Appendix A

Implementation–Defined Behavior

The following sections describe implementation–defined behavior. Each section is keyed to the ANSI C Standard (ANSI X3.159–1989), Appendix F, and each point is keyed to the section number of the ANSI C Standard. The italicized lines, usually marked with bullets, are items from Appendix F of the ANSI C Standard. Text following the italic lines describes the Silicon Graphics implementation.

Translation (F.3.1)

- *Whether each nonempty sequence of white–space characters other than newline is retained or replaced by one space character (2.1.1.2).*

  A nonempty sequence of white–space characters (other than newline) is retained.

- *How a diagnostic is identified (2.1.1.3).*

  Successful compilations are silent. Diagnostics are, in general, emitted to standard error. Diagnostic messages have the general pattern of *file–name,line–number:severity(number): message* in 64–bit mode. Diagnostics have a slightly different pattern in 32–bit mode. Also, the range of numbers in 32–bit mode are disjointed from the range 64–bit mode.

  For example, typical messages from the ANSI C compiler front end in 64–bit mode look like this:

  ```
  "t4.c", line 4: error(1020):identifier "x" is undefined
  "t4.c", line 5: warning(1551):variable "y" is used before its value
   is set
  ```

  Messages can also be issued by other internal compiler passes.

- *Classes of diagnostic messages, their return codes and control over them*

  Three classes of messages exist: warning, error, and remark. Warning messages include the notation "warning" (which can be capitalized), and allow the compilation to continue (return code 0). Error messages cause the compilation to fail (return code 1).

  Remark messages appear in 64–bit mode only. Typically, remarks are issued only if the **-fullwarn** option appears on the command line. More control is available with the **-diag_warning**, **-diag_remark**, and **-diag_error** options. (See the *cc* reference page for more information.)

  Warning messages from the compiler front end have a unique diagnostic number. You can suppress these messages individually by putting the number in the numberlist of a **-woff***numberlist* switch to *cc*(1). *numberlist* is a comma–separated list of warning numbers and ranges of warning numbers. For example, to suppress the warning message in the previous example, type:

  ```
  -woff 1551
  ```

  To suppress warning messages numbered 1642, 1643, 1644, and 1759, type:

  ```
  -woff 1642-1644,1759
  ```

Environment (F.3.2)

- *Support of freestanding environments.*

  No support is provided for a freestanding environment.

- *The semantics of the arguments to main (2.1.2.2.1).*

  main is defined to have the two required parameters **argc** and **argv**. A third parameter, **envp**, is provided as an extension. That is, **main** would have the equivalent of the prototype *int main(int argc, char *argv[], char *envp[]*). The parameters have the following semantics:

  – **argc** is the number of arguments on the command line.

  – **argv**[0..**argc**–1] are pointers to the command–line arguments (strings).

  – **argv**[0] is the program name, as it appeared on the command line.

  – **argv[argc]** is a null pointer.

  – **envp** is an array of pointers to strings of the form *NAME=value*, where *NAME* is the name of an environment variable and *value* is its value. The array is terminated by a null pointer.

- *What constitutes an interactive device (2.1.2.3).*

  Asynchronous terminals, including windows, are interactive devices and are, by default, line buffered. In addition, the standard error device, **stderr**, is unbuffered by default.

Identifiers (F.3.3)

- *The number of significant initial characters (beyond 31) in an identifier without external linkage (3.1.2).*

  All characters are significant.

- *The number of significant initial characters (beyond 6) in an identifier with external linkage (3.1.2).*

  All characters are significant.

- *Whether case distinctions are significant in an identifier with external linkage (3.1.2).*

  Case distinctions are always significant.

Characters (F.3.4)

- *The members of the source and execution character sets, except as explicitly specified in the standard (2.2.1).*

  Only the mandated characters are present. The source character set includes all printable ASCII characters, hexadecimal 0x20 through 0x7e, and 0x7 through 0xc (the standard escape sequences).

- *The values to which the standard escape sequences are translated (2.2.2).*

  The escape sequences are translated as specified for standard ASCII: \a = 0x7, \b = 0x8, \f = 0xc, \n =

0xa, \r = 0xd, \t = 0x9, \v=0xb

- *The shift states used for the encoding of multibyte characters (2.2.1.2)*

  The multibyte character set is identical to the source and execution character sets. There are no shift states.

- *The number of bits in a character in the execution character set (2.2.4.2.1).*

  There are eight bits per character.

- *The mapping of members of the source character set (in character constants and string literals) to members of the execution character set (3.1.3.4).*

  The mapping is the identity mapping.

- *The value of an integer character constant that contains a character or escape sequence not represented in the basic execution character set or in the extended character set for a wide character constant (3.1.3.4).*

  With the exception of newline (0xa), backslash ('\'), and 0xff (end−of−file), eight−bit values appearing in an integer character constant are placed in the resultant integer in the same fashion as are characters which are members of the execution character set (see below). A backslash, newline, or 0xff can be placed in a character constant by preceding it with a backslash (that is, "escaping" it).

- *The value of an integer character constant that contains more than one character or a wide character constant that contains more than one multibyte character (3.1.3.4).*

  You can assign up to four characters to an **int** using a character constant. The encoding of multiple characters in an integer consists of the assignment of the corresponding character values of the*n* characters in the constant to the least−significant*n* bytes of the integer, filling any unused bytes with zeros. The most significant byte assigned contains the value of the lexically first character in the constant. For example:

  ```
  int t = 'a'; /* integer value 0x61 */
  int t2 = 'ab'; /* integer value 0x6162 */
  int t4 = 'abcd'; /* integer value 0x61626364 */
  int t4 = 'abcde'; /* error: too many characters for */
   /* character constant */
  ```

  Since the multibyte character set is identical to the source and execution character sets, the above discussion applies to the assignment of more than one multibyte character to a wide character constant.

- *The current locale used to convert multibyte characters into corresponding wide character (codes) for a wide character constant (3.1.3.4).*

  The mapping is the identity mapping to the standard ASCII character set. The C locale is used.

- *Whether a "plain" char has the same range of values as signed char or unsigned char.*

  Plain char is the same as unsigned char by default. Use the -signed option to *cc* to switch the range to be that of **signed char.**

Integers (F.3.5)

- *The representations and sets of values of the various types of integers (3.1.2.5).*

  Integers are two's complement binary. Table A−1lists the sizes and ranges of the various types of integer. The use of **long long** results in a warning in **-ansi** and **-ansiposix** modes.
  In the 32−bit implementation, to take full advantage of the support for 64 bits integral values in **-ansi** and **-ansiposix** modes, you can define the macro **_LONGLONG** on the *cc*(1) command line when using the types **__uint64_t**, **__int64_t**, or library routines that are prototyped in terms of these types.

Table A−1 Integer Types and Ranges

type	range: low	high	size (bits)
signed char	-128	127	8
char, unsigned char	0	255	8
short, signed short	-32768	32767	16
unsigned short int	0	65535	16
int, signed int	-2147483648	2147483647	32
unsigned int	0	4294967295	32
long, signed long int	-2147483648 (**-32** mode)	2147483647 (**-32** mode)	32
	-9223372036854775808 (**-64** mode)	9223372036854775807 (**-64** mode)	64
unsigned long int	0	4294967295 (**-32** mode)	32
		1844674407370955161 5 (**-64** mode)	64
long long signed long long int	-9223372036854775808	9223372036854775807	64
unsigned long long int	0	1844674407370955161 5	64

- *The result of converting an integer to a shorter signed integer, or the result of converting an unsigned integer to a signed integer of equal length, if the value cannot be represented (3.2.1.2).*

  The least significant n bits (n being the length of the result integer) of the source are copied to the result.

- *The results of bitwise operations on signed integers (3.3).*

  With the exception of right−shift of a negative signed integer (defined below), operations on signed and unsigned integers produce the same bitwise results.

- *The sign of the remainder on integer division (3.3.5)*

  The sign of the remainder is that of the numerator.

- *The result of a right shift of a negative−valued signed integral type (3.3.7).*

  The sign bit is propagated, so the result value is still negative.

Floating Point (F.3.6)

- *The representations and sets of values of the various types of floating−point numbers (3.1.2.5).*

  The representation is IEEE:

  – single (for **float** values)

  – double (for **double** values and for **long double** values in 32−bit mode)

  – quad precision (for **longdouble** values in 64−bit mode).

  See ANSI/IEEE Standard 754−1985 and IEEE Standard for Binary Floating−Point ArithmeticTable A−2lists ranges of floating−point types.

Table A−2 Ranges of Floating−Point Types

type	range: min	max	size (bits)
float	1.1755e−38	3.4028e+38	32
double	2.225e−308	1.7977e+308	64
long double	2.225e−308	1.7977e+308	128 (**-64** mode)

- *The type of rounding or truncation used when representing a floating−point constant which is within its range.*

  Per IEEE, the rounding is round−to−nearest (IEEE Standard 754, sections 4.1 and 5.5). If the two values are equally near, then the one with the least significant bit zero is chosen.

- *The direction of truncation when an integral number is converted to a floating−point number that cannot exactly represent the original value(3.2.1.3).*

  Conversion of an integral type to a float type, if the integral value is too large to be exactly represented, gives the next higher value.

- *The direction of truncation or rounding when a floating−point number is converted to a narrower floating−point number.*

  Per IEEE, the rounding is round−to−nearest (IEEE Standard 754, Section 4.1 and 5.5). If the two values are equally near, then the one with the least significant bit zero is chosen.

Arrays and Pointers (F.3.7)

- *The type of integer required to hold the maximum size of an array— that is, the type of the sizeof operator, size_t (3.3.3.4,4.1.1).*

  An **unsigned long** holds the maximum array size.

- *The size of integer required for a pointer to be converted to an integer type (3.3.4).*

  long ints are large enough to hold pointers in **-32** mode. Both are 32 bits wide.

  long ints are large enough to hold pointers in **-64** mode. Both are 64 bits wide.

- *The result of casting a pointer to an integer or vice versa (3.3.4).*

  The result is bitwise exact provided the integer type is large enough to hold a pointer.

- *The type of integer required to hold the difference between two pointers to elements of the same array, ptrdiff_t (3.3.6, 4.1.1).*

  An **int** is large enough to hold the difference between two pointers to elements of the same array in **-32** mode.

  A **longint** is large enough to hold the difference between two pointers to elements of the same array in both **-32** and **-64** modes.

Registers (F.3.8)

- *The extent to which objects can actually be placed in registers by use of the register storage−class specifier (3.5.1).*

  The compilation system can use up to eight of the register storage−class specifiers for nonoptimized code in **-32** mode, and it ignores register specifiers for formal parameters. Use of register specifiers is not recommended.

  The **register** storage−class specifier is always ignored and the compilation system makes its own decision about what should be in registers for optimized code (**-O2** and above).

Structures, Unions, Enumerations, and Bitfields (F.3.9)

- *What is the result if a member of a union object is accessed using a member of a different type (3.3.2.3).*

  The bits of the accessed member are interpreted according to the type used to access the member. For integral types, the N bits of the type are simply accessed. For floating types, the access might cause a trap if the bits are not a legal floating−point value. For pointer types, the 32 bits (64 bits if in **-64** mode) of the pointer are picked up. The usability of the pointer depends on whether it points to a valid object or function, and whether it is used appropriately. For example, a pointer whose least−significant bit is set can point to a character, but not to an integer.

- *The padding and alignment of members of structures (3.5.2.1).*

  This should present no problem unless binary data written by one implementation are read by another.

  Members of structures are on the same boundaries as the base data type alignments anywhere else. A word is 32 bits and is aligned on an address, which is a multiple of 4. Unsigned and signed versions of a basic type use identical alignment. Type alignments are given in Table A−3

Table A−3 Alignment of Structure Members

type	alignment
long double	double− word boundary (**-32** mode) quad−word boundary (**-64** mode)

double	double−word boundary
float	word boundary
long long	double−word boundary
long	word boundary (**-32** mode) double−word boundary (**-64** mode)
int	word boundary
pointer	word boundary
short	half−word boundary
char	byte boundary

- *Whether a "plain" int bit−field is treated as a signed int bit−field or as an unsigned int bit−field (3.5.2.1).*

  A "plain" int bit−field is treated as a **signed int** bit−field.

- *The order of allocation of bitfields within a unit (3.5.2.1).*

  Bits in a bitfield are allocated with the most−significant bit first within a unit.

- *Whether a bitfield can straddle a storage−unit boundary (3.5.2.1).*

  Bitfields cannot straddle storage unit boundaries (relative to the beginning of the struct or union), where a storage unit can be of size 8, 16, 32, or 64 bits.

- *The integer type chosen to represent the values of an enumeration type (3.5.2.2).*

  The int type is always used. Note that **long** or **long long** enumerations are not supported.

Qualifiers (F.3.10)

- *What constitutes an access to an object that has volatile−qualified type (3.5.3).*

  Objects of **volatile**−qualified type are accessed only as specified by the abstract semantics, and as would be expected on a RISC architecture. No complex instructions exist (for example, read−modify−write). **volatile** objects appearing on the left side of an assignment expression are accessed once for the write. If the assignment is not simple, an additional read access is performed. **volatile** objects appearing in other contexts are accessed once per instance. Incrementation and decrementation require both a read and a write access.

  volatile objects that are memory−mapped are accessed only as specified: if such an object is of size **char**, for example, adjacent bytes are not accessed. If the object is a bitfield, a read may access the entire storage unit containing the field. A write of an unaligned field necessitates a read and write of the storage unit that contains it.

Declarators (F.3.11)

- *The maximum number of declarators that can modify an arithmetic, structure, or union type (3.5.4).*

  There is no limit.

Statements (F.3.12)

- *The maximum number of case values in a switch statement (3.6.4.2).*

  There is no limit.

Preprocessing Directives (F.3.13)

- *Whether the value of a single−character character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set. Whether such a character constant can have a negative value (3.8.1).*

  The preprocessing and execution phases use exactly the same meanings for character constants.

  A single−character character constant is always positive.

- *The method for locating includable source files (3.8.2).*

  For file names surrounded by <>, the includable source files are searched for in /usr/include.

  The default search list includes /usr/include. You can change this list with various compiler options. See cc(1), the **-I**, and **-nostdinc** options.

- *The support of quoted names for includable source files (3.8.2).*

  Quoted names are supported for includable source files. For file names surrounded by "", the includable source files are searched for in the directory of the current include file, then in*/usr/include* .

  The default search list includes /usr/include. You can change this list with various compiler options. See cc(1), the **-I**, and **-nostdinc** options.

- *The mapping of source file character sequences (3.8.2).*

  The mapping is the identity mapping.

- *The behavior on each recognized #pragma directive.*

  #pragma weak *weak_symbol* **=** *strong_symbol*

  The *weak_symbol* is an alias that denotes the same function or data object denoted by the *strong_symbol*, unless a defining declaration for the *weak_symbol* is encountered at static link time. If encountered, the defining declaration preempts the weak denotation.

  You must define the *strong_symbol* within the same compilation unit in which the pragma occurs. You should also declare the *weak_symbol* with **extern** linkage in the same compilation unit. The **extern** declaration of the weak symbol is not required, unless the symbol is referenced within the compilation unit, but Silicon Graphics recommends it for type−checking purposes. The weak and strong symbols must be declared with compatible types. When the strong symbol is a data object, its declaration must be initialized.

  Weak **extern** declarations are typically used to export non−ANSI C symbols from a library without polluting the ANSI C name−space. As an example, *libc* may export a weak symbol *read()*, which aliases a strong symbol *_read()*, where *_read()* is used in the implementation of the exported symbol *fread()*. You can either use the exported (weak) version of *read()*, or define your own version of *read()* thereby preempting the weak denotation of this symbol. This will not alter the definition of *fread()*, since it only depends on the (strong) symbol *_read()*, which is outside the ANSI C name−space.

#pragma weak *weak_symbol*

The **pragma weak** *weak_symbol* tells the link editor not to complain if it does not find a defining declaration of the *weak_symbol*. References to the symbol use the appropriate *lvalue* if the symbol is defined; otherwise, it uses memory location zero (0).

#pragmaonce

This pragma has no effect in **-32** mode, but will ensure idempotent *include* files in **-64** mode (i.e. that an *include* file is included at most once in one compilation unit). Silicon Graphics recommends enclosing the contents of an include file *afile.h* with an **#ifdef** directive similar to:

```
#ifndef afile_INCLUDED
#define afile_INCLUDED
<contents of afile.h>
#endif
```

#pragma pack(*n*)

This pragma controls the layout of structure offsets, such that the strictest alignment for any structure member will be *n* bytes, where *n* is 0, 1, 2, 4, 8, or 16. When *n* is 0, the compiler returns to default alignment for any subsequent *struct* definitions.

A *struct* type defined in the scope of a **#pragma pack**(*n*) has at most an alignment of *n* bytes, and the packed characteristics of the type apply wherever the type is used, even outside the scope of the pragma in which the type was declared. The scope of a **#pragma pack** ends with the next **#pragma pack**, hence this pragma does not nest: There is no way to "return" from one instance of the pragma to a lexically earlier instance of the pragma.

A structure declaration must be subjected to identical instances of a **#pragma pack** in all files, or else misaligned memory accesses and erroneous struct member dereferencing may ensue.

Silicon Graphics strongly discourages the use of **#pragma pack**, since it is a nonportable feature and the semantics of this pragma may change in future compiler releases. Note that references to fields in *#packed structs* may be less efficient than references to fields in unpacked structs.

#pragma intrinsic(*a_function*)

This pragma allows certain preselected functions from *math.h*, *stdio.h*, and *string.h* to be inlined at a call−site for execution efficiency. The **#pragma intrinsic** has no effect on functions other than the preselected ones. Exactly which functions may be inlined, how they are inlined, and under what circumstances inlining occurs is implementation defined and may vary from one release of the compilers to the next. The inlining of intrinsics may violate some aspect of the ANSI C standard (e.g., the *errno* setting for *math.h* functions). All intrinsics are activated through pragmas in the respective standard header files and only when the preprocessor symbol **__INLINE_INTRINSICS** is defined and the appropriate include files are included. __INLINE_INTRINSICS is predefined by default only in **-cckr** and **-xansi** mode.

#pragmahdrstop

If *-pch* is on, **#pragma hdrstop** indicates the point at which the precompiled header mechanism snapshots the headers. If *-pch* is off, **#pragma hdrstop** is ignored. See the *Compiling and Performance Tuning Guide* for details on the precompiled header mechanism.

The MIPSpro compilers also silently recognize many commonly used pragmas; however, they have no effect. Some of these include:

– **#pragma no side effects**(*a_function*)
  Tells the compiler that a call to a function of the given name does not cause any modifications to objects accessible outside the function body. Such information can be useful for optimization and parallelization purposes.

– **#pragma***ident version*
  Adds a *.comment* section in the object file and puts the revision string inside it.

– **#pragma***int_to_unsigned identifier*
  Identifies *identifier* as a function whose type was **int** in a previous releases of the compilation system, but whose type is **unsigned int** in the MIPSpro compiler release. The declaration of the identifier must precede the pragma:

```
unsigned int strlen(const char*);
#pragma int_to_unsigned strlen
```

  This declaration makes it possible for the compiler to identify where the changed type may affect the evaluation of expressions.

Other **#pragmas** are used for C multiprocessing. They are described in the *Power C User's Guide.*

- *The definitions for* ***__DATE__*** *and* ***__TIME__*** *when, respectively, the date and time of translation are not available.*

  The date and time of translation are always available in this implementation.

- *What is the maximum nesting depth of include files (3.8.2).*

  The maximum nesting of include files is 200.

Library Functions (F.3.14)

- *The null pointer constant to which the macro NULL expands (4.1.5)*

  The NULL pointer constant expands to an **int** with value zero. That is,

```
#define NULL 0
```

- *The diagnostic printed by and the termination behavior of the assert function (4.2).*

If an assertion given by assert(EX) fails, the following message is printed on stderr using a _write to its underlying fileno.

```
Assertion failed: EX, file <filename>, line <linenumber>
```

This is followed by a call to *abort*(3c) (which exits with a SIGABRT).

- *The sets of characters tested for by the isalnum, isalpha, iscntrl, islower, isprint, andisupper functions (4.3.1).*

  The following is true when operating in the C locale. The C locale is in effect at program startup for programs compiled for pure ANSI C (that is, **-ansi**), or by invoking **setlocale(LC_ALL,"C")**. The C locale can be overridden at startup for any program that does not explicitly invoke **setlocale** by setting the value of the environment variable **CHRCLASS**. (See the man page *ctype*(3C).)

  – *isalnum* is nonzero for the 26 letters a-z and the 26 letters A-Z and the digits 0-9.

  – *isalpha* is nonzero for the 26 letters a-z and the 26 letters A-Z.

  – *islower* is nonzero for the 26 letters a-z.

  – *isupper* is nonzero for the 26 letters A-Z.

  – *isprint* is nonzero for the ASCII characters space through tilde (~) (0x20 through 0x7e).

  – *iscntrl* is nonzero for the ASCII characters NUL through US (0x0 through 0x1f).

- *The values returned by the mathematics functions on domain errors (4.5.1).*

  The value returned by the math functions on domain errors is the default IEEE Quiet NaN in all cases except the following:

  – The functions *pow* and *powf* return **-HUGE_VAL** when the first argument is zero and the second argument is negative. When both arguments are zero, *pow* and *powf* return 1.0.

  – The functions *atan2* and *atan2f* return zero when both arguments are zero.

- *Whether mathematics functions set the integer expression errno to the value of the macro ERANGE on underflow range errors (4.5.1).*

  Yes, except intrinsic functions that have been inlined. Note that *fabs*, *fabs*f, *sqrt*, *sqrtf*, *hypotf*, *fhypot*, *pow*, and *powf* are intrinsic by default in **-xansi** and **-cckr** modes and can be made intrinsic in **-ansi** mode by using the compiler option D__INLINE_INTRINSICS.

- *Whether a domain error occurs or zero is returned when the fmod function has a second argument of zero (4.5.6.4).*

  fmod(x,0) gives a domain error and returns the default IEEE Quiet NaN.

Signals

- *The set of signals for the signal function (4.7.1.1).*

  The *signal set* is listed in Table A–4 which is from the *signal*(2) man page. The set of signals conforms to the SVR4 ABI. Note that some of the signals are not defined in **-ansiposix** mode. References in square brackets beside the signal numbers are described under ‘"Signal Notes" in the discussion of signal semantics.

Table A–4 Signals

Signal	Number[Note]	Meaning
SIGHUP	01	hangup
SIGINT	02	interrupt
SIGQUIT	03[1]	quit
SIGILL	04[1]	illegal instruction (not reset when caught)
SIGTRAP	05[1][5]	race trap (not reset when caught)
SIGIOT	06	IOT instruction
SIGABRT	06[1]	abort
SIGEMT	07[1][4]	MT instruction
SIGFPE	08[1]	floating point exception
SIGKILL	09	kill (cannot be caught or ignored)
SIGBUS	10[1]	bus error
SIGSEGV	11[1]	segmentation violation
SIGSYS	12[1]	bad argument to system call
SIGPIPE	13	write on a pipe with no one to read it
SIGALRM	14	alarm clock
SIGTERM	15	software termination signal
SIGUSR1	16	user–defined signal 1
SIGUSR2	17	user–defined signal 2
SIGCLD	18[2]	termination of a child process
SIGGHLD	18	4.3 BSD/POSIX name
SIGPWR	19[2]	power fail (not reset when caught)
SIGWINCH	20[2]	window size changes
SIGURG	21[2]	urgent condition on I/O channel
SIGIO	22[2]	input/output possible
SIGPOLL	22[3]	selectable event pending
SIGSTOP	23[6]	stop (cannot be caught or ignored)
SIGTSTP	24[6]	stop signal generated from keyboard
SIGCONT	25[6]	continue after stop (cannot be ignored)
SIGTTIN	26[6]	background read from control terminal
SIGTTOU	27[6]	background write to control terminal
SIGVTALRM	28	virtual time alarm
SIGPROF	29	profiling alarm
SIGXCPU	30	cpu time limit exceeded [see *setrlimit* (2)]
SIGXFSZ	31	file size limit exceeded [see setrlimit(2)]
SIG32	32	reserved for kernel usage

- *The semantics for each signal recognized by the signal function (4.7.1.1).*

In the *signal* invocation `signal(sig, func)`, *func* can be the address of a signal handler, *handler*, or one of the two constant values (defined in <*sys/signal.h*>) SIG_DFL or SIG_IGN. The semantics of these values are:

SIG_DFL — terminate process upon receipt of signal **sig**
(This is the default if no call to *signal* for signal **sig** occurs.) Upon receipt of the signal **sig**, the receiving process is to be terminated with all of the consequences outlined in exit(2). See note 1 under "Signal Notes".

SIG_IGN — ignore signal
The signal **sig** is to be ignored.

handler — catch signal
func is the address of function *handler*.

Note: The signals SIGKILL, SIGSTOP, and SIGCONT cannot be ignored.

If **func** is the address of **handler**, upon receipt of the signal **sig**, the receiving process is to invoke **handler** as follows:

```
handler (int sig, int code, struct sigcontext *sc);
```

The remaining arguments are supplied as extensions and are optional. The value of the second argument **code** is meaningful only in the cases shown in Table A−5

Table A−5 Valid Codes in a Signal−Catching Function

Condition	Signal	Code
User breakpoint	SIGTRAP	BRK_USERBP
User breakpoint	SIGTRAP	BRK_SSTEPBP
Integer overflow	SIGTRAP	BRK_OVERFLOW
Divide by zero	SIGTRAP	BRK_DIVZERO
Multiply overflow	SIGTRAP	BRK_MULOVF
Invalid virtual address	SIGSEGV	EFAULT
Read−only address	SIGSEGV	EACCESS
Read beyond mapped object	SIGSEGV	ENXIO

The third argument, *sc*, is a pointer to a **struct sigcontext** (defined in <sys/signal.h>) that contains the processor context at the time of the signal. Upon return from *handler*, the receiving process resumes execution at the point that it was interrupted.

Before entering the signal−catching function, the value of *func* for the caught signal is set to SIG_DFL, unless the signal is SIGILL, SIGTRAP, or SIGPWR. This means that before exiting the handler, a call to *signal* is necessary to catch future signals.

Suppose a signal that is to be caught occurs during:

- a *read*(2), a *write*(2), an *open*(2)
- an *ioctl*(2) system call on a slow device (like a terminal; but not a file)

- a *pause*(2) system call
- a *wait*(2) system call that does not return immediately due to the existence of a previously stopped or zombie process

The signal catching function is executed and then the interrupted system call returns a -1 to the calling process with **errno** set to EINTR.

Note: The signals SIGKILL and SIGSTOP cannot be caught.

Signal Notes

1. If SIG_DFL is assigned for SIGQUIT, SIGILL, SIGTRAP, SIGABRT, SIGEMT, SIGFPE, SIGBUS, SIGSEGV, or SIGSYS, in addition to the process being terminated, a "core image" is constructed in the current working directory of the process, if the following conditions are met:

   The effective user ID and the real user ID of the receiving process are equal. An ordinary file named *core* exists and is writable or can be created. If the file must be created, it has the following properties:

   - a mode of 0666 modified by the file creation mask [see umask(2)]
   - a file owner ID that is the same as the effective user ID of the receiving process
   - a file group ID that is the same as the effective group ID of the receiving process

   Note: The core file can be truncated if the resultant file size would exceed either ulimit [see ulimit(2)] or the process's maximum core file size [see setrlimit(2)].

2. For the signals SIGCLD, SIGWINCH, SIGPWR, SIGURG, and SIGIO, the actions associated with each of the three possible values for **func** are:

SIG_DFL	ignore signal The signal is to be ignored.
SIG_IGN	ignore signal The signal is to be ignored. Also, if **sig** is SIGCLD, the calling process's child processes do not create zombie processes when they terminate [see exit(2)].
handler	catch signal If the signal is SIGPWR, SIGURG, SIGIO, or SIGWINCH, the action to be taken is the same as that described above when func is the address of a function. The same is true if the signal is SIGCLD with one exception: while the process is executing the signal−catching function, all terminating child processes are queued. The *wait* system call removes the first entry of the queue. If the *signal* system call is used to catch SIGCLD, the signal handler must be reattached when exiting the handler, and at that time—if the queue is not empty—SIGCLD is raised again before *signal* returns. See wait(2).

In addition, SIGCLD affects the *wait* and *exit* system calls as follows:

wait	If the handler parameter of SIGCLD is set to SIG_IGN and a *wait* is executed, the *wait* blocks until all of the calling process's child processes terminate; it then

	returns a value of −1 with *errno* set to ECHILD.
exit	If in the exiting process's parent process the handler parameter of SIGCLD is set to SIG_IGN, the exiting process does not create a zombie process.

When processing a pipeline, the shell makes the last process in the pipeline the parent of the preceding processes. A process that can be piped into in this manner (and thus become the parent of other processes) should take care not to set SIGCLD to be caught.

3. SIGPOLL is issued when a file descriptor corresponding to a STREAMS [see *intro*(2)] file has a "selectable" event pending. A process must specifically request that this signal be sent using the I_SETSIG ioctl call. Otherwise, the process never receives SIGPOLL.

4. SIGEMT is never generated on an IRIS 4D system.

5. SIGTRAP is generated for breakpoint instructions, overflows, divide by zeros, range errors, and multiply overflows. The second argument code gives specific details of the cause of the signal. Possible values are described in <*sys/signal.h*>.

6. The signals SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOU, and SIGCONT are used by command interpreters like the C shell [see *csh*(1)] to provide job control. The first four signals listed stop the receiving process unless the signal is caught or ignored. SIGCONT resumes a stopped process. SIGTSTP is sent from the terminal driver in response to the SWTCH character being entered from the keyboard [see *termio*(7)]. SIGTTIN is sent from the terminal driver when a background process attempts to read from its controlling terminal. If SIGTTIN is ignored by the process, then the read returns EIO. SIGTTOU is sent from the terminal driver when a background process attempts to write to its controlling terminal when the terminal is in TOSTOP mode. If SIGTTOU is ignored by the process, then the write succeeds, regardless of the state of the controlling terminal.

Signal does not catch an invalid function argument, **func**, and results are undefined when an attempt is made to execute the function at the bad address.

SIGKILL immediately terminates a process, regardless of its state.

Processes stopped via job control (typically **<Ctrl>−Z**) do not act upon any delivered signals other than SIGKILL until the job is restarted. Processes blocked via a *blockproc*(2) system call unblock if they receive a signal that is fatal (that is, a non−job−control signal that they are not catching). These processes remained stopped, however, if the job they are a part of is stopped. Only upon restart do they die. Any non−fatal signals received by a blocked process do *not* cause the process to be unblocked. An unblockproc(2) or unblockprocall(2) system call is necessary.

If an instance of signal **sig** is pending when *signal(sig,func)* is executed, the pending signal is cancelled unless it is SIGKILL.

signal() fails if **sig** is an illegal signal number, including SIGKILL and SIGSTOP, or if an illegal operation is requested (such as ignoring SIGCONT, which is ignored by default). In these cases, *signal()* returns SIG_ERR and sets **errno** to EINVAL.

After a fork(2), the child inherits all handlers and signal masks. If any signals are pending for the parent, they are not inherited by the child.

The exec(2) routines reset all caught signals to the default action; ignored signals remain ignored; the blocked signal mask is unchanged and pending signals remain pending.

These man pages contain other relevant information: intro(2), blockproc(2), kill(2), pause(2), ptrace(2), sigaction(2), sigset(2), wait(2), setjmp(3C), sigvec(3B), and kill(1).

Diagnostics

Upon successful completion, *signal* returns the previous value of **func** for the specified signal **sig**. Otherwise, a value of SIG_ERR is returned and **errno** is set to indicate the error. SIG_ERR is defined in the header file <sys/signal.h>.

Caution: Signals raised by the instruction stream—SIGILL, SIGEMT, SIGBUS, SIGSEGV—will cause infinite loops if their handler returns, or the action is set to SIG_IGN. The POSIX signal routines (sigaction(2), sigpending(2), sigprocmask(2), sigsuspend(2), sigsetjmp(3)), and the 4.3BSD signal routines (sigvec(3B), signal(3B), sigblock(3B), sigpause(3B), sigsetmask(3B)) must *never* be used with signal(2) or sigset(2).

Before entering the signal−catching function, the value of **func** for the caught signal is set to SIG_DFL, unless the signal is SIGILL, SIGTRAP, or SIGPWR. This means that before exiting the handler, a *signal* call is necessary to again set the disposition to catch the signal.

Note that handlers installed by *signal* execute with no signals blocked, not even the one that invoked the handler.

- *The default handling and the handling at program startup for each signal recognized by the signal function (4.7.1.1).*

  Each signal is set to SIG_DFL at program startup.

- *If the equivalent of signal(sig, SIG_DFL); is not executed prior to the call of a signal handler, the blocking of the signal that is performed(4.7.1.1).*

  The equivalent of signal(sig, SIG_DFL); is executed prior to the call of a signal handler unless the signal is SIGILL, SIGTRAP, or SIGPWR. See the *signal*(3B) man page for information on the support for the BSD 4.3 signal facilities.

- *Whether the default handling is reset if the SIGILL signal is received by a handler specified to the signal function (4.7.1.1).*

  No.

Streams and Files

- *Whether the last line of a text stream requires a terminating newline character (4.9.2).*

  There is no requirement that the last line of a text stream have a terminating newline: the output is flushed when the program terminates, if not earlier (as a result of *fflush()* call). However, subsequent processes/programs reading the text stream or file might expect the newline to be present; it customarily is in IRIX text files.

- *Whether space characters that are written out to a text stream immediately before a newline* *character appear when read in (4.9.2).*

All text characters (including spaces before a newline character) written out to a text stream appear exactly as written when read back in.

- *The number of null characters that can be appended to data written to a binary stream (4.9.2).*

  The library never appends nulls to data written to a binary stream. Only the characters written by the application are written to the output stream, whether binary or text. Text and binary streams are identical: there is no distinction.

- *Whether the file position indicator of an append mode stream is initially positioned at the beginning or end of the file (4.9.2).*

  The file position indicator of an append stream is initially positioned at the end of the file.

- *Whether a write on a text stream causes the associated file to be truncated beyond that point (4.9.3).*

  A write on a text stream does not cause the associated file to be truncated.

- *The characteristics of file buffering (4.9.3).*

  Files are fully buffered, as described in paragraph 3, section 4.9.3, of ANSI X3.159−1989.

- *Whether a zero−length file actually exists (4.9.3).*

  Zero−length files exist, but have no data, so a read on such a file gets an immediate EOF.

- *The rules for composing valid file names (4.9.3).*

  Filenames consist of 1 to FILENAME_MAX characters. These characters can be selected from the set of all character values excluding \0 (null) and the ASCII code for / (slash).

  Note that it is generally unwise to use *, ?, [, or] as part of file names because of the special meaning attached to these characters by the shell (see **sh**(1)). Although permitted, the use of unprintable characters should be avoided.

- *Whether the same file can be opened multiple times (4.9.3).*

  A file can be open any number of times.

- *The effect of the remove function on an open file (4.9.4.1).*

  For local disk files, a remove removes a directory entry pointing to the file but has no effect on the file or the program with the file open. For files remotely mounted via NFS software, the effect is unpredictable (the file might be removed making further I/O impossible through open streams, or it might behave like a local disk file) and might depend on the version(s) of NFS involved.

- *The effect if a file with the new name exists prior to a call to the rename function (4.9.4.2).*

  If the new name exists, the file with that new name is removed (See rm(1)) before the rename is done.

- *The output for %p conversion in the fprintf function (4.9.6.1).*

  %p is treated the same as **%x**.

- *The input for %p conversion in the fscanf function (4.9.6.2).*

  %p is treated the same as **%x**.

- *The interpretation of a - character that is neither the first nor the last character in the scanlist for %[conversion in the fscanf function (4.9.6.2).*

  A - character that does not fit the pattern mentioned above is used as a shorthand for ranges of characters. For example, [xabcdefgh] and [xa–h] mean that characters a through h and the character x are in the range (called a scanset in 4.9.6.2).

Temporary Files

- *Whether a temporary file is removed if a program terminates abnormally (4.9.4.3).*

  Temporary files are removed if a program terminates abnormally.

errno and *perror*

- *The value to which the macro errno is set by the fgetpos or ftell function on failure (4.9.9.1, 4.9.9.4).*

  errno is set to EBADF (9) by the *fgetpos* or *ftell* function on failure.

- *The messages generated by the perror function (4.9.10.4).*

  The message generated is simply a string. The content of the message given for each legal value of **errno** is given in the list below, which is of the format *errno_value:message.*

  1: No permission match (**-32** mode)
  1: Not privileged (**-64** mode)

  2: No such file or directory

  3: No such process

  4: Interrupted system call

  5: I/O error

  6: No such device or address

  7: Arg list too long

  8: Exec format error

  9: Bad file number

  10: No child processes

  11: Resource temporarily unavailable

  12: Not enough space

  13: Permission denied

  14: Bad address

15: Block device required

16: Device or resource busy (**-32** mode)
16: Device busy (**-64** mode)

17: File exists

18: Cross−device link

19: No such device

20: Not a directory

21: Is a directory

22: Invalid argument

23: Too many open files in system (**-32** mode)
23: File table overflow (**-64** mode)

24: Too many open files in a process (**-32** mode)
24: Too many open files (**-64** mode)

25: Inappropriate IOCTL operation (**-32** mode)
25: Not a typewriter (**-64** mode)

26: Text file busy

27: File too large

28: No space left on device

29: Illegal seek

30: Read−only file system

31: Too many links

32: Broken pipe

33: Argument out of domain

34: Result too large

35: No message of desired type

36: Identifier removed

37: Channel number out of range

38: Level 2 not synchronized

39: Level 3 halted

40: Level 3 reset

41: Link number out of range

42: Protocol driver not attached

43: No CSI structure available

44: Level 2 halted

45: Deadlock situation detected/avoided

46: No record locks available

47: Error 47

48: Error 48

49: Error 49

50: Bad exchange descriptor

51: Bad request descriptor

52: Message tables full

53: Anode table overflow

54: Bad request code

55: Invalid slot

56: File locking deadlock

57: Bad font file format

58: Error 58

59: Error 59

60: Not a stream device

61: No data available

62: Timer expired

63: Out of stream resources

64: Machine is not on the network

65: Package not installed

66: Object is remote

67: Link has been severed

68: Advertise error

69: Srmount error

70: Communication error on send

71: Protocol error

72: Error 72

73: Error 73

74: Multihop attempted

75: Error 75

76: Error 76

77: Not a data message

78: Error 78 (**-32** mode)
78: File name too long (**-64** mode)

79: Error 79 (**-32** mode)
79: Value too large for defined data type (**-64** mode)

80: Name not unique on network

81: File descriptor in bad state

82: Remote address changed

83: Cannot access a needed shared library

84: Accessing a corrupted shared library

85: .lib section in a.out corrupted

86: Attempting to link in more shared libraries than system limit

87: Cannot exec a shared library directly

88: Invalid System Call (**-32** mode)
88: Illegal byte sequence (**-64** mode)

89: Error 89 (**-32** mode)
89: Operation not applicable

90: Error 90 (**-32** mode)
90: Too many symbolic links in path name traversal (**-64** mode)

91: Error 91 (**-32** mode)
91: Restartable system call (**-64** mode)

92: Error 92 (**-32** mode)
92: If pipe/FIFO, don't sleep in stream head (**-64** mode)

93: Error 93 (**-32** mode)
93: Directory not empty (**-64** mode)

94: Error 94 (**-32** mode)
94: Too many users (**-64** mode)

95: Error 95 (**-32** mode)
95: Socket operation on non−socket (**-64** mode)

96: Error 96 (**-32** mode)
96: Destination address required (**-64** mode)

97: Error 97 (**-32** mode)
97: Message too long (**-64** mode)

98: Error 98 (**-32** mode)
98: Protocol wrong type for socket (**-64** mode)

99: Error 99 (**-32** mode)
99: Option not supported by protocol (**-64** mode)

100: Error 100

101: Operation would block (**-32** mode)
101: Error 101 (**-64** mode)

102: Operation now in progress (**-32** mode)
102: Error 102 (**-64** mode)

103: Operation already in progress (**-32** mode)
103: Error 103 (**-64** mode)

104: Socket operation on non−socket (**-32** mode)
104: Error 104 (**-64** mode)

105: Destination address required (**-32** mode)
105: Error 105 (**-64** mode)

106: Message too long (**-32** mode)
106: Error 106 (**-64** mode)

107: Protocol wrong type for socket (**-32** mode)
107: Error 107 (**-64** mode)

108: Option not supported by protocol (**-32** mode)
108: Error 108 (**-64** mode)

109: Protocol not supported (**-32** mode)
109: Error 109 (**-64** mode)

110: Socket type not supported (**-32** mode)
110: Error 110 (**-64** mode)

111: Operation not supported on socket (**-32** mode)
111: Error 111 (**-64** mode)

112: Protocol family not supported (**-32** mode)
112: Error 112 (**-64** mode)

113: Address family not supported by protocol family (**-32** mode)
113: Error 113 (**-64** mode)

114: Address already in use (**-32** mode)
114: Error 114 (**-64** mode)

115: Can’t assign requested address (**-32** mode)
115: Error 115 (**-64** mode)

116: Network is down (**-32** mode)
116: Error 116 (**-64** mode)

117: Network is unreachable (**-32** mode)
117: Error 117 (**-64** mode)

118: Network dropped connection on reset (**-32** mode)
118: Error 118 (**-64** mode)

119: Software caused connection abort (**-32** mode)
119: Error 119 (**-64** mode)

120: Connection reset by peer (**-32** mode)
120: Protocol not supported (**-64** mode)

121: No buffer space available (**-32** mode)
121: Socket type not supported (**-64** mode)

122: Socket is already connected (**-32** mode)
122: Operation not supported on transport endpoint (**-64** mode)

123: Socket is not connected (**-32** mode)
123: Protocol family not supported (**-64** mode)

124: Can’t send after socket shutdown (**-32** mode)
124: Address family not supported by protocol family (**-64** mode)

125: Too many references: can’t splice (**-32** mode)
125: Address already in use (**-64** mode)

126: Connection timed out (**-32** mode)
126: Cannot assign requested address (**-64** mode)

127: Connection refused (**-32** mode)
127: Network is down (**-64** mode)

128: Host is down (**-32** mode)
128: Network is unreachable (**-64** mode)

129: Host is unreachable (**-32** mode)
129: Network dropped connection because of reset (**-64** mode)

130: Too many levels of symbolic links (**-32** mode)
130: Software caused connection abort (**-64** mode)

131: File name too long (**-32** mode)
131: Connection reset by peer (**-64** mode)

132: Directory not empty (**-32** mode)
132: No buffer space available (**-64** mode)

133: Disk quota exceeded (**-32** mode)
133: Transport endpoint is already connected (**-64** mode)

134: Stale NFS file handle (**-32** mode)
133: Transport endpoint is already connected (**-64** mode)

134: Transport endpoint is not connected (**-64** mode)

135: Structure needs cleaning (**-64** mode)

136: Error 136 (**-64** mode)

137: Not a name file (**-64** mode)

138: Not available (**-64** mode)

139: Is a name file (**-64** mode)

140: Remote I/O error (**-64** mode)

141: Reserved for future use (**-64** mode)

142: Error 142 (**-64** mode)

143: Cannot send after socket shutdown (**-64** mode)

144: Too many references: cannot splice (**-64** mode)

145: Connection timed out (**-64** mode)

146: Connection refused (**-64** mode)

147: Host is down (**-64** mode)

148: No route to host (**-64** mode)

149: Operation already in progress (**-64** mode)

150: Operation now in progress (**-64** mode)

151: Stale NFS file handle (**-64** mode)

See the perror(3C) man page for further information.

Memory Allocation

The behavior of the calloc, malloc, or realloc function if the size requested is zero (4.10.3).

The *malloc* in *libc.a* returns a pointer to a zero−length space if a size of zero is requested. Successive calls to *malloc* return different zero−length pointers. If the library*libmalloc.a* is used, *malloc* returns 0 (the NULL pointer).

The *abort* Function

The behavior of the abort function with regard to open and temporary files (4.10.4.1).

Open files are not flushed, but are closed. Temporary files are removed.

The *exit* Function

The status returned by the exit function if the value of the argument is other than zero, EXIT_SUCCESS or EXIT_FAILURE (4.10.4.3).

The status returned to the environment is the least significant eight bits of the value passed to *exit.*

The *getenv* Function

The set of environment names and the method for altering the environment list used by the **getenv** *function (4.10.4.4).*

Any string can be used as the name of an environment variable, and any string can be used for its value. The function *putenv* alters the environment list of the application. For example,

```
putenv("MYNAME=foo")
```

This sets the value of the environment variable **MYNAME** to "foo." If the environment variable **MYNAME** already existed, its value is changed. If it did not exist, it is added. The string passed to *putenv* actually becomes part of the environment, and changing it later alters the environment. Further, the string should not be space that was automatically allocated (for example, an **auto** array); rather, it should be space that is either global or *malloc*ed. For more information, see the *putenv*(3C) man page.

It is not wise to alter the value of well−known environment variables. For the current list, see the man page for *environ*(3c).

The *system* Function

The contents and mode of execution of the string passed to the system function (4.10.4.5).

The contents of the string should be a command string, as if typed to a normal IRIX shell, such as sh(1). A shell (*sh*(1)) is forked, and the string is passed to it. The current process waits until the shell has completed and returns the exit status of the shell as the return value.

The *strerror* Function

The contents of the error message strings returned by the strerror function (4.11.6.2).

The string is exactly the same as the string output by *perror*, which is documented in "errno and perror".

Timezones and the *clock* Function.

- *The local time zone and daylight saving time (4.12.1).*

  Local time and daylight saving time are determined by the value of the **TZ** environment variable. **TZ** is set by *init*(1) to the default value indicated in the file */etc/TIMEZONE*, and this value is inherited in the environment of all processes. If **TZ** is `unset`, the local time zone defaults to GMT (Greenwich mean time, or coordinated universal time), and daylight saving time is not in effect. See the man pages ctime(3C), time(2), timezone(4), environ(5), getenv(3), and other related man pages for the format of **TZ**.

- *The era for the clock function (4.12.2.1).*

clock counts seconds from 00:00:00: GMT, January 1, 1970. What was once known as Greenwich mean time (GMT) is now known as coordinated universal time, though the man pages do not reflect this change yet. See the ctime(3C) man page for further information.

Locale−Specific Behavior (F.4)

For information on locale−specific behavior, see the chapter titled "Internationalizing Your Application" in *Topics in IRIX Programming.* That chapter covers some locale−specific topics to consider when internationalizing an application. Topics include:

- Overview of Locale−Specific Behavior
- Native Language Support and the NLS Database
- Using Regular Expressions
- Cultural Data

Also, that chapter describes setting a locale, location of locale−specific data, cultural items to consider, and GUI concerns.

For additional information on locale−specific behavior, refer to the*X/Open Portability Guide*, *Volume 3*, "XSI *Supplementary Definitions,"* published by Prentice Hall, Englewood Cliffs, New Jersey 07632, ISBN 0−13−685−850−3.

Common Extensions (F.5)

The following extensions are widely used in many systems, but are not portable to all implementations. The inclusion of any extension that can cause a strictly conforming program to become invalid renders an implementation nonconforming. Examples of such extensions are new keywords, or library functions declared in standard headers or predefined macros with names that do not begin with an underscore. The Standard's description of each extension is followed by a definition of any Silicon Graphics support/nonsupport of each common extension.

Environment Arguments (F.5.1)

*In a hosted environment, the main function receives a third argument, char *envp[], that points to a null−terminated array of pointers to char. Each of these pointers points to a string that provides information about the environment for this execution of the process (2.1.2.1.1).*

This extension is supported.

Specialized Identifiers

Characters other than the underscore _, letters, and digits, that are not defined in the required source character set (such as dollar sign $, or characters in national character sets) can appear in an identifier.

If the -dollar option is given to *cc*, then the dollar sign ($) is allowed in identifiers.

Lengths and Cases of Identifiers

All characters in identifiers (with or without external linkage) are significant and case distinctions are observed (3.1.2).

All characters are significant. Case distinctions are observed.

Scopes of Identifiers (F.5.4)

A function identifier, or the identifier of an object (the declaration of which contains the keywordextern) has file scope.

This is true of the compiler when invoked with cc -cckr (that is, when requesting traditional C). When compiling in ANSI mode (by default or with one of the ANSI options) function identifiers (and all other identifiers) have block scope when declared at block level.

Writable String Literals (F.5.5)

String literals are modifiable. Identical string literals shall be distinct (3.1.4).

All string literals are distinct and writable when the **-use_readwrite_const** option is in effect. Otherwise, string literals may not be writable.

Other Arithmetic Types (F.5.6)

Other arithmetic types, such as long long int and their appropriate conversions, are defined (3.2.2.1).

Yes.

Function Pointer Casts (F.5.7)

A pointer to an object or to void can be cast to a pointer to a function, allowing data to be invoked as a function (3.3.4). A pointer to a function can be cast to a pointer to an object, or to void, allowing a function to be inspected or modified (for example, by a debugger) (3.3.4).

Function pointers can be cast to a pointer to an object, or to void, and vice versa.

Data can be invoked as a function.

Casting a pointer to a function to a pointer to an object or void does allow a function to be inspected. Normally, functions cannot be written to, since text space is read−only. Dynamically loaded functions are loaded (by a user program) into data space and can be written to.

Non−int Bit−Field Types (F.5.8)

Types other than int, unsigned int, and signed int can be declared as bitfields, with appropriate maximum widths (3.5.2.1).

A bitfield can be any integral type in **-xansi** and **-cckr** modes. However, bitfields of types other than **int**, **signed int**, and **unsigned int** result in a warning diagnostic in **-ansi** mode.

The fortran Keyword (F.5.9)

The fortran declaration specifier can be used in a function declaration to indicate that calls suitable for Fortran should be generated, or that different representations for external names are to be generated (3.5.4.3).

The fortran keyword is not supported in this ANSI C. With cc -cckr, that keyword is accepted but ignored.

The asm Keyword (F.5.10)

The asm keyword can be used to insert assembly language code directly into the translator output. The most common implementation is via statement of the form asm (character−string−literal)(3.6).

The asm keyword is not supported.

Multiple External Definitions (F.5.11)

There can be more than one external definition for the identifier of an object, with or without the explicit use of the keyword extern. If the definitions disagree, or more than one is initialized, the behavior is undefined (3.7.2).

With ANSI C, only one external definition of the object is permitted. If more than one is present, the linker (ld(1) gives a warning message. The Strict Ref/Def model is followed (ANSI C Rationale, 3.1.2.2, page 23).

With cc -cckr, the Relaxed Ref/Def model is followed (ANSI C Rationale, 3.1.2.2, page 23): multiple definitions of the same identifier of an object in different files are accepted and all but one of the definitions are treated (silently) as if they had the `extern` keyword.

If the definitions in different source units disagree, the mismatch is not currently detected by the linker (ld), and the resulting program will probably not work correctly.

Empty Macro Arguments (F.5.12)

A macro argument can consist of no preprocessing tokens (3.8.3).

This extension is supported. For example, one could define a macro such as

```
#define notokargs() macrovalue
```

Predefined Macro Names (F.5.13)

Macro names that do not begin with an underscore, describing the translation and execution environments, may be defined by the implementation before translation begins (3.8.8).

This is *not* true for **cc −ansi,** which defines ANSI C. Only macro names beginning with two underscores or a single underscore followed by a capital letter are predefined by the implementation before translation begins. The name space is not polluted.

With **cc -cckr** (traditional C), a C preprocessor is used with a full set of the predefined symbols. For example, **sgi** is predefined.

With **cc -xansi** (which is the default for cc), an ANSI C preprocessor and compiler are used and a full set of predefined symbols is defined (including **sgi**, for example).

Extra Arguments for Signal Handlers (F.5.14)

Handlers for specific signals can be called with extra arguments in addition to the signal number.

Silicon Graphics supports System V, POSIX, and BSD signal handlers. Extra arguments to the handler are available for your use. See the signal man page.

Additional Stream Types and File−Opening Modes (F.5.15)

Additional mappings from files to streams may be supported (4.9.2), and additional file−opening modes may be specified by characters appended to the mode argument of the fopen function (4.9.5.3).

There are no additional modes supported. There are no additional mappings. The UNIX approach is used, as mentioned in the ANSI C Rationale, Section 4.9.2, page 90.

Defined File Position Indicator (F.5.16)

The file position indicator is decremented by each successful call to the ungetc function for a text stream, except if its value was zero before a call (4.9.7.11).

Only the one character of pushback guaranteed by the standard is supported.