need to be saved.

In the following example, variable i is assigned to hardware register d7 and j is assigned to d6:

```
register int i, j;

i = 1; j = 2;
if (i == j)
        i = j;
return i;
```

This requires code at the beginning to save the old values and code at the end to restore those values. The optimizer detects that the scratch registers dØ and dl are not used, and it remaps the references to d7 and d6 to dl and dØ. This eliminates all saves and restores.

In the following example, compiler-generated code is shown on the left and the optimized code is shown on the right:

```
        movem.l #$CØ,-(sp)                 moveq    #1,dl
        moveq   #1,d7                      moveq    #2,dØ
        moveq   #2,d6                      cmp.l    dØ,dl
        cmp.l   d6,d7                      bne      .L13
        bne     .L13                       move.l   dØ,dl
        move.l  d6,d7               .L13:
.L13:                                      move.l   dl,dØ
        move.l  d7,dØ
        movem.l (sp)+,#$CØ
```

NOTE: Because dØ, dl, aØ, and al are the only scratch registers, only two data register variables and two pointer register variables can be remapped this way.

<u>LINK/UNLK</u> <u>removal</u> If a function doesn't have non-register local variables, the compiler generates a link instruction in the following form:

```
link a6,#-Ø
```

However, this serves only to save the old value of a6, thereby maintaining the function call linkage. The optimizer eliminates all links of this kind, changing references to parameters to offsets from the stack pointer instead of offsets from the now invalid a6.

Consider this C function:

```
f(a,b)
 int a, b;
{
        a = b; b = 2;
        return(a);
}
```

The compiler output from the foregoing example is shown on the left and the optimized code is shown on the right:

```
link    a6,#-Ø                   move.l  -4+12(sp),-4+8(sp)
move.l  12(a6),8(a6)             moveq   #2,dØ
moveq   #2,dØ                    move.l  dØ,-4+12(sp)
move.l  dØ,12(a6)                move.l  -4+8(sp),dØ
move.l  8(a6),dØ                 rts
unlk    a6
```

NOTE: The foregoing causes a couple of serious side effects:

Because the stack linkage is broken, debuggers no longer give a reliable stack backtrace since they rely on tracing the link chain.

Calling a user-supplied assembly language routine that changes the value of the stack pointer can change the optimizer-generated references to parameters. The only way to prevent this from happening is to declare at least one non-register local variable.

NOP suppression Due to an anomaly in the WICAT memory-management unit, it is necessary to generate a NOP instruction following a stack probe if the next instruction would modify the stack. Normally, this NOP is generated as part of the function-entry sequence triggered by the .entry pseudo-operation. Whenever the optimizer expands this pseudo-op, it will generate a NOP only when it is needed. Unfortunately, if the optimizer doesn't expand it (i.e., the function uses floating-point) the appropriate floating-point preprocessor will always generate a NOP. Hence, this optimization benefits only functions that do not use floating point.

Elimination of stack pops Parameters to functions are passed by pushing them onto the stack (i.e., decrementing the stack pointer and storing the parameter at the location it now points to). After returning from the function call, the parameters are popped (i.e., stack pointer incremented). In many cases, one function call immediately follows another.

In these cases, the first function's parameters are popped (stack pointer incremented) and then the second is pushed (stack pointer decremented). Here the optimizer will save time by not bothering to pop the first function's final parameter and, instead, overwriting it with the first parameter to the second function.

For example, consider the following segment of C code:

```
f(a);
g(b);
```

For the foregoing example, the compiler would generate the code on the left and the optimizer would generate the code on the left:

```
move.l   _a,-(sp)          move.l   _a,-(sp)
jsr      _f                jsr      _f
addq.l   #4,sp             move.l   _b,(sp)
move.l   _b,-(sp)          jsr      _g
jsr      _g                addq.l   #4,sp
addq.l   #4,sp
```

In the optimized code, the parameter of function f is not popped after the call, rather the following statement just writes over a with b, the parameter to function g. This optimization is performed even if the function calls are not consecutive, just as long as both function calls are in the same basic block (i.e., there are no intervening branches or labels.)

Peephole Optimizations The broadest class of optimizations are those performed on the moving instruction window. Peephole optimizations entail analyzing one-, two-, and three-instruction sequences and removing instructions that are not needed, or rewriting the sequence into an equivalent sequence that is faster or shorter. There are currently about 50 peephole transformations that cover a wide variety of instruction sequences.

The following examples show some peephole optimizations. The compiler-generated code is on the left, the optimized code is on the right. Comments are given after the semicolons:

```
clr.l    dØ        ->    moveq #Ø,dØ        ; moveq is faster

moveq    #-1,dØ    ->    <deleted>          ; essentially a NOP
and.l    dØ,dl

moveq    #5,dØ     ->    addq #5,dl         ; move is not necessary
add.l    dØ,dl

sub.w    #1,dØ     ->    dbra dØ,label      ; single dbra is shorter
cmp.w    #-1,dØ                             ; and much faster
bne      label

move.l   #16,dØ    ->    clr.w dl           ; the latter is equivalent
asr.l    dØ,dl           swap  dl           ; (though not necessarily
and.l    #$FFFF,dl                          ; as obvious)
```

The live/dead information is essential for many of the transformations. For example, the third optimization in the foregoing example could not be done if some instruction following the ADD used dØ (i.e., dØ is not dead after the ADD). Also, the first example is not really a NOP since the AND sets condition codes that might be required later. Therefore, in order to properly remove the AND, the appropriate condition codes must be dead after that instruction.

It should be noted that the transformations are geared toward code produced by the C compiler and are not intended to be generally applicable to all assembly-language programs. In fact, the optimizer makes a number of assumptions based on the known behavior of the compiler code generator. For example, the second optimization in the foregoing example is not done for the more obvious code sequence that follows since the optimizer knows that the compiler will generate code to first load -1 into a register:

```
and.l   #-1,dl
```

Assembler (as/wimac) Optimizations

The final phase of optimization is performed is the assembler. There are only two transformations done, both having to do with branch instructions. The general technique is known as branch shortening or span-dependent optimization. These optimizations are controlled by the -O flag of the assembler. The cc or compile command passes this flag to the assembler, so this optimization is always done. The input to the assembler is a file containing assembly-language modules (corresponding to C functions). The output is a machine-language file in either COFF or LL format. The unit of optimization is the module (function).

On the 68000, there are two types of branches. Both types are relative in that the target of the branch is specified as a byte distance from the current location. The short branch has an 8-bit displacement, allowing targets anywhere from -128 to 127 bytes from the current location. The word branch has a 16- bit displacement, allowing targets from -32768 to 32767 bytes away. (Branches requiring offsets greater than 16 bits can use the JMP instruction, which takes an absolute 32-bit target.) The compiler (and optimizer) always generate the word form of branches.

In the first optimization, the assembler determines how far away the target of each branch is and shortens it if the distance to the respective target will fit in 8 bits. The assembler only does this with intra-module branches. It does not attempt to shorten inter-module branches since it can make no assumptions about the way the linker will order modules in the final output file.

The second optimization is a side-effect of the first. If the target of a branch is the next instruction, the branch instruction is replaced by a NOP.

How to Get Good Code from the Software Generation System

The WICAT SGS produces code that is adequate for most applications under most circumstances. There are, however, situations that warrant a little extra work to coerce even better code from the SGS. There are no set rules to be followed; experience and experimentation are the best tools in these situations. However, the information in the following sections can help you produce better code.

Do not use register variables indiscriminately

Remember that in most cases creating a register variable entails saving and restoring the previous contents of the register. Typically, if a variable is accessed less than three or four times, it is not advantageous to make it a register variable.

Do not over-declare register variables

You can have only 4 register pointer variables and 6 register long/int/short/char variables. Declaring more doesn't cause a warning to be generated, but the register specification is ignored. If the most heavily used pointer is declared as a register after four others, performance can be far worse than it would be if it were declared after only three others.

Don't make functions too long

Long functions increase the probability that the long form of branches will have to be used. It also increases the probability that the optimizer will run out of memory attempting to process the function.

Consider using preprocessor macros instead of simple functions

For example, a function like the following causes a function call, register load, compare, and return from function:

```
isthree(arg) register int arg;
{
        return(arg==3);
}
```

This ocurrs because the following preprocessor macro definition requires only a single in-line comparison:

```
#define isthree(arg)      ((arg)==3)
```

However, the in-line nature is not always an advantage.

For example, consider the first segment of C code as opposed to the second segment of C code:

```
isnum(arg) register int arg;
{
        return(arg==0  arg==1  arg==2  arg==3  arg==4
       arg==5  arg==6  arg==7  arg==8  arg==9);
}


#define isnum(arg) \
      ((arg)==0  (arg)==1  (arg)==2  (arg)==3  (arg)==4   \
            (arg)==5  (arg)==6  (arg)==7  (arg)==8  (arg)==9)
```

With the following invocation, the macro definition generates code to evaluate the (rather unusual) expression for each of the 10 comparisons, whereas the function definition requires that the expression be evaluated only once with 10 register comparisons:

```
isnum(anarray[i+5*3][j/(int)sin(k)]->field.nutherfield);
```

<u>Use pre-increment/decrement in favor of post-increment/decrement</u>

The post-increment form requires that the appropriate value first be copied to a temporary location, the original incremented, and then the old value used from the temporary copy. In the pre-increment form, the value can be incremented in place, and the new value used. Note that there is no difference in the degenerate case where the value is not used (e.g., the statement ++i; vs. i++;) Another exception is when the expression can be mapped into a use of the 68000 post-increment addressing mode.

<u>Use char and short integer variables instead of int variables</u>

Since the natural word size of the 68000 is 16 bits, char (8 bit) and short (16 bit) operations are typically much faster than the analogous 32-bit operation. Using shorter integers often allows the compiler and optimizer to perform type reductions and hence generate 8- and 16-bit operations. This is extremely useful for multiplication and division since the 68000 has no hardware instructions for performing 32-bit by 32-bit multiplication and division. These operations must be done by calling runtime library functions. Note that this tactic can backfire if no type reductions can be performed. In these cases, a large portion of the generated code ends up being instructions to promote the shorter operands so that 32-bit operations can be used.

<u>Use pointers rather than indices when looping through arrays</u>

The 68000 does not have an addressing mode that is well suited to general array indexing. As a result, array indexing can be an expensive proposition. The following operations typically occur for a reference of the form array[loopindex]:

1. Load loopindex into a register.
2. Multiply this register by the size of an array element.
3. Add the address of array.
4. Put this value into an address register.
5. Reference the value by indirecting off the address register.

This computation is performed at every iteration of the loop. By using a pointer variable, references of the form *arraypointer cause the following operations to take place:

1. Load arraypointer into an address register
2. Reference the value by indirecting off the address register

This pointer is incremented each time through the loop by simply adding a constant (the array-element size). This tactic does not gain you much if the termination condition of the loop still involves an array reference (e.g., "arraypointer < &array[maxindex]") since the termination condition must be evaluated each time through the loop.

Keep the range of switch statements small and not too sparse

The range of a switch statement is the numeric difference between the lowest valued case and the highest valued case (excluding the default case). The fastest code that the compiler can generate for a switch statement is a jump table. This produces the smallest average number of comparisons per case (typically one). In order to use a jump table, the range must be less than 16384 and there must be at least four cases.

An additional constraint is that the range not be too sparse. For the compiler, this means that the range must be less than three times the number of cases. For example, this prevents a 45Ø-element jump table from being produced for a switch involving only cases 1, 2, 3, 4, and 45Ø since there would have to be at least 15Ø cases to generate one. If, for some reason, you needed a jump table in this instance you could "pad it out" by adding 145 cases between 5 and 45Ø that did nothing but break.

Some Special Tricks

If speed takes precedence over good taste, there are some tricks that can be applied in special cases.

Optimizing functions with more register variables than registers

There are at least three ways to solve this problem. The most acceptable way is to break the function into a number of sub-functions. However, if you do not want to incur the function call overhead, you could either overload register variables or use inner blocks. Overloading simply means declaring a set of generic register variables (e.g., register int i; register int *p;) and, by using explicit casts, use them in place of many different variables. This is very non-portable.

A slightly better technique is to declare register variables inside inner blocks within a function. This allows the compiler to use the same register in different blocks.

For example, the following code allows the compiler to use the same address register for cp and sp:

```
     if (addr & 1) {
             register char *cp = (char *)addr;
             .
             .
             addr++;
     }
     if (addr & 2) {
             register short *sp = (short *)addr;
             .
             .
             addr += 2;
     }
```

Generating Special Instructions and Addressing Modes

DBRA instruction If you have a loop in which the loop index serves only to control the number of iterations of the loop, it can be coded in a special way to allow the optimizer to generate a DBRA instruction. For example, consider the following loop:

```
          int i;
          for (i = 0; i < END; i++)
                  <some-stuff-not-using-i>;
```

If END < 32768, the foregoing code can be rewritten as the following:

```
          register short i;
          i = END - 1;
          do {
                  <some-stuff-not-using-i>;
          } while (--i != -1);
```

CMPM instruction When comparing bytes of data, the compiler generates a CMPM instruction for an if statement of the following form if p1 and p2 both point to objects of the same size (char/short/int/long) and both pointers are in registers:

```
     if (*p1++ == *p2++)
```

Post-increment and pre-decrement addressing modes Expressions involving references of the following form, where rp is a register variable (char/short/int/long), produce instructions using post-increment and pre-decrement addressing modes:

```
        *rp++
        *--rp
```

Speeding Up Arithmetic Operations

In instances where you cannot shorten the types of variables (as described earlier) because the range of a variable is greater than 8 or 16 bits, it is still possible in many instances to gain an advantage. Consider the following example, where there is a frequently used arithmetic statement in which the majority of the operand values are less than 8 or 16 bits:

```
     register int i1, i2;
     for (i1 = 0; i1 < 50000; i1++)
             i2 = i1 / 33;
```

You can try inserting a value test combined with an explicit cast to force a shorter operation when appropriate:

```
     for (i1 = 0; i1 < 50000; i1++)
             if (i1 < 32768)
                     i2 = (short)i1 / 33;
             else
                     i2 = i1 / 33;
```

In the foregoing example, the added cost of the test is insignificant compared to the time saved by converting the call to the runtime 32-bit division to a single hardware division instruction. In other situations, the added cost may exceed the savings. Only experimentation will tell.

Improving array access and pointer arithmetic

As mentioned earlier, using pointers to access array elements is typically more efficient than indexing. There are also some other special cases that can be improved when indexing is unavoidable. If space is not a big concern, array element definitions can be padded to a size that is a power of 2. This allows the compiler to generate a simple shift (as opposed to a long multiply or a series of shifts and adds) when selecting an array element.

Consider the following section of C code:

```
struct s {
        int type;
        char data[26];
};
struct s sarray[10];

struct s *getpointer(ix)
  int ix;
{
        return(&sarray[ix]);
}
```

The foregoing example generates the following eight-instruction sequence to compute the return value:

```
move.l       8(a6),d0
add.l        d0,d0
move.l       d0,d1
add.l        d0,d0
sub.l        d0,d1
asl.l        #3,d0
add.l        d1,d0
add.l        #_sarray,d0
```

Adding two bytes of padding to the structure definition (to bring the size to 32 bytes) generates the following three-instruction sequence:

```
move.l  8(a6),d0
asl.l   #5,d0
add.l   #_sarray,d0
```

Padding also benefits pointer addition (since that is what array indexing really is). It does not directly help pointer subtraction however. For example, the compiler implements the following array by subtracting the two pointers and then dividing by the size of an element:

```
getindex(sp)
  struct s *sp;
{
        return(sp - sarray);
}
```

The compiler generates a call to the long division routine regardless of padding. A right shift is not used because the pointers are considered signed. A DIVS instruction cannot be used since the quotient might be greater than a word (i.e., the array has more than 32767 elements), which would cause an overflow with DIVS.

When using padding, we can force a right shift by rewriting the function as:

```
getindex(sp)
  struct s *sp;
{
        return(((unsigned int)sp - (unsigned int)sarray)
              / sizeof(struct s));
}
```

Fast data copying/comparison

One of the most common operations in which generated code quality has a major impact is copying or comparing blocks of data. As might be expected, there are numerous obvious and non-obvious ways to perform these functions. For large blocks of data, the best way is probably to use the standard library functions that are written in assembly language and have been highly optimized for the best performance. Two such routines are memcpy and memcmp. Under UNIX, these routines are documented in memory(3C) of the UniPlus+ System V User's Reference Manual(Sections 2-6).

As usual, there is a trade-off point at which the overhead of the function call outweighs the benefits. This point is at about 16 bytes. For 16 bytes or less, it is often worthwhile to copy/compare data in-line.

The following examples show four possible ways to define a macro to copy c bytes from f to t:

```
#define COPY0(f,t,c)  \
      { register int n; for (n = 0; n < c; n++) t[n] = f[n]; }

#define COPY1(f,t,c)  \
        memcpy(t,f,c)

#define COPY2(f,t,c)  \
        { register short n = c-1; register char *fp = f, *tp = t; \
        do *tp++ = *fp++; while (--n != -1); }

#define COPY3(f,t)    \
        { struct hack { char space[FIXEDSIZE]; } hack; \
        *((struct hack *)t) = *((struct hack *)f); }
```

COPY0 is the approach a naive user might first try. Though simple, it takes easily twice as long as the next slowest method when copying 16 bytes. COPY1 is a call to the library routine and, for 16 bytes, takes nearly twice as long as COPY2. COPY2 is probably the best code that you can get for byte-by-byte copies. But its speed is highly dependent on having the three register variables actually in registers. It also limits the count to 65536 since it uses a short counter. COPY3 is a very special case that requires the copy size always be the same and known at compile time. It also requires that both the to and from buffers be on an even-byte boundary. COPY3 coerces the buffers into structures so that the compiler generates in-line code to do a structure copy. It more than twice as fast as COPY2 and almost 2Ø times faster than COPYØ for a 16-byte copy.

For comparing c characters of two buffers f and t (setting r to one if equal, zero if not), you can define the following analogous routines:

```
    #define CMPØ(f,t,c,r) \
            { register int n; \
            for (n = Ø; n < c; n++) \
                    if (t[n] != f[n]) break; \
            r = (n == c); }

    #define CMP1(f,t,c,r) \
            r = (memcmp(t,f,c) == Ø);

    #define CMP2(f,t,c,r) \
            { register short n = c-1; register char *fp = f, *tp = t; \
            do if (*tp++ != *fp++) break; while (--n != -1); \
            r = (n == -1); }

    #define CMP3(f,t,c,r) \
            { register int *ip1 = (int *)f, *ip2 = (int *)t; \
            r = Ø; if (*ip1++ == *ip2++) if (*ip1++ == *ip2++) \
            if (*ip1++ == *ip2++) if (*ip1 == *ip2) r = 1; }
```

The macro that is significantly difference is CMP3, which is approximately equivalent to what the compiler might do if there was such a thing as structure comparison. It has the same restrictions as COPY3. For 16-byte comparisons, in the best case where the first byte differs, there is very little difference between the three in-line versions. The function call version takes about twice as long. In the worst case where both buffers are equal, the times are about the same as those of the copy macros.

Cautions about optimizing

The C optimizer was intended only to improve code generated from the C compiler. For this reason, attempting to optimize hand-coded assembly or interfacing hand-coded assembly to optimized code is likely to cause problems. If you try to do this, you should be aware of the following items:

The peephole optimizer will miss many "obvious" optimizations because it assumes the compiler never generates them.

Many peephole optimizations can be performed only if the condition codes are dead following the window in question. Currently, true live/dead analysis is not performed on condition codes. To determine if the condition codes are dead following a window, it merely checks to see if the next instruction uses them. If not, it assumes they are dead.

The optimizer assembly language parser is not complete. It recognizes only instructions, addressing modes, and pseudo-ops produced by the compiler. Many unrecognized constructs are just copied to the output file when they are encountered. This creates problems due to the fact that recognized instructions are internalized and output at one time at the end of a function. The resulting output for a function is thus all unrecognized code for a function followed by the optimized body of the function.

Assembly code in asm statements is not handled well. Currently, the optimizer has no way of differentiating between code from asm statements and code produced by the compiler. Hence, it attempts to optimize it. Most times this works, but sometimes it messes up badly (typically because the optimizer assembly language is incomplete).

There is no way of selectively choosing which optimizations are performed on a function or even a file.

Chapter 5

Floating-point

WICAT's floating-point arithmetic conforms to a subset of the proposed IEEE standard for binary floating-point arithmetic.

WICAT's software emulation does not support such features as denormalized operands, extended precision, selection of rounding modes, NaN's, and full exception handling. The different floating-point hardware types conform to this standard in varying degrees.

How the C Compiler Handles Floating-point

The C compiler generates pseudo-code for all floating-point operations. This enables programs to use floating-point without the compiler having details of each kind of hardware. One pseudo-code statement (one line) is generated for each floating-point operation. The statement indicates the operation to be performed, and the operands to use (source and destination).

Each type of floating-point hardware has a preprocessor. The software emulation also has a preprocessor. These floating-point preprocessors convert the pseudo-code to assembly source statements. The preprocessor generates the assembly code to perform the indicated operation on its type of hardware. Also, a hardware preprocessor that is not hardware dependent converts pseudo-code to subroutine calls where the subroutine performs the indicated operation.

For information on selecting a floating-point preprocessor under UniPlus+ System V see cc in the _UniPlus+ System V User's Manual (Section 1)_.

For information on selecting a floating-point preprocessor under WMCS see the _compile_ command description in chapter 8 of this manual.

Floating-point Under WMCS

Under WMCS you can bind your program to a floating-point hardware by using its preprocessor at compile time.

Also, you can use the hardware-independent preprocessor to choose the software emulation, or the specific hardware at run time. This can be done because the libraries that support the software emulation, or the hardware, are not linked into the program's executable image at compile time.

The libraries are in shared memory and become part of your program during its execution. They are put into shared memory using the fpmgr command (Refer to the fpmgr command description in the WMCS User's Reference Manual).

The C language startup routines are automatically linked into your program. Also, they execute before the first statement of your program and connect it to the floating-point library in shared memory.

Floating-point Under UniPlus+ System V

Under UniPlus+ System V you can bind your program to a floating-point hardware by using the preprocessor for the hardware at compile time.

The hardware-independent preprocessor binds your program to the software floating-point emulation.

The choice of hardware or software cannot be made at run time.

The library that supports the software emulation, or the hardware, is linked into your program's executable image at compile time.

Getting the Best Performance

Following are some general guidelines for getting the best performance from floating-point operations.

Choosing the Floating-point Preprocessor

Hardware is faster than software, although the difference in speed depends on which operation you are doing and which hardware you are using.

Under UniPlus+ System V your program uses hardware if you compile it using the hardware preprocessor.

Under WMCS you get best performance by compiling your program with the hardware preprocessor. If you use the hardware independent preprocessor, and choose the hardware at run time, performance is not as good. The code generation is less efficient due to the indirectness of subroutine calls and parameter passing.

C Optimizer

The C optimizer makes code generated by the C compiler more efficient. It also optimizes floating-point pseudo-code.

Because the C optimizer does not recognize differences in hardware, some of the pseudo-code optimizations can degrade the program's performance on some hardware.

Your program will probably perform better if it is compiled using the C optimizer. However, try it both ways to find which way gives you best performance.

Floating-point Registers

If your floating-point hardware has general-purpose floating-point registers, declare floating-point variables as **register float** or **register double** to improve performance.

The first four floating-point registers are reserved as scratch registers for the compiler and floating-point preprocessor. The next four floating-point registers are available to be used for register declarations.

If your hardware has more than eight registers, only the first eight are used.

The maximum number of useful register declarations is four.

If your hardware does not have floating-point registers, using register declarations degrades performance.

Precision

Operations on floats (single precision) are faster than operations on doubles (double precision). The difference in speed depends on whether you are using hardware or software, which hardware is being used, and what operation is being performed.

However, the C language is designed to promote all calculations to double precision. Even if you declare all your floating-point variables as float, most calculations are still done in double precision. Once calculations are complete, a conversion is performed on the result. This converts it to a float. The WICAT C compiler makes one exception to this.

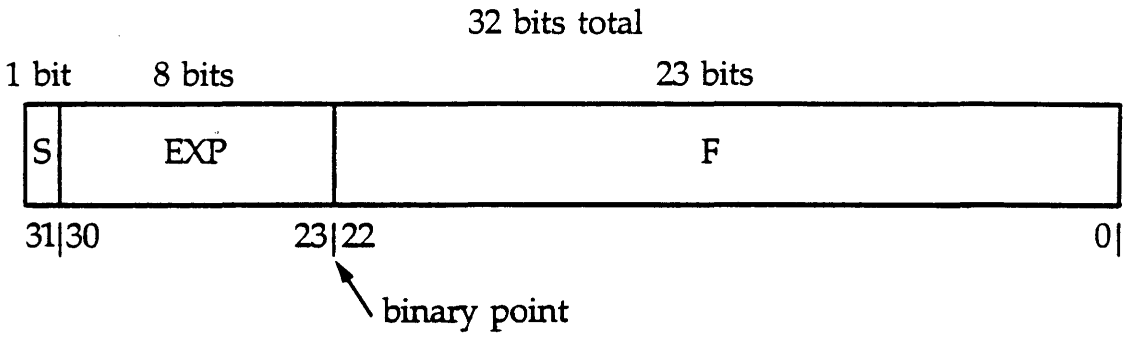
If you use the +=, -=, *=, or /= operators on float operands, the calculation is done in single precision.

C is also defined to promote all float parameters passed to subprograms to double precision. Therefore, a single precision sin, cos, exp, sqrt, etc. cannot be computed. The computation is done in double. The result is converted to single.

Floating-point Format

The storage formats for floating-point values conform to the proposed IEEE Standard for binary floating-point arithmetic. All values are normalized.

Following is a diagram of a float storage format:



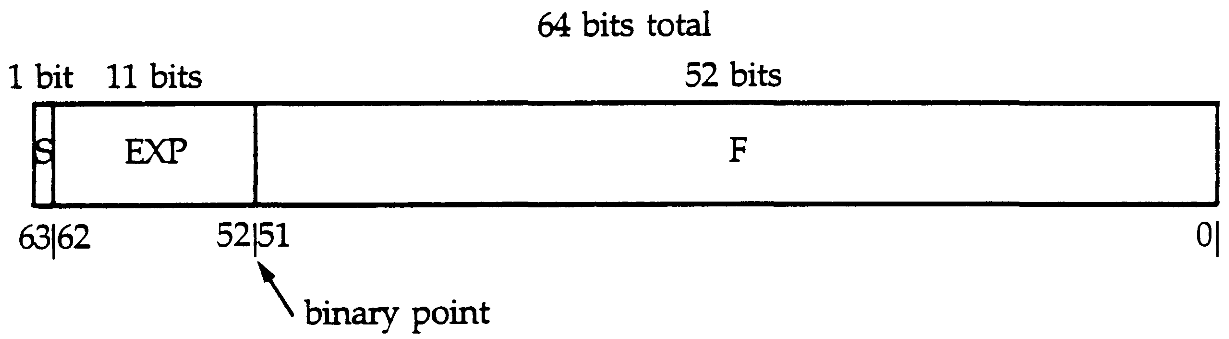
In the foregoing diagram s is sign (0=positive, 1=negative), exp is biased exponent (true exponent is biased exponent - 127), and f is binary fraction.

The float storage format is normalized with an implied 1 bit preceding the binary point.

The float storage format is interpreted (converted to decimal) as (s) 1.f x 2^ (exp-127).

It has 6 - 7 significant decimal digits, and its approximate range is 8.4e-37 <= x <=3.4e+38.

Following is a diagram of a double storage format:



In the foregoing diagram s is sign (Ø = positive, 1 = negative), exp is biased exponent (true exponent is biased exponent - 1Ø23), and f is binary fraction.

The double storage format is normalized with an implied 1 bit preceding the binary point.

The double storage format is interpreted (converted to decimal) as (s) 1.f x 2^ (exp-1Ø23).

It represents 15 to 16 significant decimal digits, and its approximate range is 4.2e-3Ø7 <= x <=1.8e+3Ø8.

Following is a table of reserved floating-point values:

		Float	Double
zero	+Ø	ØØØØØØØØ	ØØØØØØØØ ØØØØØØØØ
	-Ø	8ØØØØØØØ	8ØØØØØØØ ØØØØØØØØ
infinity	+∞	7FFFFFFF	7FFFFFFF FFFFFFFF
	-∞	FFFFFFFF	FFFFFFFF FFFFFFFF

Exception Handling

Four types of floating-point exceptions are supported:

1. Underflow occurs when the result of a calculation is too small to be represented.

2. Overflow occurs when the results of a calculation is too large to be represented.

3. Divide-by-zero occurs when you attempt to perform a division operation with a denominator of zero.

4. Illegal operation occurs when you attempt to do a mathematically undefined operation (taking the square root of a negative number).

Each of the foregoing exceptions has a standard default. A default result is returned if an error occurs and the exception is masked (the exception is turned off because the programmer does not want to know the error occurred).

The default results are used as operands in subsequent calculations as normal results would be. For example, a value of infinity generated due to an attempt to divide by zero can be propagated through subsequent calculations in a program. If the value is output, it shows up as approximately 1.79e+3Ø8. This is the largest number representable in double precision.

Underflow returns zero, overflow returns infinity, and divide-by-zero returns infinity when the software emulation exceptions are masked.

Illegal operation has no default result because it cannot be masked.

If you are using floating-point hardware, consult the hardware documentation to see what exceptions are supported, whether they can be masked, and what default results are returned.

Exception Handling under UniPlus+ System V

If an exception occurs and SIGFPE has a signal handler defined, the SIGFPE signal is generated and caught by the handler. The handler does nothing useful for floating-point. The handler cannot return its own default result and continue processing.

The startups automatically define a handler for SIGFPE for the process. However, the definition can be deleted, or the defined handler can be replaced by a user handler once inside the program.

The default handler causes a core dump with the following message:

```
Floating exception -- core dumped
```

The _errno variable is set to ERANGE.

If no handler is defined, all exceptions that can be masked are masked, default results are generated, and processing continues. You are not informed if error conditions occur.

Exceptions cannot be selectively unmasked from a C program.

The matherr(3M) routine for handling math errors is not supported.

Exception Handling under WMCS

Under WMCS exceptions that can be masked (for software or hardware) are masked. In other words, when a maskable exception occurs, the standard default result for that condition is returned. You are not informed that the error has occurred.

Exceptions cannot be unmasked from a C program.

Debugging Floating-point Programs

A floating-point operation shows up in the assembly source code produced by a floating-point preprocessor in two ways:

It can be a jsr (subroutine call) to a function name that has no underscore. In this case, it typically has a math operation within the name (e.g.,jsr add3ddxd, jsr divf, jsr sin).

When a hardware floating-point preprocessor is used, the operation can be a move (moves data), or a tst to an address from 0x2000 to 0x3fff (the address range is the user process into which the hardware is mapped). The tst compares against 0, but for hardware floating-point, it usually accesses a location to cause an operation.

If you wish to determine what operation is taking place, get a listing of the compiler output prior to the floating-point preprocessor pass. Use the cc command with the -K option under Uniplus+ System V, and the compile command with the :nofpreprocess switch under WMCS. This allows you to look at the pseudo-code.

If you are using floating-point hardware that has registers, neither adb nor WIBUG can display or modify the contents of the registers.

The debuggers are able to input and output IEEE-format single-and double-precision numbers. Refer to the WIBUG or adb documentation for details.

Debugging floating-point programs under UniPlus+ System V

Floating-point routines called by jsr instructions are linked into your program like other routines. Their symbols appear in adb. You can trace through them like any other routine.

Debugging floating-point programs under WMCS

Because you can choose the software emulation or a floating-point hardware at runtime, you should be aware of the following when debugging under WMCS:

If no libraries, or the wrong libraries, are in shared memory, one of the following error messages appears when you try to run your program:

```
Cannot share directory

Required hardware floating-point not present
```

These errors come from the C startups when they try to connect the program to shared memory.

To avoid this, use fpmgr to put the correct library into shared memory. Then, run your program again.

The symbols for floating-point routines have values in the address range 0x1000 to 0x1fff.

The symbols are entries in a jump table. The value at that location in the jump table is the starting address of the routine.

For example, if the value of symbol add3ddxd is 0x1008, the value stored at location 0x1008 is the starting address of add3ddxd. It might be something like 0x1fa448.

The symbol for the floating-point routine name is associated with an address in the jump table (not with the starting address of the routine). When you trace through the routine, addresses in floating-point routines are not displayed as offsets from the routine name.

Floating-point routines reside in address ranges 0x2000 to 0x3fff, and 0x1fa000 to 0x1fefff in the process space.

A breakpoint at a floating-point routine cannot be set until the startups have executed. Those routines are not part of your process until the startups put them there. If you try to set the breakpoint before the startups execute, WIBUG returns a memory violation.

Get past the startups by typing **xr _main.** Then set your floating-point breakpoint when you have reached _main. This works much better.

Breakpoints cannot be set inside floating-point routines because the code in shared memory is write-protected. If you attempt to set breakpoints inside floating-point routines, WIBUG gives a memory violation.

Floating-point Libraries

Following are the floating-point libraries under UniPlus+ System V:

/lib/libc.a	software emulation floating-point routines
/usr/lib/libc-skyl.a	SKY hardware floating-point routines
/usr/lib/libc-ffpl.a	FFP hardware floating-point routines

The floating-point libraries under WMCS are referenced by the fpmgr command when it puts the routines into shared memory:

/sysexe/lib2init.exe	software emulation floating-point routines
/sysexe/skylinit.exe	SKY hardware floating-point routines
/sysexe/ffplinit.exe	FFP hardware floating-point routines

Chapter 6

C Libraries

The C libraries are collections of object files, which contain the object code for one or more useful functions. When a C program (an object module) is linked by LL or ld to one of these libraries, an object file containing a referenced function is linked with the program. Both WMCS and UniPlus+ System V have several standard libraries with which to work.

The C Library

The C library contains the standard Input/Output routines, system call entry points for UniPlus+ System V, string manipulation routines, and floating-point access routines. There is a version of this library for each floating-point type.

The C library is searched by default if the linker is invoked through _cc_ or _compile_.

These are the C library files under UniPlus+ System V:

/lib/libc.a	version with software FP (default)
/usr/lib/libc-skyl.a	version with SKY FP support
/usr/lib/libc-ffpl.a	version with FFP FP support
/usr/lib/libc-nofl.a	version with no FP support

These are the C library files under WMCS:

sys$disk/comlib/libc.lib	version with FP support (default)
sys$disk/comlib/libcnofp.lib	version with no FP support
sys$disk/comlib/libcl00.lib	version with FP support and ability to have 100 files open simultaneously

The C library under WMCS has no hardware-specific versions.

The Math Library

The math library contains trigonometric, square root, and logarithmic functions.

Under WMCS, the :libraries=libm switch must be used to access the math library. The library file is SYS$DISK/COMLIB/LIBM.LIB.

The Math Library under UniPlus+ System V

Under UniPlus+ System V, the -lm option must be specified with the cc command to access the math library. The library file is /usr/lib/libm.a.

Refer to section 3 of the *UniPlus+ System V User's Manual (Sections 2-6)* for information on additional libraries under UniPlus+ System V.

Also, profiled versions of many of the libraries are available under UniPlus+ System V. Every routine (function) in a profiled library includes code to count the calls made to the function (i.e., each function that was compiled with the -p option of cc).

Chapter 7

Dictionary of C Library Routines

The C library routines contained in this chapter are for the user of C under WMCS. They are derived from the UniPlus+ System V documentation and have been modified slightly for the WMCS user.

Each entry in this dictionary is based on the format used in the UniPlus+ System V documentation.

The following sections are used.

> Name gives the name(s) of the routine(s) described under this entry and gives a very brief description of it/them. More than one routine is often described with one entry because the routines are so similar. The dictionary entry for a routine tells where a routine is described. (The index also lists where a routine is described.)
>
> Synopsis shows how a routine is set up. It shows the include files needed, if any, and shows how the routine itself is declared. These are not lines you type in your program (except the include line), but show the way the routine itself is programmed.
>
> Description describes in more detail the routines listed in the Name section.
>
> Example gives an example where appropriate.
>
> Files lists the files the routine(s) might use.
>
> See also refers to related routines.
>
> Diagnostics discusses diagnostic indications that can be produced.
>
> Warnings points out potential problems.

Bugs lists known bugs and deficiencies. Sometimes the suggested fix is also listed.

The user of C under UniPlus+ System V should consult sections 2 and 3 of the *UniPlus+ System V User's Reference Manual (Sections 2-6)*.

Name

a64l, l64a - convert between long integer and base-64 ASCII string

SYNOPSIS

```
long a64l (s)
char *s;

char *l64a (l)
long l;
```

DESCRIPTION

These functions are used to maintain numbers stored in base-64 ASCII characters. This is a notation by which long integers can be represented by up to six characters; each character represents a digit in a radix-64 notation.

The characters used to represent digits are . for Ø, /for 1, Ø through 9 for 2-11, A through Z for 12-37, and a through z for 38-63.

A64l takes a pointer to a null-terminated base-64 representation and returns a corresponding long value. If the string pointed to by s contains more than six characters, a64l will use the first six.

L64a takes a long argument and returns a pointer to the corresponding base-64 representation. If the argument is Ø, l64a returns a pointer to a null string.

BUGS

The value returned by l64a is a pointer into a static buffer, the contents of which are overwritten by each call.

Name

abs - return integer absolute value

Synopsis

```
int abs (i)
int i;
```

Description

Abs returns the absolute value of its integer operand.

Bugs

In two's-complement representation, the absolute value of the negative integer with largest magnitude is undefined. Some implementations trap the error, but others ignore it.

See Also

floor

NAME

sin, cos, tan, asin, acos, atan, atan2 - trigonometric functions

SYNOPSIS

See trig

NAME

asin, cos, tan, sin, acos, atan, atan2 - trigonometric functions

SYNOPSIS

See trig

NAME

sin, cos, tan, asin, acos, atan, atan2 - trigonometric functions

SYNOPSIS

See trig

NAME

j0, j1, jn, y0, y1, yn - Bessel functions

SYNOPSIS

```
#include <math.h>

double j0 (x)
double x;

double j1 (x)
double x;

double jn (n, x)
int n;
double x;

double y0 (x)
double x;

double y1 (x)
double x;

double yn (n, x)
int n;
double x;
```

DESCRIPTION

J0 and j1 return Bessel functions of x of the first kind of orders 0 and 1 respectively. Jn returns the Bessel function of x of the first kind of order n.

SEE ALSO

exec(2).

NAME

brk, sbrk - change data segment space allocation

SYNOPSIS

```
int brk (endds)
char *endds;

char *sbrk (incr)
int incr;
```

DESCRIPTION

Brk and sbrk are used to change dynamically the amount of space allocated for the calling process's data segment; see exec(2). The change is made by resetting the process's break value and allocating the appropriate amount of space. The break value is the address of the first location beyond the end of the data segment. The amount of allocated space increases as the break value increases. The newly allocated space is set to zero.

Brk sets the break value to endds and changes the allocated space accordingly.

Sbrk adds incr bytes to the break value and changes the allocated space accordingly. Incr can be negative, in which case the amount of allocated space is decreased.

RETURN VALUE

Upon successful completion, brk returns a value of Ø and sbrk returns the old break value. Otherwise, a value of -1 is returned and errno is set to indicate the error.

NOTES

The pointers to the key and the element at the base of the table should be of type pointer-to-element, and cast to type pointer-to-character. The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared. Although declared as type pointer-to-character, the value returned should be cast into type pointer-to-element.

SEE ALSO

lsearch, hsearch, qsort, tsearch

NAME

bsearch - binary search

SYNOPSIS

```
#include <search.h>

char *bsearch ((char *) key, (char *) base, nel, sizeof
(*key), compar)
unsigned nel;
int (*compar)( );
```

DESCRIPTION

Bsearch is a binary search routine generalized from Knuth (6.2.1) Algorithm B. It returns a pointer into a table indicating where a datum may be found. The table must be previously sorted in increasing order according to a provided comparison function. Key points to the datum to be sought in the table. Base points to the element at the base of the table. Nel is the number of elements in the table. Compar is the name of the comparison function, which is called with two arguments that point to the elements being compared. The function must return an integer less than, equal to, or greater than zero according as the first argument is to be considered less than, equal to, or greater than the second.

DIAGNOSTICS

A NULL pointer is returned if the key cannot be found in the table.

NAME

floor, ceil, fmod, fabs - floor, ceiling, remainder, absolute value functions

SYNOPSIS

See floor

NAME

chdir - change working directory

SYNOPSIS

```
int chdir (path)
char *path;
```

DESCRIPTION

Path points to the path name of a directory. Chdir causes the named directory to become the current working directory, the starting point for path searches for path names not beginning with /.

Chdir will fail and the current working directory will be unchanged if one or more of the following are true:

A component of the path name is not a directory. [ENOTDIR]

The named directory does not exist. [ENOENT]

Search permission is denied for any component of the path name. [EACCES]

Path points outside the process's allocated address space. [EFAULT]

RETURN VALUE

Upon successful completion, a value of Ø is returned. Otherwise, a value of -1 is returned and errno is set to indicate the error.

The effective user ID does not match the owner of the file. [EPERM]

Path points outside the process's allocated address space. [EFAULT]

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and errno is set to indicate the error.

chmod

NAME

chmod - change mode of file

SYNOPSIS

```
int chmod (path, mode)
char *path;
int mode;
```

DESCRIPTION

Path points to a path name naming a file. Chmod sets the access permission portion of the named file's mode according to the bit pattern contained in mode.

Access permission bits are interpreted as follows:

00400	Read by owner
00200	Write by owner
00100	Execute (or search if a directory) by owner
00070	Read, write, execute (search) by group
00007	Read, write, execute (search) by others

Chmod will fail and the file mode will be unchanged if one or more of the following are true:

A component of the path prefix is not a directory. [ENOTDIR]

The named file does not exist. [ENOENT]

Search permission is denied on a component of the path prefix. [EACCES]

NAME

clearerr, ferror, feof, fileno - stream status inquiries

SYNOPSIS

See ferror

NAME

close - close a file descriptor

SYNOPSIS

```
int close (fildes)
int fildes;
```

DESCRIPTION

Fildes is a file descriptor obtained from a creat, open, or dup system call. Close closes the file descriptor indicated by fildes.

Close will fail if fildes is not a valid open file descriptor. [EBADF]

RETURN VALUE

Upon successful completion, a value of Ø is returned. Otherwise, a value of -1 is returned and errno is set to indicate the error.

SEE ALSO

creat(2), dup(2), exec(2), open(2),

NAME

sin, cos, tan, asin, acos, atan, atan2 - trigonometric functions

SYNOPSIS

See trig

NAME

sinh, cosh, tanh - hyperbolic functions

SYNOPSIS

See sinh

The file does not exist and the directory in which the file is to be created does not permit writing. [EACCES]

The file is a pure procedure (shared text) file that is being executed. [ETXTBSY]

The named file is an existing directory. [EISDIR]

Twenty (20) file descriptors are currently open. [EMFILE]

Path points outside the process's allocated address space. [EFAULT]

RETURN VALUE

Upon successful completion, a non-negative integer, namely the file descriptor, is returned. Otherwise, a value of -1 is returned and errno is set to indicate the error.

SEE ALSO

close(2), dup(2), lseek(2), open(2), read(2), write(2).

NAME

creat - create a new file or rewrite an existing one

SYNOPSIS

```
int creat (path, mode)
char *path;
int mode;
```

DESCRIPTION

Creat creates a new ordinary file or prepares to rewrite an existing file named by the path name pointed to by path.

If the file exists, a new version is created.

Upon successful completion, a non-negative integer, namely the file descriptor, is returned. The file pointer is set to the beginning of the file. The file descriptor is set to remain open across exec system calls. No process may have more than 20 files open simultaneously. A new file may be created with a mode that forbids writing.

Creat will fail if one or more of the following are true:

A component of the path prefix is not a directory. [ENOTDIR]

A component of the path prefix does not exist. [ENOENT]

Search permission is denied on a component of the path prefix. [EACCES]

The path name is null. [ENOENT]

The argument to the encrypt entry is a character array of length 64 containing only the characters with numerical value Ø and 1. The argument array is modified in place to a similar array representing the bits of the argument after having been subjected to the DES algorithm using the key set by setkey. If edflag is zero, the argument is encrypted; if non-zero, it is decrypted.

BUGS

The return value points to static data that are overwritten by each call.

NAME

crypt, setkey, encrypt - generate DES encryption

SYNOPSIS

```
char *crypt (key, salt)
char *key, *salt;

char *key;

void encrypt (block, edflag)
char *block;
int edflag;
```

DESCRIPTION

Crypt is the password encryption function. It is based on the NBS Data Encryption Standard (DES), with variations intended (among other things) to frustrate use of hardware implementations of the DES for key search.

Key is a user's typed password. Salt is a two-character string chosen from the set [a-zA-Z0-9./]; this string is used to perturb the DES algorithm in one of 4096 different ways, after which the password is used as the key to encrypt repeatedly a constant string. The returned value points to the encrypted password. The first two characters are the salt itself.

The setkey and encrypt entries provide (rather primitive) access to the actual DES algorithm. The argument of setkey is a character array of length 64 containing only the characters with numerical value 0 and 1. If this string is divided into groups of 8, the low-order bit in each group is ignored; this gives a 56-bit key which is set into the machine. This is the key that will be used with the above mentioned algorithm to encrypt or decrypt the string block with the function encrypt.

SEE ALSO

ttyname

NAME

ctermid - generate file name for terminal

SYNOPSIS

```
#include <stdio.h>

char *ctermid(s)
char *s;
```

DESCRIPTION

Ctermid generates the path name of the controlling terminal for the current process, and stores it in a string.

If s is a NULL pointer, the string is stored in an internal static area, the contents of which are overwritten at the next call to ctermid, and the address of which is returned. Otherwise, s is assumed to point to a character array of at least L_ctermid elements; the path name is placed in this array and the value of s is returned. The constant L_ctermid is defined in the <st_io.h> header file.

NOTES

The difference between ctermid and ttyname(3C) is that ttyname must be handed a file descriptor and returns the actual name of the terminal associated with that file descriptor, while ctermid returns a string that will refer to the terminal if used as a file name. Thus ttyname is useful only if the process already has at least one file open to a terminal.

constant width.

```
     Sun Sep 16 01:03:52 1973\n\0
```

Localtime and gmtime return pointers to tm structures, described below. Localtime corrects for the time zone and possible Daylight Savings Time; gmtime converts directly to Greenwich Mean Time (GMT).

Asctime converts a tm structure to a 26-character string, as shown in the above example, and returns a pointer to the string.

Declarations of all the functions and externals, and the tm structure, are in the <time.h> header file. The structure declaration is:

```
struct tm {
        int tm_sec; /* seconds (0 - 59) */
        int tm_min; /* minutes (0 - 59) */
        int tm_hour; /* hours (0 - 23) */
        int tm_mday; /* day of month (1 - 31) */
        int tm_mon; /* month of year (0 - 11) */
        int tm_year; /* year - 1900 */
        int tm_wday; /* day of week (Sunday = 0) */
        int tm_yday; /* day of year (0 - 365) */
        int tm_isdst;
};
```

Tm_isdst is non-zero if Daylight Savings Time is in effect.

The external long variable timezone contains the difference, in seconds, between GMT and local standard time (in EST, timezone is 5*60*60); the external variable daylight is non-zero if and only if the standard U.S.A. Daylight Savings Time conversion should be applied. The program knows about the peculiarities of this conversion in 1974 and 1975; if necessary, a table for these years can be extended.

If the logical name TZ is defined, asctime uses it to override the default time zone. The value of TZ must be a three-letter time zone name, followed by a number representing the difference between local time and Greenwich Mean Time in hours, followed by an optional three-letter name for a daylight time zone. For example, the setting for New Jersey would be EST5EDT. The effects of setting TZ are thus to change the values of the external variables timezone and daylight; in addition, the time zone names contained in the external variable

```
     char *tzname[2] = { "EST", "EDT" };
```

are set from the logical name TZ. The function tzset sets these external variables from TZ; tzset is called by asctime and may also be called explicitly by the user.

NAME

ctime, localtime, gmtime, asctime, tzset - convert date and time to string

SYNOPSIS

```
#include <time.h>

char *ctime (clock)
long *clock;

struct tm *localtime (clock)
long *clock;

struct tm *gmtime (clock)
long *clock;

char *asctime (tm)
struct tm *tm;

extern long timezone;

extern int daylight;

extern char *tzname[2];

void tzset ( )
```

DESCRIPTION

Ctime converts a long integer, pointed to by clock, representing the time in seconds since 00:00:00 GMT, January 1, 1970, and returns a pointer to a 26-character string in the following form. All the fields have

SEE ALSO

time

BUGS

The return values point to static data whose content is overwritten by each call.

isspace	c is a space, tab, carriage return, new-line, vertical tab, or form-feed.
ispunct	c is a punctuation character (neither control nor alphanumeric).
isprin_	c is a printing character, code 040 (space) through 0176 (tilde).
isgraph	c is a printing character, like isprint except false for space.
iscntrl	c is a delete character (0177) or an ordinary control character (less than 040).
isascii	c is an ASCII character, code less than 0200.

DIAGNOSTICS

If the argument to any of these macros is not in the domain of the function, the result is undefined.

NAME

isalpha, isupper, islower, isdigit, isxdigit, isalnum, isspace, ispunct, isprint, isgraph, iscntrl, isascii -classify characters

SYNOPSIS

```
#include <ctype.h>

int isalpha (c)
int c;

. . .
```

DESCRIPTION

These macros classify character-coded integer values by table lookup. Each is a predicate returning nonzero for true, zero for false. Isascii is defined on all integer values; the rest are defined only where isascii is true and on the single non-ASCII value EOF (-1 - see stdio(3S)).

isalpha	c is a letter.
isupper	c is an upper-case letter.
islower	c is a lower-case letter.
isdigit	c is a digit [Ø-9].
isxdigit	c is a hexadecimal digit [Ø-9], [A-F] or [a-f].
isalnum	c is an alphanumeric (letter or digit).

NAME

cuserid - get character login name of the user

SYNOPSIS

```
#include <stdio.h>

char *cuserid (s)
char *s;
```

DESCRIPTION

Cuserid generates a character-string representation of the login name of the owner of the current process. If s is a NULL pointer, this representation is generated in an internal static area, the address of which is returned. Otherwise, s is assumed to point to an array of at least L_cuserid characters; the representation is left in this array. The constant L_cuserid is defined in the <stdio.h> header file.

DIAGNOSTICS

If the login name cannot be found, cuserid returns a NULL pointer; if s is not a NULL pointer, a null character (\Ø) will be placed at s[Ø].

SEE ALSO

getlogin

Functions drand48 and erand48 return non-negative double-precision floating-point values uniformly distributed over the interval [0.0,~1.0).

Functions lrand48 and nrand48 return non-negative long integers uniformly distributed over the interval [0,2 sup31).

Functions mrand48 and jrand48 return signed long integers uniformly distributed over the interval [-2 sup 31 ,2 sup 31).

Functions srand48, seed48 and lcong48 are initialization entry points, one of which should be invoked before either drand48, lrand48 or mrand48 is called. (Although it is not recommended practice, constant default initializer values will be supplied automatically if drand48, lrand48 or mrand48 is called without a prior call to an initialization entry point.) Functions erand48, nrand48 and jrand48 do not require an initialization entry point to be called first.

All the routines work by generating a sequence of 48-bit integer values, X sub i , according to the linear congruential formula

X sub{n+1}=(aX sub n+c) sub{roman modm}n>=0.

The parameter m=2 sup 48; hence 48-bit integer arithmetic is performed. Unless lcong48 has been invoked, the multiplier value a and the addend value c are given by

```
     amark =roman 5DEECE66Dsub 16=roman
     273673163155sub 8
     clineup =roman Bsub 16=roman 13sub 8 .
```

The value returned by any of the functions drand48, erand48, lrand48, nrand48, mrand48 or jrand48 is computed by first generating the next 48-bit $X sub i$ in the sequence. Then the appropriate number of bits, according to the type of data item to be returned, are copied from the high-order (leftmost) bits of $X sub i$ and transformed into the returned value.

The functions drand48, lrand48 and mrand48 store the last 48-bit X sub i generated in an internal buffer; that is why they must be initialized prior to being invoked. The functions erand48, nrand48 and jrand48 require the calling program to provide storage for the successive X sub i values in the array specified as an argument when the functions are invoked. That is why these routines do not have to be initialized; the calling program merely has to place the desired initial value of X sub i into the array and pass it as an argument. By using different arguments, functions erand48, nrand48 and jrand48 allow separate modules of a large program to generate several independent streams of pseudo-random numbers, i.e., the sequence of numbers in each stream will not depend upon how many times the routines have been called to generate numbers for the other streams.

drand48

NAME

drand48, erand48, lrand48, nrand48, mrand48, jrand48, srand48, seed48, lcong48 - generate uniformly distributed pseudo-random numbers

SYNOPSIS

```
double drand48 ( )

double erand48 (xsubi)
unsigned short xsubi[3];

long lrand48 ( )

long nrand48 (xsubi)
unsigned short xsubi[3];

long mrand48 ( )

long jrand48 (xsubi)
unsigned short xsubi[3];

void srand48 (seedval)
long seedval;

unsigned short *seed48 (seed16v)
unsigned short seed16v[3];

void lcong48 (param)
unsigned short param[7];
```

DESCRIPTION

This family of functions generates pseudo-random numbers using the well-known linear congruential algorithm and 48-bit integer arithmetic.

The initializer function srand48 sets the high-order 32 bits of X sub i to the 32 bits contained in its argument. The low-order 16 bits of X sub i are set to the arbitrary value roman 33ØE sub 16 .

The initializer function seed48 sets the value of X sub i to the 48-bit value specified in the argument array. In addition, the previous value of X sub i is copied into a 48-bit internal buffer, used only by seed48, and a pointer to this buffer is the value returned by seed48. This returned pointer, which can just be ignored if not needed, is useful if a program is to be restarted from a given point at some future time - use the pointer to get at and store the last X sub i value, and then use this value to reinitialize via seed48 when the program is restarted.

The initialization function lcong48 allows the user to specify the initial X sub i , the multiplier value a, and the addend value c. Argument array elements param[Ø-2] specify X sub i , param[3-5] specify the multiplier a, and param[6] specifies the 16-bit addend c. After lcong48 has been called, a subsequent call to either srand48 or seed48 will restore the standard multiplier and addend values, a and c, specified on the previous page.

SEE ALSO

rand

RETURN VALUE

Upon successful completion a non-negative integer, namely the file descriptor, is returned. Otherwise, a value of -1 is returned and errno is set to indicate the error.

SEE ALSO

creat(2), close(2), exec(2), open(2),

NAME

dup - duplicate an open file descriptor

SYNOPSIS

```
int dup (fildes)
int fildes;
```

DESCRIPTION

Fildes is a file descriptor obtained from a creat, open, or dup, system call. Dup returns a new file descriptor having the following in common with the original:

Same open file.

Same file pointer. (i.e., both file descriptors share one file pointer.)

Same access mode (read, write or read/write).

The new file descriptor is set to remain open across exec system calls.

The file descriptor returned is the lowest one available.

Dup will fail if one or more of the following are true:

Fildes is not a valid open file descriptor. [EBADF]

Twenty (2Ø) file descriptors are currently open. [EMFILE]

SEE ALSO

printf

BUGS

The return values point to static data whose content is overwritten by each call.

NAME

ecvt, fcvt, gcvt - convert floating-point number to string

SYNOPSIS

```
char *ecvt (value, ndigit, decpt, sign)
double value;
int ndigit, *decpt, *sign;

char *fcvt (value, ndigit, decpt, sign)
double value;
int ndigit, *decpt, *sign;

char *gcvt (value, ndigit, buf)
double value;
char *buf;
```

DESCRIPTION

Ecvt converts value to a null-terminated string of ndigit digits and returns a pointer thereto. The low-order digit is rounded. The position of the decimal point relative to the beginning of the string is stored indirectly through decpt (negative means to the left of the returned digits). The decimal point is not included in the returned string. If the sign of the result is negative, the word pointed to by sign is non-zero, otherwise it is zero. Fcvt is identical to ecvt, except that the correct digit has been rounded for Fortran F-format output of the number of digits specified by ndigit.

Gcvt converts the value to a null-terminated string in the array pointed to by buf and returns buf. It attempts to produce ndigit significant digits in Fortran F-format if possible, otherwise E-format, ready for printing. A minus sign, if there is one, or a decimal point will be included as part of the returned string. Trailing zeros are suppressed.

NAME

erf, erfc - error function and complementary error function

SYNOPSIS

```
#include <math.h>

double erf (x)
double x;

double erfc (x)
double x;
```

DESCRIPTION

Erf returns the error function of x, defined as {2 over sqrt pi} int from Ø to x e sup {- t sup 2} dt .

Erfc, which returns 1.Ø - erf(x), is provided because of the extreme loss of relative accuracy if erf(x) is called for large x and the result subtracted from 1.Ø (e.g. for x = 5, 12 places are lost).

SEE ALSO

exp

NAME

erf, erfc - error function and complementary error function

SYNOPSIS

See erf

When a C program is executed, it is called as follows:

```
     main (argc, argv, envp)
     int argc;
     char **argv, **envp;
```

where argc is the argument count and argv is an array of character pointers to the arguments themselves. As indicated, argc is conventionally at least one and the first member of the array points to a string containing the name of the file.

Path points to a path name that identifies the new process file. File points to the new process file.

Arg0, argl, ..., arg_ are pointers to null-terminated character strings. These strings constitute the argument list available to the new process. By convention, at least arg0 must be present and point to a string that is the same as path (or its last component).

Argv is an array of character pointers to null-terminated strings. These strings constitute the argument list available to the new process. By convention, argv must have at least one member, and it must point to a string that is the same as path (or its last component). Argv is terminated by a null pointer.

Exec will fail and return to the calling process if one or more of the following are true:

One or more components of the new process file's path name do not exist. [ENOENT]

A component of the new process file's path prefix is not a directory. [ENOTDIR]

Search permission is denied for a directory listed in the new process file's path prefix. [EACCES]

The new process file mode denies execution permission. [EACCES]

The exec is not an execlp or execvp, and the new process file has the appropriate access permission but an invalid magic number in its header. [ENOEXEC]

The new process file is a pure procedure (shared text) file that is currently open for writing by some process. [ETXTBSY]

The new process requires more memory than is allowed by the system-imposed maximum MAXMEM. [ENOMEM]

NAME

exec, execl, execv, execle, execve, execlp, execvp - execute a file

SYNOPSIS

```
int execl (path, arg0, argl, ..., argn, 0)
char *path, *arg0, *argl, ..., *argn;

int execv (path, argv)
char *path, *argv[ ];

int execle (path, arg0, argl, ..., argn, 0, envp)
char *path, *arg0, *argl, ..., *argn, *envp[ ];

int execve (path, argv, envp)
char *path, *argv[ ], *envp[ ];

int execlp (file, arg0, argl, ..., argn, 0)
char *file, *arg0, *argl, ..., *argn;

int execvp (file, argv)
char *file, *argv[ ];
```

DESCRIPTION

Exec creates a new process. The new process is constructed from an ordinary file called the new process file. This file is an executable object file, An executable object file consists of a header a text segment, and a data segment. The data segment contains an initialized portion and an uninitialized portion (bss).

The number of bytes in the new process's argument list is greater than the system-imposed limit of 5120 bytes. [E2BIG]

The new process file is not as long as indicated by the size values in its header. [EFAULT]

Path, argv, or envp point to an illegal address. [EFAULT]

RETURN VALUE

If exec returns an error, the return value will be -1 and errno will be set to indicate the error.

SEE ALSO

exit

NAME

exit, _exit - terminate process

SYNOPSIS

```
void exit (status)
int status;
void _exit (status)
int status;
```

DESCRIPTION

Exit terminates the calling process with the following consequences:

All of the file descriptors open in the calling process are closed.

Pow returns x to the y power. The values of x and y may not both be zero. If x is non-positive, y must be an integer.

Sqrt returns the square root of x. The value of x may not be negative.

DIAGNOSTICS

Exp returns HUGE when the correct value would overflow, and sets errno to ERANGE.

Log and log10 return 0 and set errno to EDOM when x is non-positive An error message is printed on the standard error output.

Pow returns 0 and sets errno to EDOM when x is non-positive and y is not an integer, or when x and y are both zero. In these cases a message indicating DOMAIN error is printed on the standard error output. When the correct value for pow would overflow, pow returns HUGE and sets errno to ERANGE.

Sqrt returns 0 and sets errno to EDOM when x is negative. A message indicating DOMAIN error is printed on the standard error output.

SEE ALSO

---

hypot, sinh

NAME

exp, log, log1Ø, pow, sqrt - exponential, logarithm, power, square root functions

SYNOPSIS

```
#include <math.h>

double exp (x)
double x;

double log (x)
double x;

double log1Ø (x)
double x;

double pow (x, y)
double x, y;

double sqrt (x)
double x;
```

DESCRIPTION

Exp returns e to the x power.

Log returns the natural logarithm of x. The value of x must be positive.

Log1Ø returns the logarithm base ten of x. The value of x must be positive.

NAME

fabs, floor, ceil, fmod, - floor, ceiling, remainder, absolute value functions

SYNOPSIS

See floor

SEE ALSO

close, exit, fopen, setbuf

NAME

fclose, fflush - close or flush a stream

SYNOPSIS

```
#include <stdio.h>

int fclose (stream)
FILE *stream;

int fflush (stream)
FILE *stream;
```

DESCRIPTION

Fclose causes any buffered data for the named stream to be written out, and the stream to be closed.

Fclose is performed automatically for all open files upon calling exit(2).

Fflush causes any buffered data for the named stream to be written to that file. The stream remains open.

DIAGNOSTICS

These functions return Ø for success, and EOF if any error (such as trying to write to a file that has not been opened for writing) was detected.

NAME

fdopen, freopen, fopen - open a stream

SYNOPSIS

See fopen

NAME

feof, ferror, clearerr, fileno - stream status inquiries

SYNOPSIS

See ferror

Fileno returns the integer file descriptor associated with the named stream; see open(2).

NOTE

All these functions are implemented as macros; they cannot be declared or redeclared.

SEE ALSO

---

open, fopen

NAME

ferror, feof, clearerr, fileno - stream status inquiries

SYNOPSIS

```
#include <stdio.h>

int feof (stream)
FILE
*stream;

int ferror (stream)
FILE
*stream;

void clearerr (stream)
FILE
*stream;

int fileno(stream)
FILE
*stream;
```

DESCRIPTION

Feof returns non-zero when EOF has previously been detected reading the named input stream, otherwise zero.

Ferror returns non-zero when an I/O error has previously occurred reading from or writing to the named stream, otherwise zero.

Clearerr resets the error indicator and EOF indicator to zero on the named stream.

NAME

fflush, fclose, - close or flush a stream

SYNOPSIS

See fclose

NAME

fgetc, getchar, getc, getw - get character or word from stream

SYNOPSIS

See getc

NAME

fgets, gets - get a string from a stream

SYNOPSIS

See gets

NAME

fileno, ferror, feof, clearerr, - stream status inquiries

SYNOPSIS

See ferror

SEE ALSO

---

abs

NAME

floor, ceil, fmod, fabs - floor, ceiling, remainder, absolute value functions

SYNOPSIS

```
#include <math.h>

double floor (x)
double x;

double ceil (x)
double x;

double fmod (x, y)
double x, y;

double fabs (x)
double x;
```

DESCRIPTION

Floor returns the largest integer (as a double-precision number) not greater than x.

Ceil returns the smallest integer not less than x.

Fmod returns x if y is zero, otherwise the number f with the same sign as x, such that x = iy + f for some integer i, and |f| < |y|.

Fabs returns |x|.

NAME

fmod, ceil, floor, fabs - floor, ceiling, remainder, absolute value functions

SYNOPSIS

See floor

"a"	append; open for writing at end of file, or create for writing
"r+"	open for update (reading and writing)
"w+"	truncate or create for update
"a+"	append; open or create for update at end-of-file

Freopen substitutes the named file in place of the open stream. The original stream is closed, regardless of whether the open ultimately succeeds. Freopen returns a pointer to the FILE structure associated with stream.

Freopen is typically used to attach the preopened streams associated with stdin, stdout and stderr to other files.

Fdopen associates a stream with a file descriptor obtained from open, dup, or creat, which will open files but not return pointers to a FILE structure stream which are necessary input for many of the section 3S library routines. The type of stream must agree with the mode of the open file.

When a file is opened for update, both input and output may be done on the resulting stream. However, output may not be directly followed by input without an intervening fseek or rewind, and input may not be directly followed by output without an intervening fseek, rewind, or an input operation which encounters end-of-file.

When a file is opened for append (i.e., when type is "a" or "a+"), it is impossible to overwrite information already in the file. Fseek may be used to reposition the file pointer to any position in the file, but when output is written to the file the current file pointer is disregarded. All output is written at the end of the file and causes the file pointer to be repositioned at the end of the output. If two separate processes open the same file for append, each process may write freely to the file without fear of destroying output being written by the other. The output from the two processes will be intermixed in the file in the order in which it is written.

SEE ALSO

open, fclose

DIAGNOSTICS

Fopen and freopen return a NULL pointer on failure.

NAME

fopen, freopen, fdopen - open a stream

SYNOPSIS

```
#include <stdio.h>

FILE *fopen (file-name, type)
char *file-name, *type;

FILE *freopen (file-name, type, stream)
char *file-name, *type;
FILE *stream;

FILE *fdopen (fildes, type)
int fildes;
char *type;
```

DESCRIPTION

Fopen opens the file named by file-name and associates a stream with it. Fopen returns a pointer to the FILE structure associated with the str_am.

File-name points to a character string that contains the name of the file to be opened.

Type is a character string having one of the following values:

"r"	open for reading
"w"	truncate or create for writing

NAME

fprintf, printf, sprintf - print formatted output

SYNOPSIS

See printf

NAME

fputc, putchar, putc, putw - put character or word on a stream

SYNOPSIS

See putc

NAME

fputs, puts - put a string on a stream

SYNOPSIS

See puts

The variable size is typically sizeof(*ptr) where the pseudo-function sizeof specifies the length of an item pointed to by ptr. If ptr points to a data type other than char it should be cast into a pointer to char.

SEE ALSO

read, write, fopen, getc, gets, printf, putc, puts, scanf

DIAGNOSTICS

Fread and fwrite return the number of items read or written. If nitems is non-positive, no characters are read or written and Ø is returned by both fread and fwrite.

NAME

fread, fwrite - binary input/output

SYNOPSIS

```
#include <stdio.h>

int fread (ptr, size, nitems, stream)
char *ptr;
int size, nitems;
FILE *stream;

int fwrite (ptr, size, nitems, stream)
char *ptr;
int size, nitems;
FILE *stream;
```

DESCRIPTION

Fread copies, into an array beginning at ptr, nitems items of data from the named input stream, where an item of data is a sequence of bytes (not necessarily terminated by a null byte) of length size. Fread stops appending bytes if an end-of-file or error condition is encountered while reading stream, or if nitems items have been read. Fread leaves the file pointer in stream, if defined, pointing to the byte following the last byte read if there is one. Fread does not change the contents of stream.

Fwrite appends at most nitems items of data from the the array pointed to by ptr to the named output stream. _write stops appending when it has appended nitems items of data or if an error condition is encountered on stream. Fwrite does not change the contents of the array pointed to by ptr.

NAME

freopen, fopen, fdopen - open a stream

SYNOPSIS

See fopen

NAME

frexp, ldexp, modf - manipulate parts of floating-point numbers

SYNOPSIS

```
double frexp (value, eptr)
double value;
int *eptr;

double ldexp (value, exp)
double value;
int exp;

double modf (value, iptr)
double value, *iptr;
```

DESCRIPTION

Every non-zero number can be written uniquely as x* 2^n, where the mantissa (fraction) x is in the range Ø.5 _ returns the mantissa of a double value, and stores the exponent indirectly in the location pointed to by eptr.

Ldexp returns the quantity value* 2 exp.

Modf returns the signed fractional part of value and stores the integral part indirectly in the location pointed to by iptr.

DIAGNOSTICS

If ldexp would cause overflow, HUGE is returned and errno is set to ERANGE.

fscanf

NAME

fscanf, scanf, sscanf - convert formatted input

SYNOPSIS

See scanf

Ftell returns the offset of the current byte relative to the beginning of the file associated with the named stream.

SEE ALSO

lseek, fopen, ungetc

DIAGNOSTICS

Fseek returns non-zero for improper seeks, otherwise zero. An improper seek can be, for example, an fseek done on a file that has not been opened via fopen; in particular, fseek may not be used on a terminal.

WARNING

Although on the UNIX System an offset returned by ftell is measured in bytes, and it is permissible to seek to positions relative to that offset, portability to non-UNIX Systems requires that an offset be used by fseek directly. Arithmetic may not meaningfully be performed on such a offset, which is not necessarily measured in bytes.

NAME

fseek, rewind, ftell - reposition a file pointer in a stream

SYNOPSIS

```
#include <stdio.h>

int fseek (stream, offset, ptrname)
FILE *stream;
long offset;
int ptrname;

void rewind (stream)
FILE *stream;

long ftell (stream)
FILE *stream;
```

DESCRIPTION

Fseek sets the position of the next input or output operation on the stream. The new position is at the signed distance offset bytes from the beginning, from the current position, or from the end of the file, according as ptrname has the value Ø, 1, or 2.

Rewind(stream) is equivalent to fseek(stream, ØL, Ø), except that no value is returned.

Fseek and rewind undo any effects of ungetc(3S).

After fseek or rewind, the next operation on a file opened for update may be either input or output.

NAME

ftell, rewind, fseek - reposition a file pointer in a stream

SYNOPSIS

See fseek

NAME

fwrite, fread, - binary input/output

SYNOPSIS

See fread

If the correct value would overflow, gamma returns HUGE and sets errno to ERANGE.

SEE ALSO

---

exp

NAME

gamma - log gamma function

SYNOPSIS

```
#include <math.h>

extern int signgam;

double gamma (x)
double x;
```

DESCRIPTION

Gamma returns ln (GAMMA(x)), where GAMMA(x) is defined as int from Ø to inf e sup { - t} t sup { x - 1 } dt. The sign of GAMMA(x) is returned in the external integer signgam. The argument x may not be a non-positive integer.

The following C program fragment might be used to calculate GAMMA:

```
     if ((y = gamma(x)) > LOGHUGE)
             error();
     y = signgam * exp(y);
```

where LOGHUGE is the least value that causes exp(3M) to return a range error.

DIAGNOSTICS

For non-negative integer arguments HUGE is returned, and errno is set to EDOM. A message indicating DOMAIN error is printed on the standard error output.

constant EOF upon end-of-file or error, but as that is a valid integer value, feof and ferror(3S) should be used to check the success of getw. Getw increments the associated file pointer, if defined, to point to the next word. Getw assumes no special alignment in the file.

SEE ALSO

fclose(3S), ferror(3S), fopen(3S), fread(3S), gets(3S), putc(3S), scanf(3S).

DIAGNOSTICS

These functions return the integer constant EOF at end-of-file or upon an error.

BUGS

Because it is implemented as a macro, getc treats incorrectly a stream argument with side effects. In particular, getc(*f++) doesn't work sensibly. Fgetc should be used instead. Because of possible differences in word length and byte ordering, files written using putw are machine-dependent, and may not be read using getw on a different processor.

NAME

getc, getchar, fgetc, getw - get character or word from stream

SYNOPSIS

```
#include <stdio.h>

int getc (stream)
FILE *stream;

int getchar ()

int fgetc (stream)
FILE *stream;

int getw (stream)
FILE *stream;
```

DESCRIPTION

Getc returns the next character (i.e. byte) from the named input stream. It also moves the file pointer, if defined, ahead one character in stream. Getc is a macro and so cannot be used if a function is necessary; for example one cannot have a function pointer point to it.

Getchar returns the next character from the standard input stream, stdin. As in the case of getc, getchar is a macro.

Fgetc performs the same function as getc, but is a genuine function. Fgetc runs more slowly than getc, but takes less space per invocation.

Getw returns the next word (i.e. integer) from the named input stream. The size of a word varies from machine to machine. It returns the

getchar

NAME

getchar, getc, fgetc, getw - get character or word from stream

SYNOPSIS

See getc

SEE ALSO

malloc

DIAGNOSTICS

Returns NULL with errno set if s_ze is not large enough, or if an error ocurrs in a lower-level function.

NAME

getcwd - get path-name of current working directory

SYNOPSIS

```
char *getcwd (buf, size)
char *buf;
int size;
```

DESCRIPTION

Getcwd returns a pointer to the current directory path-name. The value of size must be at least two greater than the length of the path-name to be returned.

If buf is a NULL pointer, getcwd will obtain size bytes of space using malloc(3C). In this case, the pointer returned by getcwd may be used as the argument in a subsequent call to free.

EXAMPLE

```
char *cwd, *getcwd();
.
.
.
if ((cwd = getcwd((char *)NULL, 64)) == NULL) {
perror("pwd");
exit(1);
}
printf("%"s\n, cwd);
```

NAME

getlogin - get login name

SYNOPSIS

```
char *getlogin ( );
```

DESCRIPTION

Getlogin returns a pointer to the login name.

If getlogin is called within a process that is not attached to a terminal, it returns a NULL pointer. The correct procedure for determining the login name is to call cuserid.

SEE ALSO

cuserid

DIAGNOSTICS

Returns the NULL pointer if name not found.

BUGS

The return values point to static data whose content is overwritten by each call.

DIAGNOSTICS

Getopt prints an error message on stderr and returns a question mark (?) when it encounters an option letter not included in optstring.

WARNING

The above routine uses <stdio.h>, which causes it to increase the size of programs, not otherwise using standard I/O, more than might be expected.

getopt

NAME

getopt - get option letter from argument vector

SYNOPSIS

```
int getopt (argc, argv, optstring)
int argc;
char **argv;
char *optstring;

extern char *optarg;
extern int optind;
```

DESCRIPTION

Getopt returns the next option letter in argv that matches a letter in optstring. Optstring is a string of recognized option letters; if a letter is followed by a colon, the option is expected to have an argument that may or may not be separated from it by white space. Optarg is set to point to the start of the option argument on return from getopt.

Getopt places in optind the argv index of the next argument to be processed. Because optind is external, it is normally initialized to zero automatically before the first call to getopt.

When all options have been processed (i.e., up to the first non-option argument), getopt returns EOF. The special option -- may be used to delimit the end of the options; EOF will be returned, and -- will be skipped.

EXAMPLE

The following code fragment shows how one might process the arguments for a command that can take the mutually exclusive options a and b, and the options f and o, both of which require arguments:

```
main (argc, argv)
int argc;
char **argv;
{
     int c;
     extern int optind;
     extern char *optarg;
      .
      .
      .
     while ((c = getopt (argc, argv, "abf:o:")) != EOF)
          switch (c) {
          case 'a':
               if (bflg)
                    errflg++;
               else
                    aflg++;
               break;
          case 'b':
               if (aflg)
                    errflg++;
               else
                    bproc( );
               break;
          case 'f':
               ifile = optarg;
               break;
          case 'o':
               ofile = optarg;
               bufsiza = 512;
               break;
          case '?':
               errflg++;
          }
     if (errflg) {
          fprintf (stderr, "usage: . . . ");
          exit (2);
     }
     for ( ; optind < argc; optind++) {
          if (access (argv[optind], 4)) {
      .
      .
      .
}
```

getpid

NAME

getpid, getpgrp, getppid - get process, process group, and parent process IDs

SYNOPSIS

```
int getpid ()

int getpgrp ()

int getppid ()
```

DESCRIPTION

Getpid returns the process ID of the calling process.

Getpgrp returns the process group ID of the calling process.

Getppid returns the parent process ID of the calling process.

SEE ALSO

exec

DIAGNOSTICS

If end-of-file is encountered and no characters have been read, no characters are transferred to s and a NULL pointer is returned. If a read error occurs, such as trying to use these functions on a file that has not been opened for reading, a NULL pointer is returned. Otherwise s is returned.

NAME

gets, fgets - get a string from a stream

SYNOPSIS

```
#include <stdio.h>

char *gets (s)
char *s;

char *fgets (s, n, stream)
char *s;
int n;
FILE *stream;
```

DESCRIPTION

Gets reads characters from the standard input stream, stdin, into the array pointed to by s, until a new-line character is read or an end-of-file condition is encountered. The new-line character is discarded and the string is terminated with a null character.

Fgets reads characters from the stream into the array pointed to by s, until n-1 characters are read, or a new-line character is read and transferred to s, or an end-of-file condition is encountered. The string is then terminated with a null character.

SEE ALSO

ferror, fopen, fread, getc, scanf

NAME

getuid, getgid, - set user and group IDs

SYNOPSIS

```
unsigned short getuid ()

unsigned short getgid ()
```

DESCRIPTION

Getuid returns the user ID of the calling process.

Getgid returns the group ID of the calling process.

NAME

getw, getchar, fgetc, getc - get character or word from stream

SYNOPSIS

See getc

entries that the table will contain. This number may be adjusted upward by the algorithm in order to obtain certain mathematically favorable circumstances.

Hdestroy destroys the search table, and may be followed by another call to hcreate.

SEE ALSO

bsearch(3C), lsearch(3C), string(3C), tsearch(3C).

DIAGNOSTICS

Hsearch returns a NULL pointer if either the action is FIND and the item could not be found or the action is ENTER and the table is full.

Hcreate returns zero if it cannot allocate sufficient space for the table.

BUGS

Only one hash search table may be active at any given time.

hsearch

NAME

hsearch, hcreate, hdestroy - manage hash search tables

SYNOPSIS

```
#include <search.h>

ENTRY *hsearch (item, action)
ENTRY item;
ACTION action;

int hcreate (nel)
unsigned nel;

void hdestroy ( )
```

DESCRIPTION

Hsearch is a hash-table search routine generalized from Knuth (6.4) Algorithm D. It returns a pointer into a hash table indicating the location at which an entry can be found. Item is a structure of type ENTRY (defined in the <search.h> header file) containing two pointers: item.key points to the comparison key, and item.data points to any other data to be associated with that key. (Pointers to types other than character should be cast to pointer-to-character.) Action is a member of an enumeration type ACTION indicating the disposition of the entry if it cannot be found in the table. ENTER indicates that the item should be inserted in the table at an appropriate point. FIND indicates that no entry should be made. Unsuccessful resolution is indicated by the return of a NULL pointer.

Hcreate allocates sufficient space for the table, and must be called before hsearch is used. _el is an estimate of the maximum number of

NAME

hypot - Euclidean distance function

SYNOPSIS

```
#include <math.h>

double hypot (x, y)
double x, y;
```

DESCRIPTION

Hypot returns

```
     sqrt(x * x + y * y),
```

taking precautions against unwarranted overflows.

DIAGNOSTICS

When the correct value would overflow, hypot returns HUGE and sets errno to ERANGE.

SEE ALSO

sqrt

NAME

jØ, j1, jn, yØ, y1, yn - Bessel functions

SYNOPSIS

See bessel

NAME

jl, j0, jn, y0, yl, yn - Bessel functions

SYNOPSIS

See bessel

NAME

jn, jl, jØ, yØ, yl, yn - Bessel functions

SYNOPSIS

See bessel

NAME

l3tol, ltol3 - convert between 3-byte integers and long integers

SYNOPSIS

```
void l3tol (lp, cp, n)
long *lp;
char *cp;
int n;

void ltol3 (cp, lp, n)
char *cp;
long *lp;
int n;
```

Description

L3tol converts a list of n three-byte integers packed into a character string pointed to by cp into a list of long integers pointed to by lp.

Ltol3 performs the reverse conversion from long integers (lp) to three-byte integers (cp).

These functions are useful for file-system maintenance where the block numbers are three bytes long.

Bugs

Because of possible differences in byte ordering, the numerical values of the long integers are machine-dependent.

NAME

l64a, a64l - convert between long integer and base-64 ASCII string

SYNOPSIS

See a64l

NAME

log, exp, log1Ø, pow, sqrt - exponential, logarithm, power, square root functions

SYNOPSIS

See exp

NAME

logl0, exp, log, pow, sqrt - exponential, logarithm, power, square root functions

SYNOPSIS

See exp

NOTES

The pointers to the key and the element at the base of the table should be of type pointer-to-element, and cast to type pointer-to-character. The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared. Although declared as type pointer-to-character, the value returned should be cast into type pointer-to-element.

SEE ALSO

bsearch(3C), hsearch(3C), tsearch(3C).

Diagnostics

If the searched-for datum is found, both lsearch and lfind return a pointer to it. Otherwise, lfind returns NULL and lsearch returns a pointer to the newly added element.

BUGS

Undefined results can occur if there is not enough room in the table to add a new item.

lsearch

NAME

lsearch - linear search and update

SYNOPSIS

```
#include <stdio.h> #include <search.h>

char *lsearch ((char *)key, (char *)base, nelp,
sizeof(*key), compar)
unsigned *nelp;
int (*compar)( );

char *lfind ((char *)key, (char *)base, nelp,
compar)
unsigned *nelp; int (*compar)( );
```

DESCRIPTION

Lsearch is a linear search routine generalized from Knuth (6.1) Algorithm S. It returns a pointer into a table indicating where a datum may be found. If the datum does not occur, it is added at the end of the table. Key points to the datum to be sought in the table. Base points to the first element in the table. Nelp points to an integer containing the current number of elements in the table. The integer is incremented if the datum is added to the table. Compar is the name of the comparison function which the user must supply (strcmp, for example). It is called with two arguments that point to the elements being compared. The function must return zero if the elements are equal and non-zero otherwise.

The resulting file pointer would be negative. [EINVAL]

Some devices are incapable of seeking. The value of the file pointer associated with such a device is undefined.

RETURN VALUE

Upon successful completion, a non-negative integer indicating the file pointer value is returned. Otherwise, a value of -1 is returned and errno is set to indicate the error.

SEE ALSO

creat(2), dup(2), open(2)

NAME

lseek - move read/write file pointer

SYNOPSIS

```
long lseek (fildes, offset, whence)
int fildes;
long offset;
int whence;
```

DESCRIPTION

Fildes is a file descriptor returned from a creat, open, or dup system call. Lseek sets the file pointer associated with fildes as follows:

> If whence is Ø, the pointer is set to offset bytes.
>
> If whence is 1, the pointer is set to its current location plus offset.
>
> If whence is 2, the pointer is set to the size of the file plus offset.

Upon successful completion, the resulting pointer location as measured in bytes from the beginning of the file is returned.

Lseek will fail and the file pointer will remain unchanged if one or more of the following are true:

> Fildes is not an open file descriptor. [EBADF]
>
> Whence is not Ø, 1 or 2. [EINVAL]

coalescing adjacent free blocks as it searches. It calls sbrk (see brk(2)) to get more memory from the system when there is no suitable space already free.

Realloc changes the size of the block pointed to by ptr to size bytes and returns a pointer to the (possibly moved) block. The contents will be unchanged up to the lesser of the new and old sizes. If no free block of size bytes is available in the storage arena, then realloc will ask malloc/^ to enlarge the arena by size bytes and will then move the data to the new space.

Realloc also works if ptr points to a block freed since the last call of malloc, realloc, or calloc; thus sequences of free, malloc and realloc can exploit the search strategy of malloc to do storage compaction.

Calloc allocates space for an array of n_lem elements of size elsize. The space is initialized to zeros.

Each of the allocation routines returns a pointer to space suitably aligned (after possible pointer coercion) for storage of any type of object.

DIAGNOSTICS

Malloc, realloc and calloc return a NULL pointer if there is no available memory or if the arena has been detectably corrupted by storing outside the bounds of a block. When this happens the block pointed to by ptr may be destroyed.

NOTES

Search time increases when many objects have been allocated; that is, if a program allocates but never frees, then each successive allocation takes longer.

NAME

malloc, free, realloc, calloc - main memory allocator

SYNOPSIS

```
char *malloc (size)
unsigned size;

void free (ptr)
char *ptr;

char *realloc (ptr, size)
char *ptr;
unsigned size;

char *calloc (nelem, elsize)
unsigned nelem, elsize;
```

DESCRIPTION

Malloc and free provide a simple general-purpose memory allocation package. Malloc returns a pointer to a block of at least size bytes suitably aligned for any use.

The argument to free is a pointer to a block previously allocated by malloc; after free is performed this space is made available for further allocation, but its contents are left undisturbed.

Undefined results will occur if the space assigned by malloc is overrun or if some random number is handed to free.

Malloc allocates the first big enough contiguous reach of free space found in a circular search from the last block allocated or freed, Memccpy copies characters from memory area s2 into s1, stopping after the first occurrence of character c has been copied, or after n characters have been copied, whichever comes first. It returns a pointer to the character after the copy of c in s1, or a NULL pointer if c was not found in the first n characters of s2.

Memchr returns a pointer to the first occurrence of character c in the first n characters of memory area s, or a NULL pointer if c does not occur.

Memcmp compares its arguments, looking at the first n characters only, and returns an integer less than, equal to, or greater than Ø, according as s1 is lexicographically less than, equal to, or greater than s2.

Memcpy copies n characters from memory area s2 to s1. It returns s1.

Memset sets the first n characters in memory area s to the value of character c. It returns s .

NOTE

For user convenience, all these functions are declared in the optional <memory.h> header file.

BUGS

Memcmp uses native character comparison, which is signed on MC68ØØØs and PDP-11s, unsigned on other machines.

Character movement is performed differently in different implementations. Thus overlapping moves may yield surprises.

NAME

memory, memccpy, memchr, memcmp, memcpy, memset - memory operations

SYNOPSIS

```
#include <memory.h>

char *memccpy (s1, s2, c, n)
char *s1, *s2;
int c, n;

char *memchr (s, c, n)
char *s;
int c, n;

int memcmp (s1, s2, n)
char *s1, *s2;
int n;

char *memcpy (s1, s2, n)
char *s1, *s2;
int n;

char *memset (s, c, n)
char *s;
int c, n;
```

DESCRIPTION

These functions operate efficiently on memory areas (arrays of characters bounded by a count, not terminated by a null character). They do not check for the overflow of any receiving memory area.

NAME

mktemp - make a unique file name

SYNOPSIS

```
char *mktemp (template)
char *template;
```

DESCRIPTION

Mktemp replaces the contents of the string pointed to by template by a unique file name, and returns the address of template. The string in template should look like a file name with six trailing Xs; mktemp will replace the Xs with a letter and the current process ID. The letter will be chosen so that the resulting name does not duplicate an existing file.

SEE ALSO

getpid(2), tmpfile(3S), tmpnam(3S).

BUGS

It is possible to run out of letters.

O_EXCL If O_EXCL and O_CREAT are set, open will fail if the file exists.

Upon successful completion a non-negative integer, the file descriptor, is returned.

The file pointer used to mark the current position within the file is set to the beginning of the file.

The new file descriptor is set to remain open across exec system calls.

No process may have more than 2Ø file descriptors open simultaneously.

The named file is opened unless one or more of the following are true:

A component of the path prefix is not a directory. [ENOTDIR]

O_CREAT is not set and the named file does not exist. [ENOENT]

A component of the path prefix denies search permission. [EACCES]

Oflag permission is denied for the named file. [EACCES]

The named file is a directory and oflag is write or read/write. [EISDIR]

Twenty (2Ø) file descriptors are currently open. [EMFILE]

Path points outside the process's allocated address space. [EFAULT]

O_CREAT and O_EXCL are set, and the named file exists. [EEXIST]

The system file table is full. [ENFILE]

RETURN VALUE

Upon successful completion, a non-negative integer, namely a file descriptor, is returned. Otherwise, a value of -1 is returned and errno is set to indicate the error.

SEE ALSO

close, creat, dup, lseek, read, write

NAME

open - open for reading or writing

SYNOPSIS

```
#include <fcntl.h>
int open (path, oflag [ , mode ] )
char *path;
int oflag, mode;
```

DESCRIPTION

Path points to a path name naming a file. Open opens a file descriptor for the named file and sets the file status flags according to the value of oflag. Oflag values are constructed by or-ing flags from the following list (only one of the first three flags below may be used):

O_RDONLY Open for reading only.

O_WRONLY Open for writing only.

O_RDWR Open for reading and writing.

O_NDELAY This flag may affect subsequent reads and writes. See read and write.

O_APPEND If set, the file pointer will be set to the end of the file prior to each write.

O_CREAT If the file exists, a new version is created.

O_TRUNC If the file exists, its length is truncated to Ø and the mode and owner are unchanged.

point to are located in the text segment so that they can be shared in programs that are loaded pure. This implies that they are read-only in pure programs and any attempts to change them will cause memory faults. However, the sys_errlist array itself is in the data aegment and hence the pointers may be changed.

NAME

perror, errno, sys_errlist, sys_nerr - system error messages

SYNOPSIS

```
void perror (s)
char *s;

extern int errno;

extern char *sys_errlist[];

extern int sys_nerr;
```

DESCRIPTION

Perror produces a message on the standard error output, describing the last error encountered during a call to a system or library function. The argument string s is printed first, then a colon and a blank, then the message and a new-line. To be of most use, the argument string should include the name of the program that incurred the error. The error number is taken from the external variable errno, which is set when errors occur. It is not cleared when non-erroneous calls are made.

To simplify variant formatting of messages, the array of message strings sys_errlist is provided. errno can be used as an index in this table to get the message string without the new-line. sys_nerr is the largest message number provided for in the table. It should be checked because new error codes may be added to the system before they are added to the table.

WARNING

The text of the error messages that the pointers in the array sys_errlist

NAME

pow, exp, log, log1Ø, sqrt - exponential, logarithm, power, square root functions

SYNOPSIS

See exp

Each conversion specification is introduced by the character %. After the %, the following appear in sequence:

Zero or more flags, which modify the meaning of the conversion specification.

An optional decimal digit string specifying a minimum field width. If the converted value has fewer characters than the field width, it will be padded on the left (or right, if the left-adjustment flag (see below) has been given) to the field width;

A precision that gives the minimum number of digits to appear for the d, o, u, x, or X conversions, the number of digits to appear after the decimal point for the e and f conversions, the maximum number of significant digits for the g conversion, or the maximum number of characters to be printed from a string in s conversion. The precision takes the form of a period (.) followed by a decimal digit string: a null digit string is treated as zero.

An optional l specifying that a following d, o, u, x, or X conversion character applies to a long integer arg.

A character that indicates the type of conversion to be applied.

A field width or precision may be indicated by an asterisk (*) instead of a digit string. In this case, an integer arg supplies the field width or precision. The arg that is actually converted is not fetched until the conversion letter is seen, so the args specifying field width or precision must appear before the arg (if any) to be converted.

The flag characters and their meanings are:

-	The result of the conversion will be left-justified within the field.
+	The result of a signed conversion will always begin with a sign (+ or -).
blank	If the first character of a signed conversion is not a sign, a blank will be prefixed to the result. This implies that if the blank and + flags both appear, the blank flag will be ignored.
#	This flag specifies that the value is to be converted to an alternate form. For c, d, s, and u conversions, the flag has no effect. For o conversion, it increases the precision to force the first digit of the result to be a zero. For x (X) conversion, a non-zero result will have Øx (ØX) prefixed to it. For e, E, f, g, and G conversions, the result will always contain a decimal

NAME

printf, fprintf, sprintf - print formatted output

SYNOPSIS

```
#include <stdio.h>

int printf (format [ , arg ] ...  )
char *format;

int fprintf (stream, format [ , arg ] ...  )
FILE *stream;
char *format;

int sprintf (s, format [ , arg ] ...  )
char *s, format;
```

DESCRIPTION

Printf places output on the standard output stream stdout. Fprintf places output on the named output stream. Sprintf places output, followed by the null character (\Ø) in consecutive bytes starting at *s; it is the user's responsibility to ensure that enough storage is available. Each function returns the number of characters transmitted (not including the \Ø in the case of sprintf), or a negative value if an output error was encountered.

Each of these functions converts, formats, and prints its args under control of the format. The format is a character string that contains two types of objects: plain characters, which are simply copied to the output stream, and conversion specifications, each of which results in fetching of zero or more args. The results are undefined if there are insufficient args for the format. If the format is exhausted while args remain, the excess args are simply ignored.

If the string pointer arg has the value zero, the result is undefined. A null arg will yield undefined results.

% Print a %; no argument is converted.

In no case does a non-existent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is simply expanded to contain the conversion result. Characters generated by printf and fprintf are printed as if putc(3S) had been called.

EXAMPLES

To print a date and time in the form Sunday, July 3, 10:02, where weekday and month are pointers to null-terminated strings:

```
printf("%s, %s %d, %.2d:%.2d", weekday, month, day, hour, min);
```

To print pi to 5 decimal places:

```
printf("pi = %.5f", 4*atan(1.0));
```

SEE ALSO

ecvt, putc, scanf, stdio

point, even if no digits follow the point (normally, a decimal point appears in the result of these conversions only if a digit follows it). For g and G conversions, trailing zeroes will not be removed from the result (which they normally are).

The conversion characters and their meanings are:

d,o,u,x,X	The integer arg is converted to signed decimal, unsigned octal, decimal, or hexadecimal notation (x and X), respectively; the letters abcdef are used for x conversion and the letters ABCDEF for X conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeroes. The default precision is 1. The result of converting a zero value with a precision of zero is a null string.
f	The float or double arg is converted to decimal notation in the style [-]ddd.ddd, where the number of digits after the decimal point is equal to the precision specification. If the precision is missing, 6 digits are output; if the precision is explicitly Ø, no decimal point appears.
e,E	The float or double arg is converted in the style [-]d.ddde_dd, where there is one digit before the decimal point and the number of digits after it is equal to the precision; when the precision is missing, 6 digits are produced; if the precision is zero, no decimal point appears. The E format code will produce a number with E instead of e introducing the exponent. The exponent always contains at least two digits.
g,G	The float or double arg is printed in style f or e (or in style E in the case of a G format code), with the precision specifying the number of significant digits. The style used depends on the value converted: style e will be used only if the exponent resulting from the conversion is less than -4 or greater than the precision. Trailing zeroes are removed from the result; a decimal point appears only if it is followed by a digit.
c	The character arg is printed.
s	The arg is taken to be a string (character pointer) and characters from the string are printed until a null character (\Ø) is encountered or the number of characters indicated by the precision specification is reached. If the precision is missing, it is taken to be infinite, so all characters up to the first null character are printed.

of a word is the size of an integer and varies from machine to machine. Putw neither assumes nor causes special alignment in the file.

Output streams, with the exception of the standard error stream stderr, are by default buffered if the output refers to a file and line-buffered if the output refers to a terminal. The standard error output stream stderr is by default unbuffered, but use of freopen(see fopen(3S)) will cause it to become buffered or line-buffered. When an output stream is unbuffered information is queued for writing on the destination file or terminal as soon as written; when it is buffered many characters are saved up and written as a block; when it is line-buffered each line of output is queued for writing on the destination terminal as soon as the line is completed (that is, as soon as a new-line character is written or terminal input is requested). Setbuf may be used to change the stream's buffering strategy.

SEE ALSO

fclose(3S), ferror(3S), fopen(3S), fread(3S), printf(3S), puts(3S), setbuf(3S).

DIAGNOSTICS

On success, these functions each return the value they have written. On failure, they return the constant EOF. This will occur if the file stream is not open for writing, or if the output file cannot be grown. Because EOF is a valid integer, ferror(3S) should be used to detect putw errors.

BUGS

Because it is implemented as a macro, putc treats incorrectly a stream argument with side effects. In particular, putc(c, *f++); doesn't work sensibly. Fputc should be used instead. Because of possible differences in word length and byte ordering, files written using putw are machine-dependent, and may not be read using getw on a different processor. For this reason the use of putw should be avoided.

NAME

putc, putchar, fputc, putw - put character or word on a stream

SYNOPSIS

```
#include <stdio.h>

int putc (c, stream)
char c;
FILE *stream;

int putchar (c)
char c;

int fputc (c, stream)
char c;
FILE *stream;

int putw (w, stream)
int w;
FILE *stream;
```

DESCRIPTION

Putc writes the character c onto the output stream (at the position where the file pointer, if defined, is pointing). Putchar(c) is defined as putc(c, stdout). Putc and putchar are macros.

Fputc behaves like putc, but is a function rather than a macro. _putc runs more slowly than putc, but takes less space per invocation.

Putw writes the word (i.e. integer) w to the output stream (at the position at which the file pointer, if defined, is pointing). The size

NAME

putchar, putc, fputc, putw - put character or word on a stream

SYNOPSIS

See putc

SEE ALSO

ferror, fopen, fread, printf, putc

NOTES

Puts appends a new-line character while fputs does not.

NAME

puts, fputs - put a string on a stream

SYNOPSIS

```
#include <stdio.h>

int puts (s)
char *s;

int fputs (s, stream)
char *s;
FILE *stream;
```

DESCRIPTION

Puts writes the null-terminated string pointed to by s, followed by a new-line character, to the standard output stream stdout.

Fputs writes the null-terminated string pointed to by s to the named output stream.

Neither function writes the terminating null character.

DIAGNOSTICS

Both routines return EOF on error. This will happen if the routines try to write on a file that has not been opened for writing.

NAME

putw, putchar, fputc, putc - put character or word on a stream

SYNOPSIS

See putc

SEE ALSO

bsearch, lsearch, string

NAME

qsort - quicker sort

SYNOPSIS

```
void qsort ((char *) base, nel, sizeof (*base), compar)
unsigned int nel;
int (*compar)( );
```

DESCRIPTION

Qsort is an implementation of the quicker-sort algorithm. It sorts a table of data in place.

Base points to the element at the base of the table. Nel is the number of elements in the table. Compar is the name of the comparison function, which is called with two arguments that point to the elements being compared. The function must return an integer less than, equal to, or greater than zero according as the first argument is to be considered less than, equal to, or greater than the second.

NOTES

The pointer to the base of the table should be of type pointer-to-element, and cast to type pointer-to-character. The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared. Although declared as type pointer-to-character, the value returned should be cast into type pointer-to-element.

NAME

rand, srand - simple random-number generator

SYNOPSIS

```
int rand ( )

void srand (seed)
unsigned seed;
```

DESCRIPTION

Rand uses a multiplicative congruential random-number generator with period 2**32 that returns successive pseudo-random numbers in the range from Ø to 2**15.

Srand can be called at any time to reset the random-number generator to a random starting point. The generator is initially seeded with a value of 1.

NOTES

The spectral properties of rand leave a great deal to be desired. Drand48(3C) provides a much better, though more elaborate, random-number generator.

SEE ALSO

drand48(3C).

NAME

rewind, fseek, ftell - reposition a file pointer in a stream

SYNOPSIS

See fseek

The control string may contain:

1. White-space characters (blanks, tabs, new-lines, or form-feeds) which, except in two cases described below, cause input to be read up to the next non-white-space character.

2. An ordinary character (not %), which must match the next character of the input stream.

3. Conversion specifications, consisting of the character %, an optional assignment suppressing character *, an optional numerical maximum field width, an optional l or h indicating the size of the receiving variable, and a conversion code.

A conversion specification directs the conversion of the next input field; the result is placed in the variable pointed to by the corresponding argument, unless assignment suppression was indicated by *. The suppression of assignment provides a way of describing an input field which is to be skipped. An input field is defined as a string of non-space characters; it extends to the next inappropriate character or until the field width, if specified, is exhausted.

The conversion code indicates the interpretation of the input field; the corresponding pointer argument must usually be of a restricted type. For a suppressed field, no pointer argument should be given. The following conversion codes are accepted:

% a single % is expected in the input at this point; no assignment is done.

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

u an unsigned decimal integer is expected; the corresponding argument should be an unsigned integer pointer.

o an octal integer is expected; the corresponding argument should be an integer pointer.

x a hexadecimal integer is expected; the corresponding argument should be an integer pointer.

e,f,g
a floating point number is expected; the next field is converted accordingly and stored through the corresponding argument, which should be a pointer to a float. The input format for floating point numbers is an optionally signed string of digits, possibly containing a decimal point, followed by an optional exponent field consisting of an E or an e, followed by an optionally signed integer.

NAME

scanf, fscanf, sscanf - convert formatted input

SYNOPSIS

```
#include <stdio.h>

int scanf (format [ , pointer ] ...  )
char *format;

int fscanf (stream, format [ , pointer ] ...  )
FILE *stream;
char *format;

int sscanf (s, format [ , pointer ] ...  )
char *s, *format;
```

DESCRIPTION

Scanf reads from the standard input stream stdin. Fscanf reads from the named input stream. Sscanf reads from the character string s. Each function reads characters, interprets them according to a format, and stores the results in its arguments. Each expects, as arguments, a control string format described below, and a set of pointer arguments indicating where the converted input should be stored.

The control string usually contains conversion specifications, which are used to direct interpretation of input sequences.

an input character and the control string. If the input ends before the first conflict or conversion, EOF is returned.

EXAMPLES

The call:

```
     int i; float x; char name[5Ø];
     scanf ("%d%f%s", &i, &x, name);
```

with the input line:

```
     25 54.32E-1 thompson
```

will assign to i the value 25, to x the value 5.432, and name will contain thompson\Ø. Or:

```
     int i; float x; char name[5Ø];
     scanf ("%2d%f%*d %[Ø-9]", &i, &x, name);
```

with input:

```
     56789 Ø123 56a72
```

will assign 56 to i, 789.Ø to x, skip Ø123, and place the string 56\Ø in name. The next call to getchar (see getc(3S)) will return a.

SEE ALSO

---

atof, getc, printf, strtol

NOTES

Trailing white space (including a new-line) is left unread unless matched in the control string.

DIAGNOSTICS

These functions return EOF on end of input and a short count for missing or illegal data items.

s a character string is expected; the corresponding argument should be a character pointer pointing to an array of characters large enough to accept the string and a terminating \Ø, which will be added automatically. The input field is terminated by a white-space character.

c a character is expected; the corresponding argument should be a character pointer. The normal skip over white space is suppressed in this case; to read the next non-space character, use %ls. If a field width is given, the corresponding argument should refer to a character array; the indicated number of characters is read.

[indicates string data and the normal skip over leading white space is suppressed. The left bracket is followed by a set of characters, which we will call the scanset, and a right bracket; the input field is the maximal sequence of input characters consisting entirely of characters in the scanset. The circumflex, (^), when it appears as the first character in the scanset, serves as a complement operator and redefines the scanset as the set of all characters not contained in the remainder of the scanset string. There are some conventions used in the construction of the scanset. A range of characters may be represented by the construct first-last, thus [Ø123456789] may be expressed [Ø-9].

Using this convention, first must be lexically less than or equal to last, or else the dash will stand for itself. The dash will also stand for itself whenever it is the first or the last character in the scanset. To include the right square bracket as an element of the scanset, it must appear as the first character (possibly preceded by a circumflex) of the scanset, and in this case it will not be syntactically interpreted as the closing bracket. The corresponding argument must point to a character array large enough to hold the data field and the terminating \Ø, which will be added automatically.

The conversion characters d, u, o, and x may be preceded by l or h to indicate that a pointer to long or to short rather than to int is in the argument list. Similarly, the conversion characters e , f , and g may be preceded by l to indicate that a pointer to double rather than to float is in the argument list.

Scanf conversion terminates at EOF, at the end of the control string, or when an input character conflicts with the control string. In the latter case, the offending character is left unread in the input stream.

Scanf returns the number of successfully matched and assigned input items; this number can be zero in the event of an early conflict between

BUGS

The success of literal matches and suppressed assignments is not directly determinable.

SEE ALSO

fopen, getc, malloc, putc

NOTES

A common source of error is allocating buffer space as an automatic variable in a code block, and then failing to close the stream in the same block.

NAME

setbuf - assign buffering to a stream

SYNOPSIS

```
#include <stdio.h>

void setbuf (stream, buf)
FILE *stream;
char *buf;
```

DESCRIPTION

Setbuf is used after a stream has been opened but before it is read or written. It causes the character array pointed to by buf to be used instead of an automatically allocated buffer. If buf is a NULL character pointer input/output will be completely unbuffered.

A constant BUFSIZ, defined in the <stdio.h> header file, tells how big an array is needed:

```
     char buf[BUFSIZ];
```

A buffer is normally obtained from malloc(3C) at the time of the first getc or putc(3S) on the file, except that the standard error stream stderr is normally not buffered.

Output streams directed to terminals are always line-buffered unless they are unbuffered.

WARNING

If longjmp is called when env was never primed by a call to setjmp, or when the last such call is in a function which has since returned, absolute chaos is guaranteed.

NAME

setjmp, longjmp - non-local goto

SYNOPSIS

```
#include <setjmp.h>

int setjmp (env)
jmp_buf env;

void longjmp (env, val)
jmp_buf env;
int val;
```

DESCRIPTION

These functions are useful for dealing with errors and interrupts encountered in a low-level subroutine of a program.

Setjmp saves its stack environment in env (whose type, jmp_buf, is defined in the <setjmp.h> header file), for later use by longjmp. It returns the value Ø.

Longjmp restores the environment saved by the last call of setjmp with the corresponding env argument. After longjmp is completed program execution continues as if the corresponding call of setjmp (which must not itself have returned in the interim) had just returned the value val. Longjmp cannot cause setjmp to return the value Ø. If longjmp is invoked with a second argument of Ø, setjmp will return 1. All accessible data have values as of the time longjmp was called.

NAME

setbuf - assign buffering to a stream

SYNOPSIS

See setbuf

NAME

sin, cos, tan, asin, acos, atan, atan2 - trigonometric functions

SYNOPSIS

See trig

NAME

sinh, cosh, tanh - hyperbolic functions

SYNOPSIS

```
#include <math.h>

double sinh (x)
double x;

double cosh (x)
double x;

double tanh (x)
double x;
```

DESCRIPTION

Sinh, cosh and tanh return respectively the hyberbolic sine, cosine and tangent of their argument.

DIAGNOSTICS

Sinh and cosh return HUGE when the correct value would overflow, and set errno to ERANGE.

NAME

sleep - suspend execution for interval

SYNOPSIS

```
unsigned sleep (seconds)
unsigned seconds;
```

DESCRIPTION

The current process is suspended from execution for the number of seconds specified by the argument. The actual suspension time may be less than that requested. Also, the suspension time may be longer than requested by an arbitrary amount due to the scheduling of other activity in the system. The value returned by sleep will be the unslept amount (the requested time minus the time actually slept) in case the caller had an alarm set to go off earlier than the end of the requested sleep time, or premature arousal.

The routine is implemented by setting an alarm and pausing until it occurs. The previous state of the alarm is saved and restored. The calling program may have set up an alarm before calling sleep; if the sleep time exceeds the time till such alarm, the process sleeps only until the alarm would have occurred, and the caller's alarm catch routine is executed just before the sleep routine returns, but if the sleep time is less than the time till such alarm, the prior alarm time is reset to go off at the same time it would have without the intervening sleep.

NAME

sprintf, fprintf, printf - print formatted output

SYNOPSIS

See printf

NAME

sqrt, exp, log, log1Ø, pow, - exponential, logarithm, power, square root functions

SYNOPSIS

See exp

NAME

sscanf, fscanf, - convert formatted input

SYNOPSIS

See scanf

The contents of the structure pointed to by buf include the following members:

```
ushort  st_mode;     /* File mode */
ino_t   st_ino;      /* FCB number */
dev_t   st_dev;      /* Not used */
dev_t   st_rdev;     /* Not used */
short   st_nlink;    /* Number of links (always 1) */
ushort  st_uid;      /* User ID of the file's owner */
ushort  st_gid;      /* Group ID of the file's group */
off_t   st_size;     /* File size in bytes */
time_t  st_atime;    /* Not used */
time_t  st_mtime;    /* Time of last data modification
*/
time_t  st_ctime;    /* Time file was created */
                     /* Times measured in seconds since */
                     /* 00:00:00 GMT, Jan. 1, 1970 */
```

st_mtime Time when data was last modified. Changed by the following system calls: creat, and write.

st_ctime Time when the file was created. Changed by the following system calls: creat,

Stat will fail if one or more of the following are true:

A component of the path prefix is not a directory. [ENOTDIR]

The named file does not exist. [ENOENT]

Search permission is denied for a component of the path prefix. [EACCES]

Buf or path points to an invalid address. [EFAULT]

Fstat will fail if one or more of the following are true:

Fildes is not a valid open file descriptor. [EBADF]

Buf points to an invalid address. [EFAULT]

RETURN VALUE

Upon successful completion a value of 0 is returned. Otherwise, a value of -1 is returned and errno is set to indicate the error.

NAME

stat, fstat - get file status

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>

int stat (path, buf)
char *path;
struct stat *buf;

int fstat (fildes, buf)
int fildes;
struct stat *buf;
```

DESCRIPTION

Path points to a path name naming a file. Read, write or execute permission of the named file is not required, but all directories listed in the path name leading to the file must be searchable. Stat obtains information about the named file.

Similarly, fstat obtains information about an open file known by the file descriptor fildes, obtained from a successful open, creat, dup, fcntl, or pipe system call.

Buf is a pointer to a stat structure into which information is placed concerning the file.

SEE ALSO

chmod, creat, read, time, unlink, write

NAME

strcat, strncat, strcmp, strncmp, strcpy, strncpy, strlen, strchr, strrchr, strpbrk, strspn, strcspn, strtok - string operations

SYNOPSIS

See string

NAME

swab - swap bytes

SYNOPSIS

```
void swab (from, to, nbytes)
char *from, *to;
int nbytes;
```

DESCRIPTION

Swab copies nbytes bytes pointed to by from to the array pointed to by to, exchanging adjacent even and odd bytes. It is useful for carrying binary data between PDP-11s and other machines. Nbytes should be even and non-negative. If nbytes is odd and positive swab uses nbytes-1 instead. If nbytes is negative swab does nothing.

DIAGNOSTICS

System exec's the cip in order to execute string. If the exec fails, system returns -1 and sets errno.

NAME

system - issue a cip command

SYNOPSIS

```
#include <stdio.h>

int system (string)
char *string;
```

DESCRIPTION

System causes the string to be given to the cip as input, as if the string had been typed as a command at a terminal. The current process waits until the cip has completed, then returns the exit status of the cip.

FILES

sys$disk/sysexe/cip.exe

SEE ALSO

exec

NAME

tan, sin, cos, asin, acos, atan, atan2 - trigonometric functions

SYNOPSIS

See trig

NAME

tanh, sinh, cosh, - hyperbolic functions

SYNOPSIS

See sinh

tempnam

NAME

tempnam, tempnam - create a name for a temporary file

SYNOPSIS

See tmpnam

NAME

time - get time

SYNOPSIS

```
long time ((long *) Ø)

long time (tloc)
long *tloc;
```

DESCRIPTION

Time returns the value of time in seconds since ØØ:ØØ:ØØ GMT, January 1, 197Ø.

If tloc (taken as an integer) is non-zero, the return value is also stored in the location to which tloc points.

Time will fail if tloc points to an illegal address. [EFAULT]

RETURN VALUE

Upon successful completion, time returns the value of time. Otherwise, a value of -1 is returned and errno is set to indicate the error.

NAME

tmpfile - create a temporary file

SYNOPSIS

```
#include <stdio.h>

FILE *tmpfile ()
```

DESCRIPTION

Tmpfile creates a temporary file and returns a corresponding FILE pointer. The file will automatically be deleted when the process using it terminates. The file is opened for update.

SEE ALSO

creat, unlink, fopen, mktemp, tmpnam

environment, whose value is a path-name for the desired temporary-file directory.

Many applications prefer their temporary files to have certain favorite initial letter sequences in their names. Use the pfx argument for this. This argument may be NULL or point to a string of up to five characters to be used as the first few characters of the temporary-file name.

Tempnam uses malloc(3C) to get space for the constructed file name, and returns a pointer to this area. Thus, any pointer value returned from tempnam may serve as an argument to free (see malloc(3C)). If tempnam cannot return the expected result for any reason, i.e. malloc failed, or none of the above mentioned attempts to find an appropriate directory was successful, a NULL pointer will be returned.

NOTES

These functions generate a different file name each time they are called.

Files created using these functions and either fopen or creat are temporary only in the sense that they reside in a directory intended for temporary use, and their names are unique. It is the user's responsibility to use unlink(2) to remove the file when its use is ended.

SEE ALSO

creat, unlink, fopen, malloc, mktemp, tmpfile

BUGS

If called more than 17,576 times in a single process, these functions will start recycling previously used names. Between the time a file name is created and the file is opened, it is possible for some other process to create a file with the same name. This can never happen if that other process is using these functions or mktemp, and the file names are chosen so as to render duplication by other means unlikely.

NAME

tmpnam, tempnam - create a name for a temporary file

SYNOPSIS

```
#include <stdio.h>

char *tmpnam (s)
char *s;

char *tempnam (dir, pfx)
char *dir, *pfx;
```

DESCRIPTION

These functions generate file names that can safely be used for a temporary file.

Tmpnam always generates a file name using the path-name defined as P_tmpdir in the <stdio.h> header file. If s is NULL, tmpnam leaves its result in an internal static area and returns a pointer to that area. The next call to tmpnam will destroy the contents of the area. If s is not NULL, it is assumed to be the address of an array of at least L_tmpnam bytes, where L_tmpnam is a constant defined in <stdio.h>; tmpnam places its result in that array and returns s.

Tempnam allows the user to control the choice of a directory. The argument dir points to the path-name of the directory in which the file is to be created. If dir is NULL or points to a string which is not a path-name for an appropriate directory, the path-name defined as P_tmpdir in the <stdio.h> header file is used. If that path-name is not accessible, /tmp will be used as a last resort. This entire sequence can be up-staged by providing a logical name TMPDIR in the user's

Asin returns the arcsine of x, in the range -pi/2 to pi/2.

Acos returns the arccosine of x, in the range Ø to pi.

Atan returns the arctangent of x, in the range -pi/2 to pi/2.

Atan2 returns the arctangent of y/x, in the range -pi to pi, using the signs of both arguments to determine the quadrant of the return value.

DIAGNOSTICS

Sin, cos and tan lose accuracy when their argument is far from zero. For arguments sufficiently large, these functions return Ø when there would otherwise be a complete loss of significance. In this case a message indicating TLOSS error is printed on the standard error output. For less extreme arguments, a PLOSS error is generated but no message is printed. In both cases, errno is set to ERANGE.

Tan returns HUGE for an argument which is near an odd multiple of pi/2 when the correct value would overflow, and sets errno to ERANGE.

Arguments of magnitude greater than 1.Ø cause asin and acos to return Ø and to set errno to EDOM. In addition, a message indicating DOMAIN error is printed on the standard error output.

NAME

trig, sin, cos, tan, asin, acos, atan, atan2 - trigonometric functions

SYNOPSIS

```
#include <math.h>

double sin (x)
double x;

double cos (x)
double x;

double tan (x)
double x;

double asin (x)
double x;

double acos (x)
double x;

double atan (x)
double x;

double atan2 (y, x)
double x, y;
```

DESCRIPTION

Sin, cos and tan return respectively the sine, cosine and tangent of their argument, which is in radians.

The variable pointed to by rootp will be changed if the deleted node was the root of the tree. Tdelete returns a pointer to the parent of the deleted node, or a NULL pointer if the node is not found.

Twalk traverses a binary search tree. Root is the root of the tree to be traversed. (Any node in a tree may be used as the root for a walk below that node.) Action is the name of a routine to be invoked at each node. This routine is, in turn, called with three arguments. The first argument is the address of the node being visited. The second argument is a value from an enumeration data type typedef enum { preorder, postorder, endorder, leaf } VISIT; (defined in the <search.h> header file), depending on whether this is the first, second or third time that the node has been visited (during a depth-first, left-to-right traversal of the tree), or whether the node is a leaf. The third argument is the level of the node in the tree, with the root being level zero.

NOTES

The pointers to the key and the root of the tree should be of type pointer-to-element, and cast to type pointer-to-character. The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared. Although declared as type pointer-to-character, the value returned should be cast into type pointer-to-element.

WARNING: The root argument to twalk is one level of indirection less than the rootp arguments to tsearch and tdelete.

DIAGNOSTICS

A NULL pointer is returned by tsearch if there is not enough space available to create a new node. A NULL pointer is returned by tsearch and tdelete if rootp is NULL on entry.

SEE ALSO

bsearch(3C), hsearch(3C), lsearch(3C).

BUGS

Awful things can happen if the calling function alters the pointer to the root.

NAME

tsearch, tdelete, twalk - manage binary search trees

SYNOPSIS

```
#include <search.h>

char *tsearch ((char *) key, (char **) rootp, compar)
int (*compar)( );

char *tdelete ((char *) key, (char **) rootp, compar)
int (*compar)( );

void twalk ((char *) root, action)
void (*action)( );
```

DESCRIPTION

Tsearch is a binary tree search routine generalized from Knuth (6.2.2) Algorithm T. It returns a pointer into a tree indicating where a datum may be found. If the datum does not occur, it is added at an appropriate point in the tree. Key points to the datum to be sought in the tree. Rootp points to a variable that points to the root of the tree. A NULL pointer value for the variable denotes an empty tree; in this case, the variable will be set to point to the datum at the root of the new tree. Compar is the name of the comparison function. It is called with two arguments that point to the elements being compared. The function must return an integer less than, equal to, or greater than zero according as the first argument is to be considered less than, equal to, or greater than the second.

Tdelete deletes a node from a binary search tree. It is generalized from Knuth (6.2.2) algorithm D. The arguments are the same as for tsearch.

BUGS

The return value points to static data whose content is overwritten by each call.

ttyname

NAME

ttyname, isatty - find name of a terminal

SYNOPSIS

```
char *ttyname (fildes)
int fildes;

int isatty (fildes)
int fildes;
```

DESCRIPTION

Ttyname returns a pointer to a string containing the null-terminated path name of the terminal device associated with file descriptor fildes.

Isatty returns 1 if fildes is associated with a terminal device, Ø otherwise.

FILES

/dev/*

DIAGNOSTICS

Ttyname returns a NULL pointer if fildes does not describe a terminal device.

DIAGNOSTICS

In order that ungetc perform correctly, a read statement must have been performed prior to the call of the ungetc function. Ungetc returns EOF if it can't insert the character. In the case that stream is stdin, ungetc will allow exactly one character to be pushed back onto the buffer without a previous read statement.

NAME

ungetc - push character back into input stream

SYNOPSIS

```
#include <stdio.h>

int ungetc (c, stream)
char c;
FILE *stream;
```

DESCRIPTION

Ungetc inserts the character c into the buffer associated with an input stream. That character, c, will be returned by the next getc call on that stream. Ungetc returns c, and leaves the file stream unchanged.

One character of pushback is guaranteed provided something has been read from the stream and the stream is actually buffered.

If c equals EOF, ungetc does nothing to the buffer and returns EOF.

Fseek(3S) erases all memory of inserted characters.

SEE ALSO

fseek, getc, setbuf

RETURN VALUE

Upon successful completion, a value of Ø is returned. Otherwise, a value of -1 is returned and errno is set to indicate the error.

SEE ALSO

close, open

NAME

unlink - remove directory entry

SYNOPSIS

```
int unlink (path)
char *path;
```

DESCRIPTION

Unlink removes the directory entry named by the path name pointed to be path.

The named file is unlinked unless one or more of the following are true:

A component of the path prefix is not a directory. [ENOTDIR]

The named file does not exist. [ENOENT]

Search permission is denied for a component of the path prefix. [EACCES]

Write permission is denied on the directory containing the link to be removed. [EACCES]

The named file is a directory. [EPERM]

Path points outside the process's allocated address space. [EFAULT]

When all links to a file have been removed and no process has the file open, the space occupied by the file is freed and the file ceases to exist. If one or more processes have the file open when the last link is removed, the removal is postponed until all references to the file have been closed.

Fildes is not a valid file descriptor open for writing. [EBADF]

If a _rite requests that more bytes be written than there is room for (e.g., the physical end of a medium), only as many bytes as there is room for will be written. For example, suppose there is space for 2Ø bytes more in a file before reaching a limit. A write of 512 bytes will return 2Ø. The next write of a non-zero number of bytes will give a failure return (except as noted below).

RETURN VALUE

Upon successful completion the number of bytes actually written is returned. Otherwise, -1 is returned and errno is set to indicate the error.

SEE ALSO

creat, dup, lseek, open

NAME

write - write on a file

SYNOPSIS

```
int write (fildes, buf, nbyte)
int fildes;
char *buf;
unsigned nbyte;
```

DESCRIPTION

Fildes is a file descriptor obtained from a creat, open, or dup system call.

Write attempts to write nbyte bytes from the buffer pointed to by buf to the file associated with the fildes.

On devices capable of seeking, the actual writing of data proceeds from the position in the file indicated by the file pointer. Upon return from write, the file pointer is incremented by the number of bytes actually written.

On devices incapable of seeking, writing always takes place starting at the current position. The value of a file pointer associated with such a device is undefined.

If the O_APPEND flag of the file status flags is set, the file pointer will be set to the end of the file prior to each write.

Write will fail and the file pointer will remain unchanged if one or more of the following are true:

NAME

y0, yl, yn, j0, jl, jn, - Bessel functions

SYNOPSIS

See bessel

NAME

y1, j0, j1, jn, y0, yn - Bessel functions

SYNOPSIS

See bessel

NAME

yn, j0, j1, jn, y0, y1, - Bessel functions

SYNOPSIS

See bessel

Chapter 8

WMCS Compilation Commands

This chapter contains the four WMCS C compilation commands. They are formatted in the WMCS command-description style. If you are using C under WMCS, read these command descriptions.

If you are using C under UniPlus+ System V, see the _cc_ command description in the _UniPlus+ System V User's Reference Manual (Section 1)_.

	:maxext=	:warnf66
	:maxhash=	:xref=
LL switches	:libraries=	:runtime=
	:prefix=	:strip
	:reloc	

Parameters

---

file list — Function: Use this parameter to specify the list of files to be compiled and loaded together.
Default: None.
Syntax: Standard syntax for file lists. Wildcards are allowed.

Switches

:assemble — Function: Use this switch to specify that assembly-language source files are to be assembled into object files. Assembly-language source files are files specified in the file list with a .S extension, plus any output files from the floating-point preprocessors.
Default: :assemble, i.e., assembly-language files are assembled into object files.
Syntax: Type :noassemble to suppress the assembler and the linker/loader phases. Files specified in the file list with a .S extension are left untouched. Assembly-language output files from the floating-point preprocessors are saved in the default directory with the same names as the corresponding source files, but with .S extensions.

:before= — Function: Use this switch to select only those files that were specified in the file list and were created/modified before the specified date and time.
Default: Selects all files that were specified in the file list.
Syntax: Type :before= followed by a date and/or time in the standard date and time syntax.

:case — Function: Use this switch with FORTRAN source files to allow upper- and lower-case names to be distinct.
Default: :nocase, i.e., upper- and lower-case names are not distinct.
Syntax: Type :case to allow upper- and lower-case names to be distinct.

compile

Description

Use this command to compile C, FORTRAN, and/or assembly language source files into object files, and to link object files together to produce an executable file.

Command Line Syntax

Mnemonic	compile		
Required parameter	File list		
Switches	File selection	:before= :exclude= :mod	:since= :uic=
	Suppress compilation	:assemble :fpreprocess :load	:preprocess :process
	General switches	:deletetemp :floatingpoint= :listing= :log :optimize :optimize=	:output= :subopt= :subpass= :tempprefix= :verbose :warnings
	C switches	:define= :include=	:undefine=
	FORTRAN switches	:case :int= :maxctl= :maxequ=	:maxstno= :onetrip :range :undefined

:fpreprocess	Function: Use this switch to specify that pseudo-assembly language source files are to be translated into actual assembly-language source files by a floating-point preprocessor. Pseudo-assembly language source files are files specified in the file list with a .K extension, plus any output files from compilers and the C optimizer. The floating-point preprocessor used is selected by the :floatingpoint= switch. Default: :fpreprocess, i.e., pseudo-assembly language files are translated to actual assembly-language files. Syntax: Type :nofpreprocess to suppress the floating-point preprocessor, assembler, and linker/loader phases. Files specified in the file list with a .K extension will be optimized if the C optimizer is selected but will otherwise be left untouched. Pseudo-assembly language output files from the compilers and the C optimizer will be left in the current default directory with the same names as the corresponding source files, but with .K extensions.
:include=	Function: Use this switch on C source files to specify additional directories in which the C preprocessor is to search for include files. Directories specified by the :include= switch are searched first, followed by predefined "standard" directories. Default: The C preprocessor searchs only the predefined "standard" directories. Syntax: Type :include= followed by a list of directory specifications, separated by commas.
:int=	Function: Use this switch on FORTRAN source files to change the default size of FORTRAN integers. Default: :int=4, i.e., FORTRAN integers are 4 bytes long. Syntax: Type :int= followed by one of the following: 2 Integers are 2 bytes long 4 Integers are 4 bytes long
:libraries=	Function: Use this switch to specify the names of library files to be used by the linker/loader in addition to the standard libraries. Default: Only the standard libraries are used. Syntax: Type :libraries= followed by a list of file names. Wildcards are not allowed. Device and directory names may not be given here but must instead be specified by the :prefix= switch.
:listing=	Function: Use this switch to specify a file in which a program listing is generated. Currently, this switch applies only to FORTRAN. To get a listing from the

:define=	Function: Use this switch with C source files to define macros for the C preprocessor. Definitions given by this switch can be cancelled by the :undefine= switch. Default: No macros are defined except some macros predefined by the C preprocessor itself. Syntax: Type :define= followed by a list of values, separated by commas. Each value must be in one of the following forms:

name=value	Assigns the value to the name
name	Assigns a value of 1 to the name

:deletetemp	Function: Use this switch to cause temporary files to be deleted automatically. Default: :deletetemp, i.e., temporary files are automatically deleted. Syntax: Type :nodeletetemp to preserve temporary files.
:exclude=	Function: Use this switch to select only those files or devices that were specified in the file list and do not match any of the files specified as the value of the :exclude= switch. Default: Selects all files or devices specified in the file list. Syntax: Type :exclude= followed by a list of file or device designations, separated by commas, any one of which may contain wildcard characters.
:floatingpoint=	Function: Use this switch to cause the compilers to generate code for a specific kind of floating-point hardware and/or software. This switch also selects the floating-point preprocessor to be used. Default: :floatingpoint=lib, i.e., the generic floating-point library is used. Syntax: Type :floatingpoint= followed by one of the following:

LIB	Current version of the generic floating-point library (same as LIB2).
LIB2	Version 2 of the generic floating-point library.
SKY	Current version of the SKY floating-point board (same as SKY1).
SKY1	Version 1 of the SKY floating-point board.
FFP	Current version of the FFP floating-point board (same as FFP1).
FFP1	Version 1 of the FFP floating-point board.
NOFP	No floating-point (produces smaller image files).

:maxhash=	Function: Use this switch with FORTRAN source files to specify the size of the FORTRAN compiler's symbol table. Default: :maxhash=4Ø1, i.e., the FORTRAN compiler's symbol table has room for 4Ø1 entries. Syntax: Type :maxhash= followed by a numeral indicating the size of the FORTRAN compiler's symbol table.
:maxstno=	Function: Use this switch with FORTRAN source files to specify the maximum number of statement numbers allowed. Default: :maxstno=4Ø1, i.e., a maximum of 4Ø1 statement numbers is allowed. Syntax: Type :maxstno= followed by a numeral indicating the maximum number of statement numbers to be allowed.
:mod	Function: Use this switch to specify that the modification date is to be used in all date and time considerations by the :before= or :since= switches. Default: :nomod, i.e., the creation date is used in all date and time considerations by the :before= or :since= switches. Syntax: Type :mod.
:onetrip	Function: Use this switch with FORTRAN source files to specify that DO loops are to be executed at least once. Default: :noonetrip, i.e., DO loops are not performed if the upper limit is less than the lower limit. Syntax: Type :onetrip to cause DO loops to be executed at least once even if the upper limit is less than the lower limit.
:optimize	Function: Use this switch to perform code optimizations where possible. This switch has the same effect as :optimize=F77,OPT and is included for convenience. Default: :nooptimize, i.e., no optimizations are performed. Syntax: Type :optimize to perform optimizations.
:optimize=	Function: Use this switch to perform specific code optimizations. Default: No optimizations are performed. Syntax: Type :optimize= followed by one or both of the following values (if both, separate with commas): F77 For FORTRAN source files only, performs FORTRAN-specific optimizations. OPT Performs optimizations on the pseudo-assembly language that is output from the compilers. This value is used for C programs.

	assembler use :subopt="as:-l filename" (where filename is the file you want to be the listing). Default: A program listing is not generated. Syntax: Type :listing= followed by a file designation. Wildcards are not allowed.
:load	Function: Use this switch to specify that object files are to be linked to create an executable file. Object files are files specified in the file list with a .W extension, plus any output files from the assembler and LLC. Default: :load, i.e., object files are linked to create an executable file. Syntax: Type :noload to suppress the linker/loader phase. Files specified in the file list with a .W extension are left untouched, and object files from the assembler and LLC are left in the current default directory with the same names as the corresponding source files, but with .W extensions.
:log	Function: Use this switch to specify whether log messages are displayed. (Log messages are informational displays that indicate what the utility is doing.) Default: The value specified by the OPTION command. Syntax: Type :log or :nolog to override the default.
:maxctl=	Function: Use this switch with FORTRAN source files to specify the maximum level that IF and DO statements can be nested. Default: :maxctl=10, i.e., IF and DO statements can be nested 10 deep. Syntax: Type :maxctl= followed by a numeral indicating the maximum number of levels that IF and DO statements can be nested.
:maxequ=	Function: Use this switch with FORTRAN source files to specify the maximum number of equivalences allowed. Default: :maxequ=150, i.e., a maximum of 150 equivalences is allowed. Syntax: Type :maxequ= followed by a numeral indicating the maximum number of equivalences to be allowed.
:maxext=	Function: Use this switch with FORTRAN source files to specify the maximum number of external symbols allowed. Default: :maxext=200, i.e., a maximum of 200 external symbols is allowed. Syntax: Type :maxext= followed by a numeral indicating the maximum number of external symbols to be allowed.

	performed at runtime. Default: :norange, i.e., runtime range-checking of subscripts is not performed. Syntax: Type :range to cause runtime range-checking of subscripts to be performed.
:reloc	Function: Use this switch to specify that relocation information is to be preserved in the executable file. Default: :noreloc, i.e., relocation information is not preserved. Syntax: Type :reloc to cause relocation information to be preserved in the executable file (useful for unmapped S125Øs).
:runtime=	Function: Use this switch to specify that the linker/ loader is to use language-specific runtime libraries. Default: The linker/loader automatically uses language-specific libraries whenever the corresponding language compiler is used. Syntax: Type :runtime= followed by a value specifying a language-specific library. Currently, the only valid value is F, which specifies the FORTRAN libraries.
:since=	Function: Use this switch to select only those files specified in the file list and were created/modified since the specified date and time. Default: Selects all files specified in the file list. Syntax: Type :since= followed by a date and/or time in the standard syntax.
:strip	Function: Use this switch to specify that all symbols are to be stripped from the executable file. Default: :strip, i.e., all symbols are stripped. Syntax: Type :nostrip to preserve symbols in the executable file.
:subpass=	Function: Use this switch to specify substitute compiler passes. The compile utility uses a three-step algorithm to determine the filename of each compiler pass. First, compiler passes specified by the :subpass= switch are used. Second, for passes not specified by the :subpass= switch, passes specified by logical names are used. Finally, for passes not specified by the :subpass switch or by logical names, the standard compiler passes are used. Default: If any of the following logical names are defined, then their definition is used as the filename of

:output=	Function: Use this switch to name the output file of the compilation process. Default: If :preprocess, :nofpreprocess, :noassemble, or :noload is specified, the output is stored in the default directory in files with the same names as the corresponding source files specified in the file list, but with extensions of .I, .K, .S, or .W respectively. If the linker/loader phase is not suppressed, the resulting executable file is stored in the default directory with the same name as the first file specified in the file list, but with an extension of .EXE. Syntax: Type :output= followed by a file name without a file extension. The correct extension (.I, .S, or .EXE, for example) is added automatically by the compiler. Wildcards are not allowed.
:prefix=	Function: Use this switch to specify additional directories in which the linker/loader is to search for libraries. Directories specified by the :prefix= switch are searched first, followed by predefined "standard" directories. Default: The linker/loader searchs only the predefined "standard" directories. Syntax: Type :prefix= followed by a list of directory specifications, separated by commas. As a special case, a value of zero causes the predefined "standard" directories to not be searched.
:preprocess	Function: Use this switch to specify that the output from the preprocessors is to be left in the current default directory in files with the same name as the corresponding source files, but with .I extensions. All other phases of the compile are suppressed. Default: :nopreprocess, i.e., preprocessor output is sent to the compilers, and other phases of the compile compile are not suppressed. Syntax: Type :preprocess to send the output of the preprocessors to .I files.
:process	Function: Use this switch to specify that the output from the preprocessors is to be sent to standard output. All other phases.of the compile are suppressed. Default: :noprocess, i.e., preprocessor output is not sent to standard output, and other phases of the compile are not suppressed. Syntax: Type :process to send the output of the preprocessors to standard output.
:range	Function: Use this switch with FORTRAN source files to specify that range-checking of array subscripts is to be

	Syntax: Type :uic= followed by a list of UIC's or usernames.
:undefine=	Function: Use this switch with C source files to cancel macro definitions for the C preprocessor given by the :define= switch. This switch can also be used to cancel the macros predefined by the C preprocessor itself. Default: No macros are cancelled. Syntax: Type :undefine= followed by a list of names, separated by commas, to be cancelled.
:undefined	Function: Use this switch with FORTRAN source files to specify that the default type of variables is undefined, rather than using the default FORTRAN rules. Default: :noundefined, i.e., the default type of variables is determined according the default FORTRAN rules. Syntax: Type :undefined to cause the default type of FORTRAN variables to be undefined.
:verbose	Function: Use this switch to display the command line for each compiler pass before it is executed. This switch also sets the :log switch. Additional information may be displayed as well depending on the situation. Default: :noverbose, i.e., command lines for compiler passes are not displayed. Syntax: Type :verbose to display command lines for each compiler pass.
:warnf66	Function: Use this switch with FORTRAN source files to suppress extensions that enhance compatibility with FORTRAN66. Default: :nowarnf66, i.e., extensions that enhance FORTRAN66 compatibility are not suppressed. Syntax: Type :warnf66 to suppress extensions that enhance FORTRAN66 compatibility.
:warnings	Function: Use this switch to display warning messages generated by the compiler. Default: :warnings, i.e., warning messages are displayed. Syntax: Type :nowarnings to suppress warnings.
:xref=	Function: Use this switch with FORTRAN source files to specify a file in which a symbol cross reference listing is generated. Default: A cross-reference listing is not generated. Syntax: Type :xref= followed by a filename. Wildcards are not allowed.

the corresponding compiler pass, otherwise the standard compiler passes are used:

Name	Corresponding Compiler Pass
CPP	C preprocessor
F77	FORTRAN compiler
F77X	FORTRAN cross reference
CC	C compiler
OPT	C optimizer
APP	Floating-point preprocessor
AS	Assembler
LLC	Compiler for linker/loader
LL	Linker/loader

Syntax: Type :subpass= followed by one or more values, separated by commas, in the form name:filename, where name is one of the names given in the above table, and filename is the filename (including its directory) of the corresponding substitute compiler pass.

:subopt= Function: Use this switch to specify arbitrary command line arguments for individual compiler passes. (This switch is provided for maximum flexibility. Take care not to misuse it.)
Default: Only arguments put on the command line by COMPILE (possibly determined by regular COMPILE switches) are passed to the compiler passes.
Syntax: Type one or more values, separated by commas, where each value is of the form "name:string". The name must be one of the following: CPP, F77, F77x, CC, OPT, APP, AS, LLC, or LL (see :subpass= for the meaning of each name). The string is the actual string to be placed on the named compiler pass's command line. If the string contains spaces, then the entire name:string value should be enclosed in double quotes.

:tempprefix= Function: Use this switch to specify the directory in which temporary files are stored.
Default: If the logical name TMPDIR is defined, then the directory it specifies is used, otherwise the directory SYS$TMP/SYSTMP/ is used.
Syntax: Type :tempprefix= followed by a directory specification. Wildcards are not allowed.

:uic= Function: Use this switch to select only those files or devices that are specified in the file list and are owned by the specified user or list of users.
Default: Selects all files specified in the file list.

At first glance, the :runtime= switch might appear useless since language-specific libraries are automatically included whenever the corresponding language compiler is used. This could be used if you previously compiled several FORTRAN source files into objects and now want to link these object files together to produce an executable file. Compile would only see a bunch of .W files and wouldn't know about their FORTRAN ancestry. You would have to use the :runtime= switch to explicitly tell COMPILE that these object files were produced from FORTRAN source files and require the use of the FORTRAN libraries.

In order to produce smaller executable files, the symbol table is normally stripped by the linker/loader. When debugging programs with WIBUG, however, it is highly recommended that you use :nostrip to preserve the symbols.

Related CIP Commands

---

lllib
llran
li

Examples

```
> compile main.c,routine1.c,routine2.c
```

This command compiles the C source files named MAIN.C, ROUTINE1.C, and ROUTINE2.C in the default directory, producing the object files MAIN.W, ROUTINE1.W, and ROUTINE2.W, and the executable file MAIN.EXE.

```
> compile *.f :prefix=/mylib/ :lib=matrix :float=sky :output=munge.exe
```

Assume that the library file /MYLIB/MATRIX.LIB exists on the default device. This command compiles all of the FORTRAN source files with a .F extension in the default directory, producing a .W object file for each one. These object files are then linked together using the library file /MYLIB/MATRIX.LIB to produce an executable file named MUNGE.EXE. The program is targeted for the SKY floating-point board.

Using Prompts

```
> Compile
File list       > main.c,routine1.c,routine2.c
```

This is the same as the first example.

Notes on Usage

The :optimize and :optimize= switches improve the ultimate efficiency of the program, but compilation usually takes longer.

Any of the five switches that suppress compilation phases may be specified at the same time. The switch that comes first in the following list is used, and any other switches in the list that are specified are ignored:

```
:process
:preprocess
:nofpreprocess
:noassemble
:noload
```

Switches

:before=	- Type a date and time. (see dates)
:exclude=	- Type a list of file designations. (see filelist)
:log	- Log messages are displayed.
:mod	- Use file modification date for comparison
:since=	- Type a date and time. (see dates)
:uic=	- Type a list of uics or usernames. (see uiclist)
:verbose	- Display detailed information about the files.

The following switches are mutually exclusive:

:add	- Add the files to the library. Create library.
:delete	- Delete the indicated files from the library.
:extract	- Extract the indicated files from the library.
:list	- List the files in the library.

lllib

Description

Use this command to create or maintain a packaged library file for the ll loader. The indicated relocatable object files generated by WiMAC are archived into a file, and llran is used to generate a symbol table.

Examples

```
> lllib develop.lib *.w :add :log
```

Generate the library file develop.lib from all the .w files in the current directory. Display the names of the files being libraried.

Parameters

Library file	> - Required. Enter the name of the library file to be used. A file extension of .lib is customary.
File list	> - Optional. Type a list of file designations which are to be displayed, added, or deleted. Wildcards are only permitted when adding. Default: With :add, all files in the current directory, with :extract or :list, all files in the library, with :delete, no files.

Display format

:filename	- Precede each symbol by the name of the file it is in
:header	- (Default) Display filename and column headers
:pause	- Pause after each screenful of display
:radix=	- Select the base in which symbol values are displayed. Specify one of the following: octal, decimal, or hexadecimal. Default: hexadecimal.
:segment=	- Select which segments to display. Specify one or more of the following (separated by commas): all, absolute, text, data, stack, uconstant, sconstant, or unknown. Default: all.
:sort=	- Select the order symbols are displayed. Specify one of the following: name, value, none. Default: name.
:truncate	- (Default) Truncates symbol names in the display if they are longer than the display field width.

Description

Use this command to display symbols in image files (.EXE files) generated by the LL linker.

Examples

```
> li
```

Display all symbols in all image files in the current directory. (Non-image files are ignored. Image files generated by the LINK linker will always have "no symbols found" as will stripped LL image files.)

Parameters:

File list	> - Optional. Default: *. Type a list of file designations whose symbols are to be displayed. (see filelist)

Switches

File selection

:before=	- Type a date and time. (see dates)
:exclude=	- Type a list of file designations. (see filelist)
:mod	- Use file modification date for comparison
:since=	- Type a date and time. (see dates)
:uic=	- Type a list of uics or usernames. (see uiclist)

llran

Description

Use this command to create or maintain a packaged library file for the ll loader. The indicated relocatable object files generated by WiMAC are archived into the file SYMTAB.LL in the current directory.

Example:

```
> llran *.w :log
```

Generate the directory library file SYMTAB.LL in the current directory from all the .w files in the current directory. Display the names of the files being libraried.

Parameters:

File list	> - Required. Type a list of file designations whose attributes are to be displayed or modified. (see filelist).

Switches

:before=	- Type a date and time. (see dates)
:exclude=	- Type a list of file designations. (see filelist)
:log	- Log messages are displayed
:mod	- Use file modification date for comparison
:quick	- Allow or disallow scrutinizing the input
:since=	- Type a date and time. (see dates)
:uic=	- Type a list of uics or usernames. (see uiclist)
:verbose	- Display detailed information about files
:warnings	- Allow or disallow warning messages

The function return value is stored in register dØ. If the return value is of type double, then the most significant half of it is in dØ. The least significant half of it is in dl.

C and FORTRAN77

To write C-language procedures that call or are called by FORTRAN77, it is necessary to know procedure names, data representation, return values, and argument lists that the compiled code uses.

Procedure Names

The name of a FORTRAN procedure has an underscore added to it by the compiler. The underscore distinguishes the procedure name from a C-language procedure or external variable with the same user-assigned name.

Also, FORTRAN-library procedure names have embedded underscores to avoid clashes with user-assigned subroutine names.

Data Representation

The following is a list of corresponding FORTRAN and C declarations:

FORTRAN	C Language
integer*2 x	short int x;
integer x	long int x;
logical x	long int x;
real x	float x;
double precision x	double x;
complex x	struct { float r, i; } x;
double complex x	struct {double dr, di; } x;
character*6 x	char x[6];

Integer, logical, and real data occupy the same amount of memory in FORTRAN.

Return Values

A function of type integer, logical, real, or double precision declared as a C function returns the corresponding type. A complex or double complex function is equivalent to a C routine with an additional initial argument that points to the place where the return value is to be stored.

Chapter 9

Calling Functions Written in Other Languages

The calling function evaluates each actual parameter and pushes it on the stack from right to left.

The following type conversions are performed on each actual parameter value before it is pushed:

1. A float is converted to a double.

2. A char or short is converted to an int.

3. An unsigned char or unsigned short is converted to an unsigned int.

4. An array name is considered a pointer to the first element in the array.

5. A entire struct or union is pushed on the stack. Structs and unions can be very large. Everything on the stack gets pushed. It may have an extra dummy byte added to the end so an even number of bytes is always pushed.

After the parameters have been pushed, the caller calls the function. The C compiler adds an underscore to the beginning of function names.

The called function preserves registers d2 through d7 and a2 through a7. Registers dØ, dl, aØ, and al are not preserved. These four registers are called scratch registers.

Once the called function has returned, the parameters are popped off the stack.

NOTE: The caller pops the parameters from the stack. This is helpful with functions that have a variable number of parameters (such as printf()) because the caller knows how many parameters to pop.

The string lengths are long int quantities passed by value.

The arguments are in the following order:

1. Extra arguments for complex and character functions

2. Address for each datum or function

3. A long int for each character argument

Because of this, the call in

```
external f
character*7 s
integer b(3)
...
call sam(f, b(2),s)
```

is equivalent to the call in the following:

```
int f();
char s[7];
long int b[3];
...
sam_(f,&b[1],s,7L);
```

NOTE: The first element of a C array always has subscript Ø.

However, FORTRAN arrays begin at 1 by default. FORTRAN arrays are stored in column-major order. C arrays are stored in row-major order.

C and Pascal

C functions cannot call Pascal routines because of unresolvable differences between the C and Pascal calling conventions. However, C functions can be called by Pascal routines because the Pascal compiler generates C style calls through the cexternal directive. Also, external C variables are not accessible from Pascal.

<u>Limitations</u>

The C library and the Pascal runtime library are not compatible. The C memory allocation routines (malloc, etc.) conflict with the Pascal memory allocation routines (mark, new, etc.).

Since many library routines allocate memory internally, it is difficult to tell what will work and what won't.

Because of this, the following:

```
complex function f(...)
```

is equivalent to

```
struct { float r, i;} temp;
f_(&temp,...)
```

A character-valued function is equivalent to a C routine with two extra initial arguments, a data address and a length. Therefore,

```
character*15 function g(...)
```

is equivalent to the following:

```
char result[];
long int length;
g_(result,length,...)
...
```

The foregoing could be invoked in C by the following:

```
char chars[15];
...
g_(chars,15L,...);
```

Subroutines are invoked as if they are integer-valued functions, whose value specifies which alternate return to use. Alternate return arguments (statement labels) are not passed to the function. They are used to do an indexed branch in the calling procedure.

The return value is undefined if the subroutine has no entry points with alternate return arguments.

The statement

```
call nret(*1, *2, *3)
```

is treated exactly as if it were the computed goto

```
goto (1, 2, 3), nret()
```

Argument Lists

All FORTRAN arguments are passed by address. In addition, for every argument that is of type character, an argument giving the length of the value is passed.

C enumerated types are always 4 bytes long. They do not correspond to Pascal enumerated types.

(3) Member Ø of the Pascal set corresponds to the least significant bit of the C unsigned char. Member 1 corresponds to the next significant bit.

(4) Member Ø of the Pascal set corresponds to the least significant bit of the first C unsigned long int. Member 63 corresponds to the most significant bit of the last unsigned long int.

Return Values

To be called from Pascal, a C function should be of type int, long int, unsigned int, unsigned long int, double, pointer, or void.

A C function of type void corresponds to a Pascal procedure.

Argument Lists

To be called from Pascal, a C function should not have formal parameters of type float, struct, or union. In addition, the Pascal compiler does not account for the type conversions listed in this chapter. Therefore, a C formal parameter of type char or short corresponds to a Pascal actual parameter of type longint.

Array parameters work as long as subscripting is declared to begin at Ø in Pascal.

In addition, some routines in each library depend on initializations done by the main program. If the main program is written in C, the Pascal initializations aren't done. If the main program is written in Pascal, the C initializations aren't done.

Therefore, you can write C functions to be called by Pascal routines, but you do so at your own risk.

The cexternal directive exists primarily so that Pascal can access UNIX system calls.

If you want to try to call C functions with Pascal routines, the following information may be of help.

Procedure names

To be called from Pascal, the name of the C function must not contain any uppercase letters. Pascal is not case sensitive. It assumes that the names of all C functions are in lowercase.

Furthermore, a C function called foo must be declared as _foo in Pascal. The C compiler puts a leading underscore on function names, but Pascal does not.

Data Representations

Following is a list of corresponding Pascal and C declarations:

Pascal	C	Notes
x: char;	unsigned char x;	
x: boolean;	char x;	(1)
x: integer;	short int x;	
x: longint;	long int x; or int x;	
x: (red, green, blue);	unsigned char x;	(2)
x: real;	double x;	
x: record i, j: integer end;	struct {short int i, j;} x;	
x: string [9];	char x[1Ø];	
x: array [Ø..9] of char;	char x[9];	
x: ^char;	char *x;	
x: set of Ø..1;	unsigned char x;	(3)
x: set of Ø..63;	unsigned long int x[2];	(4)

(1) False corresponds to Ø, true corresponds to 1 (in most cases, true corresponds to any value other than Ø).

(2) Red corresponds to Ø, green corresponds to 1, and blue corresponds to 2. Pascal enumerated types are 1 byte long if they have 256 values or less. Otherwise they are 2 bytes long.

can be changed temporarily to remove the static modifier for debugging.

Auto variables (and formal parameters)

Auto variables also have no symbols. Because an auto is a local variable, it is usually easier to find a nearby reference to it than to a static variable.

An auto variable's location can be estimated. An auto variable is located at a negative offset from register a6. Generally, the C compiler places the first auto variable nearest a6, the next auto further from a6, etc.

A formal parameter is accessed like an auto variable is accessed except it is located at a positive offset from a6.

Register variables

Register variables have no symbols, but, like auto variables, references to them can often be found in the code (see chapter 3 for details)

Also, it is possible to determine the hardware register that corresponds to a given register variable in functions that have not been optimized (see chapter 3 for details).

A register parameter is like a register variable except that an initial value gets copied into the register at the beginning of the function.

How to Locate Code

Locating a function

Non-static functions are referred to by name. The name is preceded by an underscore. The name represents the address of the first instruction in the function.

Static functions cannot be referred to by name. Like static variables, a static function can best be located by finding a reference to it in the code, then noting its address.

Locating lines in a function

If the C optimizer is not used, specific lines in a function can be located by examining the .s assembly language source file produced by the C compiler.

Chapter 10

Debugging

The WMCS debugger is WIBUG. The UniPlus+ System V debugger is adb. Both are symbolic assembly-level debuggers, that is, they allow functions and variables to be accessed by name at the assembly-language level.

For detailed information on WIBUG see the WIBUG Programmer's Reference Manual.

For detailed information on adb see the UniPlus+ System V User's Manual (Section 1).

Because WIBUG and adb have roughly equivalent capabilities, they will be referred to in this chapter as WIBUG/adb.

How to Locate Data

External variables can be referred to by name. However, static, auto, and register variables have no symbols with which they can be accessed. Following are tips for accessing each storage class:

External variables

An external variable can be referred to by its name, preceded by an underscore.

For example, a global variable called foo in C is called _foo in WIBUG/adb. The symbol represents the address of the first byte of the variable. WIBUG/adb can locate the variable, but has no information about its type or size.

Static variables

Static variables have no symbols, and their location in memory is hard to predict. A static variable is best found by locating a reference to it in the code, and then noting its address. It is also possible to declare all statics with a macro so they the optimizer performs live/dead analysis on them. A simple change in the C source can result in a drastic change in the optimized assembly code.

The optimizer removes the LINK and UNLK instructions from a function if it has no non-register local variables. This causes parameters to be at an unpredictable offset from the stack pointer, rather than at a predictable offset from register a6.

Stack backtraces in WIBUG/adb depend on the linked list of stack frames maintained by the LINK and UNLK instructions. Backtraces of optimized code can be misleading because functions without LINK and UNLK do not appear in the backtrace.

The optimizer remaps register variables to registers with lower numbers. This takes advantage of unused scratch registers. Remapping makes it more difficult to calculate which hardware register corresponds to which register variable.

To produce a .s file under WMCS, use the compile command with the :noassemble switch. To produce a .s file under UniPlus+ System V, use the cc command with the -S option.

Source file names and line numbers are indicated by .line and .file directives among the assembly-language statements.

The beginning of the C source file and each #include file is marked by a .file directive in the following format:

```
.file  file_number,"file_name"
```

In the foregoing format, file_number is a unique integer and file_name is the name of the source file. The end of each #include file is marked by an abbreviated .file directive in which the file name is omitted, and the file number is the number of the file that did the #include.

The beginning of each C statement is marked by a .line directive in the following format:

```
.line  line_number
```

In the foregoing format, line_number is the number of the line on which the statement begins.

A .line directive appears only for lines containing the beginning of a statement.

Lines containing only comments, declarations, etc. are not represented.

Debugging Optimized Code

Debugging code that has been optimized with the C optimizer is difficult. Your ability to debug optimized code depends on the optimizations that were performed.

Following are some things to watch out for:

The C optimizer removes all .file and .line directives. It does this because it moves, eliminates, and rearranges code, invalidating the line numbers. To locate lines in a function, you must look at the assembly code and try to correlate it with the source manually.

However, the optimized assembly code may not correspond in an obvious way to the original C source. The code for a C statement could be dispersed, or eliminated altogether. This is most likely to occur with the use of register variables because

ASCII CHARACTER TABLE

Abbreviations for control key functions:

NUL	- Null	DC1	- Device control 1
SOH	- Start of header	DC2	- Device control 2
STX	- Start of text	DC3	- Device control 3
ETX	- End of text	DC4	- Device control 4
EOT	- End of transmission	NAK	- Negative acknowledge
ENQ	- Enquiry	SYN	- Synchronous idle
ACK	- Acknowledge	ETB	- End of transmission block
BEL	- Bell	CAN	- Cancel
BS	- Backspace	EM	- End of medium
HT	- Horizontal tab	SUB	- Substitute
LF	- Line feed	ESC	- Escape
VT	- Vertical tab	FS	- File separator
FF	- Form feed	GS	- Group separator
CR	- Carriage return	RS	- Record separator
SO	- Shift out	US	- Unit separator
SI	- Shift in	SP	- Space
DLE	- Data link escape	DEL	- Delete

APPENDIX A

ASCII CHARACTER TABLE

Character	DEC	HEX	Char	DEC	HEX	Char	DEC	HEX	Char	DEC	HEX
[CTRL]@ NUL	000	00	SP	032	20	@	064	40	`	096	60
[CTRL]a SOH	001	01	!	033	21	A	065	41	a	097	61
[CTRL]b STX	002	02	"	034	22	B	066	42	b	098	62
[CTRL]c ETX	003	03	#	035	23	C	067	43	c	099	63
[CTRL]d EOT	004	04	$	036	24	D	068	44	d	100	64
[CTRL]e ENQ	005	05	%	037	25	E	069	45	e	101	65
[CTRL]f ACK	006	06	&	038	26	F	070	46	f	102	66
[CTRL]g BEL	007	07	'	039	27	G	071	47	g	103	67
[CTRL]h BS	008	08	(	040	28	H	072	48	h	104	68
[CTRL]i HT	009	09	)	041	29	I	073	49	i	105	69
[CTRL]j LF	010	0A	*	042	2A	J	074	4A	j	106	6A
[CTRL]k VT	011	0B	+	043	2B	K	075	4B	k	107	6B
[CTRL]l FF	012	0C	,	044	2C	L	076	4C	l	108	6C
[CTRL]m CR	013	0D	-	045	2D	M	077	4D	m	109	6D
[CTRL]n SO	014	0E	.	046	2E	N	078	4E	n	110	6E
[CTRL]o SI	015	0F	/	047	2F	O	079	4F	o	111	6F
[CTRL]p DLE	016	10	0	048	30	P	080	50	p	112	70
[CTRL]q DC1	017	11	1	049	31	Q	081	51	q	113	71
[CTRL]r DC2	018	12	2	050	32	R	082	52	r	114	72
[CTRL]s DC3	019	13	3	051	33	S	083	53	s	115	73
[CTRL]t DC4	020	14	4	052	34	T	084	54	t	116	74
[CTRL]u NAK	021	15	5	053	35	U	085	55	u	117	75
[CTRL]v SYN	022	16	6	054	36	V	086	56	v	118	76
[CTRL]w ETB	023	17	7	055	37	W	087	57	w	119	77
[CTRL]x CAN	024	18	8	056	38	X	088	58	x	120	78
[CTRL]y EM	025	19	9	057	39	Y	089	59	y	121	79
[CTRL]z SUB	026	1A	:	058	3A	Z	090	5A	z	122	7A
[CTRL][ESC	027	1B	;	059	3B	[	091	5B	{	123	7B
[CTRL]\ FS	028	1C	<	060	3C	\	092	5C	|	124	7C
[CTRL]] GS	029	1D	=	061	3D	]	093	5D	}	125	7D
[CTRL]^ RS	030	1E	>	062	3E	^	094	5E	~	126	7E
[CTRL]_ US	031	1F	?	063	3F	_	095	5F	DEL	127	7F

14 **Are any additional characters allowed in an identifier?** No additional characters are allowed in identifiers (only letters, digits, and the underscore, _, are allowed).

16 **Are enum and void implemented?** Yes, both are implemented.

16 **Is entry a reserved word?** No.

16 **What are the reserved words in WICAT Systems C?** The reserved words are listed in appendix C of this manual.

17 **Are the digits 8 and 9 allowed as octal digits?** Yes, but a warning message is generated.

17 **What are the storage sizes of the data types?** The storage sizes are given in chapter 3.

18 **How are out-of-range values handled?** Constants larger than MAXINT (2147483647) are silently truncated, and no warning or error message is generated.

19 **What representation does the 68000 use for integers?** The twol complement representation is used for integers.

21 **How is the char constant implemented?** Character constants are implemented as 8-bit signed types, i.e., the sign extends.

23 **How is the backslash treated when it is followed by an invalid escape code?** The backslash, \, is ignored when it is followed by an invalid escape-code character.

25 **Is hexadecimal notation allowed in numeric escape codes?** Yes. The hexadecimal escape code \x is allowed in character constants (e.g., "\x1A").

26 **Can the preprocessor and the compiler be operated separately?** Yes. The compilation process is described in chapter 2.

27 **Are the preprocessor commands #elif and defined supported?** The preprocessor command defined is supported but #elif is not.

28 **Is leading space and whitespace allowed with the macro commands?** No leading space, whitespace yes.

33 **Can actual argument token lists extend across multiple lines?** Backslash required.

Appendix B

Supplement to C: A Reference Manual

The book C: A Reference Manual supplied by WICAT Systems raises questions about differences between various implementations of C. This appendix clarifies those questions so that C: A Reference Manual, along with this appendix, is a complete language reference for WICAT Systems C. These answers are given in the order the questions are presented in the book.

Page #	Answer
9	**Does the character set include additional characters not in the standard set** Yes. WICAT supports the complete ASCII character set (e.g., the $, @, and ` characters are included). The characters not included in the standard set can only appear in comments, character constants, or string constants.
10	**What is the line limit of a C program?** 256 characters. Different parts of the software generation system have different line limits. The shortest limit is 256 characters for the floating-point preprocessor.
12	**Are nested comments allowed?** Nested comments are accepted. However, lint generates a warning message for nested comments.
14	**What is the maximum length of an identifier?** Identifiers of any length are allowed. However, global and static identifiers are declared in the assembly output on one line, similar to this:

```
.global <identifier>
```

The entire line must be less than 256 characters to work with the floating-point preprocessor, so the practical limit to the length of an identifier is about 200 characters.

The call CONC(INC,TAB) is ultimately expanded into this:

```
table_of_increments
```

40 **Does the preprocessor perform stringent error checking?** The preprocessor does not check for things such as an incomplete token in the macro definition.

40 **How are double quotes and angle brackets treated in #include statements?** Files listed in double quotes are only searched for in the directory the source is in. Files listed in angle brackets are searched for in the standard directories and any other directories specified with compiler options. The directories you specify are searched first.

41 **What are the standard include directories?** The standard include directories are:

```
/usr/include            (UNIX only)
sys$disk/ucc.include/    (WMCS only)
sys$disk/sysincl.sys/    (WMCS only)
```

41 **Are nested #include statements allowed?** Yes. Nesting is allowed to 16 levels.

46 **How are errors in constant expressions in preprocessor conditional commands handled?** When an error occurs in a constant expression, no warning message is generated and the value is assumed to be zero.

48 **Is # allowed for #line?** Yes. The line "# <number>" is the same as "#line <number>"

34 **Are macro formal parameters recognized within string and character constants?** Yes. Formal parameters will have the same textual form as the actual parameters when expanded with the exception that comments are deleted. For example, consider the definition and call below:

```
#define X(x)    "x"

X( a += 00400 /* foo blat */ )
```

The definition would cause the call to expand to this constant:

```
" a += 00400  "
```

34 **How are comments within macros treated?** They are not passed when a macro definition is substituted (see the example in the foregoing comment).

36 **What are WICAT's predefined macros?** The only predefined macros are:

```
mc68000
unix          (for UNIX systems)
wmcs          (for future WMCS systems)
```

36 **How is an attempt to redefine a macro handled?** Macros can be redefined. The new definition replaces the old one, and the preprocessor generates a warning message, including warnings about parameter mismatches.

37 **Are macro definitions implemented with a stack?** No. The preprocessor does not stack macro definitions (defined with #define). X would not be defined as 10 after the following three commands, as the example on p. 37; x would be undefined:

```
#define x 10
#define x 12
#undef
```

39 **Are macro bodies treated as character sequences?** Yes. Consider the following example, given on p. 39. of the text:

```
#define INC ++
#define TAB internal_table
#define INCTAB table_of_increments
#define CONC(x,y) x/**/y
CONC(INC,TAB)
```

74 **Is compile-time floating point arithmetic performed?** Yes.

75 **Are casts allowed in constant expressions?** Yes.

76 **Can automatic arrays be Initialized?** No.

77 **Are braces allowed in enumeration initialization expressions?** Yes.

78 **Can bit fields be initialized?** Yes, static and extern bit fields can be initialized.

79 **Can unions be initialized?** No.

80 **Are too few or too many braces allowed in initializer lists?** No. WICAT Systems C strictly conforms to the "Brace Eliding" rules given on p. 79 of the text.

80 **Are pointers and ints the same size?** Yes. Chapter 3 lists the storage size for each data type.

81 **When is a top-level declaration of an external name considered to be its definition?** The compiler uses the "mixed" strategy, described on p. 82 of the text, to define external names. With an initializer but no extern, it is a definition. With no initializer and no extern, it is a "common" definition. If extern is there, the definition occurs elsewhere.

87 **What are the sizes of short, int, and long?** The sizes for all data types are given in chapter 3 of this manual.

89 **What unsigned types are supported?** The compiler supports unsigned long, short, and char types.

90 **How is the char type implemented?** The char type is signed. Therefore it can assume negative values. The size of char, along with the sizes of all data types, is given in chapter 3 of this manual.

93 **Is long float allowed?** No, the compiler does not recognize "long float" as a synonym for double.

98 **What is the maximum dimension of an array?** The compiler can handle up to 13 dimensions of an array. In general, the compiler can handle 13 levels of indirection.

99 **What is the unit of measurement returned by the sizeof operator?** The sizeof operator returns the size of an array in bytes.

54 **Are labels placed in the same space as variables?** No. Labels have a separate name space from variables. In this example, given on p. 54, the integer declaration of L hides the label (it is not an illegal duplicate definition of L):

```
{   ...
    goto L;
    ...
    {   int L;
        ...
        {   ...
            L = 10;
            ...
          L:
            ...
        }
    }
}
```

56 **How are forward references to static variables handled?** Forward references to static variables with "extern" do not change the storage class of the variable, i.e., it remains static.

59 **Are the normal scoping rules different for external declarations?** Yes. The normal scoping rules for extern declarations are "violated." The following example, given on p. 59, extends the definition of E to the second assignment (it is not illegal):

```
{
    {
        extern E;
        E = 0;
    }
    E = 1;
}
```

61 **How many register variables are available?** The use of register variables is described in chapter 3 of this manual.

69 **Are zero-length arrays allowed?** No, but null-sized arrays are allowed, even in some contexts where it does not seem to make sense. For example, the following function compiles even though it is improper:

```
func() {
       char carray[];
       sprintf(carray, ...);
}
```

156 **Can the address operator be used with a register variable?** No, you cannot take the address of a register variable.

157 **Can the address operator be used with an array or function?** The compiler generates a warning message if you try to take the address of an array or a function.

163 **How is integer division with negative numbers handled?** In integer division involving negative numbers where the mathematical quotient is not an exact integer, the result is the nearest integer which is closest to zero (i.e., the same as for positive numbers).

169 **How does the signed right shift work?** Right shifts of signed numbers replicate the sign bit, i.e., the sign bit is extended.

183 **Can the result of a conditional expression be structure, union, enumeration, or void?** Yes, any of them.

185 **Are structure and union assignments allowed?** Yes.

186 **Is whitespace allowed between the characters of a compound assignment operator?** Yes. For example, "i + = 5" is allowed.

187 **Are the "old style" compound assignment operators recognized?** The "old style" compound assignment operators described on pp. 186-187 are recognized and generate a fatal error. Statements like "i=-4" produce warning messages about ambiguous assignments.

189 **Is the exclamation point, !, allowed in constant expressions?** Yes.

190 **Is casting allowed in constant expressions?** Yes.

190 **Is the comma operator allowed in constant expressions?** No.

193 **Is a warning message generated for discarded values?** No.

194 **Does the compiler optimize memory access?** No.

216 **Are the enum and long types allowed in switch statements?** Yes.

102 **What is the size of an enum type?** The size of all data types is given in chapter 3 of this manual.

102 **Can previously defined enum constants be used in enum constant expressions?** Yes.

1Ø3 **How is the enum type implemented?** The enumeration type is implemented as an integer model.

1Ø6 **What structure operations are possible?** Structures can be assigned, passed as parameters, and returned as function values.

1Ø8 **Can two structures have components with the same name?** Yes, the overloading of structure component names is allowed.

1Ø8 **How are structure components packed?** Structure components larger than the size of char always start at an even-byte offset.

1Ø9 **How are bit fields packed?** Bit fields are packed from left to right.

11Ø **How are bit fields implemented?** Bit fields must be unsigned (int or enum type). Signed declarations produce no warnings or errors and are treated as though unsigned. Fields are limited to 32 bits (one longword). Fields too large to fit entirely in the current longword are aligned to the next word boundary. Zero-length bit fields are not supported.

14Ø **How are overflow, underflow, division-by-zero, and other arithmetic exceptions handled?** Overflow, underflow, and other arithmetic exceptions do not generate error or warning messages. The (unpredictable) results are propagated through future results. Division by zero, however, generates an error message.

145 **Does enclosing an expression in parentheses force a unary conversion?** No.

148 **Can functions return structures and unions?** Yes.

15Ø **Can formal parameters be used in a function expression?** No, you cannot specify a formal parameter declared as being of type "function returning T" for some type T.

153 **Are "narrowing casts" performed by the compiler?** Yes.

154 **What is the type of the sizeof result?** The result type of the sizeof operator is unsigned int. The size is given in bytes.

Appendix C

Keywords

The keyword (reserved word) list for WICAT's C compiler conforms to the proposed ANSI standard for C.

If you use a keyword as an identifier, the compiler returns a syntax error.

Following is a list of the keywords for the WICAT C compiler. The words followed by an asterisk are reserved due to the proposed standard:

asm	enum*	struct
auto	for	sizeof
break	float	short
char	fortran	static
case	goto	typedef
const*	if	unsigned
continue	int	union
double	long	void*
default	return	volatile*
do	register	while
extern	signed*	
else	switch	

Index

WICAT Systems, Inc.
Product-documentation Comment Form

We are constantly improving our documentation, and we welcome specific comments on this manual.

Document Title: ______________________________

Part Number: ______________________________

Your Position:

☐ Novice user ☐ System manager

☐ Experienced user ☐ Systems analyst

☐ Applications programmer ☐ Hardware technician

Questions and Comments **Page No.**

Briefly describe examples, illustrations, or information that you think should be added to this manual.

______________________________ ________
______________________________ ________
______________________________ ________

What would you delete from the manual and why?

______________________________ ________
______________________________ ________
______________________________ ________

What areas need greater emphasis?

______________________________ ________
______________________________ ________
______________________________ ________

List any terms or symbols used incorrectly.

______________________________ ________
______________________________ ________
______________________________ ________

First Fold

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 00028 OREM, UTAH

POSTAGE WILL BE PAID BE ADDRESSEE

WICAT Systems, Inc.
Attn: Corporate Communications
1875 S. State St.
Orem, UT 84058

Second Fold

Tape