Release Notes for Symbolics C 1.1

Symbolics C 1.1: Introduction

The document *User's Guide to Symbolics C* describes Release 1.0 of Symbolics C, and this document describes the software enhancements which are available in Release 1.1 of C.

Note that the online documentation provided with C 1.1 is updated to include the software enhancements of C 1.1.

C 1.1 does not support the setting and clearing of breakpoints, which is documented in section ""C Frames in the Debugger"", in *User's Guide to Symbolics C*.

Exporting Technique to Reduce Size of Bin Files in C 1.1

In Release 1.0, users noticed that binary files for C programs were very large. This was due to the compiler creating a copy of definitions that are defined in include files; the compiler made a copy of each definition in each .c file that included a file. For example, each .c file that included <stdio.h> would have its own copy of a given symbol or object.

Release 1.1 offers a way to reduce the size of binary files by specifying a standard set of include files. In effect, you are guaranteeing that the definitions in these files will not change due to macros. You gather these into a single file and compile and load it; the compiler does not then copy these definitions into other files that include them.

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".

Search Lists for Predefined Include Files in C 1.1

In C Release 1.0, you could specify a search list for user-defined include files. In C Release 1.1, you can also specify a search list for predefined include files. Similarly, whereas Release 1.0 enabled you to define a default search list for user-defined include files, Release 1.1 also enables you to define a default search list for predefined include files.

A user-defined include file is one which you include 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 capability enables you to specify, with a search list, where your existing C libraries are stored. (For example, this might be the directory /usr/include on a UNIX host.) You need not store all predefined include files in sys:c;include; nor edit source files to specify complete pathnames for predefined include files.

This section includes updated documentation describing how to use search lists and default search lists.

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-system::define-default-search-list**
  - **c-system::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-system::define-search-list**

When you specify the directories in these commands, you can use a subset of wild-card 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.

Incremental Development Tools in C 1.1

C Release 1.1 includes several tools that enable C programmers to use the same powerful paradigm of incremental development in much the same way that Genera supports incremental development for Lisp programers.

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 execution 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.

The locale.h Library Is Supported in C 1.1

C Release 1.1 supports the functions and macros specified in the Draft Proposed ANSI Standard for C in the <locale.h> header file.

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).

Typing Commands in C Listener and Debugger in C 1.1

In Release 1.0, you could click on the C Listener commands in the menu, but you could not type the commands. In Release 1.1, you can enter the commands by typing them in the C Listener. You can also enter commands by typing them in the C Debugger.

Keep in mind that the menu sometimes abbreviates the name of commands. For example, the menu item [Edit] is an abbreviation for the command Edit C Definition. If you type in a command, you must type in its full name.

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.

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

You can use the function **c-system::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-system::execute**.

c-system::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-system::build-expanded-argument-list** converts this list into the corresponding argc, argv pair, and returns two values: argc and argv.

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 1 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.

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 2. Sizes and Alignments of C Data Types

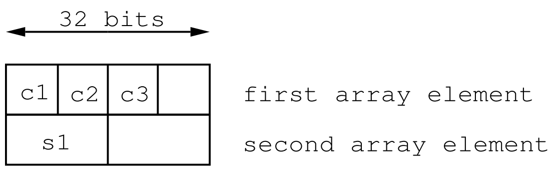
Bit fields require bit alignment. A bit field length specifier of 0 forces alignment to the nearest 8-bit boundary.

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.

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;
};
```

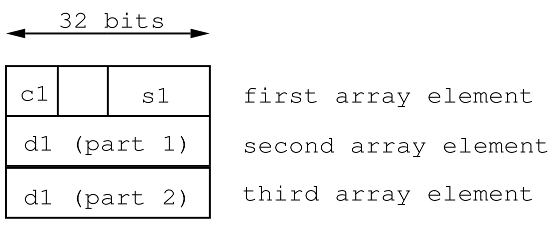
The array representing that structure is:



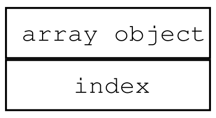
We define another structure as follows:

```
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.