# *C89

MTS C Staff
University of Michigan
Information Technology Division

## Contents

## 1 Overview

∗C89 is a C compiler derived from the C/370 compiler developed by AT&T Information Systems, although almost all of the original AT&T code has been replaced. ∗C89 now has numerous improvements and is in conformance with the *American National Standard for Information Systems – Programming Language C (ANSI X3.159-1989)*, hereafter referred to as the *ANSI C Standard*.

The ∗C89 compiler and library attempt to satisfy three goals:

1. to provide a good implementation of the standard language and library,
2. to support many of the UNIX (UNIX is a registered trademark of AT&T Information Systems) library routines, and
3. to provide support for Michigan Terminal System (MTS) routines.

## 2 Documentation

This document does not provide information on the C language or C library except where features are implementation-defined or are local extensions.

A copy of a recent ANSI C Standard can be obtained from Dollar Bill Copying on Church Street near South University (Ann Arbor, Michigan, USA). For information on the UNIX routines that are not part of the ANSI Standard, refer to *UNIX Programmer's Reference Manual (PRM) – 4.3 Berkeley Software Distribution*, from University of California, Berkeley.

## 3 How to Run ∗C89 in MTS

The MTS command to run the ∗C89 compiler is

```
        $RUN *C89 units PAR=options
```

In some cases it may be desirable to add some options to control the compilation. In particular, it may beconvenient to organize a group of include files into an include library. An include library is a mechanism that allows a single file to be partitioned into one or more "members," each of which corresponds to a single header file. These differences provide a way to compile a C program without changing the `#include` lines in the source code. There may be cases in which a file on some other system is named in such a way that it can't exist on MTS.

An include library consists of two parts, a directory and the contents of the include member. The directory starts at the first line in the file and is terminated by a line with eight zeroes. Each line of the directory has a member name and the line number (separated by one or more blanks) where the member is located in the file. Members are separated by $ENDFILEs.

Whenever a `#include` is given, *C89 will search through all specified include libraries for the "file."

Unlike normal MTS files, a member of an include library has a name that

- can be up to 255 characters.
- is case sensitive (e.g., `XYZ.h` is different from `xyz.h`).
- may consist of any characters other than blanks.

There may be cases in which a file on some other system is named in such a way that it can't exist on MTS. A program residing on such a system can be moved to MTS if the file names and the `#include` lines are changed to reflect MTS restrictions. An include library provides a way to compile such C programs without changing the `#include` lines in the source code. However, in many cases it is not necessary to use include libraries. Instead, individual include files can be placed in separate MTS files provided the restrictions on MTS file names are satisfied.

## 3.1 Compiler Unit Assignments

The *units* described above may be assigned as follows:

`INPUT=` source program.

`SERCOM=` error messages.

`PRINT=` source and object listing.

`OBJECT=` object module.

`2=` additional `#include` libraries containing header files.

`99=` if assigned, the PAR field will be read from this unit.

Typically, only `INPUT` and `OBJECT` are assigned.

*C89 does not allow `INPUT` to be assigned to the same file as any of `SERCOM`, `PRINT` or `OBJECT` nor does it allow 2 to be assigned to the same file as any of `SERCOM`, `PRINT` or `OBJECT`. Exception: if `INPUT` is the terminal, *C89 will allow `INPUT` to be the same as `PRINT`, `SERCOM`, or both.

## 3.2 Return Codes from the Compiler

The MTS return codes generated by *C89 are as follows:

**0** The compilation was successful (except for possible warning messages).

**4** A file required by *C89 couldn't be accessed. (*Exception:* `#include` files that can't be accessed cause rc=8.)

**8** One or more compile-time errors were detected or an `include` file couldn't be accessed.

**12** The compiler ran out of memory.

**16** An internal failure occurred in *C89.

Note: If two or more of these conditions apply, the condition with the highest number prevails. That is, if there are compilation errors and *C89 ran out of memory, the return code is 12.

### 3.3 Character Graphics

If a terminal doesn't support the full range of characters needed to program in C, it is possible to use the alternate character representations described in the ANSI C Standard. Some terminals may display certain characters in a form that may not be familiar. The circumflex (ˆ) appears on some terminals as an up-arrow (↑) and the tilde (˜) appears on some terminals as a logical negation(¬).

Also, the circumflex and backslash (\) will not print on the line printer and may print as other graphics in some fonts on the page printer.

## 4 Pragmas and Par Field Options

There are numerous options built into the ∗C89 compiler. These options can be specified either in the `PAR` field of an MTS $RUN command or in a `#pragma` command in the source program. Since MTS doesn't allow command lines longer than 256 characters, ∗C89 will read the `PAR` field information from unit `99` if it has been assigned. This optional method of specifying the `PAR` field is most useful for macros that generate commands. The `PAR` field and `#pragma` option names are case-insensitive and may be upper, lower, or mixed case. Most options can be negated by prefixing them with "∼" or "NO".

When these options are specified in the `PAR` field, their scope is the entire compilation unit. When specified in a pragma, the scope is indicated with each description.

The ANSI C Standard gives no guidance on the names and syntax of pragmas, so they are unique to this implementation. Note that the ANSI C Standard allows unrecognized pragmas to be ignored without an error, so many of the following pragmas may not impair portability, although some might conflict with a pragma on another compiler.

Of the multitude of options, only a few are frequently used. These are:

`SYM` to aid in debugging,

`PROPER` to have the compiler diagnose as many suspect or non-standard constructions as possible, and

**`UNIX4.3`, `UNIX` or `UNIX+`** to port programs from UNIX.

The options and pragmas are:

`AMODE`

This option can be set to `24`, `31`, or `ANY`. This tells the loader what the addressing mode of a given control section is. For example, if a program can only operate in 24 bit mode, then this option must be set. For most programs this will not be necessary. The default is AMODE=ANY.

Pragma Scope: The value at the end of a function determines how the code for the function is treated by the loader.

`ASCII` (default `NOASCII`)

This option causes all character strings and character constants to be converted to ASCII internally. It is the user's responsibility to be careful with this option. If a `PAR=ASCII` routine calls a routine that isn't `ASCII`, there may be serious problems.

For the most part, the ∗C89 run-time system assumes EBCDIC. For example, `getc`, `scanf`, etc., all return EBCDIC characters, and the `printf` and `scanf` formats must be expressed in EBCDIC. This may change at some point in the future.

Until the library supports ASCII characters (e.g., for `printf` formats), this option is of little utility.

Pragma scope: Takes effect immediately.

`CHECKIOVER` (default `NOCHECKIOVER`)

This option causes the compiler to generate code to check that the results of all signed integer computations fit into 32 bits. Turning this on would be most useful when using the `signal` facility to capture interrupts for integer overflow.

Pragma scope: The last value is used for the entire compilation unit.

`CHECKSTACK` (default `NOCHECKSTACK`)

This option causes the compiler to generate code to check that the stack doesn't overflow. If the size of the stack is exceeded from within a routine compiled with this option, an interrupt occurs. Note that ∗C89LIB is not compiled with this option.

Pragma scope: The value at the end of a function is used to determine how to generate code for that function.

`DEFINC` (default `DEFINC`)

This option causes the compiler to search the default include library (∗C89INCLUDE). `NODEFINC` would only be used to insure that all symbols are resolved by the user-specified include libraries. This is intended only for use with libraries other than ∗C89LIB.

If `NODEFINC` is used, a definition for <`unix.h`> must appear in a user-specified include member or in an include library.

Pragma scope: Takes effect immediately.

`DEFINE(x)`

This is processed as if a `#define x 1` had been issued before the source program had started. `x` may be any legal identifier. `x` is *not* translated to upper case before use.

This option cannot be negated.

Not allowed as a pragma; use `#define`.

`DEFINE(x=e)`

This is processed as if a `#define x e` had been issued before the source program had started. `x` may be any legal identifier. `e` can be any sequence of tokens. `x` and `e` are *not* translated to upper case before use.

This option cannot be negated.

Not allowed as a pragma; use `#define`.

`DEPEND (default NODEPEND)`

When this option is in effect, a list containing the names of all the source files and all the include files used by the compiler in a given compilation is printed on the logical I/O unit `PRINT`. When the `DEPEND` option is turned on, the `LIST` option is turned off and the compiler does not parse the source nor does it produce an object module (regardless of the setting of the `OBJECT` option).

The list produced may contain duplicate file names. This option is intended for production of makefile dependencies.

Pragma scope: Takes effect immediately.

`FILL=c`(default `NOFILL`)

This option specifies a `char`-sized value that is used at run-time to initialize each stack frame. This option can be used in debugging when a problem with an uninitialized variable is suspected.

The value of *c* must fit into a single character, but can be expressed as any integer constant. For example, each of the following has the same effect.

```
#pragma fill='a'
#pragma fill=0x81
#pragma fill=129
```

Pragma scope: The last value given this option within a function is used for that function.

`I=cccc` (where cccc is an MTS userID)

This affects `#include` searches. This adds the userID `cccc` to the list of userIDs to be searched when a quoted `#include` file is specified. For example,

```
$run *C89 input=... par=i=W123 i=W456
```

Where the source file contains the following include:

```
#include "ink.h"
```

*C89 will first look for the file `W123:ink.h`. If that doesn't exist, it will look for the file `W456:ink.h`, and if it isn't there, it looks for the file `ink.h` on the current userID. This option may be specified more than once and the userIDs are searched in the order specified.

This option is useful when compiling a program developed on one userID from another userID.

`#include` directives that use angle brackets are unaffected by this option.

This option cannot be negated.

Pragma scope: Takes effect immediately.

`LANG=x` (default `LANG=ANSI`)

The `LANG` option defaults to processing the ANSI C Standard language. To aid in porting programs from UNIX, a few additional syntax constructions are allowed that are non-standard but commonly accepted by traditional Kernighan and Ritchie compilers, when `LANG=K&R` is specified. For example, text appended to `#endif` will be ignored if the `LANG=K&R` option is set.

`LIST` (default `NOLIST`)
`LIST=n` (default `LIST=0`)

The `LIST` option controls echoing of source lines to `PRINT`. The `LIST` identifier by itself is the same as `LIST=1`. `NOLIST` is the same as `LIST=0`.

If `LIST=1`, everything except `#include` files is listed. If `LIST=2`, everything, including `#include` files, is listed.

If the logical I/O unit `PRINT` is assigned and the `DEPEND` option is off, `LIST` is set to `1`.

Pragma scope: Takes effect immediately.

`LOADNAME(`*cname*`,`*lname*`)`

This option causes the external name *cname* in the source program to be mapped into the name *lname* in the object module. This option can be used to allow C programs to access system names that don't conform to C syntax.

Pragma scope: The same scope as *cname*.

`LONG` (default `LONG`)

This option suppresses truncation of external names. If `NOLONG` is specified, external names are truncated to eight characters.

Pragma scope: The last value is used for the entire compilation unit.

`MC` (default `NOMC`)

Normally, external names are mapped to upper case; for example, `fopen` is changed into `FOPEN`. If `MC` is selected, then external names are left in the original case, i.e., mixed case. Note: The

∗C89 run-time system is compiled with `NOMC`, and hence, `MC` prevents the use of ∗C89LIB and makes use of *OBJUTIL, SDS and similar programs.

Pragma scope: The last value is used for the entire compilation unit.

`MTSLINE` (default `NOMTSLINE`)

This option determines how `#line` commands are interpreted and how SYM records are generated. If `MTSLINE` is in effect, the numbers in these commands are assumed to be MTS line numbers multiplied by 1000. For example:

```
        #line 1500
```

is assumed to refer to line 1.500. If `NOMTSLINE` is specified, then numbers are assumed to be integral line numbers. For example:

```
        #line 45
```

is assumed to refer to line 45. `NOMTSLINE` is useful when compiling source code that originated on a non-MTS system.

Pragma scope: Takes effect immediately. If no `#line` commands appear in the source file, `SYM` records are generated assuming MTS line numbers. Otherwise, the value of the `MTSLINE` option at the time of the last `#line` command in the source file determines how `SYM` records are produced by the compiler.

`OBJECT` (default `OBJECT`)

Object code generation may be suppressed with `NOOBJECT`.

Pragma scope: The last value is used for the entire compilation unit.

`OBJLIST` (default `NOOBJLIST`)

If `OBJLIST` is specified, an object listing is written to the I/O unit `PRINT`.

Pragma scope: The last value is used for the entire compilation unit.

`OPT=n` (default `OPT=1`)

`OPT=0` causes no analysis of the intermediate code to be made to determine code size or register usage, so code is generated using worst-case assumptions. Compilation is slightly faster than `OPT=1`.

`NOOPT` can be used as an alternative to `OPT=0`.

`OPT=1` causes two passes to be made over the intermediate code. A first quick pass makes an analysis of the code size and register usage. A second pass is then made to generate code that is usually more efficient.

`OPT=2` additionally will use peephole optimization techniques to clean up various inefficiencies in the generated object code (e.g., eliminate dead code, eliminate redundant loads) Also `OPT=2` causes variables to be assigned to registers under certain conditions (see section 5.8, "Registers," for more information).

Pragma scope: The value at the time of a declaration of a given variable determines if that variable will be considered for assignment to a register. For other properties of `OPT`, the value at the end of a function determines how that function will be compiled.

`PORT` (default `NOPORT`)

If `PORT` is specified, ∗C89 prints warnings that indicate which parts of a program might not be portable. This will not detect all violations; neither does the presence of a warning guarantee a non-portable program nor does the absence of a warning guarantee a portable program.

The following cause a portability warning. Other features may also be added to cause portability warnings.

- The `__fortran`, `__retcode`, `__prvbase`, and `__pseudoregister` extensions.
- The `FILL`, `LOADNAME`, `PROTO=0`, `RCALL` and `ZEROARG` pragmas.
- Casts between pointers and integers.
- Casts between pointers to different types.
- Uses of identifiers beginning with an underscore in ways that conflict with the ANSI standard.

Pragma scope: Takes effect immediately.

`PROPER`

The `PROPER` pragma stands for a collection of other pragmas that would all be used if one were trying to write a maximally portable and correct program. The pragmas set by `PROPER` are:

```
#pragma port standard warn=4 proto=4
```

Pragma scope: Takes effect immediately.

`PROTO=n` (default `PROTO=2`)

The `PROTO` option controls whether prototypes are required and how they are used. It can have the following values:

**PROTO=0** All prototypes are ignored.

**PROTO=1** Permits the last parameter to be omitted without generating an error or warning. This is to be used with the `ZEROARG` option for compatibility with certain UNIX implementations (e.g., that of Sun Microsystems) that allow final parameters to be omitted, and this also supplies an extra zero-valued parameter.

**PROTO=2** Prototypes are used if provided, but the compiler doesn't insist on having them.

**PROTO=3** All function declarations, but not definitions, are required to have prototypes.

**PROTO=4** All function declarations and definitions are required to have prototypes.

Pragma scope: takes effect immediately.

`RCALL(`*fname*`,`*inregs*`,`*outreg*`)`

This option is used only to provide an interface to certain MTS functions.

This option specifies that the function *fname* is to use R-call linkage. The *inregs* parameter specifies which registers are to be loaded with parameters, and the *outreg* parameter specifies which register, if any, contains the result.

*inregs* is specified as a sequence of hexadecimal digits, one for each of the general registers to be loaded with a parameter. Each of the registers specified in *inregs* is restricted to the range 0-7.

Similarly, *outreg* specifies either general register `R0` or `R1`.

For example,

```
#pragma rcall(xyz,043,1)
```

would declare the function `xyz` to use R-call linkage, loading the first parameter into `R0`, the second into `R4`, and the third into `R3`. The result will be returned in `R1`.

The register specifications are optional.

If *inregs* is omitted, parameters will be passed with an S-type linkage, but it is still possible to specify *outreg* as `1`.

If *outreg* is omitted, the value is returned in `R0`.

For example,

```
#pragma rcall(xxx,043) /* result in R0  */
#pragma rcall(yyy,,1)  /* no param regs */
```

Pragma scope: The same scope as *fname*.

`RENT` (default `NORENT`)

The option `RENT` causes all `static` and `extern` variables, as well as string constants, to have the `const` attribute. This forces ∗C89 to check for possible re-entrancy violations. If you don't know what re-entrancy is, you don't need to use this option.

Pragma scope: Takes effect immediately.

`RMODE` (default `RMODE=ANY`)

This indicates to the compiler where the code for the csect can be loaded at time execution. This can be set to `24` or `ANY`.

Pragma scope: The value at the end of a function determines how the code is treated by the loader.

`STANDARD`, `STANDARD+`, `UNIX`, and `UNIX+` (default `STANDARD`)

These four options are especially useful in porting programs to or from UNIX systems. The two effects these options have are: (1) to define compile-time symbols that control which parts of the standard header files are processed, and (2) to set other pragma options.

**STANDARD** defines only those identifiers which are in the standard header files and defined in the ANSI C Standard. No additional UNIX symbols are defined.

This also sets `LANG=ANSI` and `WARN=3`.

`STD` and `ANSI` are allowed as equivalent abbreviations.

**STANDARD+** gives all of the standard identifiers from the standard header files and *additionally* defines all identifiers that would normally be defined in the UNIX version of the header file. In the few cases that the standard and UNIX definitions conflict, the standard definitions take precedence.

This also sets `LANG=ANSI` and `WARN=3`.

`STD+` and `ANSI+` are allowed as equivalent abbreviations.

**UNIX** gives only those identifiers in the standard header files that are defined in UNIX. No additional ANSI C Standard identifiers are defined.

This also sets `LANG=K&R` and `WARN=1`.

**UNIX+** defines those identifiers in the standard header files that are defined in UNIX and *additionally* defines all non-conflicting ANSI C Standard identifiers.

This also sets `LANG=K&R` and `WARN=1`.

Note that if UNIX or UNIX+ is used and you also want to have the compiler check for warnings, the WARN option must be used after the UNIX or UNIX+ option. For example,

```
PAR=UNIX+ WARN=3
```

Pragma scope: Takes effect immediately.

`SUMMARY` (default `SUMMARY`)

This option provides a summary of the object code produced during the compilation of each function. The summary includes the number of bytes required for the generated code, constants, and stack. It also gives information about the number of unused general and floating point registers.

Pragma scope: The last value is used for the entire compilation unit.

`SYM` (default `SYM`)

If `SYM` is on, SDS `SYM` records are generated in the object module. See section 13, "Debugging With SDS."

Pragma scope: The last value is used for the entire compilation unit.

`TEST`

Synonym for `SYM`.

`UNDEF(x)`

This is processed as if a `#undef x` had been issued before the source program had started. `x` may be any legal identifier. As many `DEFINE` and `UNDEF`s can be issued as desired. If more than one is given, they are processed in order. `x` is *not* translated to upper case before use.

This option cannot be negated.

Not allowed as pragma; use `#undef`.

`UNIX`

See `STANDARD`

`UNIX4.3`

This is short-hand for `UNIX+`, `WARN=1` and `LANG=K&R`. Note that if it is desired to have the compiler check for warnings, the WARN option must be used after the UNIX4.3 option. For example,

```
PAR=UNIX4.3 WARN=3
```

`WARN=n` (default `WARN=3`)

The `WARN` option controls which warning messages are printed. The possible values range from `0` (no warnings) to `4` (all possible warnings are printed).

**0** No warnings. `NOWARN` is a synonym for `WARN=0`.

**1** Serious warnings. E.g., a loop doesn't appear to terminate.

**2** Possible confusion warnings. E.g., the assignment operator (`=`) occurs where an equality operator (`==`) would usually occur. (Also includes `WARN=1`).

**3** Default type warnings. E.g., `int` was omitted. (Also includes `WARN=1` and `WARN=2`).

**4** Stylistic warnings. E.g., a variable in an inner block hides another by the same name in an outer block. (Also includes `WARN=1`, `WARN=2` and `WARN=3`).

Pragma scope: Takes effect immediately.

`ZEROARG` (default `NOZEROARG`)

If `ZEROARG` is specified, an extra `int` zero argument is appended to every parameter list and `PROTO=1` is set. This is provided to mimic the behavior of the Sun Microsystems C compiler.

Pragma scope: Takes effect immediately.

## 5 Implementation-Defined Behavior

The ANSI C Standard specifies that every C compiler and library should describe its implementation of the following standard features.

### 5.1 Diagnostics

- All diagnostics produced by ∗C89 have the form:

```
<file-name>, line <line-number>: <message>
```

If <message> begins with the “warning:”, the error will not suppress object code generation. If it does not begin with “warning:”, it will suppress object code generation.

## 5.2 Environment

- The semantics of the arguments passed from the MTS command environment to `main` are described in section 9, "Execution Environment."
- The nature of interactive devices is described in *MTS Volume 4: Terminals and Networks in MTS*, Reference R1004.

## 5.3 Identifiers

- There may be up to 256 significant characters in identifier names. The ANSI C Standard requires only 31 significant characters.
- By default, up to 128 characters are used in external names. The ANSI C Standard requires only 6 significant characters, though most implementations support many more. If the option `PAR=NOLONG` is specified, only 8 characters will be used in external names.
- By default, case is insignificant in external names. If the option `PAR=MC` is specified, case is significant.

## 5.4 Characters

- The EBCDIC character set is used for both compilation and execution. Further information on exactly which characters are supported for the various devices can be found in *MTS Volume 1: The Michigan Terminal System*, Reference R1001.
- No alternate shift states are provided for multibyte characters.
- There are 8 bits in a character in the execution character set.
- Normally, no mapping is performed between string constants in the source file and the execution environment. If the option `ASCII` is supplied, the characters in string constants are mapped from ASCII to EBCDIC.
- All characters that might be present in a string constant can be represented in the execution environment.

- A character constant that is longer than one character is packed into an `int` right-justified and zero-filled. No more than four characters can be placed into an `int`. (This is also true for multibyte characters and wide-character constants.)
- Multibyte characters will be converted to wide-character constants with the `mbtowc` routine under all legal locales.
- A "plain" `char` is unsigned.

## 5.5 Integers

- All integers are represented in two's-complement notation and have sizes and byte (8-bit) alignments as follows. The `signed` and `unsigned` attributes do not affect the size or alignment.

  `char` is 1 byte, aligned on a byte boundary.
  `short` is 2 bytes, aligned on a two-byte boundary.
  `int` is 4 bytes, aligned on a four-byte boundary.
  `long int` is also 4 bytes, aligned on a four-byte boundary.

- Converting an integer to a signed integer of shorter length will result in a value with the same high order bit (or sign bit) and other bits removed from the left until the value fits. Converting an unsigned integer to a signed integer of the same length will result in the same bit pattern but now interpreted as a signed value. If this results in truncation, the new value will be negative.
- Bit-wise operations on signed integers simply take place on their normal two's-complement representation.
- The sign of the remainder in an integer division is the same as the sign of the dividend.
- The result of a right shift of a signed negative integer fills bit positions on the left with the sign bit.

## 5.6 Floating Point

- The representation of floating-point numbers is described in detail in the *IBM System/370 Principles of Operation.* The sizes and alignments are as follows:

  `float` is 4 bytes, aligned to a 4-byte boundary.

  `double` is 8 bytes, aligned to an 8-byte boundary except when passed as a parameter, when it is aligned to a *4-byte* boundary.

  `long double` is 16 bytes, aligned to an 8-byte boundary except when passed as a parameter, when it is aligned to a *4-byte* boundary. Currently, only the first 8 bytes are used in computations.

- If an integer is too large to be converted to a `float`, it will be *rounded* to a `float`.
- A floating-point number that is converted to a narrower floating-point value is *rounded*.

## 5.7 Arrays and Pointers

- The maximum size of an array can be held in an `unsigned int`.
- Casting may take place between pointers and integers without altering the bit pattern, but this may generate a warning.
- The difference between two pointers may be held in an `int`.

## 5.8 Registers

- Register objects are placed into registers only if `PAR=OPT=2` is specified, the type of the object is integral or pointer, and there are enough free registers ("integral" means `char`, `short`, `int`, `long` or `enum`). `GR2` through `GR7` are used for expression evaluation. Any of the registers `GR2` through `GR7` not needed for expression evaluation can be used to hold variables. For example, if `GR2` is needed to evaluate an expression and all other registers are free, then five registers are available for register variables. The number of available registers can vary anywhere from zero to six, depending on the complexity of the function.

## 5.9 Structures, Unions, Enumerations, and Bit-fields

- Members of a union accessed by a member of a different type will access the original bit pattern.

- The first byte of a `struct` or `union` is aligned to the largest alignment required by any of the members of that struct or union, except when passed as a parameter, when it is aligned to a four-byte boundary. For members of a `union` or `struct`, the alignment is as follows:

  `char` is aligned on a byte boundary.
  `short` is aligned on a two-byte boundary.
  `int` is aligned on a four-byte boundary.
  `long int` is aligned on a four-byte boundary.
  `enum` is aligned on a four-byte boundary.
  `float` is aligned to a four-byte boundary.
  `double` is aligned to an eight-byte boundary.
  `long double` is aligned to an eight-byte boundary.
  **pointers** are aligned to a four-byte boundary.
  **arrays** are aligned the same as the members of that array.
  `struct` is aligned the same as the largest alignment needed by any of the members.
  `union` is aligned the same as the largest alignment needed by any of the members.
  **bit fields** are never aligned except when the field would otherwise straddle four consecutive byte boundaries. In such cases the bit field is aligned to the next byte boundary to prevent this.

  Alignment may require the insertion of padding bytes.

- “Plain” bit fields are treated as `signed`.

- Bit fields are allocated from high-order to low-order bits within an `int`.
- A bit field may straddle a byte boundary provided that this doesn't cause the field to straddle four consecutive byte boundaries, in which case it will be aligned to the next byte boundary.
- `enum` types are implemented as `int`. Note that even though enums types are implemented as `int`, a warning message will result when enums are mixed with `int`s.

## 5.10 Qualifiers

- Currently, the `volatile` qualifier does not affect code generation. This may be changed in future versions of the compiler.

## 5.11 Declarators

- There may be no more than 13 declarators modifying a type.

## 5.12 Statements

- There is effectively no limit on the number of `case` values allowed in a `switch` statement.

## 5.13 Preprocessing Directives

- The EBCDIC character set is used in the preprocessor as well as during execution. Character constants consisting of a single character are never negative.
- The method of locating includable source is described in section 14.2, "Header Files."
- The processing of quoted `#include` files is described in section 14.2, "Header Files."

- The recognized `#pragma`s are described in section 4, "Pragmas and Par Field Options."
- If the time and date are not known (which should never happen) `__TIME__` and `__DATE__` will have the values "18:59:59" and "Dec 31 1969" respectively.

### 5.14 Library Functions

- The null pointer (`NULL`) is represented as 4 bytes of all zeros.

Other implementation-dependent aspects of the library are described in section 15, "The MTS *C89 Library, Headers, and Macros."

# 6 Local Extensions

*C89 has a number of extensions, which primarily help to provide an interface to existing MTS routines. These extensions should *not* be used if a portable program is desired.

## 6.1 Compiler Identification

Several macros are defined to make it possible for the code to identify the compiler, operating system, and hardware that are being used. These macros are:

```
_C89
_MTS
_IBM370
_SITExxx
```

(where "xxx" is a two- or three-letter site name. For example, at the University of Michigan this is "UM"; at Renssalaer Polytechnic Institute this is "RPI"; at the University of British Columbia this is "UBC.")

These macros can be used in a single source file that may be compiled either by *C89 or another, incompatible, compiler. For example:

```
#ifdef _C89
   *C89 stuff
#else
   other compiler stuff
#endif
```

It is also possible to compile a file differently depending on where it is being compiled. For example:

```
#ifdef _SITEUM
   UM stuff
#endif
#ifdef _SITERPI
   RPI stuff
#endif
```

## 6.2 FORTRAN Linkage

If a function is declared `__fortran`, linkage to it will be by means of a standard IBM 370 S-type calling sequence. The details of the calling sequence are described in *MTS Volume 3: System Subroutine Descriptions*, Reference R1003, and in section 10.1, "Calling FORTRAN Routines."

## 6.3 Saving the Return Code

It is possible to get the return code from a call. See section 10.1.2, "Return Codes."

## 6.4 Access to the Save Area

Each function has a predefined array, called `__SAVEAREA`, in its stack frame. This is an array of 16 `int`s, which overlaps the coding conventions save area. Values may be fetched from this area, and at the programmer's risk, values may be modified. For example, issuing the statement

```
__SAVEAREA[15] = 8;
```

allows a C routine to set the return code that can be tested by the caller.

## 6.5 R-call Linkage

The ability to call R-type routines is supported by means of the `RCALL` pragma.

## 6.6 Pseudo-registers

The storage class `__pseudoregister` allows variables to be allocated in the pseudo-register vector. This is necessary only if re-entrant code is desired. Example:

```
__pseudoregister int globals[100];
```

Such a variable can then be accessed just like any other variable. ∗C89 initializes the pseudo-register vector to all zeros before passing control to the main program, provided the main program is written in C. This cannot be guaranteed if the main program is written in PLUS.

A pseudo-register must not have an initializer.

∗C89 doesn't produce re-entrant code by default. However, it is possible to modify programs to be re-entrant. See the `RENT` option in section 4, "Pragmas and Par Field Options."

### 6.7 Setting the Pseudo-register Base

It is possible to specify a new pseudo-register area base on a call, for example:

```
sub(a, b, c, __prvbase w);
```

`__prvbase` sets the value of GR11 to a desired value before making the call. The expression following `__prvbase` must be of a pointer type. On return from the call, GR11 is restored to its previous value. This feature is provided primarily for MTS system programmers.

## 7 Writing Portable Code

While it is quite possible to write C programs that can easily be moved (ported) to other machines, care must be taken to achieve portability. Some of the issues that must be considered are:

- **Use the PORT Option.**

  Specify the `PORT` option so that some detectable portability violations will be flagged by the compiler. Although this will not catch all problems, it will get some of them.

- **Do Not Use int.**

  The code must not depend on the size of data elements. A common offender is `int`, which is often either 16 or 32 bits. It is better to use either `#define` or `typedef` to define a new identifier of the appropriate size to use in declarations. To use `int` alone is to allow the compiler implementor to make the size decision for you.

  The following is recommended:

  ```
  #define int8  signed char
  #define int16 short
  #define int32 int
  ```

and then use only the identifiers `int32`, `int16` and `int8` in place of `int, short` or `char`. When moving to a system that has a different definition of `int`, it is only necessary to change these definitions.

- **Make No Character Code Assumptions.**

  Avoid using the actual encoding of characters. MTS (*C89) uses EBCDIC character codes while most systems use ASCII.

- **Use Variable Length Parameter Lists.**

  Variable length parameter lists should be handled using the type and macros in <`stdarg.h`> (`va_list`, `va_start`, `va_arg`, and `va_end`). It is advisable to use these macros because other C compilers may handle variable length parameter lists in a different way or have types of different sizes. However, the parameter passing method used by *C89 is compatible with that used by most UNIX C compilers.

- **Isolate System Dependencies.**

  Avoid accessing system-dependent data structures, e.g., the fields in a FILE structure, and try to isolate the system dependencies in a separate module.

  Remember that UNIX path names are, in general, quite different from MTS file/device names. Code that refers to files or devices by name is very likely to be system-dependent.

  Don't depend on *C89 extensions such as the `FILL` option.

# 8 Porting Code From Elsewhere to *C89

## 8.1 UNIX Code

Several options have been provided to make porting C programs written for UNIX systems somewhat easier. The `UNIX4.3` pragma (option) may provide sufficient portability. See section 4, "Pragmas and Par Field Options," for details. Some UNIX C compilers append an additional zero argument to parameter lists. If the code is coming from one of those compilers (e.g., that used by Sun Microsystems), the `ZEROARG` pragma would be useful.

There are a number of UNIX functions in the library. However, the facilities of MTS do not always provide the functionality necessary to implement some of the UNIX functions.

## 8.2 Identifier Case and Length

The ANSI C Standard allows for some variation in how *external* variable names are treated. It requires only a minimum of the first six characters to be significant, ignoring any case distinctions. By default, ∗C89 allows 128 significant characters in external names but does ignore case distinctions.

Note that this is not the case for *static* and *local* identifiers.

For example, a declaration at the outer level (external)

```
int abc, ABC; /* illegal conflict of EXTERNAL identifiers */
```

is legal in many systems, but it will not work correctly in ∗C89.

The compiler may be able to diagnose some external name collisions but not those occurring between separate compilation units.

If the option `PAR=MC` is specified, then the outer level declaration

```
int abc, ABC;
```

will be legal. (`PAR=MC` must not be used indiscriminately; see section 4, “Pragmas and Par Field Options,” for more information).

## 8.3 Initialization of Variables

At the beginning of execution, external and static variables are initialized to all zeros.

No code should depend on the implicit initial values of `auto` variables, but in fact, some imported code may inadvertently depend on the fact that stack storage starts off with a value of zero (unlike MTS). The `FILL` pragma may be used to set the entire stack frame to the fill value upon function entry. Note that because the storage allocated for local variables within inner blocks may overlap, the variables in inner blocks may *not* always be set to the fill value.

The `FILL` pragma should not be used blindly except where there is a proven problem, as it will slow execution.

Do not write programs that *depend* on the `FILL` feature, since the program will not be portable.

## 8.4 Diagnosing Portability Errors

When moving a C program from elsewhere to MTS, ∗C89 will often produce many warning or error messages. Most of the warning messages are a consequence of the diagnostics in ∗C89 and can be ignored (such warnings can be controlled with the `UNIX4.3` and `WARN` options).

Here are some suggestions for dealing with those errors that cannot be ignored.

### 8.4.1 Illegal Character: xx (hex)

This probably indicates a problem was introduced as the file was transferred to MTS. For example, it may not have been correctly translated from ASCII to EBCDIC.

### 8.4.2 Missing Include File "xxx.h"

If the missing include file is a private include file, it should simply be transferred to MTS from the original system.

If the include file is a system include file from another system, the same features may be available in ∗C89INCLUDE in a different include file. If the equivalent features are not available, there may be serious difficulties in getting the program to run on MTS.

### 8.4.3 External Name Conflicts

It is possible that the originating system's method of processing external names was different from that used by ∗C89 on MTS. The following error message may indicate this problem:

```
Error: the following external names
       do not resolve to unique loader names.
...
```

This will happen when the names differ only in upper and lower case letters in the name as in `XYZ` and `xyz`.

In some cases the compiler will not recognize that external names are not unique (for example, if the conflict is between names in separate compilation units). In this case the name conflict will be diagnosed by the MTS loader. Regardless of which program gives the warning, the solutions are the same.

In such cases one of the two names can be changed. This can be done using C's `#define` feature as follows:

```
       #define name1 name2
```

Make sure the names you pick are all unique.

The `LOADNAME` option can also be used to help with these problems.

## 8.5 File and Device Name Differences

Even when a program compiles successfully, there is no guarantee that it will run successfully. One problem is that programs on other systems (such as UNIX) may use routines that do not exist in ∗C89LIB or MTS. Another problem is that file naming schemes are quite different among different operating systems. For example, if a program brought from UNIX refers to a tape as `/dev/rts8`, that reference will have to be changed to something that is a tape on MTS.

## 8.6 Character Codes

Since MTS uses EBCDIC and many other systems use ASCII, there may be problems because the program assumes too much about the character set. Problems sometimes arise because A-Z does not form a contiguous range in EBCDIC. For example, the following is true in ASCII but not in EBCDIC:

```
        'i'+1=='j'
```

# 9 The Execution Environment

Execution of a ∗C89 program usually begins in a function in ∗C89LIB. This function performs various initializations such as allocating the run-time stack and initializing the memory management and I/O routines. A call is then made to the (user) function `main`.

Entering a C function from a program written in some other language, as well as calling functions written in other languages, is discussed below.

## 9.1 Stack Allocation

One of the duties of an initialization function is to allocate a stack in which local variables and housekeeping data are stored. The default stack is 160K, but a program with larger storage requirements will need a larger stack. If the external integer variable `_stack` is defined within any of the user's compilation units, it is taken to be the number of *pages* (4K each) of stack space needed.

Upon entry, each function requires a minimum of 64 bytes of stack space. To this one must add the space required by all local variables and temporaries. The `SUMMARY` pragma will print out the required stack space for each function.

Thus, a program with substantial stack requirements might contain the following line in it:

```
   int _stack = 32;
```

which requests 32 pages (131,072 bytes) of stack space.

∗C89 protects the first page after the end of the allocated stack so that stores into this region will result in a protection exception. The storage protection exception will often occur at the first instruction of a function as it tries to save the registers. If a stack addressing exception occurs, typically either the stack may be too small or there is unlimited recursion.

The stack size is used and the final page is store-protected only when execution is started in a ∗C89 main program.

## 9.2 Passing Parameters to the Main Program

Parameters given on the `PAR` field on the $RUN command for a C main program may be accessed in the standard C fashion. Specifically, a C program that needs to examine the `PAR` parameters should be declared as

```
main(argc,argv)
int argc;
char *argv[];
```

The PAR-field parameters are placed in strings pointed to by the elements of `argv` array, starting at `argv[1]`. `argc` is one greater than the number of parameters given. The contents of `argv[0]` are undefined, which is not compatible with UNIX.

Some systems pass a third argument, `arge`, which is a pointer to an environment list. While this third parameter may be declared, *C89 always passes a `NULL` (0).

The parameters are separated by blanks (not commas as one would normally expect in an MTS PAR field). The following characters have special meaning.

**Blanks** (one or more) separate the parameters.

**Quotes** (single and double) are used to enclose characters so that the special characters (such as blank) are inactivated. The result has the quotes removed. If the quotes are unbalanced, a parameter is made of all characters from the last unbalanced quote up to the end.

**Backslash** causes the next character to be taken literally.

For example:

```
main(argc,argv)
int argc;
char *argv[];

{int i;
 for (i=1; i<argc; i++)
    printf("Argument %d is '%s'\n",i,argv[i]);
}
```

If we run this program with the MTS command:

```
        $RUN prog.o+*c89lib par=Why "doesn't" it snow?
```

the output would be

```
        Argument 1 is 'Why'
        Argument 2 is 'doesn't'
        Argument 3 is 'it'
        Argument 4 is 'snow?'
```

## 9.3 Details of Parameter Passing

When interfacing programs written in different languages, it is important to know how each language passes parameters.

∗C89 places the *value* of each argument into the parameter block (the value of an array or function is its address). This differs from many other languages on MTS that place the *address* of the argument in the parameter block.

All integer types are cast to 32-bit integers before being passed. Unsigned integer types are padded with binary zeroes on the left, if necessary. Signed integer types are sign-extended, if necessary. Note: `char`s are unsigned in ∗C89.

`float` and `double` require 64 bits in the parameter list (`float`s are cast to `double`s). `double`s are aligned to a full word boundary.

`long double` elements require 128 bits (16 bytes).

Arrays, pointers, and character-string constants cause the address of the argument to be passed. These pointers require 32 bits in the parameter list.

The values of `struct`s and `union`s are copied into the parameter block.

All parameters are aligned to 32-bit (`int`) boundaries.

The address of the parameter block is passed to the called routine in general register 1 (GR1).

# 10 Calling Non-C Routines

## 10.1 Calling FORTRAN Routines

The normal C calling conventions are not entirely compatible with the “OS S-type” calling conventions used by FORTRAN and many assembly language routines. However, a mechanism is provided so that C programs can invoke routines written in assembly language or FORTRAN, provided that they follow standard OS S-type calling conventions. This allows the use of subroutine libraries such as ∗PLOTSYS, ∗IG, and IMSL, as well as the use of most system subroutines.

To advise the compiler that an external function follows the S-type calling conventions, the function must be declared with storage class `__fortran`. For example, to declare that the subroutine `FOO` is written in FORTRAN, use the declaration

```
    __fortran void foo();
```

To declare an `INTEGER`, `INTEGER*2`, `INTEGER*4` or `LOGICAL` FORTRAN function `BAR`, write

```
    __fortran int bar();
```

To declare a `REAL` or `REAL*4` FORTRAN function `BAR2`, write

```
    __fortran float bar2();
```

To declare a `REAL*8` or `DOUBLE PRECISION` FORTRAN function `BAR3`, write

```
    __fortran double bar3();
```

(Note: `CHARACTER` functions cannot be directly called by ∗C89. `COMPLEX` functions can be called as if they were `REAL` functions, the value returned by the routine will be the real part of the return value. The imaginary part is lost. `LOGICAL` functions can be called in the same manner as `INTEGER` functions. The return value is 1 for `.TRUE.` and 0 for `.FALSE.`, which is compatible with C conventions.)

Any subsequent calls to `foo`, `bar`, `bar2` or `bar3` will use an S-type calling sequence rather than the C calling sequence.

S-type linkage differs from the C linkage in the passing of parameters and return values. S-type linkage requires that the parameters be addresses and that the high-order bit of the last parameter be set to one. The FORTRAN function may pass back a *return code* to the caller.

### 10.1.1 Type Equivalencies for FORTRAN

Because FORTRAN parameters are passed by address, it is illegal to pass a parameter to a routine declared `__fortran` unless it is a pointer to something. Some parameters, such as arrays, are naturally addresses in C. But for others, it may be useful to use the `&` operator to compute the address of a parameter. Here are FORTRAN parameter types and their C equivalencies:

| *FORTRAN parameter* | *C parameter* |
|---|---|
| `INTEGER` | `int *` |
| `INTEGER*4` | `int *` |
| `INTEGER*2` | `short *` |
| `REAL` | `float *` |
| `REAL*4` | `float *` |
| `REAL*8` | `double *` |
| `DOUBLE PRECISION` | `double *` |
| `LOGICAL` | `int *` (see below) |
| `LOGICAL*4` | `int *` (see below) |
| `LOGICAL*1` | `char *` (see below) |
| `COMPLEX` | (see below) |
| `COMPLEX*8` | (see below) |
| `COMPLEX*16` | (see below) |

| | |
|---|---|
| `array of x` | pointer to or array of appropriate type |

For `COMPLEX` and `COMPLEX*8` parameters, one can use a pointer to two `float` variables, an array of two `float` variables or a pointer to a structure containing two `float` variables. For `COMPLEX*16` parameters, one can use a pointer to two `double` variables, an array of two `double` variables or a pointer to a structure containing two `double` variables. In both cases, the real part is the first variable or element 0 of the array, and the imaginary part is the second variable or element 1 of the array.

For `LOGICAL` parameters, `.TRUE.` values should be passed into the routine as 1, and `.FALSE.` values should be passed in as 0.

`CHARACTER` parameters cannot be directly passed to a ∗C89 program.

### 10.1.2 Return Codes

Many FORTRAN routines produce a return code. In routines written in assembly language, this is a value (conventionally a multiple of 4) that is left in general register 15. In FORTRAN, return code $i$ is set using a `RETURN` $i$ statement.

The return code can be saved in a C program by writing the `__retcode` option at the end of the parameter list in a call. Example:

```
sub(a, b, c, __retcode x);
```

`__retcode` stores the return code (the value in GR15 upon return from the call) in the integer variable `x`.

#### 10.1.3 Mixing FORTRAN and RCALL

It's possible to declare an external function as both `__fortran` and having the `RCALL` (see section 4, "Pragmas and Par Field Options") linkage option. Of course, no real FORTRAN function can use register parameters, but there are some assembly language routines that can make use of this kind of linkage. When both `__fortran` and `RCALL` are declared for a function, the compiler doesn't require the parameters to be addresses.

### 10.2 Conflicts between ∗C89LIB and MTS Library Routines

Routines by the same name may occur in several libraries. When ∗C89LIB is used, routines in it may hide routines by the same name in the MTS libraries. Whether or not a routine has been declared to be a `__fortran` routine doesn't affect the loader's library search order. So something more must be done to call MTS routines that have a similarly named routine in ∗C89LIB.

### 10.3 Getting C rather than MTS Functions

The one certain way to insure that the C library routines are called instead of MTS routines by the same name is to *always include the header files* as these will insure that the proper routine is called. Omission of the header file can result in obscure errors.

The routines in <`math.h`> (`acos`, `asin`, `atan`, `atan2`, `cos`, `cosh`, `exp`, `log`, `log10`, `sin`, `sinh`, `sqrt`, `tan`, `tanh`) are remapped with the `LOADNAME` pragma to names in ∗C89LIB that begin with an underscore.

### 10.4 Getting MTS rather than C Functions

Few MTS system routines have the same names as C library functions, but when this happens and the MTS function must be called, it is best to call the MTS routine by an *alternate name*. This is true for the MTS routines `FREAD`, `FWRITE`, `MOUNT`, `READ`, `RENAME`, `REWIND`, `SYSTEM`, `TIME`, `WRITE`, and many of the mathematics routines such as `ACOS` or `SQRT`. Some MTS routines have alternate names already defined in the MTS system (e.g., `READ`, `TIME`, and `WRITE` have alternate entry points `MTSREAD`, `MTSTIME`, and `MTSWRITE`). The header file <`mts.h`> contains prototype definitions for many MTS system functions and defines the alternate entry point names `MTSMOUNT`, `MTSRENAME`, `MTSREWIND`, and `MTSSYSTEM`.

For example, you may call the MTS system subroutine `READ` (at its alternate entry point `mtsread`) by using the following code:

```
#include <mts.h>
char *buffer;
short int len;
int mods, lnum, fdub, rc;
...
mtsread(buffer, &len, &mods, &lnum, &fdub,  __retcode rc);
```

In this example, `MTSREAD` expects addresses of all its parameters. `buffer` is already an address, so `&` is not used with it.

## 10.5 Calling PLUS Routines

It is easy to call PLUS from ∗C89 provided that the data types of the PLUS arguments match the data types of the C parameters. Remember that PLUS may pack objects in a parameter list differently than C does. This can affect both the arguments to the routine and the return value. Also remember that the definition of a PLUS string and a C string are different.

## 10.6 Calling Assembly Language Routines

If an assembly language routine is to be called from C, it must first save all registers in the region pointed to by GR13. On entry, GR1 will point to a parameter list (as explained in section 9.3, "Details of Parameter Passing"). GR10 must be used as the code base register. After entry, the routine must set GR12 to the value of GR13 and then increase GR13 by the size of the routines stack frame plus 64 bytes for the save area (i.e., if the routine needs 64 bytes of stack, you must increase the value of GR13 by 128). The routine is then free to use this area for temporary storage. (If the routine doesn't need any temporary storage, it should increase GR13 by 64). After entry, the routine must not disturb the contents of GR11.

On exit the routine must restore all general registers except GR0 from the area pointed to by GR12. Floating point registers need not be restored. If the routine returns an integral type or a pointer, it should return the value in GR0. If it returns a `float` it should return the value in the high portion of FR0, the low portion will be ignored. If it returns a `double` or `long double`, it should return the value in FR0. If the routine returns a struct or union, it must move the struct or union into memory supplied by the caller. The address of this memory is the first parameter passed to the routine.

A typical routine to be called from C will look like this:

```
ROUTINE  STM 0,15,0(13)
         LR  12,13
         LR  10,15
         USING ROUTINE,10
         LA  13,stack_frame_size(13)
         ....
         L  0,return_value
         LM 1,15,4(12)
         BR 14
```

The sequence above does not do stack checking. If stack checking is desired, the following sequence can be used instead:

```
ROUTINE  STM 0,15,0(13)
         LR  12,13
         LR  10,15
         USING ROUTINE,10
         LA  13,stack_frame_size(13)
         L   15,4(11)
         C   13,0(15)
         BNL error_routine
         ....
         L  0,return_value
         LM 1,15,4(12)
         BR 14
```

This code will insure that there is enough stack for 72 bytes beyond the current stack frame (so that subroutines can be safely called). Stack checking is only necessary if more than 72 bytes of stack are used by the procedure or if the procedure contains one or more procedure calls.

## 11 Calling C From Foreign Environments

There are two ways to define a C function so that it may be called from a non-C program.

A normal C function uses the MTS coding convention linkages internally, so any calling program that uses the same linkage may call a C program subject to C environment initialization requirements given below.

Alternatively, a C function may be defined to be entered with a FORTRAN (OS S-type) linkage.

Also see section 9, “Execution Environment,” for details of parameter passing.

## 11.1 Using FORTRAN Linkage

By using the `__fortran` keyword on a function *definition*, it is possible to define a C function that can be called directly from any language that uses FORTRAN (OS S-type) linkage.

A C function that is defined this way will call an interface routine at the time of entry to initialize the C environment, if it has not already been initialized. The programmer doesn't have to be concerned with explicit initialization of the C environment (with `C89INIT`). The `_stack` and `_TRUSTME` parameters are used in the initialization process exactly like C main programs.

For example, a C function that can be called from FORTRAN and will return the sum of its arguments could be written as follows:

```
__fortran int sum(int *a, int *b)
{
   return a+b;
}
```

Note that the types of parameters and return values needed by the C program are as described in the previous section "Type Equivalences for FORTRAN." The programmer must be aware of the consequences of such a function definition.

- There is additional overhead upon entry, because of the need to establish a C environment.
- All calls to such a function must use OS S-type (FORTRAN) linkage, including calls from other C functions.

No such routine can be called asynchronously, for example calling a C function with a FORTRAN linkage from either `PGNTTRP` or `ATTNTRP` will not work correctly.

The return code for such a routine can be set by adding the following statement anywhere in the routine. If the statement is placed in a routine called by the `__fortran` routine, the return code is not preserved.

```
    __SAVEAREA[15] = e;
```

where “e” is any integer expression.

If I/O was performed while in the C environment, it may be necessary to insure that all streams are closed so that the final buffer contents are properly written. This can be done by calling the `_cleanup` function before the final C function returns.

## 11.2 Using MTS Coding Convention Linkage

This section describes how one may call a C function with normal coding conventions linkage. This type of call is generally available only from Plus or assembly language.

### 11.2.1 Initialization of the C Environment

If the main program is not written in C, the initialization routine `C89INIT` must be called at some point before entering the C-environment. Failure to do so will result in many C routines not working, in particular the I/O routines, storage routines, and the `exit` routine. `C89INIT` must be called only once and must not be called if there was a C main program.

`C89INIT` must be called using the MTS coding convention linkage. Plus uses this linkage convention, but for other languages, see *The MTS Coding Conventions* by Steve Burling.

`C89INIT` takes a single `int` parameter with the value in the range `0-7`, the sum of three bit flags.

Bit 31 (value 1) affects how `exit`s within the C code are interpreted. If `0` is given for this bit, then `exit(0)` causes the MTS routine `SYSTEM#` to be called. A call to `exit` with a non-zero value results in the MTS routine `ERROR#` being called. When `1` is given for this bit, all calls to `exit` will return to `C89INIT` which, in turn, returns to whomever called `C89INIT`.

When this is done, care must be taken. The return value from C89INIT must be checked, and the program must respond correctly. Otherwise an infinite loop will result.

If bit 30 (value 2) is on, the standard units `stdin`, `stdout`, and `stderr` are not opened within C89INIT. Suppressing this opening may be useful if the standard opening would cause problems. Even if this bit is set on, it is still possible to explicitly open these units with calls to `freopen` (See section 15.5.8, "Initial Stream Assignments.") If this bit is off, the units are opened as expected within C89INIT.

If bit 29 (value 4) is on, this will allow explicit and implicit concatenation, line number ranges and I/O modifiers to be used for files/devices opened for input. This is analagous to the _TRUSTME flag.

"Normal" returns from C89INIT give a return value of 0, returns caused by a call to exit return 1. In the latter case the return code from C89INIT will contain the value given to `exit`.

### 11.2.2 Closing the C Environment

If I/O was performed while in the C environment, it may be necessary to insure that all streams are closed so that the final buffer contents are properly written. This can be done by calling the `_cleanup` function before the final C function returns.

### 11.2.3 Calling *C89 from Plus

As an example of how C89INIT may be used, the following is a possible PLUS program which calls C89INIT:

```
  %Include(Main,Message,Message_Initialize,Integer,Boolean);
  %Punch(" ENT  MAIN");

  Procedure C89INIT is
    Procedure
```

```
      Parameter Flag is Integer,
      Result From_Where is Boolean in Register 0
   End;

   Procedure C_Stuff is
      Procedure
      End;

   Definition Main;
      Variable Mcb is Pointer to Unknown;
      Mcb := Message_Initialize();

      /* Init C environment before calling C routines */

      If C89INIT(1, return code Rc) then
        Message(Mcb, " The return code was <i></>", Rc);
        Return;
      End if;

      /* C_Stuff is a routine written in C. */
      C_Stuff();

   End;
```

### 11.2.4 Calling C from Assembly Language

Calls from assembly language to C are straightforward, provided the correct registers are set up beforehand.

GR1 must point to a parameter block. (If the called routine doesn't expect any parameters, a parameter block need not be supplied). The format of the parameter block was explained in section 9, "Execution Environment."

GR13 must point to a stack. This region must be large enough for all the routines that are called as a result of calling the particular C routine. C main programs typically allocate 160,000 bytes for this purpose.

GR11 must point to the global area. This contains all the pseudo-registers. The CXD instruction should be used to determine its size. The first two words of the region have special significance to coding convention routines. See *The MTS Coding Conventions* by Steve Burling. The global storage of C currently does not use this region, but the C run-time system does.

GR14 and GR15 must be the return address and the entry point address, respectively.

On return, the C program will restore all general registers except GR0. Floating point registers are *not* restored by the called program. If the routine returns a floating point value, it will be in FR0; if the routine returns any other type of value (except a struct), it will be in GR0.

If the C function returns a struct, there must be another parameter passed at the beginning of the parameter list that is the address of where the struct result should be stored (`GR0` will not be set in this case).

A typical call to a C routine might look like this:

```
....
L    13,A(stack_space)
L    11,A(pseudo_register_space)
....
LA   1,parameter_list
L    15,=V(routine)
BALR 14,15
....
```

## 12 Putting Debugging Tools Into the Source

The most common way to debug C programs is to insert `printf` calls at critical points in the program. It's often useful to `#define` a flag that controls whether debugging source is generated by the preprocessor. This allows the debugging code to be completely enabled or disabled with only a recompilation.

It's also easy to use the facilities of <`assert.h`> to insert optional debugging code permanently in the source program.

To determine if failure to initialize a local variable could be the source of a bug, the program can be recompiled with the `FILL` option, which will then initialize all local variables to some desired value.

## 13 Debugging With SDS

∗C89 does not come with a separate debugging program or package. However, ∗C89 programs can be monitored with the MTS symbolic debugger (`SDS`).

The following sections are not a tutorial in SDS. If you haven't used SDS, see *MTS Volume 13: The Symbolic Debugging System*, Reference R1013. The intent of this section is to point the way toward using the most useful SDS commands.

### 13.1 Invoking SDS

SDS can be invoked in two fashions. In the first, simply replace the `$RUN` command with the `$DEBUG` command, and the debugger will take over. In the second, invoke the debugger using the `$SDS` command once a program has been loaded. You may find it convenient to use the second method when a bug unexpectedly appears and the program stops with a *protection exception* or something of that sort. Issue the `$SDS` command followed by an `include` of the object to be debugged. The `include` enables `SDS` to obtain the symbol table information.

SDS obtains information about symbolic names from two sources. First, the names that the MTS loader uses are available to it. Second, there are names that the compiler told SDS via "SYM" records. These informational records are produced only when the `SYM` pragma or compile option has been given.

Note: SDS maps all names to upper case internally.

Normally, the following commands should be issued at the beginning of a debugging session to prevent the signal routines from trapping attention and program interrupts.

```
set attn=off pgnt=off
```

## 13.2 Storage Layout

The following C constructions in a compilation unit are mapped into object module entities as follows. Each object module contains one or more *control sections* (csects).

**External functions** are each assigned a separate csect. This csect contains both the code and the constants from a function. The main program is no different from any other function – it is just an external function that is called by the library when the program is executed.

**Static functions** are appended onto the control section of the first external function. There must be at least one external function in every compilation unit that has static functions.

**External variables** are placed in their own separate csects.

**Static variables** are collected into a single blank csect.

**Local variables** (`auto`) are allocated on the run-time stack.

## 13.3 Setting Breakpoints

It's possible to indicate points in the program where execution should stop. These are conventionally called *breakpoints*. A breakpoint may be set at the beginning of the function `f` by issuing

```
break f
```

where `f` is the name of the function. Note that the `LOADNAME` of the function is used here. For many C library routines this is different than the real name. The `LOADNAME` is frequently prefixed with an underscore (_). For example, `read` becomes `_read`. It's also possible to set a breakpoint at any statement. To do so both a line number and a CSECT must be specified (See section 13.2, "Storage Layout.") This can be done either by specifying the CSECT within a `CSECT` command, which may be followed by one or more `BREAK` commands, or by specifying the CSECT within the `BREAK` command. For example both of the following two sets of commands do the same thing:

```
csect f
break #13
break #13_7
```

or

```
break #13@cs=f
break #13_7@cs=f
```

The first of these `break` statements will put a breakpoint at the code for the statement that has source line number 13.000. The second will put a breakpoint at the code for source line number 13.700.

## 13.4 Continuing Execution after a Break

When execution stops at a breakpoint and variables have been examined, and it is desired to continue execution, issue the `continue` command. The breakpoint from which execution is proceeding remains in effect.

## 13.5 Removing Breakpoints

Breakpoints may be removed by means of the `restore` command. For example, to undo the effect of the previous `break` commands, issue one of the following two sets of commands:

```
restore f
restore #13@cs=f
restore #13_7@cs=f
```

or

```
csect f
restore f
restore #13
restore #13_7
```

The `clean` command can be used to remove *all* breakpoints.

## 13.6 Displaying Variables

The ease with which SDS can be used to display variables depends upon the storage class of the variable. Each kind is discussed separately. It is often useful to use the SDS `@t=` modifier to select the appropriate type. (A complete list of types that can be displayed is given in section 13.8, "Correspondence between *C89 and SDS Types.")

### 13.6.1 Displaying External Variables

To display external variables, issue the SDS `display` command with the name of the external variable. For example, to display an `int` external variable called count, use the command:

```
display count@t=f
```

### 13.6.2 Displaying Static Variables

Static variables *can not* currently be displayed from SDS.

### 13.6.3 Displaying Local (auto) Variables

When a ∗C89 function is executing, GR12 points to the current stack frame. The current stack frame holds the save area for saving the registers on entry as well as all of the function's local (`auto`) variables and temporaries. The save area occupies the first 64 bytes of the stack frame.

To display an `auto` variable, it's necessary to know its offset in the current stack frame. All `auto` variables are allocated on the execution stack (even variables that have been assigned to a register.) The first variable will have offset 64 and subsequent variables are assigned offsets with higher offsets aligned as appropriate. For more information on how the compiler calculates the offsets, see the sections on "Integers", "Floating Points", etc..

Once the offset (in hexadecimal) is known, the `display` command can be used to display `auto` variables in the *current* function as follows:

```
display $gr12+offset
```

By default SDS will display values in hexadecimal. If some other type is desired, it is necessary to append the `@t=` modifier. For example, to display 10 characters starting at offset 1C,

```
display $gr12+1C@t=cl10
```

For variables that have been assigned to registers, you must determine to which register the variable has been assigned. (This can be done by looking at the `OBJLIST` output from the compiler.) Then simply issue a command such as

```
display gr5@t=f
```

where `GR5` is replaced with the correct register number and `t=f` can be replaced with a different expression, if the value is not an integer.

### 13.6.4 Displaying Parameters

Parameters are normally allocated on the execution stack. If a parameter has been assigned to a register, its value is copied into that register soon after the beginning of the function.

When a ∗C89 function is called, `GR1` points to the parameter block. Soon after the beginning of execution, this value is copied to `GR8`, and `GR1` may be used for other purposes.

To display a parameter, it's necessary to know its offset in the parameter block, see section 9.3, "Details of Parameter Passing," to learn how to calculate such offsets.

Once the offset (in hexadecimal) is known, the `display` command can be used as follows:

```
display $gr1+offset

display $gr8+offset
```

The first command is used immediately after program entry, and the second command is used after `GR8` has been assigned a value.

As was the case for `auto` variables, by default SDS will display values in hexadecimal. This can be overridden in the same manner.

For parameters that have been assigned to registers, you must determine to which register the variable has been assigned. (This can be done by looking at the `OBJLIST` output from the compiler.) Then simply issue a command such as

```
display gr5@t=f
```

where `GR5` is replaced with the correct register number and `t=f` can be replaced with a different expression, if the value is not an integer. If this is not done, you will obtain the value the parameter had before the routine was called.

For parameters passed to an r-call routine, the parameter resides in the save-area of the routine in question. The offset is simply four times the register number. For example, the offset for `GR0` is 0, the offset for `GR1` is 4, the offset for `GR2` is 8, and so on. Such parameters can be displayed in the same way as an `auto` variable. See the previous section for more information.

### 13.6.5 Displaying Pseudo-registers

Objects which are normal external variables in other C systems have been implemented as pseudo-registers in ∗C89LIB in order that all simultaneous users may use the same copy of the library routines. In particular, `errno` has been implemented as pseudo-register. At some point before issuing a `display` command, SDS must be told which register contains the pseudo-register base. This needs to be done only once per SDS session.

```
using prarea $gr11
```

Once SDS knows where the pseudo-register base is, pseudo-registers may be displayed in the same way that any external variable is displayed. So, for example, one can issue the command

```
display errno
```

## 13.7 Other Useful SDS Commands

To print the call history after a breakpoint, one may use the following command:

```
trace stack
```

To determine the point at which execution has stopped after a breakpoint, one can issue the following command:

```
symbol $psw
```

Once SDS has stopped at the beginning of a function you can continue execution and stop when the function returns with the command

```
continue $gr14
```

you can determine the caller of that function by issuing the command

```
symbol $gr14
```

(but only at the beginning of the function).

## 13.8 Correspondence between *C89 and SDS Types

To display C variables in something approximating their C types, the following SDS types should be used.

`char @t=cl1` (see text).

`signed char @t=fl1`.

`short @t=fl2`.

`int @t=f`.

`float` `@t=e`.

`double` `@t=d`.

`long double` `@t=d` (only the first 8 bytes used).

Note that SDS does not have any support for `unsigned` types. Such data can be displayed with a signed type with the caveat that large numbers will be incorrectly displayed with negative values. They can also be displayed in hex with `@t=xl1`, `@t=xl2` or `@t=xl4` depending on the length of the data item.

Arrays of `char` can be displayed with `@t=cl`*n* with *n* replaced by the required length. For example, a string of length 10 would be displayed with `@t=cl10`. Single chars may often contain either small integers or character information, therefore `@t=cl1` or `@t=xl1` or `@t=fl1` may all be used, the choice made based on the exact situation. Note that SDS will not determine the length of a string automatically, nor does `@t=c` correctly display "non-printing characters." In order to see non-printing characters, a string or character must be displayed with `@t=xl`*n*.

For structs, arrays and unions, the object must be broken down into smaller pieces each of which can be displayed with the types indicated.

For other types such as bit fields and pointers, they can be displayed in hexadecimal with the type modifier `@t=xl`*n* (where *n* is replaced with the correct length. A bit field with length of 3 bytes would be displayed with `@t=xl3`, a pointer of length 4 would be displayed with `@t=xl4`). Bit fields that do not occupy an integral number of bytes can be display with `@t=xl`*n*, but it is up to the programmer to interpret the information.

## 14 The Library and Header Files

This section describes the library support for the run-time environment in MTS for C programs.

∗C89LIB is the run-time support library used by programs compiled with ∗C89. ∗C89INCLUDE is the header file include library with the standard headers that provide the definitions and declarations required by programs using ∗C89.

## 14.1 The ∗C89 Run-Time Library

The MTS ∗C89 run-time library (∗C89LIB) provides all the ANSI C Standard library functions. Additionally, it contains many UNIX facilities. Because MTS does not support some features available in UNIX, certain facilities are either omitted entirely or are present only in an abbreviated form.

∗C89LIB was designed to behave similarly to BSD4.3 UNIX. However, ∗C89LIB is not identical to BSD4.3, and it may be necessary to make some alterations to the source code of programs that run on UNIX to get them to run on MTS. In particular, one must recognize that there are two types of files (binary and text) in ∗C89LIB (BSD4.3 has only one type of file); the way MTS processes control characters is different from BSD4.3; and the details on how certain devices (such as printers) work is different. Also some facilities of BSD4.3 are not provided. Each of these differences may require changes to some programs. More details about these differences are described in the following sections.

It is also possible to call MTS routines directly. See sections 10.2 – 10.4.

## 14.2 Header Files

The MTS file ∗C89INCLUDE contains the standard headers (those enclosed in angle brackets).

∗C89INCLUDE contains, in the include library format, the various include files provided by BSD4.3 implementations of UNIX. In addition to this, it contains a complete set of ANSI C Standard header files and an include member called <`unix.h`>, which is read in by the compiler before reading the source file. (This provides certain translations that are necessary for the proper functioning of ∗C89LIB.) The members in this file are usually contained in separate files on UNIX systems.

By default, only the standard symbols are available from the standard header files. To access additional symbols, primarily those used in UNIX systems, the `STANDARD+`, `UNIX`, or `UNIX+` pragmas can be used. The effects of these pragmas on the *standard* header files are:

- `STANDARD`, the default, includes only the standard identifiers.
- `STANDARD+` includes all standard identifiers and all UNIX and MTS identifiers that don't conflict with the standard.
- `UNIX` includes only UNIX identifiers.
- `UNIX+` includes all UNIX identifiers and all standard and MTS identifiers that don't conflict with the UNIX definitions.

The following sections summarize the features available that are specific to *C89. A complete description of each header file can be found by examining *C89INCLUDE.

### 14.2.1 Standard Header Files

All ANSI C Standard header files are supported: <`assert.h`>, <`ctype.h`>, <`errno.h`>, <`float.h`>, <`limits.h`>, <`locale.h`>, <`math.h`>, <`setjmp.h`>, <`signal.h`>, <`stdarg.h`>, <`stddef.h`>, <`stdio.h`>, <`stdlib.h`>, <`string.h`>, <`time.h`>.

### 14.2.2 UNIX Header Files

The include files from the BSD4.3 UNIX system are also available in this library. A list of the available header files can be found by examining the beginning of *C89INCLUDE.

### 14.2.3 MTS Header File

Function prototypes, type definitions, and flag values for MTS routines may be found in <`mts.h`>.

### 14.2.4 Header File Search Order

The two kinds of header, or include, files (angle-bracket and quoted header files) are searched for in the following order. The search ends when a matching file name is found.

Angle-bracket header files are searched in the following order:

1. Search any library-organized file assigned to unit 2.
2. Search ∗C89INCLUDE (unless `NODEFINC` was specified).

Quoted header files are searched in the following order:

1. Search for the file on the MTS userIDs specified in the `I` option in the order the userIDs were specified.
2. Search the current userID.
3. Search any library-organized file assigned to unit 2.
4. Search ∗C89INCLUDE (unless `NODEFINC` was specified).

## 15 The MTS ∗C89 Library, Headers, and Macros

This section provides a summary of extensions or implementation-defined features of ∗C89LIB and ∗C89INCLUDE.

The following sections correspond to header file names and are presented in alphabetical order of those names.

Functions that have non-standard behavior and functions that are not required by the ANSI C Standard are noted as such.

## 15.1 assert.h - Debugging Aid

The header <`assert.h`> defines the `assert` macro. This is used for putting diagnostics into the program. When an assertion failure occurs, a message similar to the following is printed on `stderr`:

```
   Assertion failed.  Line 15.000 of file xyz.c.
```

After printing this message, the routine `abort` is called.

## 15.2 ctype.h - Character Handling

The header <`ctype.h`> declares several macros useful for testing and mapping characters. In all cases, the argument is an `int`, the value of which is representable as an `unsigned char` or is equal to the macro `EOF`. If the argument has any other value, the results will be unpredictable.

The term *printing character* refers to a member of an implementation-defined set of characters, each of which occupies one printing position on a display device. These include the space character, the alphabetic characters (a-z, A-Z), the decimal digits (0-9), and the following characters:

`.`, `<`, `(`, `+`, `|`, `&`, `!`, `$`, `*`, `)`, `;`, `-`, `/`, `,`, `%`, `_`, `>`, `?`, `‘`, `:`, `@`, `’`, `=`, `"`, `~`, `^`, `[`, `]`, `{`, `}`, `\`.

The term *control character* refers to one of an implementation-defined set of non-printing characters. In our implementation, the control characters are those whose values fall between the values 0X00 through 0X3F. The following control characters have special meanings:

- ‘`\n`’ (newline)
- ‘`\f`’ (form feed)
- ‘`\r`’ (carriage return)
- ‘`\b`’ (backspace)

- '`\t`' (horizontal tab)
- '`\v`' (vertical tab)

Some of these functions are implemented both as macros and functions, an `#include` of <`ctype.h`> is needed to make the macros available.

The character-testing functions described in this section return non-zero (true) if, and only if, the value of the argument `c` conforms to the description of the function.

The following standard functions are implemented:

`int isalnum(int c)` – true for upper-case, lower-case letters and digits.

`int isalpha(int c)` – true for upper-case and lower-case letters.

`int iscntrl(int c)` – true for control characters listed above.

`int isdigit(int c)` – true for decimal digits.

`int isgraph(int c)` – true for printing characters, excluding space.

`int islower(int c)` – true for lower-case letters (a-z).

`int isprint(int c)` – true for printing characters listed above.

`int ispunct(int c)` – true for printing characters, excluding space and alphanumeric characters.

`int isspace(int c)` – true for '`\f`', '`\r`', '`\n`', '`\r`', '`\t`', '`\v`', and space.

`int isupper(int c)` – true for upper-case letters (A-Z).

`int isxdigit(int c)` – true for hexadecimal digits (0-9, a-z, A-Z).

`int tolower(int c)` – if `isupper(c)` is true, returns the corresponding lower-case letter. Otherwise, the argument is returned unchanged.

`int toupper(int c)` – if `islower(c)` is true, returns the corresponding upper-case letter. Otherwise, the argument is returned unchanged.

The above macros assume an EBCDIC character set.

### 15.2.1 isascii

**Name: isascii**

**Purpose:** *Test for an ascii character*

**Include file:** <`ctype.h`>

**Prototype:** `int isascii(int c)`

**Description:** `isascii` returns true if the argument can be represented in 7 bits. If the decimal value of `c` falls in the range 0 to 127, it will return true.

**Note:** This function is not required by the ANSI C Standard. Programs using this function may not be portable. This function is included for compatibility with UNIX implementations.

### 15.2.2 toascii

**Name: toascii**

**Purpose:** *Character mapping to ASCII*

**Include file:** <`ctype.h`>

**Prototype:** `int toascii(int c)`

**Description:** The character codes used by the ∗C89 compiler are standard EBCDIC. This macro returns the value of a character's ASCII equivalent.

**Note:** This routine uses the same translate tables as MTS.

This function is not required by the ANSI C Standard. Programs using this function may not be portable.

This function is specific to the ∗C89 implementation on MTS.

## 15.3 math.h - Mathematics

The header <`math.h`> declares the floating point mathematical functions and defines three macros related to error processing.

The floating point functions described in this document take double-precision arguments and return double-precision values. The <`math.h`> header maps references of math functions (e.g., `sqrt`) into references to the actual supported names (e.g., `_sqrt`). In this way, conflicts between C function names and MTS subroutine or function names are avoided.

### 15.3.1 Treatment of Error Conditions

For all functions, a *domain error* occurs if an input argument is outside the domain over which the mathematical function is defined. When a domain error occurs, the integer variable `errno` acquires the value of the macro `EDOM`. With the exception of `log` and `log10`, all math functions return 0.0 on a domain error. Both `log` and `log10` set `errno` to `EDOM` if the argument is negative. The value returned by `log(-x)` is the same as `log(x)` for all x. The value returned by `log10(-x)` is the same as `log10(x)` for all x. `fmod` returns zero if its second argument is 0.0 and the value of `errno` is not changed.

Similarly, a *range error* occurs if the result of the function cannot be represented as a `double` value. If the result overflows (the magnitude of the result is so large that it cannot be represented in an object of the specified type), the function returns the value of the macro `HUGE_VAL`, with the same sign as the correct value of the function; the integer variable `errno` acquires the value of the macro `ERANGE`. If the result underflows (the magnitude of the result is so small that it cannot be represented in an object of the specified type), the function returns zero. This case is not considered to be an error, and the value of `errno` is not changed.

<`math.h`> contains all of the ANSI C Standard function prototypes and macros. If `UNIX+` is specified, a number of `BSD4.3` additional function prototypes and macros become available.

## 15.4 signal.h - Signal Handling

<`signal.h`> defines a number of macros and function prototypes that allow for interception of various events by a C program.

The signal parameters recognized by the `signal` facility follow. Those preceded by an * (asterisk) may result from the environment outside the library environment, e.g., an attention interrupt or a hardware program interrupt, or from a call to the `raise` function. The others currently can only be raised. Some signals can be ignored, blocked (via the user specifying a blocking mask), or caught (by the user specifying a signal handler). `SIGKILL` and `SIGSTOP` cannot be ignored or caught. `SIGKILL`, `SIGSTOP`, and `SIGCONT` cannot be blocked. For all other signals, they may be ignored, blocked, or caught.

The following is the list of the supported signals (`SIG_IGN` means ignore the signal):

* `SIGHUP` `1` Hangup. (Default = `$SIGNOFF` from `MTS`.)

* `SIGINT` `2` Attention interrupt. (Default = Error exit to `MTS` with the process still loaded.)

* `SIGQUIT` `3` Quit. The same as `SIGTERM`, but can be raised from the keyboard. (Default = Terminate the process via `exit(0)`.)

* `SIGILL` `4` Illegal operation or invalid function image (per the C Standard). This is caused by either an operation exception, privileged operation exception, execute exception, or specification exception. (Default = Error exit to `MTS` with the process still loaded.)

* `SIGIOVF` `5` Integer overflow caused by a fixed-point overflow. (Default = `SIG_IGN`.)

* `SIGABRT` `6` Abnormal termination. Causes the process to terminate without cleanup. (Default = Error exit to `MTS` with the process still loaded.)

* `SIGDECOVF` `7` Decimal overflow caused by decimal overflow exception. (Default = `SIG_IGN`.)

∗ `SIGFPE` `8` Floating point exceptions, caused by the following hardware exceptions: data exception (decimal operations), integer divide by zero, decimal divide by zero, floating exponent overflow, and floating point divide by zero. (Default = Error exit to `MTS` with the process still loaded.)

∗ `SIGKILL` `9` Terminate the process. (Cannot be caught, ignored, or blocked.) (Default = `$SIGNOFF` from `MTS`.)

∗ `SIGBUS` `10` Bus error. This is caused by a protection exception in MTS. (Default = Error exit to `MTS` with the process still loaded.)

∗ `SIGSEGV` `11` Segmentation violation (invalid storage access) caused an addressing exception in MTS. (Default = Error exit to `MTS` with the process still loaded.)

∗ `SIGSYS` `12` Bad argument to system call. (Default = Error exit to `MTS` with the process still loaded.)

∗ `SIGPIPE` `13` Write on pipe with no one to read it. (Default = Error exit to `MTS` with the process still loaded.)

∗ `SIGALRM` `14` Alarm clock. Typically used by a process so that it can be woken up at a specific time in the future. (Default = Error exit to `MTS` with the process still loaded.)

∗ `SIGTERM` `15` Software termination signal. In `UNIX` the system will send this signal to all processes shortly before a shutdown to give the processes a chance to cleanup. (Default = Terminate the process via `exit(0)`.)

∗ `SIGURG` `16` Urgent condition on I/O channel, usually an out-of-band message on a `TCP` stream. (Default = Error exit to `MTS` with the process still loaded.)

∗ `SIGSTOP` `17` A stop signal from something other than a terminal. (Cannot be caught, ignored, or blocked.) (Default = Error exit to `MTS` with the process still loaded.)

∗ `SIGTSTP` `18` Stop signal from the terminal. In UNIX this is used to checkpoint a process and place it in the background. (Default = Error exit to `MTS` with the process still loaded.)

* `SIGCONT` 19 Continue a stopped process. (Cannot be blocked.) (Default = Error exit to `MTS` with the process still loaded.)

* `SIGCHLD` 20 This signal is sent to the parent when a child stops or exits. (Default = Error exit to `MTS` with the process still loaded.)

* `SIGCLD` The same as `SIGCHLD`. Given for backwards compatibility. (Default = Error exit to `MTS` with the process still loaded.)

* `SIGTTIN` 21 Given to reader's process group upon background read from a terminal. (Default = Error exit to `MTS` with the process still loaded.)

* `SIGTTOU` 22 Like `SIGTTIN` for output. (Default = Error exit to `MTS` with the process still loaded.)

* `SIGIO` 23 Given to a process when input or output is possible (asynchronous). (Default = `SIG_IGN`.)

* `SIGXCPU` 24 Given to a process when it has exceeded its CPU time limit. (Default = Error exit to `MTS` with the process still loaded.)

* `SIGXFSZ` 25 Given to a process when it has exceeded its file size limit. (Default = Error exit to `MTS` with the process still loaded.)

* `SIGVTALRM` 26 Virtual time alarm. (Default = Error exit to `MTS` with the process still loaded.)

* `SIGPROF` 27 Profiling time alarm. (Default = Error exit to `MTS` with the process still loaded.)

* `SIGSIGNF` 28 Floating point significance caused by hardware floating point divide by zero. (Default = `SIG_IGN`.)

* `SIGUNFLO` 29 Floating point underflow caused by hardware floating point exponent underflow. (Default = `SIG_IGN`.)

* `SIGUSR1` 30 User-defined signal 1. (Default = `SIG_IGN`.)

* `SIGUSR2` 31 User-defined signal 2. (Default = `SIG_IGN`.)

The equivalent of "`signal(sig,SIG_DFL);`" is not executed prior to calling a user-defined signal handler for the signal `sig`. Further occurrences of signal `sig` are blocked until the user's handler returns. The user's signal handler function remains in effect until the user explicitly invokes the signal function to change the signal handling setting, i.e, via `SIG_IGN`, `SIG_DFL`, or another handling function.

In some systems the `SIGILL` signal default is reset upon delivery of the signal to a user-specified handler. This implementation treats `SIGILL` no differently from the other signals.

Blocking of a signal, if allowed, is accomplished by either the user calling the UNIX kernel functions `sigblock` or `sigsetmask`, or by a signal being delivered by the signal facility to a user-defined signal handler function. In the latter case, future occurrences of the same signal are implicitly blocked by the signal facility. For either case of blocking, only the first occurrence of each blocked signal is queued for processing; other occurrences of the same signal are ignored. A queued event is processed by the signal facility, when it becomes unblocked (by a user function call or by a return of a user-defined signal handler for which the signal facility implicitly blocked the signal).

If integer overflow checking is to be used, the `CHECKIOVER` option must be enabled when the program is compiled.

## 15.5 stdio.h - Input/Output

### 15.5.1 Header Definitions

The header <`stdio.h`> declares one type, several macros, and many functions for performing input and output. It contains the standard definitions and also additional definitions and macros that are contained in <`stdio.h`>, in the BSD4.3 implementation of UNIX.

### 15.5.2 Input/Output Alternatives

There are several methods of performing input/output in C:

- Use the *standard C I/O library*, which provides services in a manner compatible with other C implementations. Because this method is portable, it is preferred.
- Use *UNIX functions* which are not in the Standard C library. Many of the UNIX BSD4.3 functions are provided in ∗C89LIB. A detailed description of what is and what is not available is included in later parts of this documentation.

  More complete descriptions of these functions can be found in *UNIX Programmer's Reference Manual, April 1986*.
- Call *MTS system I/O subroutines and functions* directly. A description of these can be found in *MTS Volume 3: System Subroutine Descriptions*, Reference R1003.

### 15.5.3 MTS File Organization

The MTS file system, with its line-oriented files, is unusual among operating systems. In particular, many C programs assume a byte-oriented structure, in which newline characters are stored explicitly (rather than causing a skip to the next line of the file) and bytes may be addressed individually. The MTS file structure, however, defines the boundaries of lines, i.e., the beginning and end of each line; hence, newline characters are not used to separate lines. In UNIX systems, files may be sparsely populated, and non-existent bytes may be addressed individually. Currently MTS, the standard C I/O library, and the underlying UNIX kernel all require that for a byte to be addressed, it must physically exist within a file's bounds. Of course, in MTS, as in UNIX, one may append data to the end of a file.

The MTS file system provides features that are not in some other operating systems. Features which are not compatible with doing an `fseek` or `lseek` on a file are:

- Explicit file concatenation
- Implicit file concatenation
- $ENDFILE
- Sequential files

### 15.5.4 File Types and Input/Output Modes

∗C89LIB allows files to be used in either of two modes, *text mode* and *binary mode*. The mode affects the behavior of the following UNIX kernel routines: `read`, `write`, `open`, `lseek`, `truncate`, `ftruncate`, `stat`, `lstat`, `fstat` and most C standard I/O routines.

A file accessed in text mode is referred to as a *text file* and a file accessed in binary mode is referred to as a *binary file*. This is compatible with the ANSI C Standard. In many systems (including UNIX), both text mode and binary mode can use the same mechanism, and there need not be any difference between a text file and a binary file. However, this isn't possible on MTS.

*Binary mode* is extremely close to UNIX behavior. In binary mode, bytes are read and written unchanged, including the newline character. File streams opened in binary mode may be positioned to arbitrary byte positions using integer offsets. Almost all C programs that originated on a UNIX system and use the standard I/O library will run unchanged in binary mode.

The actual MTS records written in binary mode (except possibly the last record) will be of constant length, which is 1024 in this implementation. However, a program using ∗C89LIB need not be aware of this fact (the program can write in any size it wants).

In general, files written in binary mode cannot be directly read by other pieces of MTS software. They cannot be viewed easily by the MTS File Editor or similar methods; they can be read or written only by a program using ∗C89LIB.

*Text mode*, the default, allows files to be written that can be read by other MTS programs and vice versa. The behavior of I/O on a text file differs from I/O on a binary file in a number of respects:

- Trailing blanks at the end of a line may not appear when the file is read in.
- If you seek to a position within a text file where there is a newline '`\n`' and write a character other than a newline character, the newline character remains in the file. Replacing the very last '`\n`' in the file is possible.

- If you seek to a position within a text file where there is a character other than a newline character and write a newline character, the newline character is ignored and the original character is replaced by a blank space character. (That is, it is not possible to change the number of lines in a text file after they have been written except by use of the routines `truncate` and `ftruncate`).
- It is not possible to seek to a position beyond the end of a text file.
- If two newline characters are written one after the other to a text file, it is processed as if three characters were written: a newline character, a space, and a newline character, in that order.
- Control characters may be replaced with other characters. See section 15.5.5, "Control Characters in Output Streams," for more information.
- The character written in column one of the file may be interpreted as a carriage control character. See section 15.5.5, "Carriage Control Characters," for more information.
- No more than 32,767 non-newline characters may be written between consecutive newline characters on a file. (If this is attempted, newline characters will be inserted every 32,767 characters.) For devices other than files, there is an ananalogous limit which may be 32,767 or a smaller number, depending on the device and the circumstances.
- If a file has been opened for writing, characters have been added beyond the end of the file, and the last such character written was not a newline character, a newline character gets written automatically when the file is closed, when `_exit` is called, or when the program terminates.
- A text file can be read or written by any MTS program.

*C89LIB cannot determine whether a file is text or binary without help from the programmer. It will make assumptions and those assumptions may be wrong in some cases. There are ways to override the normal behavior of *C89LIB to make sure a specific file is processed correctly.

The kernel routines `open` and `creat` accept `O_TEXT` and `O_BINARY` for flag values to denote text or binary mode files. The standard C routines `fopen` and `freopen` use the character b in the mode parameter to indicate binary mode. Once a file has been used as a text file or a binary file, the run-time system will remember this until `_exit` is called (calls to `open` and `creat` will ignore the `O_TEXT` and `O_BINARY` parameters for such files; they will select the type of the file based on its remembered value).

If a program has calls to the routines `truncate`, `lstat`, or `stat`, the system can determine the file type if the file has been previously used by the program. In other cases the programmer can use the routine `__deffile` to specify the correct interpretation. The call `__deffile(0)` causes all future such situations to assume text mode. The call `__deffile(1)` causes all future such situations to assume binary mode.

**Warning:** If a file is not opened correctly, information will be returned incorrectly. If a file is opened for a write and opened incorrectly, the file will almost certainly be damaged in the process.

### 15.5.5 Control Characters in Output Streams

For a binary stream, all characters are passed unchanged in I/O operations with both the standard and the kernel I/O routines.

This is the default for text files also, with the exception of '`\n`', which is always interpreted as skipping to the next line, and no character, as such, gets written to the file.

For text files, a method is provided to indicate that control character interpretation is required for a given `fd`, i.e., a file descriptor. A call to `ioctl`, that is, `ioctl(fd,XHTABS,NULL)` will indicate to the run-time system that control character interpretation is expected for the file accessed by this `fd`, on I/O calls using this `fd`. If a file has been opened with `fopen`, the file descriptor number is available through the `fileno` macro. For output to the terminal, this feature becomes active by default. No call to `ioctl` is needed. Note: this feature is allowed for files or devices opened only for output.

The following apply to text files where control character interpretation has been requested.

- The tab character '`\t`' is expanded into the appropriate number of spaces to move the column position to the next tab column (the tab stops are set every 8 spaces).
- '`\r`' moves the buffer position to the beginning of the buffer unless it is already at the beginning, in which case it has no effect.
- '`\b`' moves the buffer position indicator back by one unless the buffer position indicator is already at the beginning of the buffer, in which case it has no effect.
- '`\v`' skips to the next line. That is, the current output line is written out, and the buffer position points to the same column it was on before the '`\v`' was encountered; the spaces up to that position are filled with space characters.
- '`\f`' gets replaced with a blank space if carriage control mode is not in effect. If carriage control mode is on, then the current output line is written out, and the character 1 gets written at the first position in the buffer.

### 15.5.6 Carriage Control Characters

In some cases the information in column one of a file will not be visible but, instead, will be removed and used to control the amount of spacing between lines. Whether this occurs or not depends on where the output line is being sent.

When output is sent directly to a terminal, column one is printed on the terminal; i.e., it is not interpreted. However, if the output is directed to a file and then to a terminal, column one is not printed but is used to control spacing. Normally, output sent to `*PRINT*` (the printer) will have column one interpreted for carriage controls. Output sent anywhere else is not interpreted for carriage controls.

The most common carriage controls are '`1`' to advance to the top of the next page, '` `' for single space, '`0`' for double space, and '`-`' for triple space. Other characters are specified in *MTS Volume 1: The Michigan Terminal System*, Reference R1001.

For text files, a method is provided to indicate that carriage control mode is required for a given `fd`, i.e., a file descriptor. A call to `ioctl`, that is, `ioctl(fd,CRCTL,NULL)` will indicate to the run-time system that carriage control characters need to be written for the file accessed by this `fd`, on I/O calls using this `fd`. If a file has been opened with `fopen`, the file descriptor number is available through the `fileno` macro. For output to the terminal, this feature becomes active by default. No call to `ioctl` is needed. Note: this feature is allowed for files or devices opened only for output.

When this mode is active, every output line is written with a blank or a carriage control character if a control character that effects a carriage control was part of the output (example: '`\f`').

**NOTE:** If a file that is being used for output with carriage control is opened multiple times or opened for read/write there may be unexpected interactions.

### 15.5.7 MTS File Names

MTS does not differentiate between upper-case and lower-case characters within file names. In MTS, the string argument `filename` for `open` and `fopen` may contain any of the following:

- A file name, possibly preceded by a userID and followed by I/O modifiers and a line number range, e.g., `MYID:MYFILE(1,20)`. Line number ranges and I/O modifiers are only allowed if the `_TRUSTME` flag has been set.
- A device name, e.g, ∗`PRINT`∗.
- A sequence of file or device names explicitly concatenated with "+" characters, e.g., `A+B(1,10)+ABCD:THEIRFILE`. This is normally not allowed (see below).
- A logical unit name or number preceded with a "|" (vertical bar) character, e.g., |`INPUT`, |`PRINT`, |`OBJECT`, |`0`, |`1`, ..., |`99`.

If a file or device is opened for writing, or reading and writing, then use of the MTS-specific features of I/O modifiers, line number ranges, and explicit concatenation are not allowed. Open will set errno to `EIO` and return in these cases. And implicit concatenation also is not available since, `$Continue with` and `$Endfile` are treated as data.

The above is the default behaviour for all files and devices. However, if the program contains a definition for an external variable called `_TRUSTME`, then explicit and implicit concatenation, line number ranges and I/O modifiers are allowed for files/devices opened for input only. (This does not apply when the routine `C89INIT` is used to initialize the C environment. In that case, these MTS features are allowed only if bit 3 is used when `C89INIT` is called). When `_TRUSTME` or bit 3 of `C89INIT` is used, the programmer must be aware of the following consequences.

If there is more than one instance of the file being open and one of these instances has the I/O modifier or the line number ranges, it may cause problems with synchronization, and the results of read calls may be erroneous. If one such instance is for the file to be opened with write, the behaviour is unpredictable. Seeking to random byte positions in the file will also not work, when concatenation or line number ranges are involved.

Basically, if the file is being opened multiple times and I/O is performed through multiple fps, then concatenation, line number ranges, and the other I/O modifiers should not be used. If they are, then the results are unpredictable.

For more information on valid file names, consult *MTS Volume 1: The Michigan Terminal System*, Reference R1001.

### 15.5.8 Initial Stream Assignments

At the start of execution, `stdin` is assigned to logical I/O unit `INPUT`, `stdout` to the logical I/O unit `PRINT`, and `stderr` to to the logical I/O unit `SERCOM`. By default, ∗C89 will complain if either `stdin` or `stdout` is assigned to a file/device that does not exist or to which you have no access. Also by default, `stdout` and `stderr` are emptied if they are attached to files.

If any of this behavior is not desired, it is possible to suppress the initial opening of the three I/O units. To do this, place a reference to the external variable `_noopen` anywhere in the program. Then the three units can be manually opened as desired by using the routine `freopen`. Note that the units must be opened in the following order: `stdin`, `stdout`, and then `stderr`. For example:

```
#include <stdio.h>

int _noopen;

main()
 {

   /* Open stdin first. */
   if(freopen("|INPUT", "w", stdin)==0)
     {FILE *msink;
      /* stderr hasn't been opened yet, so don't
         write an error message on it. */
      msink = fopen("*MSINK*", "w");
      fprintf(msink, " Error opening INPUT.");
      exit(4);
     }

   /* Open stdout second. */
   if(freopen("|PRINT", "w", stdout)==0)
     {FILE *msink;
      /* stderr hasn't been opened yet, so don't
         write an error message on it. */
      msink = fopen("*MSINK*", "w");
      fprintf(msink, " Error opening PRINT.");
      exit(4);
     }
   setlinebuf(stdout);

   /* Open stderr third. */
   if(freopen("|SERCOM", "w", stderr)==0)
     {FILE *msink;
      /* stderr hasn't been opened yet, so don't
```

```
         write an error message on it. */
      msink = fopen("*MSINK*", "w");
      fprintf(msink, " Error opening SERCOM.");
      exit(4);
     }
   setlinebuf(stderr);

   /* Rest of the program ... */
  }
```

Of course this may be modified as needed to suit a given situation.

### 15.5.9 Random Access

Disk files may be processed randomly by requesting that a given byte position be made the current one. The functions `fseek` and `lseek` set the byte position, while `ftell` returns the current byte position.

MTS random access organization is line-oriented, while the C library random access support is byte-oriented. To efficiently access bytes in MTS files, one of two byte-access organizations is used, depending on whether the file stream is in binary or text mode.

In binary mode the byte positions are specified by integers, making it possible to seek to a specific byte, given its offset. This is very efficient, since all lines in the file, except for possibly the last, are of the same length, which makes the byte-position computation simple.

Random access must be handled carefully. Changing the length of a line in a file opened for random access will result in an error that will not be detected by the ∗C89LIB I/O support.

Those who want to access MTS files by line number may call MTS subroutines, such as `MTSREAD` and `MTSWRITE`, directly.

### 15.5.10 Terminal Input/Output

Terminal input/output is a sensitive area for programming support. It involves the way in which the operating system handles the terminal, as well as the actual behavior of the terminal when given various control codes.

Many UNIX-based programs use a library of terminal-independent routines named `curses`. This library is not available with ∗C89. Sophisticated applications may call the MTS screen-support routines for full control of the terminal.

The terminal support provided with ∗C89 is character-based, providing the ability to read lines from the terminal and to write lines or line segments to the terminal. The normal character I/O functions are used for this purpose.

### 15.5.11 Using the MTS I/O Routines

It is possible to directly call the MTS routines `READ` and `WRITE`, but this must be done with some care. First, the routines must be called by their aliases `MTSREAD` and `MTSWRITE` so they don't interfere with the routines `READ` and `WRITE` in the C library. Second, they must not use any units that are also being used by the C I/O routines. Third, the `__fortran` linkage must be used.

### 15.5.12 Implementation Specifics for the I/O Routines

**fopen** `fopen` and `freopen` are system-dependent by their nature, since they both use a parameter that is a file name. For this reason, `fopen` and `freopen` calls are *not* generally portable. For information on constructing a valid file name, see section 15.5.7, "MTS File Names."

If the file is opened in *append* mode, the file position indicator is initially placed at the end of the file. Opening a file multiple times may cause problems with I/O synchronization, and hence, is strongly discouraged.

**remove** If the file is open, the `remove` function returns with an error, and `errno` is set to `EINTR`.

**rename** If a file with the new name already exists, the function `rename` returns with an error and `errno` is set to `EEXISTS`.

**I/O formatting** The output for `%p` conversion for `fprintf` and the input for `%p` conversion for `fscanf` are the same as those used for `%x` conversion. There is no special meaning attached to the '`-`' character in the scan list for the `fscanf` `%[` format.

**Random I/O** The functions `ftell` and `fgetpos` set `errno` to `ESPIPE` if the file is not indexable. In text mode, `errno` gets set to `ENOMEM` if memory is not available for allocating needed data structures.

**perror** `perror(s)` will produce the following output on `stderr`: if `s` is not a null pointer and the character pointed to by it is not the null character, `s` is written followed by a colon (:), a space, and the appropriate error message. The error message is produced by using the current setting of `errno` as an index into the array `sys_errlist`, which contains the error messages. For the list of the error messages, see the include member `sys/errno.h` or the *UNIX Programmer's Reference Manual, April 1986.* The error messages and error numbers produced by *C89LIB are identical to those produced by BSD 4.3 UNIX.

**Input and Output** Writing to a stream does not cause the associated file to be truncated beyond that point. MTS allows zero-length files to exist. The implementation does not set a limit on the number of null characters that may be appended to data in a binary stream. (See section 15.5.3, "MTS File Organization" and section 15.5.4, "File Types and Input/Output Modes.")

**I/O buffers** There are two I/O buffers used by the run-time system. The outer level buffer is the one modified by calls that use the `FILE` pointer. A call to `fflush` or any action that requires buffer flushing moves this buffer's contents onto an internal I/O buffer. The internal buffer is modified also by the kernel calls dealing with I/O. When the internal buffer overflows or the '`\n`' is encountered, its contents are written to the file or I/O unit. The contents of this buffer are written to the file or I/O unit when calls are made that require updating the contents of the physical unit.

## 15.6 stdlib.h - General Utilities

This header contains the macros and function prototypes required by the ANSI C Standard as general utilities.

### 15.6.1 Implementation Specifics for the Utilities

**calloc** If size requested is zero, `calloc` returns `NULL`.

**malloc** If size requested is zero, `malloc` returns `NULL`.

**realloc** If size requested is zero, `realloc` frees the pointer and returns NULL. If the ptr is NULL and the size is non-zero, `realloc` allocates memory and returns a pointer to the start of the memory. If memory is expanded, `realloc` does not initialize the extra space that is allocated.

**abort** The `abort` function flushes the I/O buffers of all open files, destroys any temporary files created by `tmpfile`, and terminates program execution with an error code of 8. It never returns to the caller.

**exit** A call to the `exit` function terminates execution of the program, and the value of its argument is returned as the MTS return code.

**getenv** The `getenv` function uses the string argument to obtain the value of a MTS command macro variable. If there is no macro variable with that name, `NULL` is returned. Otherwise, the value is converted to a string and that string is returned. To set the environment variable, use the MTS Command Macro Processor. For details refer to *MTS Volume 21: MTS Command Extensions and Macros*, Reference R1021.

**system** The `system` function takes a string parameter that is in the form of an MTS command. This command is passed to MTS for processing.

## 15.7 string.h - String Handling

This header contains all the standard macros and prototype definitions required by ANSI C Standard as well as some string function available in BSD4.3 UNIX.

### 15.7.1 memcmp

Note: Objects with "holes" (used as padding for alignment purposes within structure objects), strings that are shorter than their allocated space, and unions may cause problems in comparison with `memcmp`.

### 15.7.2 strerror

The error messages generated by `strerror` are the same as the contents of the array `sys_errlist`. The integer argument is used as an index into the array. For the list of the error messages, see the include member `sys/errno.h` or the *UNIX Programmer's Reference Manual, April 1986*. The error messages and error numbers produced by *C89LIB are identical to those produced by BSD4.3 UNIX.

### 15.7.3 stricmp

**Name: stricmp**

**Purpose:** *String comparison (case insensitive)*

**Include file:** <`string.h`>

**Prototype:** `int stricmp(const char *s1, const char *s2)`

**Description:** `stricmp` returns an integer result similar to `strcmp`.

The `stricmp` function is similar to `strcmp`, except it is case insensitive.

**Note:** This function is not required by the ANSI C Standard. Programs using this function may not be portable.

**Example:**
```
#include <string.h>
char *s1, *s2;
...
if (stricmp(s1,s2)==0)
   {
     ...
   }
   /* Here, if s1 and s2 are the same except for
      upper-case/lower-case differences in one
      or more characters, stricmp returns 0 */
```

**See also:** `strcmp, strncmp, strnicmp, memcmp`

### 15.7.4 strnicmp

**Name: strnicmp**

**Purpose:** *String comparison (case insensitive with count)*

**Include file:** <`string.h`>

**Prototype:** `int strnicmp(const char *, const char *, size_t)`

**Description:** This function returns an integer similar to `strcmp`.

The `strnicmp` function is similar to `stricmp`, except it compares not more than the third parameter number of characters from the string pointed to by the first parameter to the string pointed to by the second parameter.

**Note:** This function is not required by the ANSI C Standard. Programs using this function may not be portable.

### 15.7.5 strchr

Note: The name `index` is recognized as a synonym for this function, since some UNIX implementations use the name `index`. For portability considerations, use of the name `strchr` is preferred.

### 15.7.6 strrchr

Note: This function is known by the name `rindex` in some UNIX implementations. With ∗C89LIB, the name `rindex` is a synonym for this function. For portability considerations, use of the name `strrchr` is preferred.

### 15.7.7 strlwr

**Name: strlwr**

**Purpose:** *Convert string to lower case*

**Include file:** `<string.h>`

**Prototype:** `char *strlwr(char *s)`

**Description:** This function returns a pointer to the transformed string `s` as described below:

`strlwr` converts the upper-case characters in the string pointed to by `s` to their corresponding lower-case characters. The non-alphabetic characters are unaffected by this transformation.

**Note:** This function is not required by the ANSI C Standard. Programs using this function may not be portable.

**Example:**
```
#include <string.h>
char  *t;
char s[50];            /* Space for a string.    */
strcpy(s,"aBc;Def");   /* Initialize it.         */
...
t = strlwr(s);         /* Convert to lower case. */
  /*  Both s and t  now point to "abc;def".      */
...
```

**See also:** `strupr`

### 15.7.8 strupr

**Name: strupr**

**Purpose:** *Convert string to upper case*

**Include file:** <`string.h`>

**Prototype:** `char *strupr(char *s)`

**Description:** This function returns a pointer to the transformed string `s` as described below:

`strupr` converts the lower-case characters in the string pointed to by `s` to their corresponding upper-case characters. The non-alphabetic characters are unaffected by this transformation.

**Note:** This function is not required by the ANSI C Standard. Programs using this function may not be portable.

**Example:**
```
#include <string.h>
char  *t;
char s[50];             /* Space for a string.    */
strcpy(s,"aBc;Def");    /* Initialize it.         */
...
t = strupr(s);          /* Convert to upper case. */
  /*  Both s and t  now point to "ABC;DEF".       */
...
```

**See also:** `strlwr`

### 15.7.9 reverse

**Name: reverse**

**Purpose:** *Convert string to its reverse order*

**Include file:** <`string.h`>

**Prototype:** `char *reverse(char *s)`

**Description:** `reverse` returns a pointer to the string `s`. It reverses the order of the characters in the string `s`.

**Note:** This function is not required by the ANSI C Standard. Programs using this function may not be portable.

**Example:**
```
#include <string.h>
char *s, *p;
...
strcpy(p,s);
if (strcmp(reverse(s),p)==0)
   printf(" %s is a palindrome\\n",s);
...
```

## 15.8 time.h - Date and Time

In addition to routines required by the ANSI C Standard, the routines `ftime`, `times`, and `gettimeofday` are supplied in <`time.h`>. They are not part of the ANSI C Standard but are available in UNIX. They behave as described in the *UNIX Programmer's Reference Manual, April 1986*.

The local time zone, as used by these functions, varies depending on where the program is executed. For Ann Arbor, Michigan and Troy, New York, either EST or EDT is used. For Vancouver, British Columbia, either PST or PDT is used. The times when transitions to and from Daylight Savings time occur are based on a table, which is accurate for most of North America.

The era for the `clock` function is relative to the starting time of the MTS task that is running the program.

## 15.9 mts.h - MTS Specific Routines

A number of prototypes are defined in <`mts.h`>. It is important to use these prototypes so that the proper linkage is made to the appropriate MTS routines. Most of these routines are described in *MTS Volume 3: System Subroutine Descriptions*, Reference R1003.

Some additional C routines are described below.

### 15.9.1 atoe

**Name: atoe**

**Purpose:** *Translate ASCII to EBCDIC*

**Include file:** <`mts.h`>

**Prototype:** `char atoe(char *str, int len)`

**Description:** The function `atoe` converts the characters in the ASCII string pointed to by the argument `str` to their corresponding EBCDIC representation. It returns the last character after conversion and `NULL` if the string is empty.

This function uses the ASCII to EBCDIC conversion table provided by the system.

**Note:** This function is not required by the ANSI C Standard. Programs using this function may not be portable.

**See also:** `etoa`

### 15.9.2 etoa

**Name: etoa**

**Purpose:** *Translate EBCDIC to ASCII*

**Include file:** <`mts.h`>

**Prototype:** `char etoa(char *str, int len)`

**Description:** `etoa` converts the characters in the string pointed to by the argument `str` to their corresponding ASCII representation. It returns the last character after conversion and `NULL` if the string is empty.

Each character is converted to the corresponding 7-bit ASCII character code, with the parity bit set to zero.

**Note:** This function is not required by the ANSI C Standard. Programs using this function may not be portable.

### 15.9.3 query

**Name: query**

**Purpose:** *Prompt for terminal input*

**Include file:** <`mts.h`>

**Prototype:** `int query(char *prompt, char *reply, int maxinplen)`

**Description:** `query` displays the string pointed to by `prompt` on the `stdin` stream, if `stdin` is attached to a terminal; otherwise, the prompt is not written. Then `stdin` is read with the input being truncated, if necessary, to the length `maxinplen` - 1, to allow for appending a '`\0`' to the end of the input to make it a string. The input string result is stored in the object pointed to by `reply` and the function returns.

The length of the prompting string is limited to 79 characters, not including its '`\0`' terminator. If the string is longer, only 79 characters will be used for the prompt.

`query` returns a zero if input was successfully done; it returns `EOF`, if either an end-of-file was encountered or an input error occurred.

**Note:** This function is not required by the ANSI C Standard. Programs using this function may not be portable.

**Example:**
```
#include <mts.h>
 ...
int ret;               /* return value from query */
static char prompt[]="Input please:";
static int maxlen=100; /* maximum input length.   */
char inparea[100];     /* the input line          */
 ...
ret=query(prompt,inparea,maxlen); /* Prompt       */
if (ret!=0)
   printf("Unable to prompt and read.\n");
 ...
```

# 16 BSD4.3 UNIX Routines

This section mainly goes over the UNIX routines and their expected behaviour in this library. These fall into two categories: the kernel routines and the non-kernel routines. The kernel routines are described in chapter 2 of the *UNIX Programmer's Reference Manual, April 1986* and the others are described in chapter 3. Many of the ANSI C Standard routines are also described in chapter 3 of the *UNIX Programmer's Reference Manual, April 1986*. The ANSI C Standard also has routines not mentioned in chapter 3.

## 16.1 Kernel Routines

This section deals with the kernel routines of UNIX. Most of these routines go through a routine called `syscall` which connect the kernel and non-kernel levels of the library. This section describes how the behavior of *C89LIB is different from BSD4.3. If not otherwise specified, each routine is assumed to behave identically to BSD4.3. Full descriptions are given of extensions and implementation-defined features. The routines `read`, `write`, `open`, `close` and `lseek` are described in the section <`stdio.h`>.

### 16.1.1 File Access - Querying and Modifying

The routines that deal with file access (either specifying, modifying, or simply retrieving access information) are the routines `access`, `creat`, `open`, `chmod`, and `fchmod`. For each of these routines that requires a "pathname," an MTS filename should be supplied instead.

When an existing file is used, the access of the file is queried to determine whether a given type of use is allowed. This query determines the result of the routine `access`.

`READ` access (in the MTS sense) for the current userID, project, and program key is needed for `read access` in the UNIX sense.

`READ, WRITE/CHANGE` and `WRITE/EXPAND` access (in the MTS sense) for the current userID, project, and program key are needed for `write access` in the UNIX sense.

`READ` access (in the MTS sense) for the current userID, project, and the program key `*MTS.RUN` is needed for `execute access` in the UNIX sense.

Note that it is possible to permit a file (using MTS mechanisms) such that its access controls are more complicated than can be represented by the routine `access`.

Currently both `creat` and `open` ignore their third parameters.

In UNIX, the access of a file is changed with `chmod` or `fchmod`, which is not implemented in ∗C89LIB.

### 16.1.2 Miscellaneous I/O Routines

Various I/O related UNIX routines have been implemented:

`close`, `dup`, `dup2`, `fcntl`, `flock`, `fsync`, `ftruncate`, `getdtablesize`, `ioctl`, `lseek`, `read`, `readv`, `rename`, `sync`, `truncate`, `umask`, `unlink`, `write`, `writev`.

For each of these routines that requires a "pathname," an MTS filename should be supplied instead.

**sync** `sync` causes an `fsync` to be performed on all files open by the current process. (It has no effect on other processes).

**fcntl** For `fcntl` the `F_GETOWN` and `F_SETOWN` commands are not implemented. The `FASYNC` flag is not implemented.

**flock** Unlike UNIX, file locking is mandatory; that is, reading a file causes the equivalent of `flock(d, LOCK_SH)` to be executed. Writing to a file or modifying it causes the equivalent of `flock(d, LOCK_EX)` to be executed. Also `flock` will allow `LOCK_EX` only if the file is permitted for `write`.

A file is said to be *open* if some process has called `open` on the file and has not yet called `close` or is making some other use of the file. While a file is open, other processes may not `unlink`, `rename`, or `chmod` the file. If `flock` is called with `LOCK_UN`, the file will be unlocked but it will remain open.

**rename, unlink** If `rename` is called and the second file already exists, an error is returned and neither file is affected.

With successful calls to `rename` and `unlink`, the old file is instantly unavailable, and any attempts to do I/O on a file descriptor attached to the old file cause an error. The failure will occur only when actual I/O is done to the file and not when information is added to the buffer.

Permission to rename or unlink a file is not as described in the *UNIX Programmer's Reference Manual, April 1986.* A file can be renamed or unlinked if, and only if, the userID which calls `rename` or `unlink` has `DESTROY/RENAME` access to the file. Normally only the owner of the file will have such access. This access can be changed by using the `$PERMIT` command or the MTS `PERMIT` subroutine. It cannot be changed by use of the `chmod` routine.

### 16.1.3 Sockets in ∗C89LIB

The following socket-related routines have been implemented:

`accept`, `bind`, `connect`, `gethostname`, `getpeername`, `getsockname`, `listen`, `recvfrom`, `select`, `sendto`, `socket`.

The following socket routines have not been implemented. They set `errno` to `EFAULT` and return -1 when called:

`gethostid`, `getitimer`, `setitimer`, `getsockopt`, `recv`, `recvmsg`, `send`, `sendmsg`, `sethostid`, `sethostname`,`setsockopt`, `socketpair`

**UDP datagram** The UDP implementation on MTS can handle 1500 octets, the minimum required to be handled for Ethernet connections as per RFC 894. (Note: Seems to handle only 1472 bytes of actual data.)

The general default maximum size required by internet is only 576 octets. When sending data to hosts that are not on the same Ethernet, some hosts in the route may have to fragment the data if the length of the datagram exceeds their capacity, even if the intended receiver may be capable of handling the larger size.

**shutdown** `shutdown` closes the TCP connection. The behaviour is the same as in UNIX BSD4.3 `shutdown` when the second parameter has value 2. The partial shutdowns available with `how`=0 or `how`=1, where `how` refers to the second parameter, are not currently supported by our implementation.

### 16.1.4 Library Utility Routines

The following utility routines available in UNIX are implemented:

`_exit`, `getgid`, `getgroups`, `getegid`, `getpagesize`, `getpgrp`, `getpid`, `getppid`, `getpriority`, `getrusage`, `gettimeofday`, `getuid`, `geteuid`, `setpriority` and `syscall`.

It is assumed that the UNIX group ID is the same as the MTS project and the UNIX userID is the same as the MTS userID.

It is assumed that the UNIX process ID and the UNIX process group are the same and are both the same as the MTS task number. It is assumed that the parent to all processes is process 0. The process group cannot be changed.

**syscall** `syscall` is subject to the same constraints as the facility it calls. For example, `syscall(SYS_read,...)` has exactly the same behaviour as `read(...)`.

**getgroups** When the routine `getgroups` is called, one group is returned and it is the same as the current group userID.

**getpriority** It is not possible to modify the priority of a process. `getpriority` always returns 0.

**setpriority** `setpriority` returns 0 (OK) but does not do anything.

**setregid, setreuid** The system will not allow the UNIX group ID or the UNIX userID to be changed when `setregid` or `setreuid` is called.

### 16.1.5 Signals in *C89LIB

The routines `kill`, `killpg`, `sigblock`, `sigsetmask`, and `sigvec` have been provided.

When the routines `kill` and `killpg` are called, if the argument matches the current process, the call is processed as expected. If an attempt is made to signal another process with either `kill` or `killpg`, `EPERM` is returned.

Signal routines have some MTS-specific features, which are described earlier under <`signal.h`>.

The routines `alarm`, `siginterrupt`, `sigpause`, `sigreturn`, `sigstack`, `wait`, and `wait3` are not available.

### 16.1.6 Other Unimplemented Kernel Routines

The following routines are mainly of use to the superuser, and they set errno to `EPERM` and return -1 when called:

`acct`, `adjtime`, `chroot`, `mknod`, `mount`, `umount`, `quota`, `reboot`, `setgroups`, `settimeofday`, `setquota`, `swapon`, `vhangup`

The following routines pertain to directories. Since MTS does not have directories, they always set errno to `ENOTDIR` and return -1 when called:

`chdir`, `link`, `mkdir`, `readlink`, `rmdir`, `symlink`

The following routines have not been implemented. They return -1 and set errno to the values indicated:

```
  execve, fork, profil      -- EPERM
  ptrace, utimes, vfork     -- EPERM
```

```
brk, sbrk               -- ENOMEM
pipe                    -- EFAULT
chown, fchown           -- EPERM
setrlimit, setpgrp      -- EPERM
```

## 16.2 Non-Kernel Routines

This section deals with routines, header files, and macros other than the kernel routines. Unless otherwise specified, routines are assumed to behave identically to BSD4.3. Full descriptions are given of extensions and implementation-defined features. Some of the routines mentioned are also ones required by the ANSI C Standard, so they may have been mentioned earlier.

### 16.2.1 I/O Routines

`printf`, `fprintf`, `sprintf`, `_doprnt`, `scanf`, `fscanf`, and `sscanf` have been modified to conform to the ANSI C Standard. They all should be upward compatible with BSD4.3.

The following routines and macros have been implemented. However the details of their behaviour are analogous to the behavior for `read`, `write` and `lseek` as explained in section 16.1, "Kernel Routines."

`printf`, `fprintf`, `_doprnt`, `scanf`, `fscanf`, `fclose`, `fflush`, `ferror`, `feof`, `clearerr`, `fileno`, `fread`, `fwrite`, `ftell`, `rewind`, `getc`, `getchar`, `fgetc`, `getw`, `gets`, `fgets`, `putc`, `putchar`, `fputc`, `putw`, `puts`, `fputs`.

### 16.2.2 Signal

The routine `signal` works essentially as it does in BSD 4.3; however, there are some differences. See the earlier descriptions.

The routines `alarm` and `siginterrupt` do not work.

### 16.2.3 Character Macros

The following macros all assume an EBCDIC character set:

`isalpha`, `isupper`, `islower`, `isdigit`, `isxdigit`, `isalnum`, `isspace`, `ispunct`, `isprint`, `isgraph`, `iscntrl`, `isascii`, `toupper`, `tolower`, `toascii`. .

### 16.2.4 Other Implemented Routines

The following math routines have been implemented:

`acos`, `asin`, `atan`, `atan2`, `cabs`, `cbrt`, `ceil`, `copysign`, `cos`, `cosh`, `erf`, `erfc`, `exp`, `fabs`, `floor`, `hypot`, `infnan`, `lgamma`, `log`, `log10`, `pow`, `rint`, `sin`, `sinh`, `sqrt`, `tan`, `tanh`.

In addition, the following routines have been implemented:

`abort`, `abs`, `atof`, `atoi`, `atol`, `bcopy`, `bcmp`, `bzero`, `ffs`, `htonl`, `htons`, `ntohl`, `ntohs`, `crypt`, `setkey`, `encrypt`, `ctime`, `localtime`, `gmtime`, `asctime`, `timezone`, `ecvt`, `fcvt`, `gcvt`, `exit`, `frexp`, `ldexp`, `modf`, `fseek`, `gethostbyname`, `gethostbyaddr`, `gethostent`, `sethostent`, `endhostent`, `getopt`, `getpass`, `finite`, `inet_addr`, `inet_network`, `inet_ntoa`, `inet_makeaddr`, `inet_lnaof`, `inet_netof`, `insque`, `remque`, `malloc`, `free`, `realloc`, `calloc`, `alloca`, `mktemp`, `mkstemp`, `ns_addr`, `ns_ntoa`, `perror`, `sys_errlist`, `sys_nerr`, `psignal`, `sys_siglist`, `qsort`, `rand`, `srand`, `random`, `srandom`, `initstate`, `setstate`, `re_comp`, `re_exec`, `rexec`, `setbuf`, `setbuffer`, `setlinebuf`, `setjmp`, `longjmp`, `sleep`, `strcat`, `strncat`, `strcmp`, `strncmp`, `strcpy`, `strncpy`, `strlen`, `index`, `rindex`, `stty`, `gtty`, `swab`, `time`, `ftime`, `times`, `ttyname`, `isatty`, `ttyslot`, `ualarm`, `ungetc`, `usleep`, `utime`, `valloc`, `vlimit`, `vtimes`.

The `assert` macro and the macros defined in `varargs.h` are available.

The symbols `end`, `etext`, and `edata` are available.

### 16.2.5 Unimplemented Routines

The following routines are not implemented:

`dbm_open`, `dbm_close`, `dbm_fetch`, `dbm_store`, `dbm_delete`, `dbm_firstkey`, `dbm_nextkey`, `dbm_error`, `dbm_clearerr`, `execle`, `execlp`, `exect`, `execv`, `execve`, `execvp`, `environ`, `getfsent`, `getfsspec`, `getfsfile`, `getfstype`, `syslog`, `openlog`, `closelog`, `setlogmask`, `system`, `res_mkquery`, `res_send`, `res_init`, `dn_comp`, `dn_expand`, `scandir`, `alphasort`, `setuid`, `seteuid`, `setruid`, `setgid`, `setegid`, `setrgid`, `rcmd`, `rresvport`, `ruserok`, `popen`, `pclose`, `pause`, `nice`, `nlist`, `initgroups`, `opendir`, `readdir`, `telldir`, `seekdir`, `rewinddir`, `closedir`, `execl`, `setfsent`, `endfsent`, `getgrent`, `getgrgid`, `getgrnam`, `setgrent`, `endgrent`, `getlogin`, `getnetent`, `getnetbyaddr`, `getnetbyname`, `setnetent`, `endnetent`, `getprotoent`, `getprotobynumber`, `getprotobyname`, `setprotoent`, `endprotoent`, `getwd`, `getdiskbyname`, `getpw`, `getpwent`, `getpwuid`, `getpwnam`, `setpwent`, `endpwent`, `setpwfile`, `getservent`, `getservbyport`, `getservbyname`, `setservent`, `endservent`, `getttyent`, `getttynam`, `setttyent`, `endttyent`, `getusershell`, `setusershell`, `endusershell`.

The following math routines are not available:

`asinh`, `acosh`, `atanh`, `drem`, `expm1`, `log1p`, `logb`, `j0`, `j1`, `jn`, `scalb`, `y0`, `y1`, `yn`.

None of the multiple precision arithmetic routines are available:

`madd`, `msub`, `mult`, `mdiv`, `pow`, `gcd`, `invert`, `rpow`, `msqrt`, `mcmp`, `move`, `min`, `omin`, `fmin`, `m_in`, `mout`, `omout`, `fmout`, `m_out`, `sdiv`, `itom`.

None of the plotting routines are available:

`openpl`, `erase`, `label`, `line`, `circle`, `arc`, `move`, `cont`, `point`, `linemod`, `space`, `closepl`.

None of the termcap routines are available:

`tgetent`, `tgetnum`, `tgetflag`, `tgetstr`, `tgoto`, `tputs`.

In addition, none of the following routines are available:

`dbminit`, `fetch`, `store`, `delete`, `firstkey`, `nextkey`, `monitor`, `monstartup`, `moncontrol`.

Many UNIX-based programs use a library of terminal-independent routines named `curses`. These routines are not available with ∗C89LIB.

The routines available in some unix systems in the library `lib2648` are not available.

# 17 Incompatibilities With ∗C87

## 17.1 Incompatibilities with ∗C87LIB

Most programs using ∗C87LIB will work unchanged with ∗C89LIB if all components of the program are recompiled. There are, however, a few differences between ∗C87LIB and ∗C89LIB:

- `eat_cookie` is no longer needed and is not available.
- The values of `stdin`, `stdout`, and `stderr` cannot be modified. The routine `freopen` should be used to reassign these units.
- `fflush` only moves the level 3 buffer to the *kernel*. To do the equivalent of what `fflush` did in ∗C87LIB, one needs to call both `fflush` and `fsync`.
- The routines `read`, `write`, `open`, `lseek`, and `close` require small integers to be used as file descriptors (similar to what UNIX expects).
- `fseek` uses integer offsets instead of "magic cookies."
- `open` does not assume binary mode. The mode is passed through the appropriate flag value, namely `O_TEXT` or `O_BINARY`.
- Many more routines are available.

- By default, I/O modifiers, line number ranges, and explicit concatenation are not allowed. Any attempt to use these features will be treated as an error. `$CONTINUE WITH` and `$ENDFILE` lines are never treated as special; they are always treated as data. ∗C87LIB allowed these features to be used when files where opened with the `c` option. This is not allowed in ∗C89LIB. There is a different method of allowing `$CONTINUE WITH` and `$ENDFILE` lines, but it works on input only. This has been discussed under I/O.
- When writing to files or devices other than terminal, tab characters are not interpreted by default. An `ioctl` call is required to cause interpretation of tab characters.
- Carriage control is no longer supported via the use of the 'p' in the mode parameter for `fopen`. This is done with an `ioctl` call on files or devices opened only for output.
- `C87INIT` has been renamed `C89INIT`.
- The behaviour of `getenv` has changed.

## 17.2 Incompatibilities with ∗C87 Compiler

- ∗C89 does not predefine the macro _C87. It defines the macro _C89 instead.
- ∗C89INCLUDE is used as the default macro library instead of ∗C87INCLUDE. This means by default, you have to use ∗C89LIB to get programs to execute correctly.
- ∗C87 includes <`unix.h`> only when the option PAR=UNIX4.3 is used. ∗C89 always includes <`unix.h`> at the start of compilation.
- Object code produced by ∗C89 must be run with ∗C89LIB. It cannot be run with ∗C87LIB.
- The option `RETCODE` is no longer available. `RETCODE` users must rewrite their programs to use `__retcode` instead.
- `fortran` is no longer accepted as an alias for `__fortran`. The latter keyword must be used whenever FORTRAN linkage is required.

- When two or more function declarations have conflicting function prototypes, ∗C89 will print an error message under circumstances that ∗C87 does not.
- ∗C89 no longer allows declarations of two `typedef`s with the same name to occur in the same block.
- In some complicated initializers, ∗C89 and ∗C87 may produce different code.
- There are some subtle differences in the way the compiler determines whether an expression is signed or unsigned. This will have an effect only for the "<<", "`%`" and "`/`" operators.
- ∗C87 complained when non-portable usage was made of identifiers that began with an underscore. The rules on portability have changed since that part of the compiler was written; thus, ∗C89 uses different rules.
- ∗C89 may produce error messages in circumstances that ∗C87 does not, and conversely, ∗C89 may compile a program without complaint when ∗C87 does produce an error message.
- `LONG` defaults on. Hence by default, external identifiers are not truncated to 8 characters.
- ∗C87 always truncated pseudo-registers to 8 characters, even if the `LONG` option was turned on. ∗C89 truncates pseudo-registers only if the `LONG` option is off.

# Index