User's Guide to Symbolics C

Preface to the User's Guide to Symbolics C

0.0.1. Purpose and Scope of the User's Guide

The *User's Guide to Symbolics C* describes the Symbolics programming environment and run-time library.

Symbolics C is fully integrated with Genera. You must have a thorough understanding of Genera utilities and operations before you begin using Symbolics C. The Genera documentation set describes the editor, Command Processor, Debugger, and other utilities and features of the system.

If you are new to the Genera environment, read and use the *Genera Workbook* for an overview and tutorial introduction to the system.

0.0.2. Overview of Documentation

This guide is designed to be used with the online-only C reference manual, *C: A Reference Manual* by Samuel Harbison and Guy Steele, Jr. These two books make up the documentation for Symbolics C.

C: A Reference Manual can be found in Document Examiner in the Current Candidates pane under its title. Sections of the book are identified by the prefix *C-Ref:*

Introduction to Symbolics C

Overview of Symbolics C

0.0.3. Conforms to Draft ANSI C Standard

This implementation of Symbolics C conforms to the requirements of the draft ANSI standard. Future versions of Symbolics C will reflect the most recent ANSI standard.

Error messages refer to sections of the ANSI document.

Features of Symbolics C

The Symbolics C development environment includes these features:

- Incremental compilation from the editor
- A C editor mode that supports a template and completion facility
- A C-language source-level debugger

- A system generation facility
- Ability to build the C run-time system into applications
- A Dynamic C Listener window
- Ability to support the Metering Interface
- Online reference and user guide documentation

Installation Procedure for Symbolics C

0.0.4. Introduction

This section describes the installation procedure for Symbolics C, and provides instructions for installing and loading the Symbolics C system.

0.0.5. Procedure

Restoring the tape

Before restoring the tape, see the section "Restore Distribution Command". This fully describes the tape restoration procedure.

If you are using a machine with a tape drive, you can follow this procedure:

1. Restore the contents of the tape by placing the system distribution tape in the tape drive and typing this command at the Command processor:

   ```
   Restore Distribution
   ```

2. Press `RETURN` when you see this prompt:

   ```
   ** A tape is needed to read distribution.
    Enter a tape spec [default Local: Cart]:
   ```

After you restore the tape, create the directory, `sys:c;tmp;`

Loading the system

After the contents of the tape is loaded onto the sys host, load the C system by typing this command at the Command Processor:

```
Load System C :Query No
```

The machine informs you when the system loads successfully.

If you find that the system attempts to complete C to some other system, type:

```
Load System "C"
```

Introduction to Program Development

Introduction to Editing C Programs

This section illustrates some editing basics by taking you through the steps needed to create a simple C program.

For further information:

See the section "Zmacs Manual".
See the section "C Editor Commands".
See the section "Summary of Standard Editor Mode Commands for C".

0.0.6. Procedure

1. Select Zmacs by pressing `SELECT E`.

2. Use the Find File command, `c-X c-F`, to create a new buffer for the C program you are creating. To invoke the Find File command, hold down the `CTRL` key as you press first the `X` key, and then the `F` key.

3. Choose a name for the buffer. When prompted, type the name into the minibuffer at the bottom of the screen and press `RETURN`. Use the pathname conventions appropriate for the host operating system.

4. Example: In order to create a new C source file, quadratic.c, that you want to store in your home directory on a host machine called quabbin (q for short), type the following:

   ```
   c-X c-F q:>my-dir>quadratic.c
   ```

5. Make sure that the name includes the proper C file type (extension) for your host; for example, `quadratic.c` is the correct name for a file residing on LMFS, the Genera file system.

6. Use `m-X` C Mode to set the mode of the buffer to C. Press the `META` and `X` keys together. Then type `C mode` and press `RETURN`.

The mode line, located below the editor window, displays "C". Use m-X Update Attribute List to display the attribute list, which specifies the properties of the buffer, among them the mode and the package. The attribute list is the first line of a file and looks something like:

```
/*-*- Mode: C; Package: C-USER -*- */
```

7. Type in the program and check the code in your buffer against the example.

   For the editor commands that control cursor movement and text manipulation: see the section "Summary of Standard Editor Mode Commands for C".

8. If you want to save the source code in a disk file, use c-X c-S. If the file contains syntax errors, you are informed and asked whether you want to correct the errors or proceed with saving the file.

Compilation, Execution, and Debugging from the Editor

This section presents two simple examples of how to compile, execute, and debug C code from the editor. The first example shows how to compile and execute the quadratic function shown in figure 1. The second example asks you to edit that program to create a simple error and then takes you through the steps to compile, execute, and debug the quadratic function.

0.0.7. Procedure 1: Compiling and Executing a Function from the Editor

1. Compile the buffer with m-X Compile Buffer. M-X Compile Buffer performs two compilation steps: it compiles the preprocessor directives `#include<stdio.h>` and `#include <math.h>`, and it compiles the function `main quad`.

   The minibuffer at the bottom of the screen displays the stages of the compilation as they complete. If the compilation completes successfully, go to Step 3.

2. If the typeout window displays compiler errors or warnings, press the space bar to erase the typeout window and return to the editor window; correct the code or use m-X Edit Compiler Warnings to resolve the warnings.

3. Press SUSPEND to enter a Zmacs C mode breakpoint. A typeout window appears at the top of the screen, overlaying a portion of the editor window.

4. Type:

   ```
   Execute C Function main quad
   ```

5. The program prompts:

This program uses the quadratic formula to solve for x in a quadratic equation. It finds two roots. The root must be a real number value. Enter values based on the formula ax(2) + bx + c = 0.

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

main quad()
{
  int a, b, c, d;
  double drt, x1, x2;
  printf("\nSolves the equation ax(2) + bx + c = 0");
  printf("\nfor x.");
  printf("\nEnter values for a, b, and c: ");
  scanf("%d %d %d", &a, &b, &c);

  d=((b*b)-(4*(a*c)));
  if (d <= 0)
    {
      printf("This program cannot solve a quadratic formula whose");
      printf("\nresult is an imaginary number.");
    }
  else {
      drt = sqrt((double)d);
      x1=(-b+drt)/(2*a);
      x2=(-b-drt)/(2*a);

      printf("The value of root one is %f ", x1);
      printf("\nThe value of root two is %f ", x2);
    }
}
```

Figure 1. The main_quad Function

```
Solves the equation ax(2) + bx + c = 0;
for x.
Enter values for a, b, and c:
```

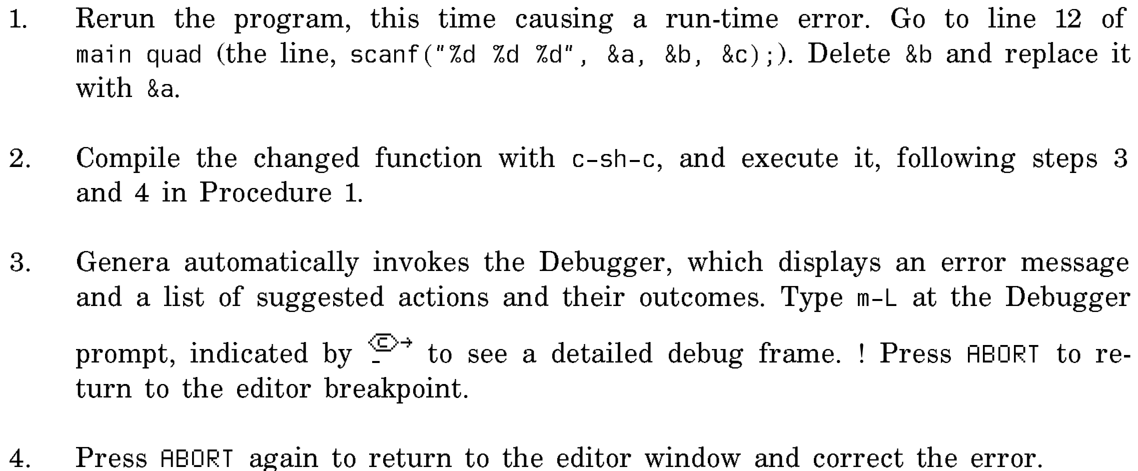
Type:

```
1 0 -9 <RETURN>
```

The window should display the following:

```
The value of root one is   3.000000
The value of root two is  -3.000000
```

0.0.8. Procedure 2: Compiling, Executing, and Debugging a Function from the Editor



1. Rerun the program, this time causing a run-time error. Go to line 12 of `main quad` (the line, `scanf("%d %d %d", &a, &b, &c);`). Delete `&b` and replace it with `&a`.

2. Compile the changed function with `c-sh-c`, and execute it, following steps 3 and 4 in Procedure 1.

3. Genera automatically invokes the Debugger, which displays an error message and a list of suggested actions and their outcomes. Type `m-L` at the Debugger prompt, indicated by → to see a detailed debug frame. ! Press `ABORT` to return to the editor breakpoint.

4. Press `ABORT` again to return to the editor window and correct the error.

This completes the edit-compile-debug cycle for the sample program. We suggest that you spend some time editing the code, recompiling, and rerunning the program until you feel comfortable with the process.

Naming C Functions

Naming conventions for Symbolics C functions differ slightly from the traditional C style. The difference in naming convention is due to the fact that in the Genera environment, functions must have unique names within a package. For example, you cannot have more than one function named *main* in C-User, the default C package. This differs from many C implementations, designed for file oriented systems, where the convention is to name the function where execution begins as *main*.

The following examples show a traditional-style C function *main* and two Genera-style solutions to the need for unique function names within packages.

Traditional C style:

```
int main(argc, argv)
```

Genera style: Using a descriptive name

```
int sort test main(argc, argv)
```

Genera style: Using the package system

1. Use the command Create C Package from the C Listener. For example:

   ```
   Create C Package sort-test
   ```

2. You may now define a function named *name* in a buffer or file with package attribute sort-test. For more information, see the section "Set Package Command". For example:

   ```
   int main(argc, argv)
   ```

```
Command: Execute C Function "main_quad"

Solves the equation ax(2) + bx + c = 0
for x.
Enter values for a, b, and c: 1 0 -9
Trap: The arguments given to the SYS:*-INTERNAL instruction, Undefined and Undefined, were neit
her one of them a number.

C Function main_quad (Package C-USER)
s-A, <RESUME>:   Supply replacement arguments
s-B:             Return a value from the *-INTERNAL instruction
s-C:             Retry the *-INTERNAL instruction
s-D, <ABORT>:    Terminate C execution of main_quad
s-E:             Return to Breakpoint ZMACS C MODE in Editor Typeout Window 1
s-F:             Editor Top Level
s-G:             Restart process Zmacs Windows
<c>→ Meta-L Show Frame :Detailed Yes :Clear Window Yes
 Frame has special variable bindings.
 Called to return, funcalled.

[main_quad]

 Locals:

 a       int       =            0
 b       int       =            <Undefined>
 c       int       =            -11
 d       int       =            <Undefined>
 drt     double    =            <Undefined:Undefined>
 x1      double    =            <Undefined:Undefined>
 x2      double    =            <Undefined:Undefined>

Source:

    int a, b, c, d;
    double drt, x1, x2;

    printf("\nSolves the equation ax(2) + bx + c = 0");
    printf("\nfor x.");
    printf("\nEnter values for a, b, and c: ");
    scanf("%d %d %d", &a, &a, &c);

⇒   d=((b*b)-(4*(a*c)));
    if (d <= 0)
```

Figure 2. Producing a run-time error in the sample program.

The routine `main` in the package `sort-test` is distinct from the routine `main` in `c-user`.

Using the Editor to Write C Programs

C Mode

0.0.9. Syntax-directed Editor

The C editor mode extension to Zmacs is based on a syntax-directed editor. The syntax-directed editor understands the syntax of C and makes use of this knowledge in providing language-specific commands and information while editing, for example, indicating the location of the next syntax error in the buffer.

The editing commands in C mode operate on C language units and on language tokens and expressions. This means, in effect, that the syntax-directed editor understands how to distinguish one unit from another.

In addition to the Zmacs textual model of editing, the syntax-directed editor provides the features of a structure or template editor as well. Unlike many structure editors, the syntax-directed editor does not restrict the size or the illegality of the contents of the buffer. However, the more syntactically correct a program, the more helpful the editor.

0.0.10. Relation to Zmacs

The syntax-directed editor provides the standard commands and capabilities of Zmacs that are applicable to C. For example, `c-N` moves the cursor to the next line in both Lisp mode and C mode as well as in text mode buffers.

One important difference is that Zmacs commands that operate on Lisp forms in a Lisp mode buffer operate on statements and larger language-specific constructs (like functions) in C modes. Separate commands operate on language expressions; others exhibit even more refined behavior, such as deleting a C language token or finding C syntax errors.

Where possible, the C editor mode commands are modeled on their analogous Lisp mode commands. For example, in Lisp mode `c-m-RUBOUT` deletes the previous Lisp form; in C mode the same command deletes the previous C language statement or definition.

0.0.11. Special C Commands in Zmacs C Mode

meta-x Kill Definition

Removes a C definition from the editor buffer and/or the current world. When you remove a definition from the current world, the command also offers to remove the definition from the editor buffer. If the source file is an include file, the command offers to remove the definition from all files containing the definition.

meta-x Resolve C Identifier

Presents the object associated with an identifier in the context of the current statement. You can obtain all possible resolutions of the identifier by supplying a numeric argument. Supplying a numeric argument places the object in the current statement into bold.

0.0.12. Online Documentation from the Editor

You can display documentation for the Symbolics C run-time functions and for C reserved words and template items. You can access documentation for template items by clicking the right mouse button when pointing the cursor at an item on a menu of template. Additionally, you can access documentation for functions and reserved words by pressing `m-sh-d` and pointing the cursor directly after a function.

C Fundamental Mode

C fundamental mode is a major editor mode, parallel to C mode. You can use a subset of C editor commands in this mode, including: m-X Compile Buffer, c-sh-C, and the directory search list commands also available in C mode. Use m-X C Fundamental Mode to set up this environment.

Since the C editor mode performs no macro expansion during parsing of a C buffer, you should use C Fundamental Mode for any C source in which C preprocessor macros hide the actual C syntax from the C editor mode.

For example, use C Fundamental Mode when you define macros of the form:

```
#define ROUTINE
#define IS
#define BEGIN {
#define END }
```

and their use ends up hiding a routine definition, such as:

```
ROUTINE main IS ()
BEGIN
   ...
END
```

Finding Syntax Errors

0.0.13. Introduction

Usually you become aware of syntax errors only when you try to compile a unit of source code. The syntax-directed editor, however, parses source code as you type it and keeps track of all syntax errors. However, the editor notifies you of such errors only in certain circumstances:

- You explicitly query about syntax errors using either c-sh-N or c-sh-P.
- Some compilation commands, for example, c-sh-C, notify you when syntax errors are encountered.
- Deletion commands like c-m-K and c-m-RUBOUT mark the region containing a syntax error in inverse video and query you about whether or not to proceed with the deletion.
- You press LINE after entering a line of source code.

0.0.14. c-sh-N and c-sh-P

- c-sh-N

  Finds the nearest syntax error to the right of the cursor, if any, and moves the cursor there. With a numeric argument, it finds the last syntax error in the buffer. c-sh-P

Finds the nearest syntax error to the left of the cursor and moves the cursor there. With a numeric argument, it finds the first syntax error in the buffer.

Sometimes a single error results in a cascade of error messages from `c-sh-N` or `c-sh-P`. In such cases, correct the errors starting with the first error.

0.0.15. LINE

In addition to indenting the current line correctly with respect to the line above it, `LINE` also detects syntax errors within that line, indicating in the minibuffer the point of error, as in:

```
j := k + ;
         ^
```

C Mode Completion and Templates

Introduction to Completion and Templates in C Mode

C editor mode provides a general completion facility over the set of C language constructs, as well as over the set of predeclared identifiers and reserved words. For example, as soon as you type enough characters in a C keyword or predefined identifier signalling a word as unique, you can ask for completion. The remaining characters of the identifier are inserted in the buffer. If you do not type enough letters to identify the word as unique, you can ask to see all possible completions to what you typed so far. The completion facility can also insert *templates*, showing the pattern of the syntactic constructs of C.

C Mode Templates

Positioning the cursor after a keyword and pressing `END` inserts the syntactic pattern (template) of the appropriate language structure into the buffer. The template consists of some combination of descriptive items and C keywords.

Where a single valid construct is possible, the `END` key inserts the template matching the keyword. Where multiple possibilities exist, `END` pops up a menu of template items for all the constructs valid in the current context.

0.0.16. Template Items

In addition to keywords, the template contains *template items*, which are syntactic constructs surrounded by horseshoes.

These template items contain constructs either required, optional and repeating, or repeating. The delimiters of the template item indicate the type of construct it describes, in accordance with extended Backus-Naur form.

Type of template item *Delimiter*

⊂{ optional repeating }⊃	braces
⊂[optional]⊃	square brackets
⊂ required ⊃	horseshoes only

0.0.17. Moving Among Template Items

You can move from one template item to another by pressing `c-m-N` to move to the beginning of the next item, or `c-m-P` to move to the beginning of the previous item.

0.0.18. Filling in a Template Item

A template item is just text, in the sense that you can write out or read in a file containing template items. The editor treats (parses) a template item as what it represents; for example, ⊂identifier⊃ acts as though an identifier is present. However, unlike regular text, the template item disappears when you begin to fill it in.

The `c-Help` Command

The `c-HELP` command displays a mouse-sensitive menu of all language constructs valid at the cursor position. The `c-HELP` mechanism understands the syntax of C and works by comparing its understanding with the relative position of the cursor in the buffer.

Because the `c-HELP` command relies heavily on language context, its behavior is altered by incompleteness or syntax errors in your program. In general, the more accurate and complete your program, the more accurate and useful the help information.

In general, the `c-HELP` command is useful for finding out which language constructs are valid at the cursor position.

Select a menu item by clicking the left, middle, or right button on the mouse. Click Left, Middle, or Right on [Exit] to return to editing.

Left	Inserts the selected template at the cursor position.
Middle	Displays the selected template in a temporary window. Move the mouse pointer off the window to make the template disappear.
Right	Displays documentation for the template item. Press `SELECT E` to return to the editor window.

Completing Reserved Words and Predeclared Identifiers

Use the `COMPLETE` command to complete reserved words and predeclared identifiers. Consider the following example: you begin typing a predeclared identifier and wish to take advantage of the completion facility. Positioning the cursor at the end

of your typein (as long as it is not inside a comment or string), and pressing the COMPLETE key compares what you typed with the set of all reserved words and predeclared identifiers. If your typein completes to a unique string, the completion facility inserts the remaining characters of the identifier and adjusts the face and case of the identifier as appropriate.

Electric C Mode

Electric C mode is an editor facility available in C-mode buffers. As you type, Electric C Mode places input into the appropriate font and case, depending on the syntactic context into which you insert the input. For example, by default, a word typed within a comment is rendered in an italic face. A reserved word used in a function, such as double is placed in lowercase and boldface.

By default, Electric C mode places comments in mixed-case italics. To change the defaults, use m-X Adjust Face and Case.

0.0.19. Using Electric C Mode

You can only use Electric mode when in C-mode buffers. If the buffer is created in such a way that the editing mode is not implicit (for example, via c-X B), set the buffer mode to C via m-X C Mode. Then turn on electric mode using m-X Electric C Mode. The mode line displays (C Electric Mode).

To turn off the mode if it is on, reissue the command.

0.0.20. Converting Code to Electric C Mode Format

The C editor mode provides a facility for applying the character style and capitalization rules of Electric C mode to code not originally written using Electric C mode. This facility changes the face and case of reserved words to lowercase bold, and the face of comments to italics.

You can convert code to Electric C Mode format using the m-X Format Language Region command. You can apply the command to an editor region, or, if no region is defined, to the current C language unit. Supplying a numeric argument reverses the effect: all formatting is removed from the specified editor region or routine.

Customizing Electric C Mode

When you use Electric C mode, typed input displays in the appropriate face and case, depending on the *syntactic context* into which you insert the input. Syntactic context refers to the following types of text:

- Body (plain) text
- Reserved words
- Comments
- Template items

Electric C mode supplies a default case and face setting for each of these contexts. This section describes the default settings provided by Electric C mode and explains how to change them, where possible.

0.0.21. Default Case and Face

The table below shows the default case and face for each syntactic context.

Context	*Case*	*Face*
Body text	Lower	Roman
Reserved words	Lower	Bold
Comments	Leave alone	Italic
Template items	Leave alone	Roman

Leave alone means that input displays exactly as typed.

0.0.22. Changing the Global Defaults

You can use the special form **zwei:change-syntax-editor-defaults** to change the C editor mode's defaults for case, face, and indentation.

zwei:change-syntax-editor-defaults *language* &key *dialect indentation case style*
Special Form

Changes the syntax editor's defaults for a language major mode, whose name is the keyword *language*. The defaults you can change are case, face, and indentation. It is recommended that you place **zwei:change-syntax-editor-defaults** within a login form in your init file; however, in this case you must have previously loaded both the **syntax-editor-support** system and the particular language system whose defaults you are changing. Alternatively, you can evaluate the login form at a Lisp Listener after loading the system.

Valid keywords are **:dialect**, **:indentation**, **:case**, and **:style**. **:dialect** and **:indentation** are not documented here, because the preferred and simpler methods for changing dialect (Pascal only) and indentation are m-X Set Dialect and m-X Save Indentation, respectively; use these commands instead of the **:dialect** and **:indentation** keywords.

Keyword	*Meaning*
:case	Specifies the default case of predeclared words, comments, and body text. The format is a list of lists; each list consists of a syntactic context and a case; both are keyword symbols. The format is thus: ((*:syntactic-context :case*) ...).

The keywords naming a syntactic context are:

Keyword	*Meaning*

:predeclared	Reserved words or predeclared identifiers. The default is **:lower**.
:template	Template items. The default is **:leave-alone**, which means "exactly as entered".
:body	Plain text. The default is **:upper**.

Note that strings and comments do not support alternate cases.

The keywords naming a case are:

Keyword	*Meaning*
:upper	All caps, as in ALL_CAPS.
:lower	All lowercase, as in all_lowercase.
:capitalize	Initial cap on first word, as in Initial_cap.
:capitalize-words	Initial cap on each word, as in Initial_Caps.

:style Specifies a list of four character styles, regulating the appearance of body text, reserved words, comments, and template items, in that order. If you specify a character style that is not currently in the environment, **zwei:change-syntax-editor-defaults** loads that style. For more information, see the section "Using Character Styles".

Example: Changing the editor defaults for Pascal mode.

```
(login-forms (zwei:change-syntax-editor-defaults
               :Pascal
               :style '((nil :roman nil) (nil :bold nil)
                        (nil :italic nil) (nil :roman :smaller)
               :case ((:predeclared :upper)
                      (:template :upper))))
```

0.0.23. Changing Indentation Globally

In order to change the global indentation of C source code, select a C mode buffer and position the cursor at the beginning of the construct whose indentation you are changing. Press SPACE or RUBOUT for as many characters as you wish to indent or outdent the construct, respectively. Then press c-I. When the change is successful the editor displays a message, for example:

```
Indentation for the construct changed to 2
```

However, when the change is not successful, such as when you cannot change the indentation for a construct, the editor displays a message to that effect. Once you

are satisfied with the indentation, use m-X Save Indentation. The command produces a Lisp form in another buffer reflecting your changes; evaluate this form after the C editor is loaded.

0.0.24. Changing Face and Case in the Current Buffer

Use m-X Adjust Face and Case to change the face and case of the syntactic contexts (templates, comments, reserved words, text) in the current C mode buffer. The command displays a menu of face and case choices for reserved words, comments, text, and template items. Boldfaced words indicate the current defaults.

Once you select new defaults for the buffer, click on Done. Invoke m-X Format Language Region to put the changes into effect in the current buffer.

Note that you cannot alter the case of comments. You can specify the case for template items, reserved words, and plain text using on of the following:

Case	*Meaning*	*Example*
Upper	All caps	PROCEDURE
Lower	All lowercase	Procedure
Capitalize	Initial cap	Initial_cap_on_first_word
Capitalize words	All initial caps	Initial_Cap_On_Each_Word
Leave alone	Exactly as typed	EXACTLY_as_tYpEd

The Dynamic C Listener

The Dynamic C Listener is a Dynamic Window from which you can issue both C-specific Genera Command Processor (CP) commands. The window has all the features of a Dynamic Lisp Listener: A command history is available, you can scroll back through previous work, and you can operate on many items on screen with the mouse. For example, you can use the mouse to reexecute a command with or without modifying its arguments.

For further information on Dynamic Windows, see the section "Introduction to Genera".

In addition to the usual features of a dynamic language listener window, the C Listener includes three menus listing abbreviations for C-specific commands.

All C-specific commands must be executed in a C Listener. You can also execute other CP commands in a C Listener. Press HELP at the C Listener prompt to see a list of the commands you can use there.

The C Listener supports tools for incremental development of C programs similar to tools that Genera provides for Lisp. For details, see the section "C Evaluation".

How to Use the Dynamic C Listener

This section discusses these aspects of the C Listener:

- How to select the C Listener window.
- The C Listener window components.
- How to issue commands.
- How to use the command menu.

Selecting the C Listener

After loading the C system, type `Select Activity C` at a Listener window. This places you at the C Listener window and makes the window accessible via the System menu and the Select Activity command in future work sessions.

If you wish to customize a key combination with which to select C, use the Select Key Selector utility. (Press `SELECT =` to get to this window). See "Customizing the `SELECT` Key" for further information.

The C Listener Window

The major components of the Dynamic C Listener window are:

- The window name.
- The command menus, which list mouse-sensitive, abbreviated forms of C commands.
- The scroll bar.
- The "C command:" prompt.
- The mouse documentation line.

Issuing Commands

You can issue commands at the C command prompt as you would any Genera command. Command completion is supported and you are prompted for missing command arguments. Note that `HELP`, `COMPLETE`, `c-/`, and `c-?` are useful to discover further information on available argument choices.

Previously typed commands are mouse-sensitive, and the mouse documentation line (in reverse video at the bottom of the screen) tells you what action occurs when you click the left, middle, or right mouse button.

In general, mouse clicks from the C command prompt have these meanings:

Left Reexecute command.

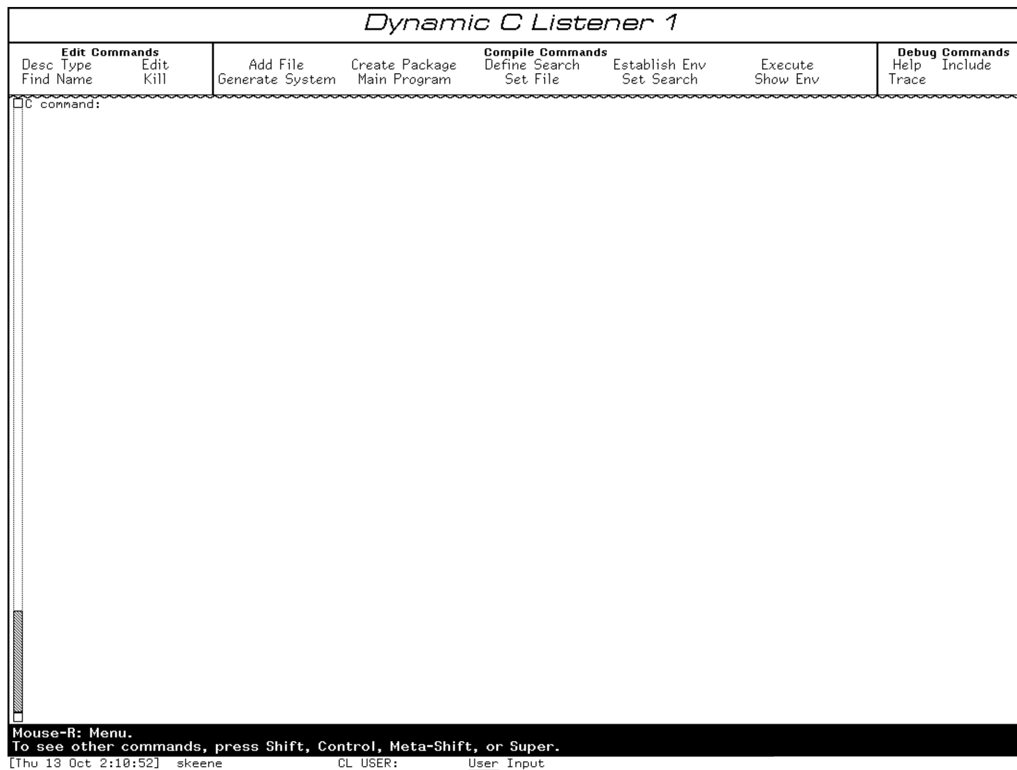


Figure 3. Dynamic C Listener Window

Middle Reexecute command after modifying arguments.

Right Menu of further options.

Using the C Command Menu

In issuing C commands from the C Listener window, you have the option to use the menu of command abbreviations. These abbreviations are mouse-sensitive. Clicking on an abbreviation activates that command and causes the C Listener to prompt you for additional arguments to the command.

In general, when pointing to a previous command, mouse clicks from the command menu have these meanings:

Left Read arguments. Uses the standard CP format to read arguments as you type them and to prompt you for missing arguments.

Middle Not in use.

Right Choose arguments. Presents command arguments in a menu form that you complete.

C Listener Commands

All the usual Genera command processor commands are available in the C Listener. In addition, the C Listener provides commands specific to C. This chapter describes those commands.

0.0.25. Edit Commands

Edit C Definition Command

Menu abbreviation: Edit

Edit C Definition (Definition name) *definition-name* *keyword*

Finds the C definition *definition-name*, places it in a Zmacs editor buffer, and brings up the editor buffer.

definition-name	Name of the definition to edit. A definition is a C procedure.
keyword	:Pathname
:Pathname	The pathname of the source file containing the C definition.

Kill C Definition Command

Menu abbreviation: Kill

Kill C Definition (Definition name) *definition-name* *keywords*

definition-name	The name of the definition to kill.
keywords	:Pathname, :Type, :Query
:Pathname	The pathname of the source file containing the C definition you are killing.
:Type	{All, Function, Macro, Variable, Type} The type of C definition you are killing.
:Query	Enables the system to ask you whether to kill a C definition.

Describe Type Command

Menu abbreviation: Desc Type

Describe Type (A procedural language type) *type*

Describes in some detail any C type.

type	A C type.

Find C Name Command

Menu abbreviation: Find Name

Find C Name (substring) *substring keywords*

Finds the globals whose names contain *substring*. You can specify that the name be of a given type and exist in a given package. The list of names produced is mouse-sensitive. You can click on a name to get further information about it, such as a variable's type.

substring	Substrings of the name to find.
keywords	:Packages, :Type
:Packages	{All, *package-name*} Package in which to search. The default is C-USER.
:Type	{All-Types, Function, Typedef, Variable} The type of definition to search for. The default is All-Types.

Show C Callers Command

Menu abbreviation: Show Callers

Show C Callers (Definition name) *definition-name keywords*

Shows the functions that call *definition-name.*

definition-name	The name of a defined function whose callers to locate.
keywords	:Package, :Pathname, :System
:Package	Lists only callers in the specified package.
:Pathname	Source file pathname for definition.
:System	Lists only callers in the specified system.

0.0.26. Compile Commands

Add File Command

Menu abbreviation: Add File

Add File (Pathname) *pathnames*

Adds the names from the indicated files to the environment and performs their preliminary initialization so that they are visible to the C evaluator.

pathnames Pathnames of one or more C source files.

Create C User Package Command

Menu abbreviation: Create Package

Create C User Package (package name) *symbol*

Creates a C package named with the specified *symbol*.

symbol The symbol that becomes the package name.

Define C Include Directory Search List Command

Menu abbreviation: Define Search

Define C Include Directory Search List (Search list name) *name*
(List of pathnames) *pathnames*

Defines the search list *name* as a list of directory pathnames. This list is used in program compilation as the basis for a search for #include files. The directories are searched in the order in which they are listed.

The directory of the current source file is implicitly the first item in a search list and the directory SYS:C;INCLUDE; is implicitly the last item.

name The name for the directory search list.

pathnames The pathnames of the directories that make up the search list.

Establish Environment Command

Menu abbreviation: Establish Environment

Establish Environment (Program Name or Pathnames) *name keywords*

Establishes an environment for a particular C Listener.

name The argument is either a function name or a list of files. For any function or list of files you must load or compile the equivalent files.

keywords :File Context, :New Name Environment, :Reinitialize, :Search List From Pathname

:File Context — A pathname of the file in whose immediate context you are compiling; in other words, all types defined in this file are visible in this environment.

:New Name Environment — Defines whether or not you inherit the names currently defined in the environment active in the C environment. A value of Yes sets up a new environment where names are not inherited. A value of No does inherit the names already defined.

:Reinitialize — A value of Yes causes reinitialization of the global variables in the environment as a result of establishing this environment. A value of No does not cause reinitialization.

:Search List From Pathname — A value of Yes resets the search list of the environment according to the search list defined by the file whose pathname serves as the immediate file context. A value of No keeps the search list as it already was in the environment.

Execute C Function Command

Menu abbreviation: Execute

Execute C Function (program name) *function-name* *keywords*

Executes a C function.

function-name
The name of the function you want to execute as a main function.

keywords
:Arguments, :Program Name, :User File Pathname, :Temporary File Pathname Defaults, :Enter Debugger On Error

:Arguments — Comma-separated list of arguments to the C program. The arguments are treated as strings and set up as an `argc/argv` array.

:Program Name — Name of the C program to execute. This corresponds to the name you provide for an executable module in conventional architectures.

:User File Pathname Defaults — A pathname for the default directory for user files during program execution.

:Temporary File Pathname Defaults — A pathname for the default directory for temporary files during program execution. The initial default is SYS:C;TMP;.

:Enter Debugger On Error
Boolean for whether to enter the Debugger on a runtime error. The default value is **nil**, not to enter the Debugger.

Generate C System Command

Menu abbreviation: Generate System

Generate C System Definition (System Name) *system-name*
(Source Pathnames) *pathnames* keywords

Prepares a C system for compilation by generating a **defsystem** form. The command creates two C editor buffers:

1. `*system-name-SYSTEM*` — contains the **defsystem** form generated by this command.

2. `*system-name-System-Warnings*` — contains any warnings produced in generating the **defsystem**.

system-name A name for the system you are generating.

pathnames The pathnames of the source files generating the system. You must specify more than one pathname, separated by commas. You must specify a file type "c". Wildcards are accepted.

keywords :Default pathname, :Searchlist name

:Default pathname The **defsystem's** default pathname. The default value is the directory named by the first pathname in the source pathname list.

:Searchlist name The name of a previously defined include file directory search list.

Make C Main Program Command

Menu abbreviation: Main Program

Make C Main Program (Main Program Name) *name*

Makes the specified C definition a main program.

name A C definition.

Note: The Execute C Function CP command automatically performs this command before executing the specified function.

Set C Environment Search List Command

Menu abbreviation: Set Search

Set C Environment Search List (Search List Name) *name keyword*

Sets the search list for the current environment.

name	Name of a search list.
keyword	Predefined
Predefined	If Yes, this command sets the predefined search list of the environment. If No, this command sets the regular search list.

Set File Context Command

Menu abbreviation: Set File

Set File Context (Pathname) *pathname keyword*

Sets the file context of the current environment as the indicated file.

pathname	Pathname of a C source file.
keyword	:Search List From Pathname
:Search List From Pathname	A value of Yes resets the search list of the environment according to the search list defined by the file whose pathname serves as the immediate file context. A value of No keeps the search list as it already was in the environment.

Show C Established Environment Command

Menu abbreviation: Show Env

Show C Established Environment

Displays the current state of the environment, including the file context, the search list, the files in the environment, and the names defined locally in the environment.

Show C Include Directory Search List Command

Menu abbreviation: Show Search

Show C Include Directory Search List (Search list names) *name*

Displays the components of the specified search lists.

name	{:All, :Default, *name*} The search list you want to display.

0.0.27. Debug Commands

Trace C Definition Command

Menu abbreviation: Trace

Trace C Definition (Definition name) *name* *keyword*

Enables the trace facility for the specified definition. Reads the name of the C function and places it in a menu of trace options. For a description of these options: See the section "Tracing Function Execution".

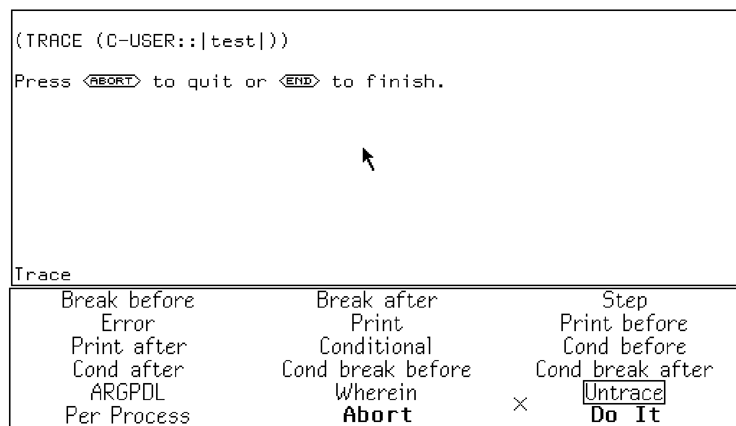


Figure 4. The trace menu

name	The name of a **static** C definition.
keyword	:Pathname
:Pathname	The pathname for the C definition.

C Include Command Menu Command

Menu abbreviation: Include

C Include Command Menu

Presents a menu of commands that work with include files. When you type or click on the command, the following menu appears:

You can click on one of these choices to activate a specific command. The mouse documentation line informs you what action takes place.

For Show Include Files, clicking left shows the source file for a loaded include file you specify from the menu presented; clicking right shows the source files for all loaded include files.

For List Include Files, any mouseclick lists the names of all include files and the time each file is read in.

For Clear Include Files, clicking left clears the specified include file from memory; clicking right clears all include files from memory.

C Evaluation

To enhance incremental development, the Symbolics C environment enables you to evaluate C statements and/or declarations within some environment at both the top level of a C Listener and from a Debugger break in a C function. This section explains how to evaluate C statements and expressions in the Symbolics environment.

The Symbolics C Development Paradigm

The normal mode of development in the Symbolics environment (under Lisp) is quite different from a more traditional environment, such as UNIX.

In traditional environments, program development repeats four steps: editing, compilation, linking, execution. The link step causes association between a global name (function or variable) and a particular storage location applicable during the execution of the program. When the program finishes execution, the association is broken and the name is no longer associated with that storage location.

In the Symbolics environment, you do not have to execute a link step to perform the association between a variable and the storage location containing the variable's value. The association is formed at the time you define the variable. We say that variables have indefinite extent, and though executing some function may change the value, the variable remains accessible after the function returns. You can then evaluate other C statements that use this value and examine the value of the variable for correctness, all of which aids incremental development.

These techniques are useful for the C programmer. Names with indefinite extent allow rapid incremental development and increased ease in debugging, because you can build and examine data structures incrementally as a sequence of actions is applied to the data. Names bound at link time are more useful when porting from another system or when there are two communicating programs that need consistent self-contained data. The Symbolics C environment supports both models of ex-

ecution by enabling evaluation to take place at the top level within a particular C environment, and by providing the Execute C Function CP command that forces name binding at function execution time.

Using C Evaluation

C evaluation enables you to type a C expression, a C statement, or a C declaration; the result from evaluating the statement/declaration is presented. C evaluation is enabled in the following contexts:

- A C listener
- A suspend break from an editor buffer in C mode
- The Debugger

To use the C evaluator:

1. Begin the C statement with a comma to distinguish between a C statement and a CP command.
2. End the C statement by pressing the END key to get the evaluator to take effect.

For example:

```
C command: ,printf("hello, world\n");[END]
hello, world
13
```

To get the value of a global variable, simply invoke the evaluator on that particular variable.

```
C command: ,CHAR MAX[END]
255
```

If the variable has more detailed values, a mouse click expands it.

Restrictions to C Evaluation

Conceptually, each C evaluation takes place as though the statement/declaration were contained inside a C function whose execution has not yet completed. Unfortunately, this implies that there are some restrictions to C evaluation. First, you cannot define new functions in a C evaluation. To achieve the equivalent functionality, define the function in an editor buffer contained in the C environment. The function becomes visible for evaluating in the C environment. Second, statements causing actions to happen at the end of program execution such as the C `atexit` statement, have no effect. Finally, statements that cause nonlocal flow-of-control such as `setjmp` and `longjmp` have no effect. Each C evaluation is invoked within the context of a C environment which controls how names are associated with values. See the section "Name Resolution in C Environments".

Name Resolution in C Environments

This section describes how Symbolics C controls the association of a C variable, function, typedef, macro, structure definition, or static variable with a value during evaluation. C is inherently file oriented. Typedefs, macros, and structure definition names have semantics only within the context of a given file.

In other operating systems, C programs usually consist of a set of files compiled and linked together into an object module. Symbolics C follows that structure in that you can define a C environment consisting of a set of loaded files that establish the name scope in which evaluation can take place.

Once the environment is defined, you can extend the names visible in the environment by:

- Evaluating declarations at top level
- Adding files to the environment
- Establishing a new environment that inherits names from this environment

You can designate an environment by a set of files or a function that you are going to execute. In the latter case, the system computes all the files needed in the environment so that this function and each of its callees are executed. Further, each environment designates a particular file called the *file context* used to resolve names to typedefs and macros when evaluating a C statement or declaration. You can change the file context (for example, to gain access to particular typedefs) without affecting the rest of the current environment. Typedefs and macros defined at top level supersede the typedefs and macros defined in a particular file context.

Environments for C Evaluation

You can establish a default environment by starting a C listener, or using a suspend break in the editor. This environment includes the C run-time system and whatever names you previously entered into the default environment; you can modify the environment used for the default environment.

Once you establish the default environment, you can establish any number of environments desired for performing incremental development. You can use the traditional model of rebinding the C environment each time the function is called by invoking by Execute C Function command. A number of CP commands enable you to query the state of a particular environment. All values and types are presented to the screen in such a way that you can examine their values using the mouse when applicable.

Creating and Manipulating C Environments

Whenever you start a C Listener or use suspend break in C mode, a C environment is established with the default environment, consisting of all the names that

exist in the C runtime library. You can then modify that environment by using the following commands:

Establish Environment Command
: Establishes an environment for a particular C listener.

Set File Context Command
: Sets the file context of the current environment as the indicated file.

Add File Command
: Adds the names from the indicated files to the environment.

Set C Environment Search List Command
: Sets the search list for the current environment.

The following commands enable you to obtain information about C environments:

Describe Type Command
: Describes in some detail any C type.

Find C Name Command
: Finds the globals whose names contain the given substring.

Show C Established Environment Command
: Displays the current state of the environment.

For more information concerning the arguments to these commands: See the section "C Listener Commands".

Debugging C Programs

Debugging Capabilities

0.0.28. Standard Debugger

The standard Debugger starts automatically when an error occurs, and provides information about the Lisp or C routine causing the error.

The Debugger provides features enabling you to:

- Examine the backtrace of routines leading to the error at the source level. If the error occurs in a C routine, the location of the error in the source file is indicated by a small arrow on the left side of the code. In other cases, you can examine the sequence of calls, at the source level, from the C program to the error site.
- Examine variables and arguments at the source level.
- Enter executable C statements and evaluate them at the debug level.

- Set and clear breakpoints.

The standard Debugger presents these capabilities in the Listener window in which the program is executed.

0.0.29. Display Debugger

You can invoke the Display Debugger from the standard Debugger. The Display Debugger has the same functionality as the standard debugger, but provides a structured framework for you to work with in the form of a in the multi-paned window. For more information concerning the Display Debugger interface, see the section "Using the Display Debugger". You can invoke the Display Debugger by typing `c-m-W` from the C Debugger prompt.

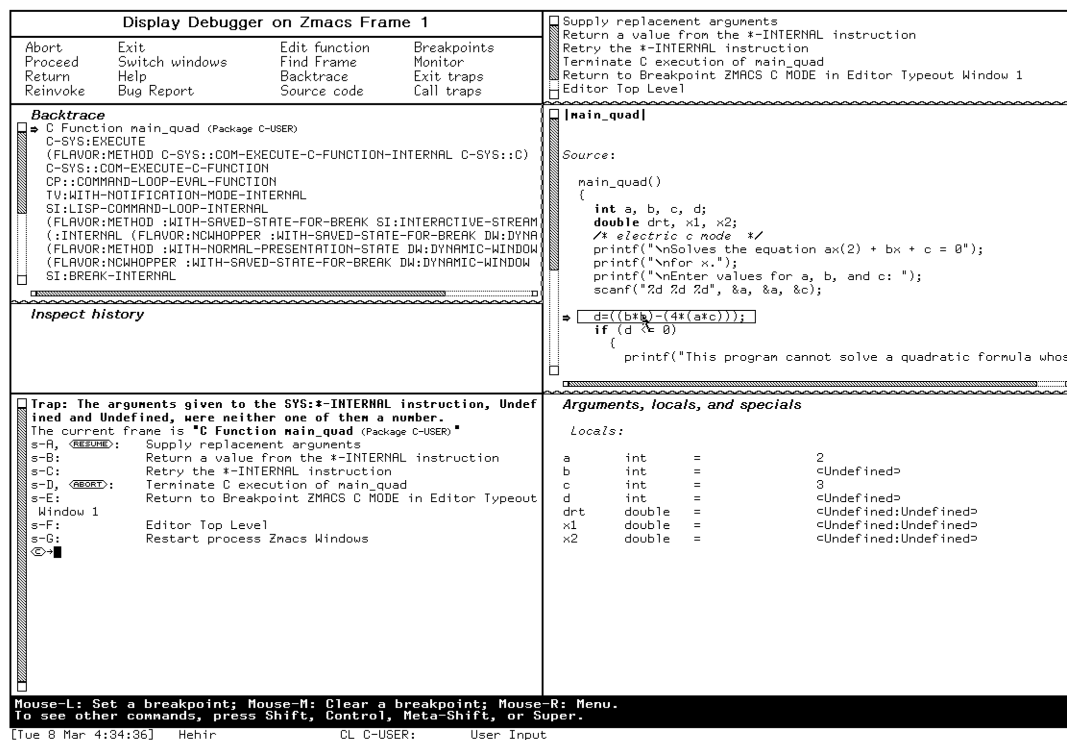


Figure 5. A C Program in the Display Debugger

Invoking the Debugger

You enter the Debugger under these circumstances:

- When you encounter a run-time error and are automatically thrown into the Debugger.

- By setting a breakpoint from the editor.

0.0.30. Exiting the Debugger

You can exit the Debugger using the ABORT key, the :Abort command, or by invoking a restart option.

If you are in the middle of a series of recursive Debugger levels, press ABORT to return to the previous level. Keep pressing ABORT until you leave the Debugger and return to the top level. Pressing m-ABORT from a recursive Debugger level brings you back to top level immediately.

0.0.31. Using Help

The Debugger offers you online help. Pressing the HELP key inside the Debugger displays several help options for you to choose:

- c-HELP displays documentation about all Debugger commands. This documentation consists of brief command descriptions and available key-binding accelerators.
- The ABORT key takes you out of the Debugger. (You can enter the :Abort command or press c-Z instead of pressing ABORT.)
- c-m-W brings you into the Window Debugger. (You can enter the :Window Debugger command instead of pressing c-m-W.)

The REFRESH key, the :Show Frame command, or the :Show Frame command accelerator c-L clears the screen, then redisplays the error message for the current stack frame.

You can also ask for help with keywords. If you do not remember what keywords are available for the command you are entering, press the HELP key after you receive the keywords prompt. The Debugger displays a list of keywords for that command. For example:

```
→ :Previous Frame (keywords) HELP
You are being asked to enter a keyword argument

These are the possible keyword arguments:
:Detailed             Show locals and disassembled code
:Internal             Show internal interpreter frames
:Nframes              Move this many frames
:To Interesting       Move out to an interesting frame
```

C Frames in the Debugger

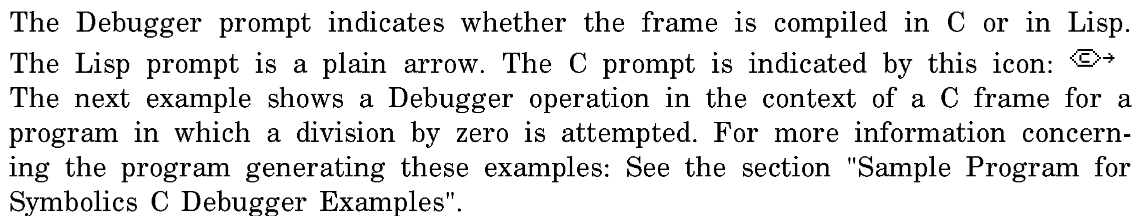
If you are unfamiliar with the Genera Debugger, you can refer to the Genera documentation set for background information. See the section "Debugger". This discussion assumes you have some knowledge of Debugger concepts and capabilities. In particular, it refers to:

- Stack frame — A frame from the control stack that holds the local variables for the routine.

- Current stack frame — The context within which Debugger commands operate. The Debugger uses the current frame environment for performing operations according to the suspended state of your program. Also, the Debugger evaluates forms in the context of the suspended state of the current function's stack frame.

When you use the Debugger on a frame compiled in C, you can get information about local and global variables and about the type and value of variables at various points in the source. You can also evaluate expressions and statements.

The Debugger prompt indicates whether the frame is compiled in C or in Lisp. The Lisp prompt is a plain arrow. The C prompt is indicated by this icon: The next example shows a Debugger operation in the context of a C frame for a program in which a division by zero is attempted. For more information concerning the program generating these examples: See the section "Sample Program for Symbolics C Debugger Examples".



```
Command: Execute C Function (function name) main_quad

Solves the equation ax(2) + bx + c = 0
for x.
Enter values for a, b, and c: 0 1 2
Error: An invalid floating-point operation was attempted:
       %FLOAT-OPERATION-DIVIDE with arguments (0.0d0 0.0d0)

C Function main_quad (Package C-USER)
 Debugger was entered because of an error in the program
s-A, RESUME:   Use the not-a-number (non-trap) result: #<+DOUBLE-NAN 17777777777777777>
s-B:           Ask for a number to use in place of the result
s-C, ABORT:    Terminate C execution of main_quad
s-D:           Return to Breakpoint ZMACS C MODE in Editor Typeout Window 1
s-E:           Editor Top Level
s-F:           Restart process Zmacs Windows
Meta-L Show Frame :Detailed Yes :Clear Window Yes
 Called to return, funcalled.
```

You can perform a variety of Debugger operations from this prompt. Most multilanguage Debugger operations are performed easily using the mouse. The mouse documentation line describes the meaning of a mouse click in the context of the current location of the cursor and also provides help messages for some operations. Most mouse gestures have typein command equivalents or are bound to command accelerators. For further information: See the section "C Language Debugger Commands".

You can begin a debugging session using the command :Show Frame :Detailed (or the command accelerator `m-L`) at the C Debugger prompt. This shows a list of local variables and their values for the function in question as shown in the next example, a continuation of the previous one:

```
©→ Meta-L Show Frame :Detailed Yes :Clear Window Yes
 Called to return, funcalled.

|main_quad|  (from R:>hehir>c>quadratic.c)


 Locals:

 d       int       =              1
 drt     double    =              1.0d0
 x       double[2]  =             ≤double[2] 16892671≥
 coeffs  pcoeffs   =              (*≤coeffs 22765759≥)

Source:

    d=((coeffs->coeff2*coeffs->coeff2)-(4*(coeffs->coeff1*coeffs->ccoeff)));
    if (d <= 0)
      {
        printf("This program cannot solve a quadratic formula whose");
        printf("\nresult is an imaginary number.");
      }
    else {
        drt = sqrt((double)d);
⇒       x[0]=(-coeffs->coeff2+drt)/(2*coeffs->coeff1);
        x[1]=(-coeffs->coeff2-drt)/(2*coeffs->coeff1);

        printf("The value of root one is %f ", x[0]);
        printf("\nThe value of root two is %f ", x[1]);
      }
  }
```

From this point you have a number of options.

For example, you can see a translation of the summarized value *≤coeffs 96781403≥*, by using the :Show Detailed Value command and clickingon this value from the Show Frame display when prompted. In this case, you can see that the summarized value represents a structure with three members, coeff1, coeff2, and ccoeff.

```
©→ :Show Detailed Value (a value) "≤coeffs 96781483≥"
struct {
   coeff1 : 0 ;
   coeff2 : 1 ;
   ccoeff : 2 ;
} ;
```

Additionally, you can perform operations such as displaying detailed information for a variable's type. The next example shows how to find the type of the variable `coeff`. As a first step, examine its type, `pcoeff`, by pointing the cursor at `pcoeff` in the Show Frame display and clicking `mouste`-middle invoking the :Show Type Detailed command. The presented result indicates that `pcoeff` declares a pointer to a type named `coeffs`. Clicking `Mouse`-middle on `coeffs` in the presented result shows that `coeffs` is a structure type with three members each of type `int`.

```
©→ Describe Type Detailed pcoeffs
(*coeffs)
©→ Describe Type Detailed coeffs
struct {
   coeff1 : int ;
   coeff2 : int ;
   ccoeff : int ;
} ;
```

0.0.32. Setting and Clearing Breakpoints

You can set a breakpoint in source code clicking left on a line of code. Clear the breakpoint by clicking middle. From the editor, set a breakpoint clicking `c-m-Left`, while pointing to a line of code. Clear the breakpoint by clicking `c-m-Left`, while pointing to a line of code.

Looking at Variables, Values, and Types

Local variables and their values are mouse-sensitive items. The mouse documentation line displays the action associated with each mouse click. The following table summarizes information for variables, values, and types.

Language Object	*Mouse click*		
	Left	*Middle*	*Right*
variable	Show value	:Show type	menu
value	returns the value	Describes the value	menu
type	Show type	:Show type :detailed	menu

Using the mouse offers a quick and efficient way to inspect a variable, value, or type. In particular, you can get a complete description of a complex type.

How Values Are Displayed in the Debugger

Uninitialized values An uninitialized value prints out as the symbol: *undefined*

Values that exist out of range of an object

When you attempt to access a value beyond the range of the *underlying* allocated hardware object, the value prints out as the symbol: *unallocated_memory*

Note there is no one-to-one correspondence between a hardware object and a C language object, since many C language objects are allocated within a hardware object. Thus, violating the bounds of a language object does not always yield the symbol *unallocated_memory*.

Summarized values Summarized values are given for objects whose values are too large for printing by default. For instance, unless requested explicitly by the user, arrays and structures are printed out in summary form between the characters ≤≥. The summary form contains an abbreviated type indication followed by a unique number that helps distinguish two different values.

For example,

≤struct {...} 97541456≥
≤struct {...} 12345678≥

Values not of the Declared Type

A value printed between horseshoes when the value, as indicated by the tags in the hardware, does not correspond to the declared type for the value. For example, obtaining the value of a variable declared as an integer but actually containing a real yields a result in this form.

For example,

⊂1.3⊃

C Language Debugger Commands

0.0.33. Variables, Values, and Types

The following tables summarize the Debugger commands available in C frames and provide specific information for C variables, values, and types. The left column represents Command Processor commands and accelerators, the right column shows corresponding menu choices in the first table, and a definition of the command in the second table.

Commands	*Menu choices*
Commands for variables	
:Show Locals	not applicable
:Show Frame	not applicable
:Show Typed Variable	not applicable
:Show Typed Value	not applicable
Commands for values	
:Show Variable's Value	Examines the value associated with this variable
Commands for types	
:Show Variable's Type	Examines the type associated with this variable
:Describe Type Detailed	Describe the type in greater detail
:Show Type Name	Show the type name
Other:	
from menu only	Edit Viewspecs

:Show Source Code

Commands *Definition*

Commands for stepping

:Statement Step For Function

Program execution stops in the debugger before the execution of each statement

:Clear Statement Step For Function

Clears the :Statement Step For Function enabling the program to execute normally

The following debugging commands are useful when using the stepping feature. In order to see a complete list of all debugging commands, specify :language help from the debugger.

```
c-X c-D       Show the source code for the function in the current frame.
```

:Next Frame

```
c-N,      Move down a frame (takes numeric argument), skipping invisible frames.
m-sh-N        Move down a frame, not skipping invisible frames.
m-N           Move down a frame, displaying detailed information about it.
c-m-N         Move down a frame, not hiding internal interpreter frames.
```

:Previous Frame

```
c-P           Move up a frame (takes numeric argument), skipping invisible frames.
m-sh-P        Move up a frame, not skipping invisible frames.
m-P           Move up a frame, displaying detailed information about it.
c-m-P         Move up a frame, not hiding internal interpreter frames.
c-m-U         Move to the next frame that is not an internal interpreter frame.
```

:Show Backtrace

```
c-B           Displays a brief backtrace, hiding invisible frames,
              but not censoring continuation frames.
c-sh-B        Displays a brief backtrace of the stack, censoring invisible
              internal (continuation) frames.  Use a numeric
              argument to indicate how many frames to display.
m-B           Displays a detailed backtrace of the stack.
m-sh-B        Displays a brief backtrace, without censoring invisible
              or continuation frames.
c-m-B         Displays a backtrace of the stack, including internal frames.
```

0.0.34. C Listener Commands Available in the Debugger

The C listener commands available in the debugger are as follows:

```
:Show C Callers
:Find C Name
:Execute C Function
:Make C Main Program
:Show C Include Directory Search List
:Define C Include Directory Search List
:Create C User Package
:Trace C Definition
:C Include Command Menu
:Kill C Definition
:List C Include Files
:Show C Include Files
:Clear C Include Files
```

For more information on these commands: See the section "C Listener Commands".

0.0.35. Monitor Expression Commands

:Monitor C Expression

:Monitor C Expression *expression keywords*

Monitors a C address associated with a specified C *expression*.

expression	The C expression you are monitoring.
keywords	:Read, :Write, :Query
:Read	Enables the command to trap the reads.
:Write	Enables the command to trap the writes.
:Query	Enables you to specify whether the command queries for resolution field types.

:Unmonitor C Expression

:Unmonitor C Expression *expression keywords*

Stops monitoring a C address associated with a specified C *expression*.

expression	The C expression you want to stop monitoring.
keywords	:Query
:Query	Enables you to specify whether the command queries for resolution union field types.

0.0.36. Expressions and Statements

You can evaluate expressions and statements from the Debugger. Type the expression or statement after the C debugger prompt, then press END for evaluation. The result reflects the values and conditions present in the Debugger context: the point at which program execution stopped (see the following example).

The example evaluates the members of the structure `coeffs` (note that you must use the END key and not RETURN).

```
→ Eval (program): (coeffs->coeff1)
0
→ Eval (program): (coeffs->coeff2)
1
→ Eval (program): (coeffs->ccoeff)
2
```

Genera Debugger Commands for Use with C

You can use the following kinds of Genera Debugger commands to help debug C programs. These commands are described in the Genera documentation set. For more information, see the section "Debugging C Programs".

- Commands for viewing a stack frame.
- Commands for stack motion.
- Commands for general information display.
- Commands to continue execution.
- Trap commands.
- Commands for breakpoints and single stepping.
- Commands for system transfer.

You can choose to use Lisp mode debugging at any point in a debugging session by using the :Use Lisp Mode command. This command is similar to choosing to debug a program at the assembly language level in conventional machines.

Sample Program for Symbolics C Debugger Examples

This program solves for x in the quadratic formula `ax(2) +bx +c = 0`. The program is written using arrays, structures, and derived types as an illustration of how these constructs appear in various debugging scenarios.

```
/*-*- Mode: C; Package: C-USER -*- */
```

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

struct coeffs {
  int coeff1, coeff2, ccoeff;
};

typedef struct coeffs *pcoeffs;

main quad()
{
  int d;
  double drt, x[2];
  pcoeffs coeffs;

  coeffs = malloc(sizeof(struct coeffs));

  printf("\nSolves the equation ax(2) + bx + c = 0");
  printf("\nfor x.");
  printf("\nEnter values for a, b, and c: ");
  scanf("%d %d %d", &coeffs->coeff1, &coeffs->coeff2, &coeffs->ccoeff);

  d=((coeffs->coeff2*coeffs->coeff2)-(4*(coeffs->coeff1*coeffs->ccoeff)));
  if (d <= 0)
    {
      printf("This program cannot solve a quadratic formula whose");
      printf("\nresult is an imaginary number.");
    }
  else {
      drt = sqrt((double)d);
      x[0]=(-coeffs->coeff2+drt)/(2*coeffs->coeff1);
      x[1]=(-coeffs->coeff2-drt)/(2*coeffs->coeff1);

      printf("The value of root one is %f ", x[0]);
      printf("\nThe value of root two is %f ", x[1]);
    }
}
```

Using Include Files with Symbolics C

Symbolics C provides several means for manipulating directories of user-defined and predefined include files.

A user-defined include file is included with the double-quote syntax, as follows:

`#include "`*filename*`"`

A predefined include file is one which you include with the angle-bracket syntax, as follows:

`#include <`*filename*`>`

This section describes how you can manipulate include files, and describes how Genera caches include files.

Search Lists for Include File Directories

You can define *search lists* for include files. A search list tells the compiler where to look for include files. A search list has a name and an ordered list of directories. You first define the search list, and then you use it by associating the search list with a file or buffer. You can associate a file and its search lists via file attributes. See the section "Defining Search Lists for Include Files". See the section "Setting the Search Lists of a Source File".

Each C source file can have two different search lists: one for user-defined include files (which we call the *regular search list*), and one for predefined include files (which we call the *predefined search list*).

You can also define *default search lists*. A *default regular search list* is searched when a source files has no regular search list associated with it. Similarly, you can define a *default predefined search list* that is searched when a source file has no predefined search list associated with it. See the section "Defining Default Search Lists for Include Files".

When the compiler looks for user-defined include files (which use the double-quote syntax with #include), it does the following:

1. Checks the directory in which the current source file exists.

2. If it is not found there, checks each directory in the regular search list associated with the source file. If the file has no regular search list, the directories in the default regular search list are checked.

3. If it is not found there, checks the SYS:C;INCLUDE; directory.

4. If it is not found there, signals an error.

When the compiler looks for predefined include files (which use the angle-bracket syntax with #include), it does the following:

1. Checks each directory in the predefined search list associated with the source file. If the file has no predefined search list, the compiler checks the directories in the default predefined search list.

2. If it is not found there, checks the SYS:C;INCLUDE; direcory.

3. If it is not found there, signals an error.

Commands and Functions for Using Search Lists

You can use the following commands and functions to create and use search lists for directories of C include files:

- **C Listener commands**
  - Define C Include Directory Search List
  - Set C Environment Search List
  - Show C Include Directory Search List
- **Editor commands**
  - m-X Define C Search List
  - m-X Set C Search List for Buffer
  - m-X Show C Search List
  - m-X Undefine C Search List
- **Functions**
  - **c-sys:define-default-search-list**
  - **c-sys:define-search-list**

See individual commands for further descriptions.

Defining Search Lists for Include Files

You can define a search list in three ways:

From the C Listener, with Define C Include Directory Search List
From the editor, with m-X Define Search List
With the function, **c-sys:define-search-list**

When you specify the directories in these commands, you can use a subset of wildcard syntax. Specifically, you can use this syntax:

```
>*.*.*
```

Wildcard directory mapping is not supported, nor is specifying a portion of the pathname as a wildcard.

c-sys:define-search-list *name* &rest *directories* *Function*

Defines a search list of C include file directories, using *name* as the name for the search list and the specified *directories* as its components. It lists the directories in the order in which you want them searched.

See also: "Defining Default Search Lists for Include Files"

Setting the Search Lists of a Source File

To ensure that the compiler uses a given search list for a C source file, you have to associate that search list with the source file (use the m-X Set C Search List for Buffer command). This command gives the file an attribute specifying the name of the search list. Note that this does not associate the list of pathnames with the file.

If you are using the search list for user-defined include files, use the m-X Set Using the Set C Search List for Buffer command with no argument. This sets the Search-List file attribute as the given search list.

If the search list is used for predefined include files, use the m-X Set C Search List for Buffer with a numeric argument. This sets the Predefined-Include-Search-List file attribute as the given search list.

Defining Default Search Lists for Include Files

In addition to defining explicitly named search lists, you can also define a search list as the default search list for include files.

You can define a *default regular search list* searched for user-defined include files if the source file has no regular search list associated with it.

Similarly, you can define a *default predefined search list* searched for predefined include files, if the source file has no predefined search list associated with it.

Default search lists do not have names. There is at most one default regular search list and one default predefined search list in effect in any given Lisp world. You can define or redefine them with the functions described below. Note that if you define a default search list within **login-forms**, the effects are automatically undone when you log out.

c-sys:define-default-search-list &rest *directories* *Function*

Defines a default search list for user-defined C include files to be the specified *directories*. It lists the directories in the order in which they are searched.

To undo the effects of calling this function to set up a default search list, call the function again with no arguments.

c-sys:define-predefined-default-search-list &rest *directories* *Function*

Defines a default search list for predefined C include files as the specified *directories*. It lists the directories in the order in which they are searched.

To undo the effects of calling this function to set up a default search list, call the function again with no arguments.

Exporting Include Files for Shared Use

We recommend that you "export" include files intended for use across a set of C source files. This prevents the C binary files from becoming very large, due to unnecessary copying of definitions. When you are sure that the definitions in an include file remain the same when compiled with each C source file that is part of the system, then the compiler compiles and loads one set of those definitions, which are shared by all C files that include them.

Consider what happens if you do not export include files. The compiler makes a copy of the definitions from an include file in each .c file that includes that file. For example, each .c file that includes stdio.h has its own copy of a given symbol or object. This results in very large binary files.

The procedure for exporting include files is simple. You perform the following steps:

1. Create a file including a set of include files; this is called the *export file*. Be sure that the definitions in each file included in the export file remain the same when compiled on each C source file which is part of the system.

2. Set the Export attribute of the buffer to be yes using the command `m-X` Set Export for Buffer.

3. Include the export file as a module of your system. The export file is a C source file; you must compile and load it before all other C sources. Compiling and loading the export file defines the objects shared across subsequent files in the system.

Symbolics C supplies a predefined export file including all the standard predefined include files. Include this file as part of the system definition of any system using standard include files, even if it only uses one or two of the standard include files. The name of the predefined export file is:

SYS:C;EXPORT-C-LIBRARY.BIN

Most C applications include the predefined export file and a separate export file corresponding to application-specific data as part of their system definitions.

Here we give an example of an include file that is **not** a good candidate for exporting. The include file named include.h contains this definition:

```
struct x {
  TWO WORD TYPE f;
};
```

One file in the system contains the following:

```
#define TWO WORD TYPE double
#include "include.h"
```

Another file in the system contains the following:

```
#define TWO WORD TYPE char *
#include "include.h"
```

Since the two source files define TWO WORD TYPE differently, do not export the header file that uses TWO WORD TYPE. The default behavior of the compiler (to textually include the definitions from include files for each C source file) is appropriate for this situation.

Note: Using one export file reduces the size of C binary files with symbol information. It has no effect on the size of run-time-only binary files. See the section "Minimizing the Size of Compiled Files for C Programs".

Caching Include Files

The Symbolics C system caches include files in these ways:

Compiling to Memory

When compiling to memory, the include cache is ignored if a corresponding Zmacs file buffer exists. Instead, information from the buffer is used. If there is no file buffer and a new version of the file is written to disk, the system updates the cache.

Compiling to File

When compiling to file, using the Compile File command, the cache is updated if a new version of the file is written to disk. Include files cached in memory are updated to match the latest version written to disk. Zmacs file buffers are ignored.

When compiling to file using the Compile System command, the current state of any include files on disk is updated once at the start of the compile system, and that cached state is used throughout the system compilation. Any new include files written to disk during the compile system operation do not update the state of the include file cache.

Using the Package System with C Programs

An important characteristic of the Genera environment is that the "operating system," the editor, and a wide variety of other programs exist in one environment where their names and the names of their functions and variables are each associated with a symbol. The programs you use during a work session are also loaded into this environment. This presents the potential for a conflict in names. For example, if the function **foo** already exists in the environment and you load a program that has a function named **foo**, the original **foo** is redefined by the new function **foo**.

Genera provides a mechanism for separating like-named functions by assigning each to a distinct context or namespace. The namespace is called a package.

Example: c-user:foo and **cl-user:foo** define two functions named **foo**, one in the package **c-user**, the other in the package **cl-user**. These functions can coexist in the same Genera environment.

In a more conventional environment, these correspond to two different **foo**s in different programs where the programs may not coexist at the same time in memory or to two different **foo**s that exist in separate processes.

Effect of the Package System on Naming C Functions

It is common practice in C programming to use the name *main* for the function at which execution of a program begins. You can use this practice in the Genera environment if you place each C program in its own package. However, it is advantageous to place closely related programs in the same package. See the section "Sharing of Symbols Among Packages".

Consequently, it is most efficient to give the function at which execution begins a name more descriptive of the purpose of the function, such as *allocate-main*.

Predeclared C Package

Symbolics C recognizes one predeclared packages, **c-user**, which is the default package for Symbolics C.

How to Declare a C Package

You can create a C package with the appropriate attributes with the C Listener command Create C User Package or with the function **c-sys:package-declare-with-no-superiors.** After you create a package declaration, you must compile it for it to become effective.

Create C User Package Command

Menu abbreviation: Create Package

Create C User Package (package name) *symbol*

Creates a C package named with the specified *symbol.*

symbol The symbol that becomes the package name.

c-sys:package-declare-with-no-superiors *name* *Function*

Creates a package for use with C. The package has no inheritance from other packages.

name A symbol assigned as the package name.

How to Assign a C Package

You can assign a package name as part of a file's attribute list or as a value in a system declaration.

Adding the package name to a file's attribute list enables editor-based compilation of routines in the file without referencing a system declaration or the default package.

To change the package name in the attribute list, use `m-X` Set Package and type the package name. Alternatively, you can edit the attribute list directly by typing `; Package:` and the package name. Use `m-X` Reparse Attribute List to make this change take effect.

You may specify a default package for a C system of several files in a system declaration. See the section "**:default-package** option for **defsystem**".

See the function **defsubsystem**. Any package attribute in a file's attribute list overrides the default package you have specified in your system definitions.

Using the System Construction Tool with C Programs

Managing a large program is easier by splitting it into several files. The Genera System Construction Tool (SCT) provides a way to manage such a program. It lets you construct a *system* by specifying a set of source files and a set of rules and procedures defining the relations among these files. This system makes up a complete program. For an overview of SCT: See the section "Introduction to the System Construction Tool". There are three basic steps in using a system:

- Generating a system definition.
- Compiling the system.
- Loading the system.

You define the system using SCT's **defsystem** special form. The definition, called a *system declaration*, specifies such information as the names of the source files, modules, or both in your system and what operations you must perform on each module in what order. For example, the system declaration specifies compilation order for modules.

After creating a system declaration and evaluating it, you can compile the system. You can then choose to load the system, edit it, or distribute it by tape.

The command Compile System and the **compile-system** function provide a means to compile a system you define it.

You can load a C system using the Load System command or **load-system** function. See the section "Load System Command".

See the function **load-system**.

Creating Defsystems for C Programs

There are two ways to generate defsystems for C programs. If you are describing a simple program, you can define a **defsystem** of your own. If you have a large amount of code, use the Generate C System Definition command or **c-sys:create-c-defsystem-from-pathnames** function to automate the process. See the section "Creating Defsystems for Large C Programs". To create your own **defsystem**:

1. Define a module corresponding to each file.

2. For each include file dependency, define a C compilation dependency using the **:uses-definitions-from** keyword.

The C system provides the module types **:c-include** and **:c** and the parameter type **:searchlist**. For more information:

See the section "Long-form Module Specifications".
See the section "**:module** Keyword Options".
See the section "**:parameters** option for **defsystem**".

After creating the **defsystem** that corresponds to your system, you must compile or evaluate the **defsystem** to actually define the system.

Creating Defsystems for Large C Programs

The C Listener command Generate C System Definition and the **c-sys:create-c-defsystem-from-pathnames** form provide a simple way to generate SCT **defsystem** forms for large or complicated programs. These both use the arguments you supply to create a system declaration. They read a set of C files, deduce the proper compilation dependencies, and create a **defsystem** form.

The **defsystem** form is written to an editor buffer whose name follows the form **system-name*-System*. Any warnings produced in generating the **defsystem** appear in the buffer **system-name*-System-Warnings*. Edit the generated system definition in the **system-name-definitions** buffer to correct any errors found in the **system-name-System-Warnings** buffer. Save the contents of the **system-name-definitions** buffer to a file for preservation between reboots.

Note that you cannot specify information concerning include file search lists or component systems with Generate C System definition or for **c-sys:create-c-defsystem-from-pathnames**. In such cases, edit the **defsystem** form directly.

Generate C System Command

Menu abbreviation: Generate System

Generate C System Definition (System Name) *system-name*
(Source Pathnames) *pathnames* keywords

Prepares a C system for compilation by generating a **defsystem** form. The command creates two C editor buffers:

1. `*system-name-SYSTEM*` — contains the **defsystem** form generated by this command.

2. `*system-name-System-Warnings*` — contains any warnings produced in generating the **defsystem**.

system-name — A name for the system you are generating.

pathnames — The pathnames of the source files generating the system. You must specify more than one pathname, separated by commas. You must specify a file type "c". Wildcards are accepted.

keywords — :Default pathname, :Searchlist name

:Default pathname — The **defsystem's** default pathname. The default value is the directory named by the first pathname in the source pathname list.

:Searchlist name — The name of a previously defined include file directory search list.

c-sys:create-c-defsystem-from-pathnames *system-name pathnames* &key *(:default-pathname* **(send (first c-sys::pathnames) :new-pathname :name nil :type nil :version nil)***) :component-systems :search-list-name :buffer-or-file-name (:buffer-p* **'t***)*
Function

Prepares a C system for compilation by generating a **defsystem** form.

system-name — A name for the system.

pathnames — The pathnames of the source files generating the system. You can specify more than one pathname as a Lisp list. Files that are C source files must have the file type "c". Wildcards are accepted.

keywords — :default-pathname, searchlist-name, :buffer-or-file-name

:default-pathname — The defsystem's default pathname. The default value is the directory named by the first pathname in the source pathname list.

:searchlist-name — The name of a previously defined include file directory search list.

Generate C System Example

This example illustrates the use of the Generate C System Definition command to create a **defsystem**. The case described is one in which we are given the files `board.h` , `board.c`, `check.h`, `check.c`, and `life.c` in local:>. The Generate C System Definition command reads through the files, checking for dependencies, and calculates an accurate **defsystem** form.

The next section shows each of these files, the CP command used to generate a **defsystem** for them, and the **defsystem** form that results.

Example

The file contents are:

board.c

```
/*-*- Mode: C; Package: C-USER -*- */
                                                /**board.c**/

/** this file contains no include files **/
```

check.c

```
/*-*- Mode: C; Package: C-USER -*- */

#include <time.h>                               /**check.c**
#include "board.h"
        .
        .
        .
```

life.c

```
/*-*- Mode: C; Package: C-USER -*- */

#include <stdio.h>                              /**life.c**/
#include "board.h"
#include "check.h"

        .
        .
        .
```

The following command creates a **defsystem** named `life` and places the **defsystem** form in the Zmacs buffer `*life-system*`.

```
C Command: Generate C System Definition (System Name) life (Source
   Pathnames) local:>*.c
```

```
Output found in the Zmacs buffer: *life-system*.
Warnings found in the Zmacs buffer: *life-system-warnings*
```

the *life-system* editor buffer

```
;;; -*- mode: lisp; syntax: common-lisp; package: user; base: 10 -*-

(DEFSYSTEM
 LIFE
 (:DEFAULT-PATHNAME "local:>" :DEFAULT-MODULE-TYPE :C)
 (:MODULE C-MODULE-0 "local:>board.c" (:TYPE :C))
 (:MODULE C-MODULE-2 "local:>check.h" (:TYPE :C-INCLUDE))
 (:MODULE C-MODULE-3 "local:>board.h" (:TYPE :C-INCLUDE))
 (:MODULE C-MODULE-4
          "local:>check.c"
          (:TYPE :C)
          (:USES-DEFINITIONS-FROM C-MODULE-3))
 (:MODULE C-MODULE-1
          "local:>life.c"
          (:TYPE :C)
          (:USES-DEFINITIONS-FROM C-MODULE-2 C-MODULE-3))
 (:PARALLEL C-MODULE-0 C-MODULE-2 C-MODULE-3 C-MODULE-4 C-MODULE-1))
```

The :maintain-journals Option in C Defsystems

Do not use the **(:maintain-journals nil)** option of **defsystem** when you create systems whose modules (or some subset of whose modules) are C source and C include files. The compilation dependencies on C include files are not properly computed, and unnecessary recompilation of C source files occurs.

Note that the Generate C System Definition command produces system definitions without the **:maintain-journals** option, so it defaults to **t**, which is correct for systems whose modules are C files.

Compiling and Loading C Systems

You can compile a system with the Genera command Compile System or with the **compile-system** function. You can load a system with the Load System command, with a keyword argument to the Compile System command, or with the **compile-system** function.

Building Applications with C Run-time Systems

Symbolics C supports features enabling you to build and distribute minimally sized C applications including the C run-time system. You can run applications including this system in environments not running the C development system.

Customers who distribute an application with the C run-time system **must** sign a *Sublicense Addendum to the Terms and Conditions of Sale, License, and Service.* For more information, see the section "Sublicense Addendum for Symbolics C".

0.0.37. Components of a Run-time System

A run-time system (as opposed to the development system) for a language is made up of the minimal subset of the development system software required to load and execute a program. From a user's perspective, it contains the library routines defined for the language, the loader, and the function that initiates execution. The following functionality, normally present in the development system, is absent in the run-time system:

- Language-specific Zmacs Editor Mode
- Compiler
- Language-specific Source Level debugger
- CP window, to support language-specific CP commands

The C run-time system is called *C-Runtime*.

0.0.38. Creating Applications

Normally, you follow these steps to develop and deliver an application that includes the C run-time system:

1. Develop the application using the full development environment.
2. As an option, during program compilation, set a global variable to filter out debugging information from binary files. This reduces the size of the finished application.
3. Include the C run-time system as a component system in the system definition when you write the system declaration for the C application.

Minimizing the Size of Compiled Files for C Programs

During a normal compilation, the compiler produces information that supports debugging and incremental compilation. This information is normally written out to the bin file, a binary file identified by the file extension *bin*, or *ibin* on Ivory based machines. You can exclude this information from the bin file by setting the special variable **cts:*compile-for-run-time-only*** to the Lisp boolean **t**. Doing so minimizes the size of the binary files produced for an application.

By convention, binary files produced in this manner are referred to as run-time only (*rto*) bins (but are assigned the file extension). Using *rto* binary files limits your ability to debug and compile source code, so use this facility judiciously. Use of

this facility does not change the generated code. The section "Program Configurations: Development and Run-time System Options for C Systems" specifies the capabilities of *rto* binary files.

Incorporating the Runtime System Into a C Application

Package the run-time system as a dependent component system of the application. You must meet these requirements when defining such a packaged system:

- The packaged system definition must cause the run-time system to load before any of the application program loads. Specifying the appropriate `:serial` dependency loads the run-time system correctly.

- The definition must cause the reading of the system declaration (*sysdcl*) file for the run-time system before encountering any C file in an application system definition.

The following example illustrates how an application named *a1* is packaged. Note that *a1* is a component system (with accompanying separate system declaration, or *sysdcl* file) and not a separate subsystem when the sysdcl contains references to objects defined in the **system** or **user** package defined by the run-time system in question.

```
(defsystem a1
    (:default-pathname "foo:bar;"
     :distribute-binaries t
     :default-module-type :C)
  (:serial "f1.C" "f2.C"))

(defsystem packaged-a1
    (:default-pathname "foo:pkg-bar;"
     :distribute-binaries t)
  (:module C-runtime "C-runtime" (:type :system))
  (:module a1 "a1" (:type :system))
  (:serial C-runtime a1))
```

You can use the distribution software to distribute the packaged software.

For further information, see the sections:

"Distribute Systems Command"
"Distribute Systems Frame"
"Restore Distribution Command"
"Restore Distribution Frame"

Program Configuration Options for C Systems

Given the capabilities of a run-time system and the ability to produce *rto* bins, you can have a program in a configuration obtained by the following cross product:

(normal bin, rto bin) X (development system, run-time system)

The (normal bin, development system) configuration is the usual configuration and the one that makes the full functionality of the development system available. Other configurations limit the functionality in various ways.

The following table describes the properties of each possible configuration.

0.0.39. Program Configurations: Development System and Run-time System Options

	Development System	*Run-time System*
Normal Bin	Incremental Compilation: Yes Batch compilation: Yes Language-specific debugging: Yes	Incremental Compilation: No Batch compilation: No Language-specific debugging: No
Rto Bin	Incremental Compilation: * Batch compilation: * Language-specific debugging: No	Incremental Compilation: No Batch compilation: No Language-specific debugging: No

*Incremental compilation is possible, after all references external to the unit being incrementally compiled are compiled. For C, this means that a file or buffer is compiled before an individual function within it is recompiled.

0.0.40. Purpose of Configurations

Normal bin, Development system:
This is the normal configuration for software development.

Normal bin, Run-time system: This configuration is advantageous when software is actively developed, but is also simultaneously used in a run-time system.

Rto bin, Run-time system: This is the desired configuration for software of minimal size that is released.

Rto bin, Development system: This is not a recommended configuration. You should re-create normal *bin* files if you plan to do any debugging or development work with these files.

C - Lisp Interaction

This chapter discusses the interface between Lisp and C and presents information on these areas:

- `lispobj`, a new type specifier for representing Lisp objects in C programs
- `lisp`, a new storage class specifier used in a function directive to declare a Lisp function so that you can call it from a C procedure
- Passing `double` values
- Calling C from Lisp
- Calling Lisp from C

`lispobj`: C Type Specifier to Represent Lisp Data Objects

In addition to the type specifiers defined in ANSI C, Symbolics C supports a new type specifier called `lispobj`. By declaring a variable a `lispobj`, you can represent any Lisp data object. You can form arrays of `lispobjs` and declare `lispobj` functions.

The only valid operations on objects of the `lispobj` type are assignment and parameter passing; you cannot read, write, or compare `lispobjs`. You cannot coerce a `lispobj` into another type, nor can you coerce another type into a `lispobj`.

`lisp`: C Function Directive

Symbolics C supports a new function directive, `lisp`, allowing you to declare a Lisp function that you can call from a C function.

Note: The function directive is similar to a macro in that when a directive changes, you must recompile all callers of the function defined in the directive.

The format for declaring a Lisp function in a C function is:

`lisp` [*"Lisp-function"*] *output-type c-name* ([*parameter-list*]);

0.0.41. Example of Function Declaration

For example, to declare a Lisp function `length` for use with a C function, you might use:

```
lisp "global:length" int length (lispobj list);
```

This specifies the routine directive `lisp` and declares the function whose Lisp name is **global:length**, where **global** is the package name. It assigns the type `int` to any value returned from **global:length**, specifies the C name for the Lisp routine, and names a parameter value `list` of type `lispobj`.

Note that you use a lisp directive much like you use a macro. When a change is made to the directive, you must recompile all callers.

0.0.42. Description of Fields

Lisp-function

A string argument that is the name of the Lisp function you are declaring. This string argument reflects Lisp naming convention, rather than C convention in determining the name of the Lisp function. So that, unlike C, which is a case-sensitive language, you can use "global:length" or "GLOBAL:LENGTH" to specify the same Lisp function.

When the C name for the Lisp routine is identical to the Lisp function name, you do not have to specify the string argument. If the string argument is not specified, the Lisp function must have the same name as *c-name* and the same package as the C file. Do not compile C in a package that inherits from **global** or **Slc**.

For further information: See the section "How to Use the Lisp-function Field".

type

The C type of any return value from the function specified by *lisp-routine*.

c-name

Specifies the name you want to use in your C code to call the Lisp routine. Make sure *c-name* is a valid C identifier.

Note that you can specify only one C name per directive. Each Lisp function that you call from C requires its own declaration statement.

parameter-list

Specifies a C parameter list, which can include the types and names of the parameters. All C data types are permitted, including `lispobj`. You have to pass these scalar types by value to Lisp, *not* by reference. See the section "Passing Double Values in C".

Structure parameters are passed either by value or by reference. Note that aggregate objects do not necessarily have one element per Lisp array word. Also, arrays of `doubles` are stored unpacked.

How to Use the Lisp-function Field

You use the optional Lisp-function field in the case where the Lisp function

1. Is not in the same package as that specified in the attribute line of the C source file.

2. Has a name that uses a character that is illegal in a C identifier.

0.0.43. Case Sensitivity

If you define a function in the default **c-user** package such as:

```
(defun c-user::|my-lisp-proc| (arg1 arg2) ...)
```

and the package into which you are compiling C routines is the **c-user** package, then

```
lisp "|my-lisp-proc|" void my lisp proc();
```

is a correct declaration. On the other hand, if the Lisp routine is defined as:

```
(defun c-user::my-lisp-proc (arg1 arg2) ...)
```

you can use the `lisp` storage-class-specifier as follows:

```
lisp "MY-LISP-PROC" void my lisp proc();

or

lisp "my-lisp-proc" void my lisp proc();
```

As a final example, if you define two Lisp routines as

```
(defun c-user::|my lisp proc| (arg1 arg2) ...)
(defun c-user::|MY LISP PROC| (arg1 arg2) ...)
```

the C declarations

```
lisp void my lisp proc();
lisp void MY LISP PROC();
```

establish proper linkage to the two different Lisp routines.

Passing Double Values in C

Values of the type `double` are represented as boxed numbers in the Genera environment, and as unboxed numbers in C. C procedures expect to receive data of type `double` in unboxed form. They also return their results unboxed.

Example:

```
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: USER; Base: 10 -*-
```

```
(defun c-user::dsr (double-hi double-lo)
  (declare (values double-hi double-lo))
  (si:with-double-components
    ((let ((to-be-squared (si:%make-double double-hi double-lo)))
       (* to-be-squared to-be-squared))
     ret-hi ret-lo)
    (values ret-hi ret-lo)))

(defun c-square-and-add (i j)
  (declare (values double-precision-float))
  (multiple-value-bind (ret-hi ret-lo)
      (c-user::|dsquare and add| i j)
    (si:%make-double ret-hi ret-lo)))
```

```
/*-*- Mode: C; Package: C-USER -*- */

lisp double DSR(double to_be_quartered);

double dsquare_and_add(int i, int j)
{
   return DSR(i) + j;
}
```

Passing Structures Between C and Lisp

There are three cases you should concern yourself with when passing structures from C to Lisp:

1. Structures of size 1 word or less.

2. Structures of size 2 words or less.

3. Structures of a size greater than 2 words.

In the first case, structures are passed by value and appear as a single Lisp argument. In the second case, structures are passed by value and appear as two consecutive Lisp arguments. In the third case, structures are passed by reference. For these cases, the Symbolics C compiler guarantees preservation of by-value semantics for C routines. EXTERN functions defined in Lisp should avoid violating structure by-value semantics.

The following examples show how to access the various fields of a structure for each of these cases.

0.0.44. Example 1: Structures of Size One Word or Less

For

```
struct ts { int f1; } s = { 1 };
extern int add one to f1(struct ts s);
```

you can

```
(defun |add one to f1| (s) (1+ s))
```

0.0.45. Example 2: Structures of Size Two Words or Less

For

```
struct ts { int f1, f2; } s = { 1 };
extern int add one to f1 plus f2(struct ts s);
```

you can

```
(defun |add one to f1 plus f2| (s1 s2) (+ s1 s2 1))
```

0.0.46. Example 3: Structures of A Size Greater than Two Words

For

```
struct ts { int f1, f2, f3; } s = { 1 };
extern int add one to f1 plus f2 plus f3(struct ts s);
```

you can

```
  (defun |add one to f1 plus f2 plus f3| (c-array c-offset)
    (let ((lisp-offset (rot c-offset -2)))
      (+ (aref c-array lisp-offset)
         (aref c-array (+ lisp-offset 1))
         (aref c-array (+ lisp-offset 2))
         1)))
```

Returning Structures in C

Calling Lisp From C

A Lisp function can call a C main program directly.

The following program passes a C array to a Lisp function **listarray**, returning a list of the elements. It then calls the Lisp function **reverse** to reverse the `lispobj` list returned, and finally calls the Lisp function **print** displaying the result on the console:

```
/*-*- Mode: C; Package: C-USER -*- */
```

```
void revarray()
{
  #define ASIZE 10

  int a[ASIZE], i;

  extern lispobj listarray(int a[], int count);
  lisp "global:reverse" lispobj reverse(lispobj list);
  lisp "global:print" void print(lispobj anyobj);

  for (i = 0; i < 10; i++) a[i] = i;
  print(reverse(listarray(a, ASIZE)));
}
```

The Lisp code:

```
;;; -*- Mode: LISP; Syntax: Common-Lisp; Base: 10; Package: USER; Lowercase: Yes-*-

(defun c-user::|listarray| (array-pointer-object array-object-byte-offset count)
   (loop for i from (floor array-object-byte-offset 4)
         for count from count above 0
         collect (aref array-pointer-object i)))
```

Calling C From Lisp

To call a C function from Lisp, you place a C function called from Lisp on the call tree of a C function invoked by **c-sys:execute** if that function has any static data needing initialization. Data initialization occurs when the main program is executes.

Static data includes:

1. Any data of static storage duration defined by or referenced in the C function.

2. Any string literals referenced in the C function.

Passing Arguments to a C Main Program in Argc, Argv Format

You can use the function **c-sys:build-expanded-argument-list** for translating Lisp strings into the argc, argv format needed by C main programs. This function is useful when calling the C program by **c-sys:execute**.

c-sys:build-expanded-argument-list takes two arguments. The first argument is a string naming the C program; this is the same as the :Program Name keyword to the Execute C Function command. The second argument is a list of Lisp strings corresponding to argv strings. **c-sys:build-expanded-argument-list** converts this list into the corresponding argc, argv pair, and returns two values: argc and argv.

Porting C Programs

Overview of Porting C Programs

This chapter discusses considerations in porting programs developed on other compilers to the 3600-series of machines for use in the Genera environment. In particular, it describes some effects of run-time data type-checking, the treatment of uninitialized variables in the Genera environment, and presents a table showing the size of language data types in this implementation.

Run-time Data Type-Checking in C

Run-time data type-checking is the most noticeable difference if you are accustomed to conventional untagged architectures. Operations that are meaningless, but performed undetected in conventional hardware, are trapped by the Symbolics 3600-series machine.

Attempts at pointer operations on non-pointer values in C are trapped by the tagged hardware. In addition, references via pointers are checked to ensure that the access is restricted to the allocated object and does not corrupt storage.

Uninitialized Variables in C

In Genera, all variables start out with the distinguished value "undefined" unless explicitly initialized or assigned. You cannot write or coerce an undefined value. You can also initialize data to undefined values, which is true for both the I/O case and the non-I/O case. Signalling an error, in such a case, is preferable to picking up a random machine-dependent value, and actually eases the porting process.

Size and Alignment of Symbolics C Language Data Types

You can use the information in this section is for porting C applications. If you have an existing C application, you can use this information to access the data it contains from Lisp. You can convert C data into Lisp objects.

Table ! shows the sizes and alignments of C language data types.

All C structures are allocated in **sys:art-q** arrays. Each element is a 32-bit Lisp word. The alignment column below shows how the various C data types are aligned within the 32-bit words.

Bit fields require bit alignment. A bit field length specifier of 0 forces alignment to the nearest 8-bit boundary.

Pointers and integers are different sizes on the Symbolics 3600 series. Pointers are represented as an [array-object, index] pair and are two words in length; integers are one word long.

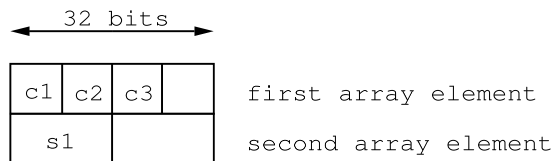
Type	Size	Alignment
char	8 bits	8-bit boundary
long	1 word	8-bit boundary
int	1 word	1-word boundary
short	16 bits	16-bit boundary
float	1 word	1-word boundary
double	2 words	1-word boundary
pointer	2 words	1-word boundary

Table 1. Sizes and Alignments of C Data Types

Here we give examples of how you can pack structures into arrays. We define one structure as follows:

```
struct {
  char c1;
  char c2;
  char c3;
  short s1;
};
```

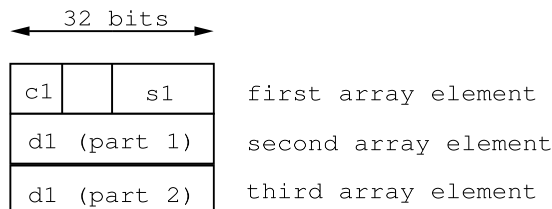
The array representing that structure is:



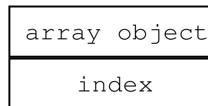
We define another structure as follows:

```
struct {
  char c1;
  short s1;
  double d1;
};
```

The array representing that structure is:



As mentioned earlier, an [array-object, index] pair represents a pointer and occupies two words:



You can access a pointer from Lisp by defining a function which takes two arguments, array-object and index.

Symbolics C Run-time Libraries

Overview of the Run-time Libraries

This section lists and describes run-time routines defined in the Symbolics C Run-time Library. These routines conform to the ANSI standard definition of C.

The routines are grouped into these libraries:

assert.h	math.h	stdio.h
ctype.	setjmp.h	stdlib.h
float.h	signal.h	string.h
limits.h	stdarg.h	time.h
locale.h	stddef.h	

The routines are grouped alphabetically by library, and are described in this form:

Synopsis:

```
#include <commonlib.h>
type routine-name(parameters);
```

Description: What the routine does, what it is used for.

Returns: Values returned.

The on-line reference, *C: A Reference Manual*, also describes C run-time library routines. Use these as a resource for additional description and discussion of these routines, in particular, for a comparison of ANSI and non-ANSI implementations. Note that the routine descriptions in the Symbolics C Run-time Library take precedence in describing specifics of the Symoblics C run-time libraries.

Run-time Library Table

The following table is an alphabetical list of run-time routines and the libraries where they are located.

Function	Library	Purpose
abort	stdlib.h	environment function
abs	stdlib.h	integer arithmetic function
acos	math.h	trigonometric function
asctime	time.h	date and time function
asin	math.h	trigonometric function
assert	assert.h	diagnostic function
atan	math.h	trigonometric function
atan2	math.h	trigonometric function
atexit	stdlib.h	environment function
atof	stdlib.h	string conversion function
atoi	stdlib.h	string conversion function
atol	stdlib.h	string conversion function
bsearch	stdlib.h	searching and sorting function
calloc	stdlib.h	memory management function
ceil	math.h	exponential and logarithmic function
clearerr	stdio.h	I/O function
clock	time.h	date and time function
cos	math.h	trigonometric function
cosh	math.h	trigonometric function
ctime	time.h	date and time function
difftime	time.h	date and time function
exit	stdlib.h	environment function
exp	math.h	exponential and logarithmic function
fabs	math.h	exponential and logarithmic function
fclose	stdio.h	I/O function
feof	stdio.h	I/O function
ferror	stdio.h	I/O function
fflush	stdio.h	I/O function
fgetc	stdio.h	I/O function
fgetpos	stdio.h	I/O function
fgets	stdio.h	I/O function
floor	math.h	exponential and logarithmic function

fmod	math.h	exponential and logarithmic function
fopen	stdio.h	I/O function
fprintf	stdio.h	I/O function
fputc	stdio.h	I/O function
fputs	stdio.h	I/O function
fread	stdio.h	I/O function
free	stdlib.h	memory management function
freopen	stdio.h	I/O function
frexp	math.h	exponential and logarithmic function
fscanf	stdio.h	I/O function
fseek	stdio.h	I/O function
fsetpos	stdio.h	I/O function
ftell	stdio.h	I/O function
fwrite	stdio.h	I/O function
getc	stdio.h	I/O function
getchar	stdio.h	I/O function
getenv	stdlib.h	environment function
gets	stdio.h	I/O function
gmtime	time.h	date and time function
isalnum	ctype.h	character processing function
isalpha	ctype.h	character processing function
iscntrl	ctype.h	character processing function
isdigit	ctype.h	character processing function
isgraph	ctype.h	character processing function
islower	ctype.h	character processing function
isprint	ctype.h	character processing function
ispunct	ctype.h	character processing function
isspace	ctype.h	character processing function
isupper	ctype.h	character processing function
isxdigit	ctype.h	character processing function
labs	stdlib.h	integer arithmetic function
ldexp	math.h	exponential and logarithmic function
ldiv	stdlib.h	integer arithmetic function

localtime	time.h	date and time function
log10	math.h	exponential and logarithmic function
longjmp	setjmp.h	non-local jump function
malloc	stdlib.h	memory management function
memchr	string.h	string handling function
memcmp	string.h	string handling function
memcpy	string.h	string handling function
memmove	string.h	string handling function
memset	string.h	string handling function
mktime	time.h	date and time function
modf	math.h	exponential and logarithmic function
perror	stdio.h	I/O function
pow	math.h	exponential and logarithmic function
printf	stdio.h	I/O function
putc	stdio.h	I/O function
putchar	stdio.h	I/O functions
puts	stdio.h	I/O function
qsort	stdlib.h	searching and sorting function
raise	signal.h	signal handling function
rand	stdlib.h	pseudo-random number generator
realloc	stdlib.h	memory management function
remove	stdio.h	I/O function
rename	stdio.h	I/O function
rewind	stdio.h	I/O function
scanf	stdio.h	I/O function
setbuf	stdio.h	I/O function
setjmp	setjmp.h	non-local jump function
setlocale	locale.h	locale parameters function
setvbuf	stdio.h	I/O function
signal	signal.h	signal handling function
sin	math.h	trigonometric function
sinh	math.h	trigonometric function
sprintf	stdio.h	I/O function

sqrt	math.h	exponential and logarithmic function
srand	stdlib.h	pseudo-random number generator
sscanf	stdio.h	I/O function
strcat	string.h	string handling function
strchr	string.h	string handling function
strcmp	string.h	string handling function
strcoll	string.h	string handling function
strcpy	string.h	string handling function
strcspn	string.h	string handling function
strerror	string.h	string handling function
strftime	time.h	date and time function
strlen	string.h	string handling function
strncat	string.h	string handling function
strncmp	string.h	string handling function
strncpy	string.h	string handling function
strpbrk	string.h	string handling function
strrchr	string.h	string handling function
strspn	string.h	string handling function
strstr	string.h	string handling function
strtod	stdlib.h	string conversion function
strtok	string.h	string handling function
strtol	stdlib.h	string conversion function
strtoul	stdlib.h	string conversion function
strtou	stdio.h	I/O function
system	stdlib.h	environment function
tan	math.h	trigonometric function
tanh	math.h	trigonometric function
time	time.h	date and time function
tmpfile	stdio.h	I/O function
tmpnam	stdio.h	I/O function
tolower	ctype.h	character processing function
toupper	ctype.h	character processing function
ungetc	stdio.h	I/O function

va_arg	stdarg.h	variable arguments list function
va_end	stdarg.h	variable arguments lists
va_start	stdarg.h	variable arguments list function
vfprintf	stdio.h	I/O function
vprintf	stdio.h	I/O function
vsprintf	stdio.h	I/O function

The limits.h library

The limits provided in <limits.h> for Symbolics C are as follows:

CHAR BIT	8	width in number of bits for the type char
SCHAR MIN	-127	minimum value of a signed char
SCHAR MAX	+127	maximum value of a signed char
UCHAR MAX	255U	maximum value of an unsigned char
CHAR MIN	0	minimum value of a char
CHAR MAX	UCHAR MAX	maximum value of a char
SHRT MIN	-32767	minimum value of a short int
SHRT MAX	+32767	maximum value of a short int
USHRT MAX	65535U	maximum value of an unsigned short
INT MIN	-2147483648	minimum value of an int
INT MAX	+2147483647	maximum value of an int
UINT MAX	4294967295U	maximum value of an unsigned int
LONG MIN	-2147483648	minimum value of a long int
LONG MAX	+2147483647	maximum value of a long int
ULONG MAX	4294967295U	maximum value of an unsigned long

The float.h Library

The type characteristics of floating-point types defined in <**float.h**> are as follows:

```
radix of exponent representation for all floating types
 FLT RADIX          +2

addition rounds
 FLT ROUNDS        +1

number of (base-FLT RADIX) digits in the floating-point mantissa
 FLT MANT DIG       +24
 DBL MANT DIG       +53
 LDBL MANT DIG      +53

minimum x>0.0 such that 1.0+x !=1.0
 FLT EPSILON        +5.960465E-8F
```

```
 DBL EPSILON        +1.1102230246251568E-16
 LDBL EPSILON       +1.1102230246251568E-16

number of decimal digits of precision
 FLT DIG            +6
 DBL DIG            +15
 LDBL DIG           +15
```

maximum x such that 2^{x-1} approximates FLT MAX

```
 FLT MAX EXP        +128
 DBL MAX EXP        +1024
 LDBL MAX EXP       +1024
```

minimum x such that 2^{x-1} approximates FLT MIN

```
 FLT MIN EXP        -125
 DBL MIN EXP        -1021
 LDBL MIN EXP       -1021

maximum representable finite number
 FLT MAX            +3.4028235e38F
 DBL MAX            +1.7976931348623157E308
 LDBL MAX           +1.7976931348623157E308

minimum normalized positive number
 FLT MIN            +1.1754944e-38F
 DBL MIN            +2.2250738585072014E-308
 LDBL MIN           +2.2250738585072014E-308

maximum x such that 1Ex approximates FLT MAX
 FLT MAX 10 EXP     +38
 DBL MAX 10 EXP     +308
 LDBL MAX 10 EXP    +308

minimum x such that 1Ex approximates FLT MIN
 FLT MIN 10 EXP     -37
 DBL MIN 10 EXP     -307
 LDBL MIN 10 EXP    -307
```

The stddef.h Library

This header file defines these types and macros:

`ptrdiff t` The type that is the result of subtracting two pointers.

`size t` The type that is the result of the `sizeof` operator.

`NULL` A macro that defines the value of the null pointer constant.

`offsetof(type, identifier)` Describes in bytes the offset of `identifier` in `type`.

`errno` A macro that returns an indicator for error conditions. When you first run a program, the value of the indicator is set to zero. During the course of a program, various library functions can set the value to a positive. Because a call to a library function cannot set the value of `errno` to zero, the program has to reset the indicator to zero before a call, and inspect it before subsequent calls.

The assert.h library

The assert Macro

Synopsis: `#include <assert.h>   void assert(int expression);`

Description: Tests whether an `expression` is true or false at that point in the program. If the expression evaluates to false, this macro writes information to the standard error file for the faulty call, using this format:

```
**** C assertion assert expr string failed at
  line assert line in file assert file
```

It then calls the **abort** function.

Note: The assert facility disables when the NDEBUG macro is defined in the header file <assert.h>.

Returns: Returns no value.

The ctype.h Library

The isalnum Function

Synopsis: `#include <ctype.h>    int isalnum(int c);`

Description: Tests whether `c` is an alphanumeric character. An alphanumeric character is one of:

The digits 0 through 9
The letters a through z
The letters A through Z

Returns: Returns a nonzero value if this condition is true, otherwise, returns a value of zero.

The isalpha Function

Synopsis: #include <ctype.h>   int isalpha(int c);

Description: Tests whether c is an alphabetic character. An alphabetic character is either a through z or A through Z.

Returns: Returns a nonzero value if the condition is true, otherwise, returns a value of zero.

The iscntrl Function

Synopsis: #include <ctype.h>   int iscntrl(int c);

Description: Tests whether c is a control character.

Returns: Returns a nonzero value if the condition is true, otherwise, returns a value of zero.

The isdigit Function

Synopsis: #include <ctype.h>   int isdigit(int c);

Description: Tests whether c is a decimal digit character.

Returns: Returns a nonzero value if the condition is true, otherwise, returns a value of zero.

The isgraph Function

Synopsis: #include <ctype.h>   int isgraph(int c);

Description: Tests whether c is a graphic character. A graphic character is any printing character with the exception of the space (‘ ’) character.

Returns: Returns a nonzero value if the condition is true, otherwise, returns a value of zero.

The islower Function

Synopsis: `#include <ctype.h>    int islower(int c);`

Description: Tests whether `c` is a lowercase alphabetic character.

Returns: Returns a nonzero value if the condition is true, otherwise, returns a value of zero.

The isprint Function

Synopsis: `#include <ctype.h>    int isprint(int c);`

Description: Tests whether `c` is a printing character, including the space (' ') character.

Returns: Returns a nonzero value if the condition is true, otherwise, returns a value of zero.

The ispunct Function

Synopsis: `#include <ctype.h>    int ispunct(int c);`

Description: Tests whether `c` is a punctuation character.

Returns: Returns a nonzero value if the condition is true, otherwise, returns a value of zero.

The isspace Function

Synopsis: `#include <ctype.h>    int isspace(int c);`

Description: Tests whether `c` is a whitespace character. A whitespace character is one of:

space (' ')
form feed ('\f')
newline ('\n')
carriage return ('\r')
horizontal tab ('\t')
vertical tab ('\v')

Returns: Returns a nonzero value if the condition is true, otherwise, returns a value of zero.

The isupper Function

Synopsis: `#include <ctype.h>    int isupper(int c);`

Description: Tests whether `c` is an uppercase letter.

Returns: Returns a nonzero value if the condition is true, otherwise, returns a value of zero.

The isxdigit Function

Synopsis: `#include <ctype.h    int isxdigit(int c);`

Description: Tests whether `c` is a hexadecimal digit character.

Returns: Returns a nonzero value if the condition is true, otherwise, returns a value of zero.

The tolower Function

Synopsis: `#include <ctype.h>    int tolower(int c);`

Description: Converts `c` from an uppercase letter to a lowercase letter.

Returns: Returns the corresponding lowercase letter. If there is no corresponding lowercase letter, the argument `c` is returned unchanged.

The toupper Function

Synopsis: `#include <ctype.h>    int toupper(int c);`

Description: Converts `c` from a lowercase letter to an uppercase letter.

Returns: Returns the corresponding uppercase letter. If there is no corresponding uppercase letter, the argument `c` is returned unchanged.

The locale.h Library

The `locale.h` library enables you to change the way certain functions behave, based on local conventions of language and culture. It is the hook for targeting a program to users in one or more countries in the international community.

This library includes the setlocale function and the following macros, all of which are used as the category argument to setlocale:

LC ALL
LC COLLATE
LC CTYPE
LC NUMERIC
LC TIME

The setlocale Function

Synopsis: #include <locale.h> char *setlocale(int category, const char *locale);

Description: Selects the locale of the program, according to the category and locale arguments. The category argument indicates which portion of the program's current locale is changed or queried.

category value	Affects
LC ALL	program's entire current locale
LC COLLATE	strcoll
LC TYPE	character-handling functions
LC NUMERIC	the decimal-point character for I/O and string conversion functions
LC TIME	strftime

The locale argument specifies the desired locale. A null pointer for locale simply queries for the current locale, without changing it. Other values for locale request to change the current locale. A value of "C" for locale specifies the minimal environment for C translation; this is the English C locale. A value of "" for locale specifies the implementation-defined native environment. You can also make this argument implementation-defined string values.

Returns: If a pointer to a string is given for locale and the selection is honored, this function returns the string specifying the category for the new locale. If the selection is not supported, a null pointer is returned, and the program's locale remains unchanged.

If a null pointer is given for locale, this function returns the string associated with the category of the program's current locale, and the locale remains unchanged.

Note: If you specify a category other than LC ALL, the specific subcategory is defined, but the overall category LC ALL is undefined. In this case, an inquiry for the category LC ALL results in a null pointer being returned.

Note: Symbolics C supports two locales: the minimal locale and the C locale (which are the same in this implementation).

The math.h Library

The acos Function

Synopsis: #include <math.h>   double acos(double x);

Description: Computes the principal value of the arc cosine of x. If an argument is not in the range [-1,1], a domain error occurs.

Returns: Returns the arc cosine in the range [0, π].

The asin Function

Synopsis: #include <math.h>   double asin(double x);

Description: Computes the principal value of the arc sine of x. If an argument is not in the range [-1,1], a domain error occurs.

Returns: Returns the arc sine in the range [-π/2, π/2].

The atan Function

Synopsis: #include <math.h>   double atan(double x);

Description: Computes the principal value of the arc tangent of x.

Returns: Returns the arc tangent in the range [-π/2, π/2].

The atan2 Function

Synopsis: #include <math.h>   double atan2(double y, double x);

Description: Computes the principal value of the arc tangent of the value y/x. The function uses the signs of both arguments in determining the quadrant for the return value. The value is a floating-point value. If both arguments are zero, a domain error occurs.

Returns: Returns a result in radians for the arc tangent of y/x whose value is in the range [-π, π].

The ceil Function

Synopsis: #include <math.h> double ceil(double x);

Description: Computes the smallest integer not less than x.

Returns: Returns the smallest integer not less than x, where x is expressed as a double type.

The cos Function

Synopsis: #include <math.h> double cos(double x);

Description: Computes the cosine of x (measured in radians).

Returns: Returns the cosine value. In cases where you give a large magnitude argument, the result is of little or no significance.

The cosh Function

Synopsis: #include <math.h> double cosh(double x);

Description: Computes the hyperbolic cosine of x. If the magnitude of x is too large, a range error occurs.

Returns: Returns the hyperbolic cosine value.

The exp Function

Synopsis: #include <math.h> double exp(double x);

Description: Computes the exponential function of x. If the magnitude of x is too large, a range error occurs.

Returns: Returns the exponential value.

The fabs Function

Synopsis: #include <math.h> double fabs(double x);

Description: Computes the absolute value of floating-point number x.

Returns: Returns the absolute value of x.

The floor Function

Synopsis: #include <math.h> double floor(double x);

Description: Computes the largest integer not greater than `x`.

Returns: Returns the absolute value of `x`.

The fmod Function

Synopsis: `#include <math.h>    double fmod(double x, double y);`

Description: Computes the floating-point remainder of `x/y`.

Returns: Returns the remainder of `x/y` when `y` is nonzero. Returns zero if `y` is zero.

The frexp Function

Synopsis: `#include <math.h>    double frexp(double value, int *exp);`

Description: Breaks a floating-point number into an integral power of 2 and a normalized fraction. The integer is stored in the `int` object pointed to by `exp`.

Returns: Returns the value `x`. The value is a double with magnitude in the interval [1/2, 1] or zero and is equal to `x` times 2 raised to the power `*exp`. If value is zero, both parts of the result are zero.

The ldexp Function

Synopsis: `#include <math.h>    double ldexp(double x, int exp);`

Description: Multiplies a floating-point number by an integral power of 2. A range error may occur.

Returns: Returns the value of `x` times 2 raised to the power `exp`.

The ldiv Function

Synopsis: `#include <stdlib.h>   ldiv t ldiv(long int numer, long int denom);`

Description: Computes the quotient and remainder of the division of `numer` by `denom`.

Returns: Returns a structure of type `ldiv t`. The type `ldiv t` is a structure whose members are named `quot` and `rem` and that contains `long int` members.

The log Function

Synopsis: `#include <math.h>    double log(double x);`

Description: Computes the natural logarithm of x. If the argument is negative, a domain error occurs. If the argument is zero, a range error occurs.

Returns: Returns the natural logarithm.

The log10 Function

Synopsis: #include <math.h>   double log10(double x);

Description: Computes the base-ten logarithm of x. If the argument is negative, a domain error occurs. If the argument is zero, a range error occurs.

Returns: Returns the base-ten logarithm.

The modf Function

Synopsis: #include <math.h>   double modf(double value, double *iptr);

Description: Breaks the argument value into fractional and integral parts, each of which has the same sign as the argument. It stores the integral part as a double in the object pointed to by iptr.

Returns: Returns the signed fractional part of value.

The pow Function

Synopsis: #include <math.h>   double pow(double x, double y);

Description: Computes x raised to the power y. If x is zero and y is less than or equal to zero, or if x is negative and y is not an integer, a domain error occurs. A range error may occur.

Returns: Returns the value of x raised to the power of y.

The sin Function

Synopsis: #include <math.h> double sin(double x);

Description: Computes the sine of x (measured in radians). A result of little or no significance is returned from a large magnitude argument.

Returns: Returns the sine value.

The sinh Function

Synopsis: #include <math.h>   double sinh(double x);

Description: Computes the hyperbolic sine of `x`. If the magnitude of `x` is too large, a range error occurs.

Returns: Returns the hyperbolic sine value.

The sqrt Function

Synopsis: `#include <math.h>     double sqrt(double x);`

Description: Computes the non-negative square root of `x`. If the argument is negative, a domain error occurs.

Returns: Returns the value of the square root.

The tan Function

Synopsis: `#include <math.h>     double tan(double x);`

Description: Computes the tangent of `x` (measured in radians). A result of little or no significance is returned from a large magnitude argument.

Returns: Returns the tangent value.

The tanh Function

Synopsis: `#include <math.h>     double tanh(double x);`

Description: Computes the hyperbolic tangent of `x`.

Returns: Returns the hyperbolic tangent value.

The setjmp.h Library

The setjmp Function

Synopsis: `#include <setjmp.h>   int setjmp(jmp buf env);`

Description: Saves the calling environment in `env`, the "jump buffer."

Returns: Returns zero, if called directly. Returns 1, if called from **longjmp**.

The longjmp function

Synopsis: `#include <setjmp.h>   void longjmp(jmp buf env, int val);`

Description: Restores the environment saved in `env` with **setjmp**.

Returns: Returns the value of `val`.

The signal.h Library

This header file defines several macros and one type. The type is:

`sig atomic t` Identifies an object that you can modify atomically in the presence of in the presence of asynchronous interrupts.

The macros are:

Macro	Meaning
SIG_DFL	Passed as second argument to `signal`; requests default handling for the signal.
SIG_ERR	Possible returned value for `signal`; indicates that the signal was not generated.
SIG_IGN	Passed as second argument to `signal`; requests that the signal be ignored.

Macro	Action	Meaning
SIGABRT	The program is aborted.	abort signal
SIGFPE	The Debugger is entered.	floating-point exception signal
SIGILL	The program is aborted.	illegal instruction signal
SIGINT	The Debugger is entered.	interrupt signal
SIGSEGV	The Debugger is entered.	segment violation signal
SIGTERM	The program is aborted.	termination signal

The raise Function

Synopsis: `#include <signal.h>   int raise(int sig);`

Description: Sends the signal `sig` to the executing program.

Returns: Returns zero if successful, otherwise, returns nonzero.

The signal Function

Synopsis: `#include <signal.h> void (*signal(int sig, void (*func)(int)))(int);`

Description: Specifies the way the signal number `sig` is handled when it is raised. If the value of `func` is SIG_DFL, default handling for that signal occurs. If the value of `func` is SIG_IGN, the signal is ignored. If the value of `func` is other than these, `func` acts as a pointer to a function called when that signal occurs.

Returns: If the request is successful, returns the value of `func` for the previous call to `signal` for the signal named by `sig`. Otherwise, a value of SIG_ERR returns and `errno` is set to indicate an error.

The stdarg.h Library

Type Declaration in the <stdarg.h> Header File

This header declares the type `va list`.

va_list An array type used to declare the local state variable `ap`, used by functions in the stdarg.h library to traverse parameters.

The va_arg Macro

Synopsis: `#include <stdarg.h>` *type* `va arg(va list ap,` *type*`);`

Description: Expands to an expression that has the value and type of the next argument in the call. The internal argument pointer `ap` moves to the next argument, if any exist.

Returns: Returns the argument that follows *parmN* (represented in function variable lists by: ...). Successive calls return successive arguments from this list.

The va_end Function

Synopsis: `#include <stdarg.h> void va end(va list ap);`

Description: Performs clean-up operations after all arguments are read by **va_arg**.

Returns: Returns no value.

The va_start Macro

Synopsis: `#include <stdarg.h>` void va_start(va_list `ap`, *parmN*);

Description: Initializes `ap`, the va_list variable. *ParmN* is the argument for the name of the right-most parameter in the variable parameter list for a function. Use **va_start** before any calls to **va_arg** or **va_end**.

Returns: Returns no value.

The stdio.h Library

Macros and Types Declared in the <stdio.h> Header

The following macros and types are declared in the <stdio.h> header:

`FILE` A type used to record the information to control a stream.

`fpos t` A type used to record the information that uniquely specifies every position within a file. For more information, see the section "The **fgetpos** Function".

`IOFBF, IOLBF, IONBF` Macros representing full buffering, line buffering, and no buffering, respectively. See the section "The **setvbuf** Function".

`BUFSIZE` Macro representing the size of the buffer used by the **setbuf** function.

`EOF` Represents a negative integral constant.

`L tmpnam` Represents the size of a file name array. See the section "The **tmpnam** Function".

`SEEK CUR` Represents the current file position.

`SEEK END` Represents the end of file.

`SEEK SET` Represents the start of file.

`OPEN MAX` Represents the minimum number of simultaneously open files.

`TMP MAX` Macro representing the number of times successive calls to **tmpnam** generates unique names. See the section "The **tmpnam** Function".

Streams:

```
stderr The output stream that receives error messages
stdin  The stream for normal input
stdout The stream for normal output
```

These streams are intialized before the start of an application program.

The clearerr Function

Synopsis: `#include <stdio.h>    void clearerr(FILE *stream);`

Description: Clears the end-of-file and error indicators for the stream being pointed to.

Returns: Returns no value.

The fclose Function

Synopsis: `#include <stdio.h>    int fclose(FILE *stream);`

Description: Closes the specified file and disassociates it from the specified stream object. In the Genera environment, performs any equivalent **:finish** operations required on the file descriptor and sends the Genera stream object the **:finish** message. It then sends the stream object the **:close** message and marks the file descriptor as closed. For an input stream, the **:finish** operations are omitted.

Returns: Returns zero if the file is closed successfully. Returns a nonzero value if an error returns or if the file is closed when you call this function.

The feof Function

Synopsis: `#include <stdio.h>    int feof(FILE *stream);`

Description: Tests for an end-of-file indicator for the stream to which you are pointing. End-of-file is indicated when an attempt is made to read beyond the last character.

Returns: Returns a nonzero value if the end-of-file indicator for the stream is found. Otherwise, returns zero.

The ferror Function

Synopsis: `#include <stdio.h>    int ferror(FILE *stream);`

Description: Tests for an error indicator for the stream to which you are pointing and returns its status. Use the **clearerr** or **fclose** function to reset the error indicator.

Returns: Returns a nonzero value if the error indicator is set, indicating that an error occurred while writing to or reading from the stream. Otherwise, returns zero.

The fflush Function

Synopsis: #include <stdio.h>   int fflush(FILE *stream);

Description: For an output or update stream, finalizes file content by writing to file any unwritten data for stream. That is, it performs any equivalent **:finish** operations required on the file descriptor and sends the Genera stream object the **:finish** message. Performs any internal file descriptor bookkeeping required to note that no pending data transfers to the stream are outstanding.

For an input or update stream, undoes the effect of a previous **ungetc** operation on the file descriptor or stream.

Returns: Returns a nonzero value if a write error occurs. Otherwise, returns zero.

The fgetc Function

Synopsis: #include <stdio.h>   int fgetc(FILE *stream);

Description: At the point indicated by the file position indicator, reads the next character from stream, returns the value of that character as an int, and advances to the next character in the stream. Use the **feof** facility to determine if the end-of-file is reached. See the section "The **getc** Function". See the section "The **getchar** Function".

Returns: Returns the value of the character as an int type. If the stream is at the end of file, returns EOF. If a read error occurs, returns EOF and sets the error indicator.

The fgetpos Function

Synopsis: #include <stdio.h>   int fgetpos(FILE *stream, fpos t *pos);

Description: Gets the current value of the file position indicator for stream and returns it as a value that can be used by **fsetpos**.

Returns: If successful, returns zero. If unsuccessful, returns a nonzero value and sets **errno**.

The fgets Function

Synopsis: `#include <stdio.h>    char *fgets(char *s, int n, FILE *stream);`

Description: Reads `n - 1` characters from the stream `s` into `stream`. Characters are read in at the point specified by `FILE`, if you set `FILE`. If the specified string contains a newline character or an end-of-file, the function stops reading the string. The function appends a null character to follow the last character read into the array `stream`.

Unlike the **gets** function, newline characters are read into the array.

Returns: Returns `s`, if successful. Returns a null pointer if no string is read, or if a read error occurs.

The fopen Function

Synopsis: `#include <stdio.h>    FILE *fopen(const char *filename, const char *mode);`

Description: Associates the file named by `filename` with a stream. The file is opened or created with the modes specified by the argument `mode`. This argument must be a character string that begins with one of the options described in the next table.

Returns: Returns a pointer to the object controlling the stream. If unsuccessful, returns a null pointer.

The following table lists the C file mode options and their effects and also maps the C file mode options into the Genera open options.

r — Open an existing text file for reading (input)

Option	*Value*
:element-type	string-char
:direct	nil
:direction	:input
:if-exists	:dont-care
:if-does-not-exist	:error

w — Create a text file for writing (output), or truncate an existing one to zero length

Option	*Value*
:element-type	string-char
:direct	nil
:direction	:output
:if-exists	:truncate

:if-does-not-exist	:create

a — Append, or open or create a text file for writing (output) at end-of-file

Option	*Value*
:element-type	string-char
:direct	nil
:direction	:output
:if-exists	:append
:if-does-not-exist	:error

v — Create a new text file for writing (output) for file systems that support file version numbers, or truncate an existing one to zero length for file systems that do not support file version numbers

Option	*Value*
:element-type	string-char
:direct	nil
:direction	:output
:if-exists	:new-version
:if-does-not-exist	:create

rb — Open an existing file for reading (input)

Option	*Value*
:element-type	(unsigned-byte 8)
:direct	t
:direction	:input
:if-exists	:dont-care
:if-does-not-exist	:error

wb — Create a binary file for writing (output) or truncate an existing one to zero length

Option	*Value*
:element-type	(unsigned-byte 8)
:direct	t
:direction	:output
:if-exists	:truncate

	:if-does-not-exist	:create

ab — Append, or open or create a text file for writing (output) at end-of-file

Option	*Value*
:element-type	(unsigned-byte 8)
:direct	t
:direction	:output
:if-exists	:append
:if-does-not-exist	:create

vb — Create a binary file for writing (output) for file systems that support file version numbers, or truncate an existing one to zero length for file systems not supporting file version numbers

Option	*Value*
:element-type	(unsigned-byte 8)
:direct	t
:direction	:output
:if-exists	:new-version
:if-does-not-exist	:create

r+ — Open an existing text file for updating (reading or writing)

Option	*Value*
:element-type	string-char
:direct	nil
:direction	:io
:if-exists	:overwrite
:if-does-not-exist	:error

w+ — Create a text file for update, or truncate an existing file to zero length for update

Option	*Value*
:element-type	string-char
:direct	nil
:direction	:io
:if-exists	:truncate

:if-does-not-exist	:create

a+ Append, or open or create a text file for writing (output) at end-of-file

Option	*Value*
:element-type	string-char
:direct	nil
:direction	:io
:if-exists	:append
:if-does-not-exist	:create

v+ Create a text file for update for file systems supporting file version numbers, or truncate an existing file to zero length for update for file systems not supporting file version numbers

Option	*Value*
:element-type	string-char
:direct	nil
:direction	:io
:if-exists	:new-version
:if-does-not-exist	:create

r+b Open an existing binary file for updating (reading or writing)

Option	*Value*
:element-type	(unsigned-byte 8)
:direct	t
:direction	:io
:if-exists	:overwrite
:if-does-not-exist	:error

w+b Create a binary file for update, or truncate an existing file to zero length for update

Option	*Value*
:element-type	(unsigned-byte 8)
:direct	t
:direction	:io
:if-exists	:truncate

:if-does-not-exist	:create

a+b — Append, or open or create a text file for writing (output) at end-of-file

Option	*Value*
:element-type	(unsigned-byte 8)
:direct	t
:direction	:io
:if-exists	:append
:if-does-not-exist	:create

v+b — Create a binary file for update for file systems supporting file version numbers, or truncate an existing file to zero length for update for file systems not supporting file version numbers

Option	*Value*
:element-type	(unsigned-byte 8)
:direct	t
:direction	:io
:if-exists	:new-version
:if-does-not-exist	:create

The fprintf Function

Synopsis:

```
#include <stdio.h>    int fprintf(FILE *stream, const char *format, ...);
```

Description: Performs output formatting and writes output to the specified `stream`. The `format` argument is the format control string and specifies how arguments are converted for output. See also: **printf, sprintf**.

Returns: Returns the number of characters sent to the output stream, if no error occurs. Returns EOF if an error occurs.

Summary of Conversion for Print Functions

The format control string can consist of text and conversion specifiers.

A conversion specification is introduced by the % character. It can consist of the following items in this order:

- A flag character, which can be one of

- -, specifying left-justification of a field.
- 0, specifying zero as the pad character.
- +, specifying that signed conversions are preceded with a + or - sign.
- space, which prepends a space if the first character of a conversion is not a + or - sign.
- #, which converts the result to one of the variant forms specified for conversion characters.

- A minimum width field, represented by a decimal integer constant.
- A precision specifier, represented by a "." character and an optional decimal integer.
- A conversion operation, represented by one of the characters described in the next table.

Conversion Characters	*Description*
d,i	signed decimal conversion
u	unsigned decimal conversion
o	unsigned octal conversion
x,X	unsigned hexadecimal conversion
c	prints as a character
s	prints as a string
p	for an argument that is a pointer to void, prints as an object pointed to followed by a byte offset
n	writes the number of characters output to an argument of type `int *;`
f	signed decimal floating-point conversion
e,E	signed decimal floating-point conversion
g,G	signed decimal point conversion
%	prints a percent sign

For a more detailed description of conversion specifiers for print functions: See the section "C-Ref: **FPRINTF, PRINTF, SPRINTF**".

The fputc Function

Synopsis: #include <stdio.h> int fputc(int c, FILE *stream);

Description: Writes the character c to the output stream pointed to by stream and advances to the next character. If the file position indicator is defined, the character is placed at that point in the stream. Otherwise, the character is appended to the stream. Use the **feof** facility in determining whether the end-of-file is reached. See the section "The **putc** Function". See the section "The **putchar** Function".

Returns: Returns the character c as an int type. If a write error occurs, returns EOF and sets the error indicator.

The fputs Function

Synopsis: #include <stdio.h> int fputs(const char *s, FILE *stream);

Description: Copies the string pointed to by s to the stream pointed to by stream. Adds a newline character to string after it is copied.

Returns: Returns zero, if successful. Otherwise, returns a nonzero value.

The fread Function

Synopsis: #include <stdio.h> size t fread(void *ptr, size t size, size t nmemb, FILE *stream);

Description: Reads up to nmemb number of objects from stream into the array pointed to by ptr. The size argument specifies the size of nmemb.

Returns: Returns the number of objects read.

The freopen Function

Synopsis: #include <stdio.h> FILE *freopen(const char *filename, const char *mode, FILE *stream);

Description: This function associates the file pointed to by *filename with the stream pointed to by *stream. It opens or creates the file with with the modes specified by the argument mode. See the section "The **fopen** Function".

Returns: If successful, returns the value of stream. Otherwise, returns a null pointer.

The fscanf Function

Synopsis: `#include <stdio.h>    int fscanf(FILE *stream, const char *format, ...);`

Description: Reads characters from `stream`, interpreting them according to the string `format`. The `format` field consists of one or more of:

- A whitespace character
- A text character
- A conversion specification

Compare **scanf** and **sscanf**.

Returns: Returns the number of characters assigned to `stream`, if successful. Otherwise, returns EOF.

Summary of Conversion Specifiers for Scan Functions

The `format` field can consist of one or more of:

- A whitespace character
- A text character
- A conversion specification

A conversion specification is introduced by the % character. It can consist of the following items:

1. An assignment suppression flag, represented by the * character.
2. A maximum field width, represented by an unsigned decimal integer greater than zero.
3. A size specification, represented by the character `h`, meaning `short`, or by the character `l`, meaning `long`.
4. A conversion operation, represented by one of the characters described in the next table.

Conversion Characters	*Description*
d	signed decimal conversion
i	signed, based integer conversion
u	unsigned decimal conversion
o	unsigned octal conversion
x,X	unsigned hexadecimal conversion

c	reads one or more characters
s	reads a string delimited by whitespace
p	converts a pointer value
n	writes out the number of characters read
f,e,E,g,G	signed decimal conversion
%	accepts a % character as input
[	scans a sequence of characters

For a detailed description of conversion specifiers for scan functions, see the section "C-Ref: **FSCANF, SCANF, SSCANF**".

The fsetpos Function

Synopsis: `#include <stdio.h>    int fsetpos(FILE *stream, const fpos t *pos);`

Description: Sets the stream's position specified by its pos argument. The function verifies that pos was obtained by a previous call to **fgetpos** on the same stream and hence verifies that the stream saved in the pos argument is **eq** to the stream specified in the stream file descriptor.

Returns: Returns zero if successful. If unsuccessful, returns a non-zero value and sets **errno**.

The fseek Function

Synopsis: `#include <stdio.h>    int fseek(FILE *stream, long int offset, int whence);`

Description: Sets the file position indicator for stream. The value of wherefrom can be on of the constants SEEK_SET, SEEK_CUR, or SEEK_END. The new file position is offset number of characters from wherefrom.

Returns: Returns a value of zero, if successful. Returns a nonzero value, if unsuccessful.

The ftell Function

Synopsis: `#include <stdio.h>    long int ftell(FILE *stream);`

Description: Gets the current value of the file position indicator for stream. You can use the value as the second argument to the **fseek** function.

Returns: If successful, returns the current value of the file position indicator. If unsuccessful, returns -1L and sets **errno**.

The fwrite Function

Synopsis: #include <stdio.h> size t fwrite(const void *ptr, size t size size t nmemb, FILE *stream);

Description: Writes up to nmemb number of objects from the array pointed to by ptr.

Returns: Returns the number of objects written. size specifies the size of nmemb.

The getc Function

Synopsis: #include <stdio.h> int getc(FILE *stream);

Description: Equivalent to **fgetc**. Reads the next character from the stream given as an argument, returns the value of that character as an int, and advances to the next character in the stream. See the section "The **fgetc** Function". See the section "The **getchar** Function".

Returns: Returns the value of the character as an int type. If the stream is at the end-of-file, returns EOF. If a read error occurs, returns EOF and sets the error indicator. Use the **feof** facility to determine if the end-of-file is reached.

The getchar Function

Synopsis: #include <stdio.h> int getchar(void);

Description: Equivalent to **fgetc**, except that the stdin stream is used as the input stream. Reads the next character from the stdin stream given as an argument, returns the value of that character as an int, and advances to the next character in the stream. Use the **feof** facility in determining whether the end-of-file is reached. See the section "The **fgetc** Function". See the section "The **getc** Function".

Returns: Returns the value of the character as an int type. Returns EOF if the stream is at end-of-file, or if a read error occurs.

The gets Function

Synopsis: #include <stdio.h> char *gets(char *s);

Description: Reads characters from the standard input stream, stdin, into the array pointed to by s. The string of characters terminates when it reaches an end-of-file or a newline character. When a newline character is read, a null character is written to the array, and the newline character is discarded.

Returns: Returns `s` if successful. Returns a null pointer if no string is read, or if a read error occurs.

The perror Function

Synopsis: `#include <stdio.h>    void perror(const char *s);`

Description: Writes information concerning errors detected with **errno**.

Returns: Returns no value.

The printf Function

Synopsis: `#include <stdio.h>    int printf(const char *format, ...);`

Description: Sends the formatted output of the string format control string `format` to the standard output stream `stdout`, and then to any additional arguments. The format control string can contain text and conversion specifiers. For a summary of conversion specifiers: See the section "The **fprintf** Function".

Returns: If no error occurs, returns the number of characters sent, otherwise, returns a negative number.

The putc Function

Synopsis: `#include <stdio.h>    int putc(int c, FILE *stream);`

Description: Equivalent to **fputc**: Writes the character `c` to the output stream pointed to by `stream`, and advances to the next character. If you defined the file position indicator, the character is placed at that point in the stream. Otherwise, the character appends to the stream. Use the **feof** facility to determine whether the end-of-file is reached. See the section "The **fputc** Function". See the section "The **putchar** Function".

Returns: Returns the character `c` as an `int` type. If a write error occurs, returns EOF and sets the error indicator.

The putchar Function

Synopsis: `#include <stdio.h>    int putchar(int c);`

Description: Equivalent to **fputc**, but uses the `stdout` stream as the output stream. Writes the character `c` to the standard output stream and advances to the next character. If you define the file position indicator, the character is placed at that point in the stream, otherwise, the character appends to the stream. Use the **feof** facility to determine whether the end-of-file is reached. See the section "The **fputc** Function". See the section "The **putc** Function".

Returns: Returns the character `c` as an `int` type. If a write error occurs, returns EOF and sets the error indicator.

The puts Function

Synopsis: `#include <stdio.h>    int puts(const char *s);`

Description: Copies the string pointed to by `s` to the standard output stream, `stdout`. Adds a newline character to the string after it is copied.

Returns: Returns zero if successful, otherwise, returns a nonzero value.

The remove Function

Synopsis: `#include <stdio.h>    int remove(const char *filename);`

Description: Removes the files specified by `filename`.

Returns: Returns zero if successful, otherwise, returns a nonzero value.

The rename Function

Synopsis: `#include <stdio.h>    int rename(const char *old, const char *new);`

Description: Renames the file pointed to by `old` to the name pointed to by `new`.

Returns: Returns zero if successful, otherwise, returns a nonzero value.

The rewind Function

Synopsis: `#include <stdio.h>    void rewind(FILE *stream);`

Description: Resets the file position indicator for `stream` to the beginning of the file.

Returns: Returns no value.

The scanf Function

Synopsis: `#include <stdio.h> int scanf(const char *format, ...);`

Description: Reads characters from the standard input stream `stdin` and interprets them according to the string `format`. The `format` field can consist of one or more of:

- A whitespace character
- A text character
- A conversion specification

For further information on conversion specifications: See the section "Conversion Specifiers for Scan Functions". Compare **fscanf** and **sscanf**.

Returns: Returns the number of characters read, if successful. Otherwise, it returns EOF.

The setbuf Function

Synopsis: `#include <stdio.h> void setbuf(FILE *stream, char *buf);`

Description: Equivalent to the **setvbuf** function, controls the buffering of the `stream`, using `buf` instead of an automatically assigned buffer. This function assumes _IOFBF for mode and BUFSIZE size if `buf` is not a null pointer. In such a case, _IONBF is the value of mode. For more information, see the section "The **setvbuf** Function".

Returns: Returns no value.

The setvbuf Function

Synopsis: `#include <stdio.h> int setvbuf(FILE *stream, char *buf, int mode, size t size);`

Description: Sets the buffering for a stream to full buffering, line buffering, or no buffering. Use this function after associating a stream with an open file and before reading from or writing to the stream. The argument mode sets buffering for the stream pointed to by `stream`.

Mode can be one of these:

Mode	*Meaning*
_IOFBF	Input/output is fully buffered.
_IOLBF	Input/output is line-buffered. The buffer is

flushed when a newline character is written, when input is requested, or when the buffer is full.

_IONBF Input/output is not buffered.

Returns: Returns zero, if successful. Otherwise, returns a nonzero value.

The sprintf Function

Synopsis: #include <stdio.h> int sprintf(char *s, const char *format, ...);

Description: Similar to **fprintf**, except that formatted output is sent to the array s, rather than to a stream. For a summary of conversion specifiers: See the section "The **fprintf** Function".

Returns: Returns the number of characters written to the array.

The sscanf Function

Synopsis: #include <stdio.h> int sscanf(const char *s, const char *format, ...);

Description: Reads characters from the string s and interprets them according to the string format. The format field can consist of one or more of:

- A whitespace character
- A text character
- A conversion specification

For further information on conversion specifications: See the section "Conversion Specifiers for Scan Functions". Compare **fscanf** and **scanf**.

Returns: Returns the number of characters assigned to s, if successful. Otherwise, returns EOF.

The tmpfile Function

Synopsis: #include <stdio.h> FILE *tmpfile(void);

Description: Creates a temporary file open for output. This binary file exists for the duration of the program (or until closed).

Returns: Returns a pointer to the new file, if successful. Otherwise, returns a null pointer.

The tmpnam Function

Synopsis: `#include <stdio.h>    char *tmpnam(char *s);`

Description: Generates a unique file-name string. If `s` is a null pointer, the new file name is stored in a static object. If `s` is not a null pointer, the new file name is stored in `s`.

Returns: If successful, returns a pointer to the new file name. Otherwise, returns a null pointer.

The ungetc Function

Synopsis: `#include <stdio.h>    int ungetc(int c, FILE *stream);`

Description: Pushes the character `c` back onto the input stream named in the argument `stream`. One character of pushback is guaranteed, and the character `c` is returned by the next call to **fgetc**, **getc**, or **getchar** on that stream. An intervening call to **fflush** or to any file positioning function removes any characters that have been pushed back.

Returns: Returns `c` if successful. Otherwise, returns EOF.

The vfprintf Function

Synopsis: `#include <stdio.h>    int vfprintf(FILE *stream, const char *format, va list arg);`

Description: Similar to **fprintf**, except that additional arguments are given as a variable argument list. For further information on variable argument lists: See the section "The **va_arg** Macro".

Returns: Returns the number of characters sent to the output stream, if no error occurs. Returns EOF if an error occurs.

The vprintf Function

Synopsis: `#include <stdio.h>    int vprintf(const char *format, va list arg);`

Description: Similar to **printf**, except that additional arguments are given as a variable argument list. For further information on variable argument lists: See the section "The **va_arg** Macro".

Returns: Returns the number of characters sent, if successful. Otherwise, returns a negative number.

The vsprintf Function

Synopsis: `#include <stdio.h>    int vsprintf(char *s, const char *format, va_list arg);`

Description: Similar to **sprintf**, except that additional arguments are given as a variable argument list. For further information on variable argument lists: See the section "The **va_arg** Macro".

Returns: Returns the number of characters written in the array.

The stdlib.h Library

The abort Function

Synopsis: `#include <stdlib.h>   void abort(void);`

Description: Causes the unsuccessful termination of a program.

Returns: Returns a nonzero value.

The abs Function

Synopsis: `#include <stdlib.h>   int abs(int j);`

Description: Computes the absolute value of `j`.

Returns: Returns the absolute value of `j` as an integer type.

The atexit Function

Synopsis: `#include <stdlib.h>   int atexit(void (*func)(void));`

Description: Places the function pointed to by `func` on a list of functions invoked upon normal program termination.

Returns: Returns a zero value if successful. Otherwise, returns a nonzero value.

The atof Function

Synopsis: `#include <stdlib.h>   double atof(const char *nptr);`

Description: Converts the string pointed to by `*nptr` to a `double` representation. The function ignores leading whitespace when it begins reading the string and stops reading the string when it reaches an unrecognized

character. This function is equivalent to the **strtod** function, except for its treatment of error conditions. For further information, see the section "The **strtod** Function".

Returns: Returns the converted value.

The atoi Function

Synopsis: #include <stdlib.h> int atoi(const char *nptr);

Description: Converts the string pointed to by *nptr to an int representation. The function ignores leading whitespace when it begins reading the string, and stops reading the string when it reaches an unrecognized character. This function is equivalent to the **strtol** function, except for its treatment of error conditions. For further information, see the section "The **strtol** Function".

Returns: Returns the converted value.

The atol Function

Synopsis: #include <stdlib.h> long int atol(const char *nptr);

Description: Converts the string pointed to by *nptr to a long int representation. The function ignores leading whitespace when it begins reading the string, and stops reading the string when it reaches an unrecognized character. This function is equivalent to the **strtol** function, except for its treatment of error conditions. See the section "The **strtol** Function".

Returns: Returns the converted value.

The bsearch Function

Synopsis: #include <stdlib.h> void *bsearch(const void *key, const void *base, size t nmemb, size t size, int (*compar)(const void *, const void *));

Description: Searches an array for an object that matches the one pointed to by *key. The argument base is the initial object in the array, nmemb specifies the number of objects to search in the array, and size specifies the size of each object.

Returns: Returns a pointer to the matching array member. If no match is found, the function returns a null pointer.

The calloc Function

Synopsis: #include <stdlib.h> void *calloc(size t nmemb, size t size);

Description: Allocates memory for `nmemb` objects of size `size` where the unit of `size` is in bytes. All bits in the region are initialized to zero.

Returns: Returns a pointer to the first element of the region, if successful. Otherwise, returns a null pointer.

The div Function

Synopsis: `#include <stdlib.h>   div t div(int numer, int denom);`

Description: Computes the quotient and remainder of the division of `numer` by `denom`.

Returns: Returns a structure of type `div t`. The type `div t` is a structure whose members are named `quot` and `rem` and that contains `int` members.

The exit Function

Synopsis: `#include <stdlib.h>   void exit(int status);`

Description: Causes the normal termination of a program by following these steps: Calls all functions registered by **atexit** (in reverse order from registration), flushes all open output streams, closes all open streams, and removes all files created by the **tmpfile** function.

Returns: Returns no value.

The free Function

Synopsis: `#include <stdlib.h>   void free(void *ptr);`

Description: Frees a region of memory previously allocated by **calloc**, **malloc**, or **realloc**. The region is pointed to by `ptr`. If `ptr` is null, no action is taken.

Returns: Returns no value.

The getenv Function

Synopsis: `#include <stdlib.h>   char *getenv(const char *name);`

Description: Searches an environment list for a string matching the string pointed to by `name`. The **getenv** function in Symbolics C can return values for one of the following "key" strings:

- user-file-pathname-defaults

- temporary-file-pathname-defaults

Returns: If successful, returns a value that is a pointer to a C string whose contents are the converted versions (from Lisp string to C string) specified. Otherwise, returns a null pointer.

The labs Function

Synopsis: `#include <stdlib.h>   long int labs(long int j);`

Description: Computes the absolute value of `j`.

Returns: Returns the absolute value of `j` as a `long int`.

The malloc Function

Synopsis: `#include <stdlib.h>   void *malloc(size t size);`

Description: Allocates an area of memory of size `size`.

Returns: Returns a pointer to the first element of the the region allocated, if successful. Otherwise, returns a null pointer.

The qsort Function

Synopsis: `#include <stdlib.h> void qsort(void *base, size t nmemb, size t size, int (*compar)(const void *, const void *));`

Description: Sorts an array of objects in ascending order. The argument `base` is the initial object in the array. `nmemb` specifies the number of objects to sort, and `size` specifies the size of each object.

Returns:

The rand Function

Synopsis: `#include <stdlib.h>   int rand(void);`

Description: Computes a pseudo-random number in the range 0 to RAND_MAX. See the section "The **srand** Function".

Returns: Returns a pseudo-random number of type `int`.

The realloc Function

Synopsis: `#include <stdlib.h>   void *realloc(void *ptr, size t size);`

Description: Changes the size of a memory region previously allocated with **malloc** or **calloc** by reallocating the region. In doing so, may move the region. The contents of the region are the same up to the lesser of the old or new size.

If `*ptr` is specified as a null pointer, the function behaves like **malloc**. That is, it allocates a region of the specified size.

If `size` is specified as zero and `*ptr` is not a null pointer, the region is deallocated and the function returns a null pointer.

Returns: Returns a pointer to the start of the reallocated object. Note that the object may have been moved. If the object cannot be reallocated, returns a null pointer and leaves the region unchanged.

The srand Function

Synopsis: `#include <stdlib.h> void srand(unsigned int seed);`

Description: Reinitializes the pseudo-random number generator by specifying the seed for a sequence of pseudo-random numbers generated by the next call to **rand**.

After a call to **srand**, successive calls to **rand** produce a reproducible series of pseudo-random numbers. You can reproduce the series by calling **srand** again with the same argument, and then using **rand**.

Use the argument 1 with this function to replicate calls to **rand** prior to the first time **srand** was called.

Returns: Returns no value.

The strtod Function

Synopsis: `#include <stdlib.h> double strtod(const char *nptr, char **endptr);`

Description: Converts the string pointed to by `*nptr` to a `double` representation and returns its value. After converting the string, it sets `**endptr`, if not null, to point to the first character in `*nptr` immediately following the converted part of the string.

The function decomposes a string by breaking it down into three parts in this order:

- Whitespace characters, if any, as defined by **isspace**.
- A `subject sequence` of recognized characters.

- A final group of one or more unrecognized characters.

Unrecognized characters include the whitespace character and the terminating null character. The `subject sequence` is the object being converted. It is defined as the longest sequence of characters consisting of:

1. An optional plus or minus sign.

2. A sequence of decimal digits that can include a single decimal point.

3. An optional exponent, made up of an `e` or E and a sequence of decimal digits.

That is, a legal subject sequence may be one of a decimal-constant, an octal-constant, or a floating-constant. A type-marker is not recognized as part of a subject sequence.

Returns: If successful, returns the result of the conversion. If no conversion is possible, returns zero, sets `endptr` (if not null) to the value of `nptr`, and sets **errno** to ERANGE. If the number converted causes overflow, returns HUGE_VAL (correctly signed), and sets **errno** to ERANGE. If the number converted causes underflow, returns zero and sets **errno** to ERANGE.

The strtol Function

Synopsis: `#include <stdlib.h> long int strtol(const char *nptr, char **endptr, int base);`

Description: Converts the initial part of the string pointed to by `*nptr` to an integer of type `long int` and returns its value. After converting the string, it sets `**endptr`, if not null, to point to the first character in `*nptr` immediately following the converted part of the string.

The function decomposes a string by breaking it down into three parts in this order:

- Whitespace characters, if any, as defined by **isspace**.

- A `subject sequence` of recognized characters.

- A final group of one or more unrecognized characters.

Unrecognized characters include the whitespace character and the terminating null character. The `subject sequence` is the object being converted.

If the value of base is 0, the subject sequence is defined as the longest sequence of characters consisting of an integer-constant with an optional preceding sign. Type-markers are not considered part of the sequence. The radix is derived from the format of the number.

If the value of base is 2 through 36, the subject sequence consists of a nonzero sequence of letters and digits representing an integer in the specified base with an optional preceding sign. Letters a through z (or A through Z) represent values from 10 through 35. Hex-markers are considered part of the sequence. The radix is derived from the value of base.

Returns: Returns the result of the conversion, if successful. If no conversion is possible, returns zero, sets endptr (if not null) to the value of nptr, and sets **errno** to ERANGE. If the number converted causes an overflow, returns LONG_MAX or LONG_MIN (depending on sign of the result) and sets **errno** to ERANGE. If the number converted causes an underflow, returns zero and sets **errno** to ERANGE.

The strtoul Function

Synopsis:

```
#include <stdlib.h>   unsigned long int strtoul(const char *nptr,
char **endptr, int base);
```

Description: Converts the initial part of the string pointed to by *nptr to an integer of type unsigned long int and returns its value. After converting the string, it sets **endptr, if not null, to point to the first character in *nptr immediately following the converted part of the string.

The function decomposes a string by breaking it down into three parts in this order:

- Whitespace characters, if any, as defined by **isspace**.
- A subject sequence of recognized characters.
- A final group of one or more unrecognized characters.

Unrecognized characters include the whitespace character and the terminating null character. The subject sequence is the object being converted.

If the value of base is 0, the subject sequence is defined as the longest sequence of characters consisting of an integer-constant. Type-markers are not considered part of the sequence. The radix is derived from the format of the number.

If the value of base is 2 through 36, the subject sequence consists of a nonzero sequence of letters and digits representing an integer in the specified base. Letters a through z (or A through Z) represent values from 10 through 35. Hex-markers are considered part of the sequence. The radix is derived from the value of base.

Returns: If successful, returns the result of the conversion. If no conversion is possible, returns zero, sets endptr to the value of nptr, and sets **errno** to ERANGE. If the number converted causes an overflow, returns ULONG_MAX and sets **errno** to ERANGE.

The system Function

Synopsis: `#include <stdlib.h>   int system(const char *string);`

Description: Executes a Command Processor command given in string. The function works by passing string to the Genera environment.

Returns: Returns 1 if an error is signalled, otherwise, returns 0.

The string.h Library

The memchr Function

Synopsis: `#include <string.h>   void *memchr(const void *s, int c, size t n);`

Description: Searches for the first occurrence of the character c in the first n characters of *s.

Returns: If the character is located, returns a pointer to it. If the character is not located, returns a null pointer.

The memcmp Function

Synopsis: `#include <string.h>   int *memcmp(const void *s1, const void *s2, size t n);`

Description: Compares the first n characters of *s1 with the first n characters of *s2.

Returns: Returns an integer indicating whether *s1 is greater than, less than, or equal to *s2. If *s1 is greater than *s2, returns an integer greater than zero; if *s1 is less than *s2, returns an integer less than zero; if *s1 is equal to *s2, returns zero.

The memcpy Function

Synopsis: `#include <string.h>   void *memcpy(void *s1, void *s2, size t n);`

Description: Copies `n` characters from the object pointed to by `s2` into the object pointed to by `s1`. Behavior is undefined if the objects `*s1` and `*s2` overlap. See the section "The **memmove** Function".

Returns: Returns the value of `s1`, the object into which `n` characters are copied.

The memmove Function

Synopsis: `#include <string.h>   void *memmove(void *s1, void *s2, size t n);`

Description: Copies `n` characters from the object pointed to by `s2` into the object pointed to by `s1`. Unlike the **memcpy** function, this function supports copying between overlapping strings.

Returns: Returns the value of `s1`, the object into which `n` characters are copied.

The memset Function

Synopsis: `#include <string.h>   void *memset(const void *s, int c, size t n);`

Description: Copies the value of `c` into each of the first `n` characters of the object pointed to by S.

Returns: Returns the value of `s`.

The strcat Function

Synopsis: `#include <string.h>   char *strcat(char *s1, const char *s2);`

Description: Appends a copy of the string `s2` to `s1`. The terminating null character of s1 is overwritten by `s2`, and the terminating null character of `s2` is copied as part of that string.

Returns: Returns the value of `s1`.

The strchr Function

Synopsis: `#include <string.h>   char *strchr(const char *s, int c);`

Description: Finds the first occurrence of the character `c` in the string pointed to by `s`. (`c` is converted to a `char`).

Returns: Returns a pointer to the first occurrence of `c`. If `c` is not found, the function returns a null pointer.

The strcmp Function

Synopsis: `#include <string.h>   int *strcmp(const char *s1, const char *s2);`

Description: Compares the string `s1` to the string `s2`.

Returns: If `s1` is equal to `s2`, returns zero. If `s1` is greater than `s2`, returns an integer greater than zero. If `s1` is less than `s2`, returns an integer less than zero.

The strcoll Function

Synopsis: `#include <string.h>   size t strcoll(char *to, size t maxsize, const char *from);`

Description: Takes the string `from`, transforms it, and places the transformed string into `to`. You can use the transformed string as an argument to **strcmp** or **memcmp**.

Returns: Returns the number of characters in `to`, if that number is less than `maxsize`. Otherwise, this function returns zero.

The strcpy Function

Synopsis: `#include <string.h>   char *strcpy(char *s1, const char *s2);`

Description: Copies the string `s2` into the string `s1`.

Returns: Returns the value of `s1`.

The strcspn Function

Synopsis: `#include <string.h>   size t  strcspn(const  char  *s1,  const  char  *s2);`

Description: Finds the length of the initial segment of `s1` that does not contain any of the same characters as `s2`. Compare: "The **strspn** Function". **Returns:** Returns the length of the segment.

The strerror Function

Synopsis: `#include <string.h>   char *strerror(int errnum);`

Description: Maps the error number in errnum to an error message string.

Returns: Returns a pointer to the string.

The strlen Function

Synopsis: `#include <string.h>   size t strlen(const char *s);`

Description: Finds the number of characters in the string pointed to by s, excluding the terminating null character, and returns the value as the type size t.

Returns: Returns the number of characters in the string s.

The strncat Function

Synopsis: `#include <string.h>   char *strncat(char *s1, const char *s2, size t n);`

Description: Appends up to n characters of s2 to s1, and terminates the catenation with the null character.

Returns: Returns the value of s1.

The strncmp Function

Synopsis: `#include <string.h>   int *strncmp(const char *s1, const char *s2, size t n);`

Description: Compares n characters of the string s1 to the string s2.

Returns: If s1 is equal to s2, returns zero. If s1 is greater than s2, returns an integer greater than zero. If s1 is less than s2, returns an integer less than zero.

The strncpy Function

Synopsis: `#include <string.h>   char *strncpy(char *s1, const char *s2, size t n);`

Description: Copies n characters of the string s2 to the array s1. If the number of characters in s2 is less than n, n - s2 null characters are appended.

Returns: Retruns the value of s1.

The strpbrk Function

Synopsis: `#include <string.h> char *strpbrk(const char *s1, const char *s2);`

Description: Finds the first occurrence of any character in the string s2 in the string s1. Compare: "The **strcspn** Function". **Returns:**
Returns a pointer to the first character found from s2.

The strrchr Function

Synopsis: `#include <string.h> char *strrchr(const char *s, int c);`

Description: Finds the last occurrence of the character c in the string s. (c is converted to char).

Returns: Returns a pointer to the first occurrence of c. If c is not found, the function returns a null pointer.

The strspn Function

Synopsis: `#include <string.h> size t strspn(const char *s1, const char *s2);`

Description: Compares the string s1 with s2 and finds the number of characters in the initial segment of s1 that match the characters in s2.

Returns: Returns the number of characters.

The strstr Function

Synopsis: `#include <string.h> char *strstr(const char *s1, const char *s2);`

Description: Finds the first occurrence of the string s2 in the string s1.

Returns: Returns a pointer to s1. If s2 is not found, returns a null pointer.

The strtok Function

Synopsis: `#include <string.h> char *strtok(char *s1, const char *s2);`

Description: Separates the string s1 into tokens separated by a character from the string s2. For further information, see the section "C-Ref: **STRSTR, STRTOK**".

Returns: Returns a pointer to the first character of a token. If there is no token, returns a null pointer.

The time.h Library

Macros and Types Declared in the <time.h> Header

The header <time.h> declares:

- The type `clock t`, for representing CPU times.
- The type `time t`, for representing calendar time.
- The macro `CLK TCK`, which defines the number of `clock t` units per second.
- The structure `tm`, which contains elements that describe components of calendar time. A time described by a `tm` structure is called *broken-down time*.

	Meaning	Normal Range
struct tm {		
int tm_sec,	seconds after the minute	0 - 59
tm_min,	minutes after the hour	0 - 59
tm_hour,	hours since midnight	0 - 23
tm_mday,	day of the month	1 - 31
tm_mon,	months since January	0 - 11
tm_year,	years since 1900	---
tm_wday,	days since Sunday	0 - 6
tm_yday,	days since January 1	0 - 365
tm_isdst;	Daylight Savings Time flag	positive for DST
	zero for no DST	
	negative for no information	
} /*--- struct tm ---*/		

The asctime Function

Synopsis: `#include <time.h>     char *asctime(const struct tm *timeptr);`

Description: Converts the time represented in the structure timeptr to a string of the form:

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

Returns: Returns a pointer to the `asctime` string.

The clock Function

Synopsis: `#include <time.h>     clock t clock(void);`

Description: Determines the processor time used.

Returns: Returns the processor time used by the program since program invocation. If processor time is undetermined, returns the value (clock t)-1.

The ctime Function

Synopsis: #include <time.h> char *ctime(const time t *timer);

Description: Converts the calendar time pointed by timer to local time. This function is equivalent to:

```
asctime(localtime(timer))
```

Returns: Returns a pointer to the equivalent of the asctime argument.

The difftime Function

Synopsis: #include <time.h> double difftime(time t time1, time t time0);

Description: Computes the difference in seconds between time1 and time0.

Returns: Returns the difference in seconds.

The gmtime Function

Synopsis: #include <time.h> struct tm *gmtime(const time t *timer);

Description: Converts the calendar time pointed to by timer into a time expressed as Greenwich Mean Time (GMT).

Returns: Returns a pointer to the structure containing components of the converted time. If GMT cannot be calculated, returns a null pointer.

The localtime Function

Synopsis: #include <time.h> struct tm *localtime(const time t *timer);

Description: Converts the calendar time pointed to by timer into a time expressed as local time.

Returns: Returns a pointer to the structure containing components of the converted time.

The mktime Function

Synopsis: #include <time.h> time t mktime(struct tm *timeptr);

Description: Converts the time components in timeptr into calendar time.

Returns: Returns the new time value and sets the `tm wday` and `tm day` components. Returns the value `(time t)-1`, if unsuccessful.

The strftime Function

Synopsis:

```
#include <time.h>     size t strftime(char *s, size t maxsize, const char *(format, const struct tm *timeptr);
```

Description: Prints dates and times in a locale-determined format. The function converts the `timeptr` argument into `s`, which is of size `maxsize`, using format specifiers given in `format`. The `format` argument can consist of format specifiers and characters. The `*timeptr` argument points to a structure that describes locale values for the format specifiers. The following table lists the format specifiers and their values.

%a	Abbreviated weekday name.
%A	Full weekday name.
%b	Abbreviated month name.
%B	Full month name.
%c	Date and time representation.
%d	Decimal number representing the day of the month.
%H	Decimal number representing the hour, based on a 24-hour clock.
%I	Decimal number representing the hour, based on a 12-hour clock.
%j	Decimal number representing the day of the year.
%m	Decimal number representing the month.
%M	Decimal number representing the minute.
%p	A value equivalent to AM or PM.
%S	Decimal number representing the second.
%U	Decimal number representing the week number of the year, using Sunday as the first day of the week.
%w	Decimal number representing the day of the week, using Sunday as the first day of the week (0 - 6).
%W	Decimal number representing the week number of the year, using Monday as the first day of the week.

%x	A date representation.
%X	A time representation.
%y	A decimal number representing the year without the century.
%Y	A decimal number representing the year without the century.
%Z	The timezone name. If no timezone exists, no value is supplied.
%%	Represents %.

Returns: Returns the number of characters sent to the array `s` if that number is not greater than `maxsize`. Otherwise, returns zero. In such a case, `s` array contents are indeterminate.

The time Function

Synopsis: `#include <time.h>     time t time(time t *timer);`

Description: Determines the current calendar time.

Returns: Returns the current calendar time and assigns this value to the object pointed to by `timer`. If the calendar time is undeterminable, returns the value `time t - 1`.

Summary of Standard Editor Mode Commands

0.0.47. Cursor Movement Commands

Keystroke	*Meaning*
`c-m-F`	Moves the cursor to the end of the current or next language-specific unit.
`c-m-B`	Moves the cursor to the start of the current or previous language-specific unit.
`c-m-A`	Moves the cursor to the start of the current or previous language definition.
`c-m-E`	Moves the cursor to the end of the current or next language definition.
`c-sh-F`	Moves the cursor to the end of the current or next language expression.

`c-sh-B`	Moves the cursor to the start of the current or previous language expression.
`c-m-H`	Moves the cursor to the start of the current language definition and marks the entire definition as a region. The editor underlines the region.
`c-m-N`	Moves the cursor to the next template in the buffer, if any.
`c-m-P`	Moves the cursor to the previous template in the buffer, if any.

0.0.48. Deletion Commands

Keystroke	*Meaning*
`m-sh-X`	Deletes the language expression to the left of the cursor.
`c-sh-X`	Deletes the language expression to the right of the cursor.
`c-m-RUBOUT`	Deletes the language construct to the left of the cursor.
`c-m-K`	Deletes the language construct to the right of the cursor.
`c-sh-K`	Deletes the language construct around point (the cursor).
`c-sh-T`	Deletes the language token (for example, an identifier or comment) to the right of the cursor.
`m-sh-T`	Deletes the language token (for example, an identifier or comment) to the left of the cursor.

0.0.49. Syntax Error Detection Commands

Keystroke	*Meaning*
`c-sh-N`	Finds the nearest syntax error to the right of the cursor, if any, and moves the cursor there. With a numeric argument, it finds the last syntax error in the buffer.
`c-sh-P`	Finds the nearest syntax error to the left of the cursor and moves the cursor there. With a numeric argument, it finds the first syntax error in the buffer.

0.0.50. Template and Completion Commands

Keystroke	*Command*
`END`	Inserts a template that matches the keyword to the right of the cursor.
`c-END`	Inserts whatever uniquely closes a language construct to the left of the cursor. For example, `c-END` inserts a close bracket "]" to match a "[", or a `then` to match an `if`.

COMPLETE	Completes a keyword to the left of the cursor or further fills in the current template.
c-HELP	Provides a list of templates for valid language constructs inserted at the cursor.
c-?	Lists in an editor typeout window the possible completions of predeclared identifiers for the name immediately to the left of the cursor.
m-X Remove Template Item	Deletes the next template to the right of the cursor.
SPACE	Deletes the next template item to the right of the cursor.
c-m-N	Moves the cursor to the next template in the buffer.
c-m-P	Moves the cursor to the previous template in the buffer.

0.0.51. Indentation Commands

Keystroke	*Command*
c-m-Q	Corrects the indentation of the language structure following point (cursor position).
LINE	Indents the current line correctly with respect to the line above it. It also positions the cursor on the next line and aligns it with the preceding line. LINE opens a new blank line. If a syntax error is found on that line, the editor points out the error.
TAB	Indents the current line correctly with respect to the line above it and positions the cursor at the first character on the line.
m-X Save Indentation	After using c-I to change global indentation of C language constructs, the command produces a Lisp form reflecting the new indentation values. Evaluate this form after the C editor loads.

0.0.52. Formatting Commands

Keystroke	*Command*
m-X Adjust Face and Case	Modifies the face and case settings for a particular language or dialect.
m-X Electric C Mode	Turns on Electric C mode, or, if it is on, turns it off. Once the mode is on, you can use the Adjust Face and Case command. The Electric C Mode command works only when the buffer is in C mode.
m-X Format Language Region	Conforms the face and case in the region to the settings for the buffer. A numeric argument removes any special typefaces from the region but leaves the case untouched.

C Editor Commands

This chapter summarizes some Zmacs commands specific to the C editor mode.

m-X C Fundamental Mode Sets the editor to C fundamental mode, a major editing mode, parallel to C mode, that binds the C compiler. You can use a subset of C commands in this mode, including Compile Buffer, Compile Region, and the C mode include directory search list commands.

m-X C Mode Sets the editor buffer to C mode. C mode supports template completion.

m-X Electric C Mode Toggles Electric C mode. Electric mode places keywords and comments in their own font.

m-X Compile and Execute C Function
Compiles and executes the C function that the cursor is on.

m-X Compile Buffer Compiles the current C buffer to memory. With a numeric argument, compiles from the point indicated by the cursor to the end of the buffer.

m-X Compile Changed Definitions of Buffer or m-sh-C
Compiles any definitions changed in the current buffer. With a numeric argument, it prompts individually about whether to compile particular changed definitions. The default compiles all changed definitions.

c-sh-C Compiles the currently defined region, a contiguous delimited section of text in the editor buffer. If none is defined, it compiles the C declaration or definition the cursor is on.

m-X Compile File Compiles a file, offering to save it first if the buffer is modified. It prompts for a file name in the minibuffer, using the file associated with the current buffer as the default. The command writes a compiled-code file to disk but does not change or create any data or function in memory.

m-X Compiler Warnings Places all pending compiler warnings in a buffer and selects that buffer. Loads the compiler warnings database into a buffer called *Compiler-Warnings-1*, creating that buffer if it does not exist.

m-X Edit Compiler Warnings Allows you to edit some or all routines whose compilation caused a warning message. For each file mentioned in the compiler warnings database, you are asked whether you want to edit the warnings for the routines in that file. It splits the screen, placing the warning message in the top window and source code whose compilation caused the error in the bottom window. Use c-. to move to the next pair of warning and source code.

m-X Edit C Definition (m-.) Edits the definition of a compiled C unit. When you are prompted for the name of a unit, you can: 1) type the name in the minibuffer at the bottom of the screen, or 2) use the mouse to select a name in the current buffer. The command finds the unit, places it in an editor buffer, and positions the cursor there.

The echo area displays a message indicating multiple occurrences of the definition, if any. Use c-. to move to the next occurrence.

You can use this command to edit any C definition currently loaded regardless of whether the file that contains it is currently in a buffer.

m-X Execute C Function Checks to see that the cursor is positioned near a valid, compiled C program. Then executes the program without run-time options, and with predefined files input and output bound to the editor typeout window.

Format Language Region (c-m-Q) Places the contents of the marked region in Electric C Mode format. If no region is defined, it acts on the current C definition. With a numeric argument, this command removes Electric C Mode formatting.

Mark C Definition (c-m-H) Marks the definition of a compiled C declaration or definition.

m-X Reparse Attribute List Reparses the attribute list for a buffer, causing any changed attributes to take effect.

m-X Define C Search List Defines the search list *name* as a list of directory pathnames. When a source file uses this search list, the compiler searches these directories for #include files. The directories are searched in the order in which they are listed.

m-X Set C Search List for Buffer Sets the include directory search list for the current buffer. With a numeric argument, sets the search list for predefined include files (those that are included with the angle-bracket syntax).

m-X Set Export for Buffer Sets the Export attribute of the buffer to Yes.

m-X Set Package Sets the package for the buffer.

m-X Show C Search List Displays the current list of directories associated with the specified search lists.

Show Documentation (m-sh-D) Displays documentation for the C library function or reserved word preceding the cursor.

m-X Undefine C Search List Removes a defined C include directory search list.

m-X Update Attribute List Creates or updates the attribute list for a file.

Sublicense Addendum for Symbolics C

Your purchase of Symbolics C under the *Terms and Conditions of Sale, License, and Service* (3/89) allows you to use this product on a designated processor. Customers who distribute an application that includes the Symbolics C run-time system **must** sign a *Sublicense Addendum to the Terms and Conditions of Sale, License and Service* (3/89). This agreement spells out the terms and conditions under which you can sublicense any application that contains the Symbolics C run-time system. The Sublicense Addendum appears on the next page. If you have not done so already, read the Sublicense Addendum carefully, sign it, detach it, and return it to your Symbolics sales representative.

Note: *You are required to sign the sublicense agreement even if you only distribute your application internally — to end users who work for your company.*

Sublicense Addendum to Symbolics Inc. Terms and Conditions of Sale, License and Service (3/89)

Addendum made this _____ day of __________, 199__, ("this Addendum") to Symbolics, Inc. Standard Terms and Conditions of Sale, License and Service (3/89) dated, _______, 199__, (the "Agreement"), both of which are by and between Symbolics Inc. and Customer. All capitalized terms used in this Addendum, if not defined in this Addendum, shall have the meanings assigned to them in the Agreement.

1. **The Software**. The Software to which this Addendum applies is defined as follows:

Model	Description	Release
SLAN-C	Symbolics C	1.2

2. **Right of Sublicense**.

Customer may sublicense all or any portion of the run-time system binary code (the "Code") of the Software to Customer's end users provided that:

(i) such Code is part of Customer's application software program sublicensed to such end users;

(ii) the Customer's application software program is licensed by Customer to Customer's end users to run on a Symbolics computer system or processor; and

(iii) Symbolics' copyright and trademark notices shall not be removed from the Software.

3. **End User**.

The term "end user" for the purposes of this Addendum shall mean Customer's customers and includes Customer's own internal end users of its application software programs.

CUSTOMER	SYMBOLICS, INC.
______________________________	______________________________
Name	Name
______________________________	______________________________
Title	Title
______________________________	______________________________
(Address)	(Address)