Reference Manual

Volume II
Advanced Programming Guide

Version 6.30

March 17th 2015

CLIPS Advanced Programming Guide

Version 6.30 March 17th 2015

CONTENTS

License Information

Preface

The History of CLIPS

The origins of the C Language Integrated Production System (CLIPS) date back to 1984 at NASA's Johnson Space Center. At this time, the Artificial Intelligence Section (now the Software Technology Branch) had developed over a dozen prototype expert systems applications using state-of-the-art hardware and software. However, despite extensive demonstrations of the potential of expert systems, few of these applications were put into regular use. This failure to provide expert systems technology within NASA's operational computing constraints could largely be traced to the use of LISP as the base language for nearly all expert system software tools at that time. In particular, three problems hindered the use of LISP based expert system tools within NASA: the low availability of LISP on a wide variety of conventional computers, the high cost of state-of-the-art LISP tools and hardware, and the poor integration of LISP with other languages (making embedded applications difficult).

The Artificial Intelligence Section felt that the use of a conventional language, such as C, would eliminate most of these problems, and initially looked to the expert system tool vendors to provide an expert system tool written using a conventional language. Although a number of tool vendors started converting their tools to run in C, the cost of each tool was still very high, most were restricted to a small variety of computers, and the projected availability times were discouraging. To meet all of its needs in a timely and cost effective manner, it became evident that the Artificial Intelligence Section would have to develop its own C based expert system tool.

The prototype version of CLIPS was developed in the spring of 1985 in a little over two months. Particular attention was given to making the tool compatible with expert systems under development at that time by the Artificial Intelligence Section. Thus, the syntax of CLIPS was made to very closely resemble the syntax of a subset of the ART expert system tool developed by Inference Corporation. Although originally modeled from ART, CLIPS was developed entirely without assistance from Inference or access to the ART source code.

The original intent for CLIPS was to gain useful insight and knowledge about the construction of expert system tools and to lay the groundwork for the construction of a replacement tool for the commercial tools currently being used. Version 1.0 demonstrated the feasibility of the project concept. After additional development, it became apparent that CLIPS would be a low cost expert system tool ideal for the purposes of training. Another year of development and internal use went into CLIPS improving its portability, performance, functionality, and supporting documentation. Version 3.0 of CLIPS was made available to groups outside of NASA in the summer of 1986.

Further enhancements transformed CLIPS from a training tool into a tool useful for the development and delivery of expert systems as well. Versions 4.0 and 4.1 of CLIPS, released respectively in the summer and fall of 1987, featured greatly improved performance, external language integration, and delivery capabilities. Version 4.2 of CLIPS, released in the summer of 1988, was a complete rewrite of CLIPS for code modularity. Also included with this release were an architecture manual providing a detailed description of the CLIPS software architecture and a utility program for aiding in the verification and validation of rule-based programs. Version 4.3 of CLIPS, released in the summer of 1989, added still more functionality.

Originally, the primary representation methodology in CLIPS was a forward chaining rule language based on the Rete algorithm (hence the Production System part of the CLIPS acronym). Version 5.0 of CLIPS, released in the spring of 1991, introduced two new programming paradigms: procedural programming (as found in languages such as C and Ada) and object-oriented programming (as found in languages such as the Common Lisp Object System and Smalltalk). The object-oriented programming language provided within CLIPS is called the CLIPS Object-Oriented Language (COOL). Version 5.1 of CLIPS, released in the fall of 1991, was primarily a software maintenance upgrade required to support the newly developed and/or enhanced X Window, MS-DOS, and Macintosh interfaces. Version 6.0 of CLIPS, released in 1993, provided support for the development of modular programs and tight integration between the object-oriented and rule-based programming capabilities of CLIPS. Version 6.1 of CLIPS, released in 1998, removed support for older non-ANSI C Compilers and added support for C++ compilers. Commands to profile the time spent in constructs and user-defined functions were also added. Version 6.2 of CLIPS, released in 2002, added support for multiple environments into which programs can be loaded and improved Windows XP and MacOS development interfaces.

Because of its portability, extensibility, capabilities, and low cost, CLIPS has received widespread acceptance throughout the government, industry, and academia. The development of CLIPS has helped to improve the ability to deliver expert system technology throughout the public and private sectors for a wide range of applications and diverse computing environments. CLIPS is being used by numerous users throughout the public and private community including: all NASA sites and branches of the military, numerous federal bureaus, government contractors, universities, and many private companies.

CLIPS is now maintained as public domain software by the main program authors who no longer work for NASA. See appendix A for information on obtaining CLIPS and support.

CLIPS Version 6.3

Version 6.3 of CLIPS contains one major enhancement (improved rule performance) and several minor enhancements/changes. For a detailed listing of differences between releases of CLIPS, refer to appendix B of the *Basic Programming Guide* and appendix B of the *Advanced Programming Guide*.

CLIPS Documentation

Two documents are provided with CLIPS.

- The *CLIPS Reference Manual* which is split into the following parts:
  - *Volume I - The Basic Programming Guide*, which provides the definitive description of CLIPS syntax and examples of usage.
  - *Volume II - The Advanced Programming Guide*, which provides detailed discussions of the more sophisticated features in CLIPS and is intended for people with extensive programming experience who are using CLIPS for advanced applications.
  - *Volume III - The Interfaces Guide*, which provides information on machine-specific interfaces.
- The *CLIPS User's Guide* which provides an introduction to CLIPS rule-based and object-oriented programming and is intended for people with little or no expert system experience.

Acknowledgements

As with any large project, CLIPS is the result of the efforts of numerous people. The primary contributors have been: Robert Savely, who conceived the project and provided overall direction and support; Chris Culbert, who managed the project and wrote the original *CLIPS Reference Manual*; Gary Riley, who designed and developed the rule-based portion of CLIPS, co-authored the *CLIPS Reference Manual*, and developed the Macintosh interface for CLIPS; Brian Donnell, who designed and developed the CLIPS Object Oriented Language (COOL) and co-authored the *CLIPS Reference Manual*; Bebe Ly, who developed the X Window interface for CLIPS; Chris Ortiz, who developed the original Windows 95 interface for CLIPS; Dr. Joseph Giarratano of the University of Houston-Clear Lake, who wrote the *CLIPS User's Guide*; and Frank Lopez, who designed and developed CLIPS version 1.0 and wrote the CLIPS 1.0 User's Guide.

Many other individuals contributed to the design, development, review, and general support of CLIPS, including: Jack Aldridge, Carla Colangelo, Paul Baffes, Ann Baker, Stephen Baudendistel, Les Berke, Tom Blinn, Marlon Boarnet, Dan Bochsler, Bob Brown, Barry Cameron, Tim Cleghorn, Major Paul Condit, Major Steve Cross, Andy Cunningham, Dan Danley, Mark Engelberg, Kirt Fields, Ken Freeman, Kevin Greiner, Ervin Grice, Sharon Hecht, Patti Herrick, Mark Hoffman, Grace Hua, Gordon Johnson, Phillip Johnston, Sam Juliano, Ed Lineberry, Bowen Loftin, Linda Martin, Daniel McCoy, Terry McGregor, Becky McGuire, Scott Meadows, C. J. Melebeck, Paul Mitchell, Steve Mueller, Bill Paseman, Cynthia Rathjen, Eric Raymond, Reza Razavipour, Marsha Renals, Monica Rua, Tim Saito, Michael Sullivan, Gregg Swietek, Eric Taylor, James Villarreal, Lui Wang, Bob Way, Jim Wescott, Charlie Wheeler, and Wes White.

Section 1:
Introduction

This manual is the *Advanced Programming Guide* for CLIPS. It describes the Application Programmer Interface (API) that allows users to integrate their programs with CLIPS and use some of the more sophisticated features of CLIPS. It is written with the assumption that the user has a complete understanding of the basic features of CLIPS and a background in programming. Many sections will not be understandable without a working knowledge of C. Knowledge of other languages also may be helpful. The information presented here will require some experience to understand, but every effort has been made to implement capabilities in a simple manner consistent with the portability and efficiency goals of CLIPS.

Section 2 describes how to install and tailor CLIPS to meet specific needs. Section 3 of this document describes how to add user-defined functions to a CLIPS expert system. Section 4 describes how to embed a CLIPS application in a C program. Section 5 describes how to create run-time CLIPS programs. Section 6 discusses integrating CLIPS with languages other than C. Section 7 details the input/ output (I/O) router system used by CLIPS and how the user can define his own I/O routers. Section 8 discusses CLIPS memory management. Section 9 discusses environments which allow multiple expert systems to be loaded and run concurrently.

Not all of the features documented here will be of use to all users. Users should pick those areas which are of specific use to them. It is advised that users complete the *Basic Programming Guide* before reading this manual.

1.1 Warning About Interfacing With CLIPS

CLIPS provides numerous methods for integrating with user-defined code. As with any powerful capability, some care must be taken when using these features. By providing users with the ability to access internal information, we have also opened the door to the possibility of users corrupting or destroying data that CLIPS needs to work properly. Users are advised to be careful when dealing with data structures or strings which are returned from calls to CLIPS functions. Generally, these data structures represent useful information to CLIPS and should not be modified or changed in any way except as described in this manual. A good rule of thumb is to duplicate in user-defined storage space every piece of information taken out of or passed into CLIPS. In particular, *do not* store pointers to strings returned by CLIPS as part of a permanent data structure. When CLIPS performs garbage collection on symbols and strings, the pointer reference to the string may be rendered invalid. To store a permanent reference to a string, allocate storage for a copy of the string and then copy the string returned by CLIPS to the copy's storage area.

1.2 C++ Compatibility

The CLIPS source code can now be compiled using either an ANSI C or C++ compiler. Minimally, non-ANSI C compilers must support full ANSI style function prototypes and the void data type in order to compile CLIPS. If you want to make CLIPS API calls from a C++ program, it is usually easier to do the integration by compiling the CLIPS source files as C++ files. This removes the need to make an *extern "C"* declaration in your C++ program for the CLIPS APIs. Some programming environments allow you to specify the whether a file should be compiled as C or C++ code based on the file extension. Other environments allow you to explicitly specify which compiler to use regardless of the extension (e.g. in gcc the option "-x c++" will compile .c files as C++ files). In some environments, the same compiler is used to compile both C and C++ programs and the compiler uses the file extension to determine whether the file should be compiled as a C or C++ program. In this situation, changing the .c extension of the CLIPS source files to .cpp usually allows the source to be compiled as a C++ program.

1.3 Threads and Concurrency

The CLIPS architecture is designed to support multiple expert systems running concurrently using a single CLIPS application engine. The environment API, described in section 9, is used to implement this functionality. In order to use multiple environments, CLIPS must be embedded within your program either by linking the CLIPS source code with your program or using a shared library such as a Dynamic Link Library (DLL). The standard command line version of CLIPS as well as the operating system specific development interfaces for Windows provide access to a single environment. It is not possible to load and run multiple expert systems using these versions of CLIPS.

If multiple environments are created, a single thread of execution can be used to run each expert system. In this situation, one environment must finish executing before control can be passed to another environment. The user explicitly specifies which environment should process each API call. Once execution of an API call for that environment begins, the user must wait for completion of the API call before passing control to another environment.

Most likely, this type of execution control will be used when you need to make several expert systems available to a single end user, but don't want to go through the process of clearing the current expert system from a single environment, loading another expert system into it, and then resetting the environment. Instead, each expert system is loaded into its own environment, so to change expert systems it is only necessary to switch to the new environment and reset it.

A less likely scenario for this type of execution control is to simulate multiple expert systems running concurrently. In this scenario, each environment is allowed to execute a number of rules before control is switched to the next environment.

Instead of simulating multiple expert systems running concurrently, using the multi-threading capabilities native to the operating system on which CLIPS is running allows concurrent execution to occur efficiently and prevents one environment from blocking the execution of another. In this scenario, each environment uses a single thread of execution. Since each environment maintains its own set of data structures, it is safe to run a separate thread on each environment. This use of environments is most likely for a shared library where it is desirable to have a single CLIPS engine running that is shared by multiple applications.

❖ Warning

Each environment can have at most *one* thread of execution. The CLIPS internal data structures can become corrupted if two CLIPS API calls are executing at the same time for a single environment. For example, you can't have one thread executing rules and another thread asserting facts for the same environment without some synchronization between the two threads.

1.4 Garbage Collection

Garbage collection is a process used by CLIPS to manage memory that most users do not need to understand to use CLIPS. In some cases, when users embed CLIPS within their applications, a knowledge of the garbage collection process is necessary to understand when values returned by CLIPS to an embedding program can be safely accessed.

As a CLIPS program executes, it allocates and deallocates numerous types of data structures. In many cases, some data structures cannot be immediately deallocated because either outstanding references to the data structure still exist or the need to deallocate the data structure is questionable. Data which has been marked for later deallocation is referred to as **garbage**. The process of deallocating this garbage is referred to as **garbage collection**. CLIPS only performs garbage collection when it can determine that it is safe to deallocate the data structures marked for deallocation.

The following example illustrates several important concepts:

```
CLIPS>
(defrule gc-example
   ?f <- (factoid ?g)
   =>
   (retract ?f)
   (printout t "The value is " ?g crlf))
CLIPS> (assert (factoid (gensym*)))
<Fact-0>
CLIPS> (run)
The value is gen1
CLIPS>
```

First the *gc-example* rule is entered at the command prompt. The RHS of this rule retracts the *factoid* fact bound on the LHS of the rule. It then prints out one of the field values contained in this fact. The next command creates a *factoid* fact that activates the rule. This fact contains the unique symbol *gen1* returned by the **gensym*** function. The *gen1* symbol is initially considered to be garbage when created since nothing refers to it, but when it is asserted as part of the *factoid* fact it's no longer considered as garbage and isn't subject to garbage collection.

When the *run* command is issued, the *gc-example* rule fires. The first action of the rule retracts the *factoid* fact bound on the LHS of the rule. The fact is now considered to be garbage. The symbol *gen1* contained in the fact is also marked as being garbage since the fact contains the only reference to it. The next action in the rule prints the value from the *factoid* fact bound to the variable ?g. Since CLIPS directly retrieves this value from the fact, if the fact and symbols associated with it had been immediately deallocated when the **retract** command was executed, these values would not be available when the **printout** command is executed.

Since garbage created by the RHS actions may be accessed by other RHS actions, CLIPS does not initiate garbage collection for garbage created by RHS actions until the rule has finished firing. In this example, once the *gc-example* rule has finished firing, since there are no outstanding references to the *factoid* fact or the *gen1* symbol the data structures associated with these can be deallocated.

The garbage collection behavior would be changed by adding an **assert** command to the rule RHS:

```
(defrule gc-example
   ?f <- (factoid ?g)
   =>
   (retract ?f)
   (printout t "The value is " ?g crlf)
   (assert (info ?g)))
```

In this case, the *factoid* fact and the *gen1* symbol would be marked as garbage as a result of the **retract** command, but the assertion of the *info* fact with the *gen1* symbol removes the symbol from consideration for garbage collection. Once the rule finishes executing, however, the other data structures associated with the fact are still subject to garbage collection.

This next example is a simpler example of garbage collection that will be used to compare and contrast garbage collection triggered by the command prompt to that triggered by an embedding application.

```
CLIPS> (gensym*)
gen2
CLIPS>
```

The **gensym*** function entered at the command prompt returns the unique symbol *gen2*. This newly created symbol is assumed to be garbage until an outstanding reference to the symbol is established. In this case, once the return value has been displayed and control returned to the command prompt, garbage collection is initiated as part of the command prompt loop and the data structures associated with the symbol can be deallocated,

The following *main* routine is an equivalent embedded program that makes a call to the **gensym*** function.

```
#include "clips.h"

int main()
  {
   void *theEnv;
   DATA_OBJECT rtn;

   theEnv = CreateEnvironment();

   EnvFunctionCall(theEnv,"gensym*",NULL,&rtn);
  }
```

The key difference between this example and the command loop example is that the *gen2* symbol returned to the command loop can be garbage collected after it is printed, but the value returned to the embedding main program can not be safely garbage collected until the embedding program has finished using it.

If the values returned to an embedding program are never garbage collected, continuous execution would result in a program eventually running out of memory. CLIPS addresses this issue by automatically invoking garbage collection for the following embedded functions: **EnvAssert**, **EnvAssertString**, **EnvBuild**, **EnvClear**, **EnvDecrementGCLocks**, **EnvDeleteInstance**, **EnvDirectGetSlot**, **EnvDirectPutSlot**,, **EnvEval**, **EnvFunctionCall**, **EnvMakeInstance**, **EnvReset**, **EnvRetract**, **EnvSend**, **EnvSetDefglobalValue**, **EnvUndefclass**, **EnvUndeffacts**, **EnvUndeffunction**, **EnvUndefgeneric**, **EnvUndefglobal**, **EnvUndefinstances**, **EnvUndefmethod**, **EnvUndefrule**, **EnvUndeftemplate**, and **EnvUnmakeInstance**. Calling one of these functions will not garbage collect any data returned from that call, but it could garbage collect data returned from prior calls.

The following *main* routine is an example of how garbage collection affects whether you can safely access data returned by CLIPS.

```
#include "clips.h"

int main()
  {
   void *theEnv;
   DATA_OBJECT rtn;
   const char *str1, *str2;
```

```
   theEnv = CreateEnvironment();

   EnvFunctionCall(theEnv,"gensym*",NULL,&rtn);
   str1 = DOToString(rtn);

   /* Safe to refer to str1 here. */

   EnvFunctionCall(theEnv,"gensym*",NULL,&rtn);
   str2 = DOToString(rtn);

   /* Not safe to refer to str1 here. */
   /* Safe to refer to str2 here. */
 }
```

The first call to **EnvFunctionCall** could trigger garbage collection, but since no data has been returned yet to the embedding program this does not cause any problems. The next call to DOToString stores the string value in the DATA_OBJECT *rtn* in the *str1* variable. At this point, *str1* can be safely referenced.

The second call to **EnvFunctionCall** could also trigger garbage collection. In this case, however, the value returned by the prior call to **EnvFunctionCall** could be garbage collected as a result. Therefore it is not safe to reference the value stored in *str1* after this point. This is a problem if, for example, you want to compare the value of *str1* to *str2*.

There are two ways to work around this problem. The first is to create your own copies of *str1* and *str2*. This is somewhat inconvenient since you have to determine the size of the strings, allocate space for them, copy them, and then delete them once they're no longer needed. The second way is more convenient. CLIPS provides two functions, **EnvIncrementGCLocks** and **EnvDecrementGCLocks**, which allow you to temporarily disable garbage collection on values returned by API calls. Each call to **EnvIncrementGCLocks** places a lock on the garbage collector for the specified environment. Each call to **EnvDecrementGCLocks** removes a lock from the garbage collector for the specified environment. If the garbage collector has one or more locks placed on it, garbage collection does not occur on values previously returned by CLIPS.

```
   void    EnvIncrementGCLocks(void *);
   void    EnvDecrementGCLocks(void *);
```

The use of these functions is demonstrated in the following revised *main* routine:

```
#include "clips.h"

int main()
  {
   void *theEnv;
   DATA_OBJECT rtn;
```

```
   const char *str1, *str2;

   theEnv = CreateEnvironment();

   EnvIncrementGCLocks(theEnv);

   EnvFunctionCall(theEnv,"gensym*",NULL,&rtn);
   str1 = DOToString(rtn);

   /* Safe to refer to str1 here. */

   EnvFunctionCall(theEnv,"gensym*",NULL,&rtn);
   str2 = DOToString(rtn);

   /* Safe to refer to str1 here. */
   /* Safe to refer to str2 here. */

   EnvDecrementGCLocks(theEnv);

   /* Not safe to refer to str1 here. */
   /* Not safe to refer to str2 here. */
  }
```

In this case, the second call to **EnvFunctionCall** can't garbage collect the string referenced by *str1*, so it is safe to refer to this string after the call. The same effect could also be achieved by moving the **IncrementGCLocks** call after the first call to **FunctionCall**. Calling **EnvDecrementGCLocks** will trigger garbage collection if the number of locks is reduced to 0, so it is no longer safe to reference str1 after this function is called.

Locks shoud not be placed on the garbage collector indiscriminately as shown in the following example:

```
#include "clips.h"

int main()
  {
   void *theEnv;

   theEnv = CreateEnvironment();

   EnvIncrementGCLocks(theEnv);

   EnvLoad(theEnv,"mab.clp");
   EnvReset(theEnv);
   EnvRun(theEnv,-1);

   EnvDecrementGCLocks(theEnv);
  }
```

While calling **EnvReset** could trigger garbage collection on values returned to the embedding program, in this case there are no such values. The **EnvLoad** and **EnvRun** calls won't trigger garbage collection on values previously returned to the embedding program, but they will trigger garbage collection to remove garbage generated during their execution.

In version of CLIPS prior to 6.3, placing locks on the garbage collector disabled *all* garbage collection, not just garbage collection on values returned by CLIPS. Indiscriminate use of locks could cause performance issues if CLIPS was allowed to continuously run with garbage collection disabled.

Locks in version 6.3 only disable garbage collection for values returned by API calls. In this example, the use of locks will not effect performance since none of the API calls return values that are garbage collected. However, it is very important in an embedded program that each call to **EnvIncrementGCLocks** be balanced with a call to **EnvDecrementGCLocks** and that the lock count is periodically reduced to 0. For example, if you have a loop which creates several thousands facts and need to request several pieces of data from CLIPS using API calls to create each fact, it's better to increment/decrement the locks inside the loop (allowing garbage collection to occur after each fact is created) than to increment/decrement the locks outside the loop.

It is only necessary to consider the effects of garbage collection when an embedding program is retrieving data from CLIPS. When calls to a user function by CLIPS are made (such as to a user-defined function), the possible consequences of garbage collection do not have to be considered. In this case, garbage collection will not be triggered for any data retrieved by the user function until after the user function has exited.

Section 2:
Installing and Tailoring CLIPS

This section describes how to install and tailor CLIPS to meet specific needs.

2.1 Installing CLIPS

CLIPS executables for DOS, Windows XP, and MacOS are available for download from the internet. See Appendix A for details. To tailor CLIPS or to install it on another machine, the user must port the source code and create a new executable version.

Testing of CLIPS 6.30 included the following software environments:

- Windows 7 Home Premium 32-bit Operating System with Visual C++ 2010 Express and Windows 7 Professional 64-bit Operating System with Visual Studio 2013.

- MacOS X 10.10.2 using Xcode 6.2.

CLIPS was designed specifically for portability and has been installed on numerous other computers without making modifications to the source code. It *should* run on any system which supports an ANSI C or C++ compiler. Some compilers have extended syntax to support a particular platform which will add additional reserved words to the C language. In the event that this extended syntax conflicts with the CLIPS source, the user will have to edit the code. This usually only involves a global search-and-replace of the particular reserved word. The following steps describe how to create a new executable version of CLIPS:

1) **Load the source code onto the user's system**
   The following C source files are necessary to set up the basic CLIPS system:

agenda.h	dfinsbin.h	incrrset.h	prcdrpsr.h
analysis.h	dfinscmp.h	inherpsr.h	prdctfun.h
argacces.h	drive.h	inscom.h	prntutil.h
bload.h	emathfun.h	insfile.h	proflfun.h
bmathfun.h	engine.h	insfun.h	reorder.h
bsave.h	envrnmnt.h	insmngr.h	reteutil.h
classcom.h	evaluatn.h	insmoddp.h	retract.h
classexm.h	expressn.h	insmult.h	router.h
classfun.h	exprnbin.h	inspsr.h	rulebin.h
classinf.h	exprnops.h	insquery.h	rulebld.h

classini.h
classpsr.h
clips.h
clsltpsr.h
commline.h
conscomp.h
constant.h
constrct.h
constrnt.h
crstrtgy.h
cstrcbin.h
cstrccmp.h
cstrccom.h
cstrcpsr.h
cstrnbin.h
cstrnchk.h
cstrncmp.h
cstrnops.h
cstrnpsr.h
cstrnutl.h
default.h
defins.h
developr.h
dffctbin.h
dffctbsc.h
dffctcmp.h
dffctdef.h
dffctpsr.h
dffnxbin.h
dffnxcmp.h
dffnxexe.h
dffnxfun.h
dffnxpsr.h
exprnpsr.h
extnfunc.h
factbin.h
factbld.h
factcmp.h
factcom.h
factfun.h
factgen.h
facthsh.h
factlhs.h
factmch.h
factmngr.h
factqpsr.h
factqury.h
factprt.h
factrete.h
factrhs.h
filecom.h
filertr.h
generate.h
genrcbin.h
genrccmp.h
genrccom.h
genrcexe.h
genrcfun.h
genrcpsr.h
globlbin.h
globlbsc.h
globlcmp.h
globlcom.h
globldef.h
globlpsr.h
immthpsr.h
insqypsr.h
iofun.h
lgcldpnd.h
match.h
memalloc.h
miscfun.h
modulbin.h
modulbsc.h
modulcmp.h
moduldef.h
modulpsr.h
modulutl.h
msgcom.h
msgfun.h
msgpass.h
msgpsr.h
multifld.h
multifun.h
network.h
objbin.h
objcmp.h
object.h
objrtbin.h
objrtbld.h
objrtcmp.h
objrtfnx.h
objrtgen.h
objrtmch.h
parsefun.h
pattern.h
pprint.h
prccode.h
prcdrfun.h
rulebsc.h
rulecmp.h
rulecom.h
rulecstr.h
ruledef.h
ruledlt.h
rulelhs.h
rulepsr.h
scanner.h
setup.h
sortfun.h
strngfun.h
strngrtr.h
symblbin.h
symblcmp.h
symbol.h
sysdep.h
textpro.h
tmpltbin.h
tmpltbsc.h
tmpltcmp.h
tmpltdef.h
tmpltfun.h
tmpltlhs.h
tmpltpsr.h
tmpltrhs.h
tmpltutl.h
userdata.h
utility.h
watch.h

agenda.c
analysis.c
argacces.c
bload.c
bmathfun.c
bsave.c
classcom.c
classexm.c
classfun.c
dfinscmp.c
drive.c
emathfun.c
engine.c
envrnmnt.c
evaluatn.c
expressn.c
exprnbin.c
exprnops.c
immthpsr.c
incrrset.c
inherpsr.c
inscom.c
insfile.c
insfun.c
insmngr.c
insmoddp.c
insmult.c
prcdrfun.c
prcdrpsr.c
prdctfun.c
prntutil.c
proflfun.c
reorder.c
reteutil.c
retract.c
router.c

classinf.c	exprnpsr.c	inspsr.c	rulebin.c
classini.c	extnfunc.c	insquery.c	rulebld.c
classpsr.c	factbin.c	insqypsr.c	rulebsc.c
clsltpsr.c	factbld.c	iofun.c	rulecmp.c
commline.c	factcmp.c	lgcldpnd.c	rulecom.c
conscomp.c	factcom.c	main.c	rulecstr.c
constrct.c	factfun.c	memalloc.c	ruledef.c
constrnt.c	factgen.c	miscfun.c	ruledlt.c
crstrtgy.c	facthsh.c	modulbin.c	rulelhs.c
cstrcbin.c	factlhs.c	modulbsc.c	rulepsr.c
cstrccom.c	factmch.c	modulcmp.c	scanner.c
cstrcpsr.c	factmngr.c	moduldef.c	sortfun.c
cstrnbin.c	factprt.c	modulpsr.c	strngfun.c
cstrnchk.c	factqpsr.c	modulutl.c	strngrtr.c
cstrncmp.c	factqury.c	msgcom.c	symblbin.c
cstrnops.c	factrete.c	msgfun.c	symblcmp.c
cstrnpsr.c	factrhs.c	msgpass.c	symbol.c
cstrnutl.c	filecom.c	msgpsr.c	sysdep.c
default.c	filertr.c	multifld.c	textpro.c
defins.c	generate.c	multifun.c	tmpltbin.c
developr.c	genrcbin.c	objbin.c	tmpltbsc.c
dffctbin.c	genrccmp.c	objcmp.c	tmpltcmp.c
dffctbsc.c	genrccom.c	objrtbin.c	tmpltdef.c
dffctcmp.c	genrcexe.c	objrtbld.c	tmpltfun.c
dffctdef.c	genrcfun.c	objrtcmp.c	tmpltlhs.c
dffctpsr.c	genrcpsr.c	objrtfnx.c	tmpltpsr.c
dffnxbin.c	globlbin.c	objrtgen.c	tmpltrhs.c
dffnxcmp.c	globlbsc.c	objrtmch.c	tmpltutl.c
dffnxexe.c	globlcmp.c	parsefun.c	userdata.c
dffnxfun.c	globlcom.c	pattern.c	userfunctions.c
dffnxpsr.c	globldef.c	pprint.c	utility.c
dfinsbin.c	globlpsr.c	prccode.c	watch.c

Additional files must also be included if one of the machine specific user interfaces is to be set up. See the *Utilities and Interfaces Guide* for details on compiling the machine specific interfaces.

2) **Modify all include statements (if necessary)**
All of the ".c" files and most of the ".h" files have #include statements. These #include statements may have to be changed to either match the way the compiler searches for include files or to include a different ".h" file.

3) **Tailor CLIPS environment and/or features**

Edit the setup.h file and set any special options. CLIPS uses preprocessor definitions to allow machine-dependent features. The first set of definitions in the setup.h file tells CLIPS on what kind of machine the code is being compiled. The default setting for this definition is GENERIC, which will create a version of CLIPS that will run on any computer. The user may set the definition for the user's type of system. If the system type is unknown, the definition should be set to GENERIC (so for this situation you do not need to edit setup.h). If you change the system type to anything other than GENERIC, make sure that the version number of your compiler is greater than or equal to the version number listed in the setup.h file (as earlier versions of a compiler may not support some system dependent features). Other preprocessor definitions in the setup.h file also allow a user to tailor the features in CLIPS to specific needs. For more information on using the flags, see section 2.2.

Optionally, preprocessor definitions can be set using the appropriate command line argument used by your compiler, removing the need to directly edit the setup.h file. For example, the command line option –DUNIX_7 will work on many compilers to set the preprocessor definition of UNIX_7 to 1.

4) **Compile all of the ".c" files to object code**
Use the standard compiler syntax for the user's machine. The ".h" files are include files used by the other files and do not need to be compiled. Some options may have to be set, depending on the compiler.

If user-defined functions are needed, compile the source code for those functions as well and modify the EnvUserFunctions definition in userfunctions.c to reflect the user's functions (see section 3 for more on user-defined functions).

5) **Create the interactive CLIPS executable element**
To create the interactive CLIPS executable, link together all of the object files. This executable will provide the interactive interface defined in section 2.1 of the *Basic Programming Guide*.

2.1.1 Additional Considerations

Although compiling CLIPS should not be difficult even for inexperienced C programmers, some non-obvious problems can occur. One type of problem is linking with inappropriate system libraries. Normally, default libraries are specified through the environment; i.e., not specified as a part of the compile/link process. On occasion, the default system libraries are inappropriate for use with CLIPS. For example, when using a compiler which supports different memory models, be sure to link with the system libraries that match the memory model under which the CLIPS code was compiled. The same can be said for floating-point models. Some computers provide multiple ways of storing floating-point numbers (typically differing in accuracy or speed of processing). Be sure to link with system libraries that use the same storage formats with which the

CLIPS code was compiled. Some additional considerations for compiling CLIPS with specific compilers and/or operating systems are described following.

UNIX
If the **EXTENDED_MATH_FUNCTIONS** compiler directive is enabled, then the **-lm** option must be used when compiling CLIPS with the gcc command. If all of the CLIPS source code is contained in the same directory and the compiler directives are set to their default values in the **setup.h** file, then the following command line will compile CLIPS.

```
gcc -o clips *.c -lm
```

GCC
If the **–O** optimization option is specified, then the **-fno-strict-aliasing** option should also be specified. The **–x c++** option can be used to force compilation of CLIPS as a C++ program. If used the **-lstdc++** option should also be used to link with C++ libraries. The following command line will compile CLIPS as a C++ program.

```
gcc -o clips -x c++ *.c -lstdc++
```

Visual C++
The following steps assume you have Microsoft Visual Studio 2013 installed and want to compile the core CLIPS source code from the DOS command prompt rather than using the Visual Studio environment.

First, launch the Command Prompt application (select Start > All Programs > Accessories > Command Prompt). Next, execute the script that sets up the environment variables for the appropriate target machine. For example, the vcvars64.bat batch file in the directory “Program Files (x86)/Microsoft Visual Studio 12.0/VC/bin/amd64”. Use the cd command to change the current directory to the one containing the core CLIPS source code. The following command will then create the CLIPS executable.

```
cl /Feclips.exe *.c
```

2.2 Tailoring CLIPS

CLIPS makes use of **preprocessor definitions** (also referred to in this document as **compiler directives** or **setup flags**) to allow easier porting and recompiling of CLIPS. Compiler directives allow the incorporation of system-dependent features into CLIPS and also make it easier to tailor CLIPS to specific applications. All available compiler options are controlled by a set of flags defined in the **setup.h** file.

The first flag in **setup.h** indicates on what type of compiler/machine CLIPS is to run. The source code is sent out with the flag for GENERIC CLIPS turned on. When compiled in this mode, all system-dependent features of CLIPS are excluded and the program should run on any system. A

number of other flags are available in this file, indicating the types of compilers/machines on which CLIPS has been compiled previously. If the user's implementation matches one of the available flags, set that flag to 1 and turn the **GENERIC** flag off (set it to 0). The code for most of the features controlled by the compiler/machine-type flag is in the **sysdep.c** file.

Many other flags are provided in **setup.h**. Each flag is described below.

BLOAD This flag controls access to the binary load command (bload). This would be used to save some memory in systems which require binary load but not save capability. This is off in the standard CLIPS executable.

BLOAD_AND_BSAVE
This flag controls access to the binary load and save commands. This would be used to save some memory in systems which require neither binary load nor binary save capability. This is on in the standard CLIPS executable.

BLOAD_INSTANCES
This flag controls the ability to load instances in binary format from a file via the **bload-instances** command (see section 13.11.4.7 of the *Basic Programming Guide*). This is on in the standard CLIPS executable. Turning this flag off can save some memory.

BLOAD_ONLY This flag controls access to the binary and ASCII load commands (bload and load). This would be used to save some memory in systems which require binary load capability only. This flag is off in the standard CLIPS executable.

BSAVE_INSTANCES
This flag controls the ability to save instances in binary format to a file via the **bsave-instances** command (see section 13.11.4.4 of the *Basic Programming Guide*). This is on in the standard CLIPS executable. Turning this flag off can save some memory.

CONSTRUCT_COMPILER
This flag controls the construct compiler functions. If it is turned on, constructs may be compiled to C code for use in a run-time module (see section 5). This is off in the standard CLIPS executable.

DEBUGGING_FUNCTIONS
This flag controls access to commands such as agenda, facts, ppdefrule, ppdeffacts, etc. This would be used to save some

memory in BLOAD_ONLY or RUN_TIME systems. This flag is on in the standard CLIPS executable.

DEFFACTS_CONSTRUCT

This flag controls the use of deffacts. If it is off, deffacts are not allowed which can save some memory and performance during resets. This is on in the standard CLIPS executable. If this flag is off, the (initial-fact) fact is still created during a reset if the DEFTEMPLATE_CONSTRUCT flag is on.

DEFFUNCTION_CONSTRUCT

This flag controls the use of deffunction. If it is off, deffunction is not allowed which can save some memory. This is on in the standard CLIPS executable.

DEFGENERIC_CONSTRUCT

This flag controls the use of defgeneric and defmethod. If it is off, defgeneric and defmethod are not allowed which can save some memory. This is on in the standard CLIPS executable.

DEFGLOBAL_CONSTRUCT

This flag controls the use of defglobal. If it is off, defglobal is not allowed which can save some memory. This is on in the standard CLIPS executable.

DEFINSTANCES_CONSTRUCT

This flag controls the use of definstances (see section 9.6.1.1 of the *Basic Programming Guide*). If it is off, definstances are not allowed which can save some memory and performance during resets. This is on in the standard CLIPS executable. If this flag is off, the [initial-object] instance is still created during a reset if the INSTANCE_PATTERN_MATCHING flag is on.

DEFMODULE_CONSTRUCT

This flag controls the use of the defmodule construct. If it is off, then new defmodules cannot be defined (however the MAIN module will exist). This is on in the standard CLIPS executable.

DEFRULE_CONSTRUCT

This flag controls the use of the defrule construct. If it is off, the defrule construct is not recognized by CLIPS. This is on in the standard CLIPS executable.

DEFTEMPLATE_CONSTRUCT

This flag controls the use of deftemplate. If it is off, deftemplate is not allowed which can save some memory. This is on in the standard CLIPS executable.

EXTENDED_MATH_FUNCTIONS

This flag indicates whether the extended math package should be included in the compilation. If this flag is turned off (set to 0), the final executable will be about 25-30K smaller, a consideration for machines with limited memory. This is on in the standard CLIPS executable.

FACT_SET_QUERIES

This flag determines if the fact-set query functions are available. These functions are **any-factp**, **do-for-fact**, **do-for-all-facts**, **delayed-do-for-all-facts**, **find-fact**, and **find-all-facts**,. This is on in the standard CLIPS executable. Turning this flag off can save some memory.

INSTANCE_SET_QUERIES

This flag determines if the instance-set query functions are available. These functions are **any-instancep**, **do-for-instance**, **do-for-all-instances**, **delayed-do-for-all-instances**, **find-instance**, and **find-all-instances**,. This is on in the standard CLIPS executable. Turning this flag off can save some memory.

IO_FUNCTIONS

This flag controls access to the I/O functions in CLIPS. These functions are **printout**, **read**, **open**, **close**, **format**, and **readline**. If this If this flag is off, these functions are not available. This would be used to save some memory in systems which used custom I/O routines. This is on in the standard CLIPS executable.

MULTIFIELD_FUNCTIONS

This flag controls access to the multifield manipulation functions in CLIPS. These functions are **subseq$**, **delete$**, **insert$**, **replace$**, **explode$**, **implode$**, **nth$**, **member$**, **first$**, **rest$**, **progn$**, and **subsetp**. The function **create$** is always available regardless of the setting of this flag. This would be used to save some memory in systems which performed limited or no operations with multifield values. This flag is on in the standard CLIPS executable.

OBJECT_SYSTEM

This flag controls the use of defclass, definstances, and defmessage-handler. If it is off, these constructs are not allowed which can save

some memory. If this flag is on, the MULTIFIELD_FUNCTIONS flag should also be on if you want to be able to manipulate multifield slots. This is on in the standard CLIPS executable.

PROFILING_FUNCTIONS

This flag controls access to the profiling functions in CLIPS. These functions are **get-profile-percent-threshold**, **profile**, **profile-info**, **profile-reset**, and **set-profile-percent-threshold**. This flag is on in the standard CLIPS executable.

RUN_TIME

This flag will create a run-time version of CLIPS for use with compiled constructs. It should be turned on only *after* the constructs-to-c function has been used to generate the C code representation of the constructs, but *before* compiling the constructs C code. When used, about 90K of memory can be saved from the basic CLIPS executable. See section 5 for a description of how to use this. This is off in the standard CLIPS executable.

STRING_FUNCTIONS

This flag controls access to the string manipulation functions in CLIPS. These functions are **str-cat**, **sym-cat**, **str-length**, **str-compare**, **upcase**, **lowcase**, **sub-string**, **str-index**, **eval**, and **build**. This would be used to save some memory in systems which perform limited or no operations with strings. This flag is on in the standard CLIPS executable.

TEXTPRO_FUNCTIONS

This flag controls the CLIPS text-processing functions. It must be turned on to use the **fetch**, **toss**, and **print-region** functions in a user-defined help system. This is on in the standard CLIPS executable.

WINDOW_INTERFACE

This flag indicates that a windowed interface is being used. In some cases, this may include CLIPS console applications (for example Win32 console applications as opposed to a DOS application). This is off in the standard CLIPS executable.

Section 3:
Integrating CLIPS with External Functions

One of the most important features of CLIPS is an ability to integrate CLIPS with **external functions** or applications. This section discusses how to add external functions to CLIPS and how to pass arguments to them and return values from them. A user can define external functions for use by CLIPS at any place a function can normally be called. In fact, the vast majority of system defined functions and commands provided by CLIPS are integrated with CLIPS in the exact same manner described in this section. The examples shown in this section are in C, but section 6 discusses how other languages can be combined with CLIPS. Prototypes for the functions listed in this section can be included by using the **clips.h** header file.

3.1 Declaring User-Defined External Functions

All external functions must be described to CLIPS so they can be properly accessed by CLIPS programs. User-defined functions are described to CLIPS by modifying the function **EnvUserFunctions** which resides in the CLIPS **userfunctions.c** file. Within **EnvUserFunctions**, a call should be made to the **EnvDefineFunction** routine for every function which is to be integrated with CLIPS. The user's source code then can be compiled and linked with CLIPS. Alternately, the user can call **EnvDefineFunction** from their own initialization code—the only restrictions is that it must be called after CLIPS has been initialized and before the user-defined function is referenced.

```
int            EnvDefineFunction(environment,functionName,functionType,
                                 functionPointer,actualFunctionName);

void          *environment;
const char    *functionName, *actualFunctionName;
char           functionType;
int           (*functionPointer)(void *);
```

An example **UserFunctions** declaration follows:

```
void EnvUserFunctions(
  void *environment)
  {
   /*=======================================*/
   /* Declare your C functions if necessary. */
   /*=======================================*/

   extern double rta(void *);
```

```
   extern long long mul(void *);

   /*=========================================================*/
   /* Call DefineFunction to register user-defined functions. */
   /*=========================================================*/

   EnvDefineFunction(environment,"rta",'d',PTIEF rta,"rta");
   EnvDefineFunction(environment,"mul",'g',PTIEF mul,"mul");
  }
```

The first argument to **EnvDefineFunction** is the CLIPS environment in which the function is to be defined.

The second argument to **EnvDefineFunction** is the CLIPS function name, a string representation of the name that will be used when calling the function from within CLIPS.

The third argument is the type of the value which will be returned to CLIPS. Note that this is not necessarily the same as the function type. Allowable return types are shown as follows:

Return Code	Return Type Expected
a	External Address
b	Boolean
c	Character
d	Double Precision Float
f	Single Precision Float
g	Long Long Integer
i	Integer
j	Unknown Data Type (Symbol, String, or Instance Name Expected)
k	Unknown Data Type (Symbol or String Expected)
l	Long Integer
m	Multifield
n	Unknown Data Type (Integer or Float Expected)
o	Instance Name
s	String
u	Unknown Data Type (Any Type Expected)
v	Void—No Return Value
w	Symbol
x	Instance Address
y	Fact Address

Boolean functions should return a value of type int (0 for the symbol FALSE and any other value for the symbol TRUE). String, symbol, instance name, external address, fact address, and instance address functions should return a pointer of type void *. Character return values are converted by CLIPS to a symbol of length one. Integer and long return values are converted by CLIPS to long long integers for internal storage. Single precision float values are converted by

CLIPS to double precision float values for internal storage. If a user function is not going to return a value to CLIPS, the function should be defined as type void and this argument should be v for void. Return types *o* and *x* are only available if the object system has been enabled (see section 2.2).

Function types *j*, *k*, *m*, *n*, and *u* are all passed a data object as an argument in which the return value of function is stored. This allows a user defined function to return one of several possible return types. Function type *u* is the most general and can return any data type. By convention, function types *j*, *k*, *m*, and *n* return specific data types. CLIPS will signal an error if one of these functions return a disallowed type. See section 3.3.5 for more details on returning unknown data types.

The fourth argument is a pointer to the actual function, the compiled function name (an **extern** declaration of the function may be appropriate). The CLIPS name (second argument) need not be the same as the actual function name (fourth argument). The macro identifier PTIEF can be placed in front of a function name to cast it as a pointer to a function returning an integer (primarily to prevent warnings from compilers which allow function prototypes).

The fifth argument is a string representation of the fourth argument (the pointer to the actual C function). This name *should be identical* to the third argument, but enclosed in quotation marks.

EnvDefineFunction returns zero if the function was unsuccessfully called (e.g. bad function type parameter), otherwise a non-zero value is returned.

User-defined functions are searched before system functions. If the user defines a function which is the same as one of the defined functions already provided, the user function will be executed in its place. Appendix H of the *Basic Programming Guide* contains a list of function names used by CLIPS.

In place of **EnvDefineFunction**, the **EnvDefineFunction2** function can be used to provide additional information to CLIPS about the number and types of arguments expected by a CLIPS function or command.

```
int            EnvDefineFunction2(environment,functionName,functionType,
                                  functionPointer,actualFunctionName,
                                  functionRestrictions);

void          *environment;
const char    *functionName, functionType, *actualFunctionName;
int           (*functionPointer)(void *);
const char    *functionRestrictions;
```

The first five arguments to **DefineFunction2** are identical to the five arguments for **DefineFunction**. The sixth argument is a restriction string which indicates the number and types of arguments that the CLIPS function expects. The syntax format for the restriction string is

```
<min-args> <max-args> [<default-type> <types>*]
```

The values <min-args> and <max-args> must be specified in the string. Both values must either be a character digit (0-9) or the character *. A digit specified for <min-args> indicates that the function must have at least <min-args> arguments when called. The character * for this value indicates that the function does not require a minimum number of arguments. A digit specified for <max-args> indicates that the function must have no more than <max-args> arguments when called. The character * for this value indicates that the function does not prohibit a maximum number of arguments. The optional <default-type> is the assumed type for each argument for a function call. Following the <default-type>, additional type values may be supplied to indicate specific type values for each argument. The type codes for the arguments are as follows:

Type Code	Allowed Types
a	External Address
d	Float
e	Instance Address, Instance Name, or Symbol
f	Float
g	Integer, Float, or Symbol
h	Instance Address, Instance Name, Fact Address, Integer, or Symbol
i	Integer
j	Symbol, String, or Instance Name
k	Symbol or String
l	Integer
m	Multifield
n	Integer or Float
o	Instance Name
p	Instance Name or Symbol
q	Symbol, String, or Multifield
s	String
u	Any Data Type
w	Symbol
x	Instance Address
y	Fact Address
z	Fact address, Integer, or Symbol

Examples

The restriction string for a function requiring a minimum of three arguments is:

```
"3*"
```

The restriction string for a function requiring no more than five arguments is:

```
"*5"
```

The restriction string for a function requiring at least three and no more than five arguments (each of which must be an integer or float) is:

```
"35n"
```

The restriction string for a function requiring exactly six arguments (of which the first must be a string, the third an integer, and the remaining arguments floats) is:

```
"66fsui"
```

3.2 Passing Arguments from CLIPS to External Functions

Although arguments are listed directly following a function name within a function call, CLIPS actually calls the function without any arguments. The arguments are stored internally by CLIPS and can be accessed by calling the argument access functions. Access functions are provided to determine both the number and types of arguments.

3.2.1 Determining the Number of Passed Arguments

User-defined functions should first determine that they have been passed the correct number of arguments. Several functions are provided for this purpose.

```
int            EnvRtnArgCount(environment);
int            EnvArgCountCheck(environment,functionName,restriction,count);
int            EnvArgRangeCheck(environment,functionName,min,max);

void          *environment;
int            restriction, count, min, max;
const char    *functionName;
```

A call to **EnvRtnArgCount** will return an integer telling how many arguments with which the function was called. The function **EnvArgCountCheck** can be used for error checking if a function expects a minimum, maximum, or exact number of arguments (but not a combination of these restrictions). It returns an integer telling how many arguments with which the function was called (or -1 if the argument restriction for the function was unsatisfied). The second argument is the name of the function to be printed within the error message if the restriction is unsatisfied. The *restriction* argument should be one of the values NO_MORE_THAN, AT_LEAST, or EXACTLY. The *count* argument should contain a value for the number of arguments to be used

in the restriction test. The function **EnvArgRangeCheck** can be used for error checking if a function expects a range of arguments. It returns an integer telling how many arguments with which the function was called (or -1 if the argument restriction for the function was unsatisfied). The second argument is the name of the function to be printed within the error message if the restriction is unsatisfied. The third argument is the minimum number of arguments and the fourth argument is the maximum number of arguments.

3.2.2 Passing Symbols, Strings, Instance Names, Floats, and Integers

Several access functions are provided to retrieve arguments that are symbols, strings, instance names, floats, and integers.

```
const char    *EnvRtnLexeme(environment,argumentPosition);
double         EnvRtnDouble(environment,argumentPosition);
long long      EnvRtnLong(environment,argumentPosition);

void          *environment;
int            argumentPosition;
```

A call to **EnvRtnLexeme** returns a character pointer from either a symbol, string, or instance name data type (NULL is returned if the type is not SYMBOL, STRING, or INSTANCE_NAME), **EnvRtnDouble** returns a floating-point number from either an INTEGER or FLOAT data type, and **EnvRtnLong** returns a long long integer from either an INTEGER or FLOAT data type. The arguments have to be requested one at a time by specifying each argument's position number as the *argumentPosition* to **EnvRtnLexeme**, **EnvRtnDouble**, or **EnvRtnLong**. If the type of argument is unknown, another function can be called to determine the type. See section 3.2.3 for a further discussion of unknown argument types. *Do not* store the pointer returned by **EnvRtnLexeme** as part of a permanent data structure. When CLIPS performs garbage collection on symbols and strings, the pointer reference to the string may be rendered invalid. To store a permanent reference to a string, allocate storage for a copy of the string and then copy the string returned by **EnvRtnLexeme** to the copy's storage area.

Example

The following code is for a function to be called from CLIPS called **rta** which will return the area of a right triangle.

```
                                /* This include definition      */
#include "clips.h"              /* should start each file which */
                                /* has CLIPS functions in it    */

/*
Use EnvDefineFunction2(environment,"rta",'d',PTIEF rta,"rta","22n");
*/

double rta(
```

```
void *environment)
{
 double base, height;

 /*==================================*/
 /* Check for exactly two arguments. */
 /*==================================*/

 if (EnvArgCountCheck(environment,"rta",EXACTLY,2) == -1) return(-1.0);

 /*================================================*/
 /* Get the values for the 1st and 2nd arguments. */
 /*================================================*/

 base = EnvRtnDouble(environment,1);
 height = EnvRtnDouble(environment,2);

 /*==================================*/
 /* Return the area of the triangle. */
 /*==================================*/

 return(0.5 * base * height);
}
```

As previously shown, **rta** also should be defined in **EnvUserFunctions**. If the value passed from CLIPS is not the data type expected, an error occurs. Section 3.2.3 describes a method for testing the data type of the passed arguments which would allow user-defined functions to do their own error handling. Once compiled and linked with CLIPS, the function **rta** could be called as shown following.

```
CLIPS> (rta 5.0 10.0)
25.0
CLIPS> (assert (right-triangle-area (rta 20.0 10.0)))
<Fact-1>
CLIPS> (facts)
f-0     (initial-fact)
f-1     (right-triangle-area 100.0)
For a total of 2 facts.
CLIPS>
```

3.2.3 Passing Unknown Data Types

Section 3.2.2 described how to pass data to and from CLIPS when the type of data is explicitly known. It also is possible to pass parameters of an unknown data type to and from external functions. To pass an unknown parameter *to* an external function, use the **RtnUnknown** function.

```
#include "clips.h"

DATA_OBJECT   *EnvRtnUnknown(environment,argumentPosition,&argument);
```

```
int            GetType(argument);
int            GetpType(&argument);
int            ArgTypeCheck(environment,functionName,argumentPosition,
                            expectedType,&argument);

const char    *DOToString(argument);
const char    *DOPToString(&argument);
double         DOToDouble(argument);
double         DOPToDouble(&argument);
long long      DOToLong(argument);
long long      DOPToLong(&argument);
void          *DOToPointer(argument);
void          *DOPToPointer(&argument);
void          *DOToExternalAddress(argument);
void          *DOPToExternalAddress(&argument);

void          *environment;
const char    *functionName;
int            argumentPosition, expectedType;
DATA_OBJECT    argument;
```

Function **RtnUnknown** should be called first. It copies the elements of the internal CLIPS structure that represent the unknown-type argument into the DATA_OBJECT structure pointed to by the third argument. It also returns a pointer to that same structure, passed as the third argument. After obtaining a pointer to the DATA_OBJECT structure, a number of macros can be used to extract type information and the arguments value.

Macros **GetType** or **GetpType** can be used to determine the type of argument and will return an integer (STRING, SYMBOL, FLOAT, INTEGER, MULTIFIELD, FACT_ADDRESS, INSTANCE_ADDRESS, INSTANCE_NAME, or EXTERNAL_ADDRESS) defined in the **clips.h** file. Once the data type is known, the functions **DOToDouble**, **DOPToDouble** (for FLOAT), **DOToString**, or **DOPToString** (for STRING, SYMBOL, or INSTANCE_NAME), **DOToLong**, **DOPToLong** (for INTEGER), and **DOToPointer** and **DOPToPointer** (for INSTANCE_ADDRESS and FACT_ADDRESS), and **DOToExternalAddress** and **DOPToExternalAddress** (for EXTERNAL_ADDRESS) can be used to extract the actual value of the variable from the DATA_OBJECT structure. Accessing multifield values is discussed in section 3.2.4. *Do not* store the pointer returned by **DOToString** or **DOPToString** as part of a permanent data structure. When CLIPS performs garbage collection on symbols and strings, the pointer reference to the string may be rendered invalid. To store a permanent reference to a string, allocate storage for a copy of the string and then copy the string returned by **DOToString** or **DOPToString** to the copy's storage area.

The function **EnvArgTypeCheck** can be used for error checking if a function expects a specific type of argument for a particular parameter. It returns a non-zero integer value if the parameter was of the specified type, otherwise it returns zero. The first argument is a pointer to the environment in which the function was invoked. The second argument is the name of the

function to be printed within the error message if the type restriction is unsatisfied. The third argument is the index of the parameter to be tested. The fourth argument is the type restriction and must be one of the following CLIPS defined constants: STRING, SYMBOL, SYMBOL_OR_STRING, FLOAT, INTEGER, INTEGER_OR_FLOAT, MULTIFIELD, EXTERNAL_ADDRESS, FACT_ADDRESS, INSTANCE_ADDRESS, INSTANCE_NAME, or INSTANCE_OR_INSTANCE_NAME. If the FLOAT type restriction is used, then integer values will be converted to floating-point numbers. If the INTEGER type restriction is used, then floating-point values will be converted to integers. The fifth argument is a pointer to a DATA_OBJECT structure in which the unknown parameter will be stored.

Example

The following function **mul** takes two arguments from CLIPS. Each argument should be either an integer or a float. Float arguments are rounded and converted to the nearest integer. Once converted, the two arguments are multiplied together and this value is returned. If an error occurs (wrong type or number of arguments), then the value 1 is returned.

```
#include <math.h>          /* ANSI C library header file */
#include "clips.h"

/*
Use EnvDefineFunction2(environment,"mul",'g',PTIEF mul,"mul","22n");
*/

long long mul(
  void *environment)
  {
   DATA_OBJECT temp;
   long long firstNumber, secondNumber;

   /*==================================*/
   /* Check for exactly two arguments. */
   /*==================================*/

   if (EnvArgCountCheck(environment,"mul",EXACTLY,2) == -1)
     { return(1LL); }

   /*============================================================*/
   /* Get the first argument using the EnvArgTypeCheck function. */
   /* Return if the correct type has not been passed.            */
   /*============================================================*/

   if (EnvArgTypeCheck(environment,"mul",1,INTEGER_OR_FLOAT,&temp) == 0)
     { return(1LL); }

   /*============================================================*/
   /* Convert the first argument to a long long integer. If it's */
   /* not an integer, then it must be a float (so round it to    */
   /* the nearest integer using the C library ceil function.     */
   /*============================================================*/
```

```
   if (GetType(temp) == INTEGER)
     { firstNumber = DOToLong(temp); }
   else /* the type must be FLOAT */
     { firstNumber = (long long) ceil(DOToDouble(temp) - 0.5); }

   /*=========================================================*/
   /* Get the second argument using the EnvRtnUnknown function. */
   /* Note that no type error checking is performed.            */
   /*=========================================================*/

   EnvRtnUnknown(environment,2,&temp);

   /*=========================================================*/
   /* Convert the second argument to a long integer. If it's  */
   /* not an integer or a float, then it's the wrong type.    */
   /*=========================================================*/

   if (GetType(temp) == INTEGER)
     { secondNumber = DOToLong(temp); }
   else if (GetType(temp) == FLOAT)
     { secondNumber = (long long) ceil(DOToDouble(temp) - 0.5); }
   else
     { return(1LL); }

   /*=========================================================*/
   /* Multiply the two values together and return the result. */
   /*=========================================================*/

   return (firstNumber * secondNumber);
  }
```

Once compiled and linked with CLIPS, the function **mul** could be called as shown following.

```
CLIPS> (mul 3 3)
9
CLIPS> (mul 3.1 3.1)
9
CLIPS> (mul 3.8 3.1)
12
CLIPS> (mul 3.8 4.2)
16
CLIPS>
```

3.2.4 Passing Multifield Values

Data passed from CLIPS to an external function may be stored in multifield values. To access a multifield value, the user first must call **EnvRtnUnknown** or **EnvArgTypeCheck** to get the pointer. If the argument is of type MULTIFIELD, several macros can be used to access the values of the multifield value.

```
#include "clips.h

long           GetDOLength(argument);
long           GetpDOLength(&argument);
long           GetDOBegin(argument);
long           GetpDOBegin(&argument);
long           GetDOEnd(argument);
long           GetpDOEnd(&argument);
int            GetMFType(multifieldPtr,fieldPosition);
void          *GetMFValue(multifieldPtr,fieldPosition);

DATA_OBJECT    argument;
void          *multifieldPtr;
int            fieldPosition;
```

Macros **GetDOLength** and **GetpDOLength** can be used to determine the length of a DATA_OBJECT or DATA_OBJECT_PTR respectively. The macros **GetDOBegin**, **GetpDOBegin**, **GetDOEnd**, **GetpDOEnd** can be used to determine the beginning and ending indices of a DATA_OBJECT or DATA_OBJECT_PTR containing a multifield value. Since multifield values are often extracted from arrays of other data structures (such as facts), these indices are used to indicate the beginning and ending positions within the array. Thus it is very important when traversing a multifield value to use indices that run from the begin index to the end index and not from one to the length of the multifield value. The begin index points to the first element in the multifield value and the end index points to the last element in the multifield value. A multifield value of length one will have the same values for the begin and end indices. A multifield value of length zero will have an end index that is one less than the begin index.

The macros **GetMFType** and **GetMFValue** can be used to examine the types and values of fields within a multifield value. The first argument to these macros should be the value retrieved from a DATA_OBJECT or DATA_OBJECT_PTR using the **GetValue** and **GetpValue** macros. The second argument is the index of the field within the multifield value. Once again, this argument should fall in the range between the begin index and the end index for the DATA_OBJECT from which the multifield value is stored. Macros **ValueToString**, **ValueToDouble**, and **ValueToLong** can be used to convert the retrieved value from **GetMFValue** to a C object of type char * (for types SYMBOL, STRING, and INSTANCE_NAME), double (for type FLOAT), and long long (for type INTEGER) respectively. **ValueToPointer** can be used to convert the retrieved value from **GetMFValue** to a C object of type void * (for types FACT_ADDRESS and INSTANCE_ADDRESS). **ValueToExternalAddress** can be used to convert the retrieved value from **GetMFValue** to a C object of type void * (for type EXTERNAL_ADDRESS). *Do not* store the pointer returned by **ValueToString** as part of a permanent data structure. When CLIPS performs garbage collection on symbols and strings, the pointer reference to the string may be rendered invalid. To store a permanent reference to a string, allocate storage for a copy of the string and then copy the string returned by **ValueToString** to the copy's storage area.

The multifield macros should only be used on DATA_OBJECTs that have type MULTIFIELD (e.g. the macro GetDOLength returns erroneous values if the type is not MULTIFIELD).

Examples

The following function returns the length of a multifield value. It returns -1 if an error occurs.

```
#include "clips.h"

/*
Use EnvDefineFunction2(environment,"mfl",'g',PTIEF MFLength,"MFLength","11m");
*/

long long MFLength(
  void *environment)
  {
   DATA_OBJECT argument;

   /*==================================*/
   /* Check for exactly one argument. */
   /*==================================*/

   if (EnvArgCountCheck(environment,"mfl",EXACTLY,1) == -1) return(-1LL);

   /*=====================================================*/
   /* Check that the 1st argument is a multifield value. */
   /*=====================================================*/

   if (EnvArgTypeCheck(environment,"mfl",1,MULTIFIELD,&argument) == 0)
     { return(-1LL); }

   /*===========================================*/
   /* Return the length of the multifield value. */
   /*===========================================*/

   return ((long long) GetDOLength(argument));
  }
```

The following function counts the number of characters in the symbols and strings contained within a multifield value.

```
#include "clips.h"

/*
Use EnvDefineFunction2(environment,"cmfc",'g',PTIEF CntMFChars,"CntMFChars","11m");
*/

long long CntMFChars(
  void *environment)
  {
   DATA_OBJECT argument;
   void *multifieldPtr;
```

```
  long end, i;
  long long count = 0;
  const char *tempPtr;

  /*=================================*/
  /* Check for exactly one argument. */
  /*=================================*/

  if (EnvArgCountCheck(environment,"cmfc",EXACTLY,1) == -1) return(0LL);

  /*=======================================================*/
  /* Check that the first argument is a multifield value. */
  /*=======================================================*/

  if (EnvArgTypeCheck(environment,"cmfc",1,MULTIFIELD,&argument) == 0)
    { return(0LL); }

  /*====================================*/
  /* Count the characters in each field. */
  /*====================================*/

  end = GetDOEnd(argument);
  multifieldPtr = GetValue(argument);
  for (i = GetDOBegin(argument); i <= end; i++)
    {
     if ((GetMFType(multifieldPtr,i) == STRING) ||
         (GetMFType(multifieldPtr,i) == SYMBOL))
       {
        tempPtr = ValueToString(GetMFValue(multifieldPtr,i));
        count += strlen(tempPtr);
       }
    }

  /*============================*/
  /* Return the character count. */
  /*============================*/

  return(count);
 }
```

3.3 Returning Values To CLIPS From External Functions

Functions which return doubles, floats, integers, long integers, characters, external addresses, and instance addresses can directly return these values to CLIPS. Other data types including the unknown (or unspecified) data type and multifield data type, must use functions provided by CLIPS to construct return values.

3.3.1 Returning Symbols, Strings, and Instance Names

CLIPS uses symbol tables to store all symbols, strings, and instance names. Symbol tables increase both performance and memory efficiency during execution. If a user-defined function returns a symbol, string, or an instance name (type 's', 'w', or 'o' in **EnvDefineFunction**), the symbol must be stored in the CLIPS symbol table prior to use. Other types of returns (such as unknown and multifield values) may also contain symbols which must be added to the symbol table. These symbols can be added by calling the function **EnvAddSymbol** and using the returned pointer value.

```
#include "clips.h"

void          *EnvAddSymbol(environment,string);

void          *environment;
const char    *string;
```

Example

This function reverses the character ordering in a string and returns the reversed string. The null string is returned if an error occurs.

```
#include <stdlib.h>            /* ANSI C library header file */
#include <stddef.h>            /* ANSI C library header file */
#include "clips.h"

/*
Use EnvDefineFunction2(environment,"reverse-str",'s',PTIEF Reverse,"Reverse","11s");
*/

void *Reverse(
  void *environment)
  {
   DATA_OBJECT temp;
   const char *lexeme;
   char *tempString;
   void *returnValue;
   size_t length, i;

   /*==================================*/
   /* Check for exactly one argument. */
   /*==================================*/

   if (EnvArgCountCheck(environment,"reverse-str",EXACTLY,1) == -1)
     { return(EnvAddSymbol(environment,"")); }

   /*=========================================================*/
   /* Get the first argument using the ArgTypeCheck function. */
   /*=========================================================*/

   if (EnvArgTypeCheck(environment,"reverse-str",1,STRING,&temp) == 0)
```

```
    { return(EnvAddSymbol(environment,"")); }
  lexeme = DOToString(temp);

  /*=======================================================*/
  /* Allocate temporary space to store the reversed string. */
  /*=======================================================*/

  length = strlen(lexeme);
  tempString = (char *) malloc(length + 1);

  /*====================*/
  /* Reverse the string. */
  /*====================*/

  for (i = 0; i < length; i++)
    { tempString[length - (i + 1)] = lexeme[i]; }
  tempString[length] = '\0';

  /*============================*/
  /* Return the reversed string. */
  /*============================*/

  returnValue = EnvAddSymbol(environment,tempString);
  free(tempString);
  return(returnValue);
 }
```

3.3.2 Returning Boolean Values

A user function may return a boolean value in one of two ways. The user may define an integer function and use **EnvDefineFunction** to declare it as a BOOLEAN type ('b'). The function should then either return the value **TRUE** or **FALSE**. Alternatively, the function may be declare to return a SYMBOL type ('w') or UNKNOWN type ('u') and return the value of **EnvFalseSymbol** or **EnvTrueSymbol** macro.

```
#include "clips.h"

void          *EnvFalseSymbol(environment);
void          *EnvTrueSymbol(environment);

void          *environment;
```

Examples

This function returns true if its first argument is a number greater than zero. It uses a boolean return value.

```
#include "clips.h"

/*
```

```
Use EnvDefineFunction2(environment,"positivep1",'b',positivep1,"positivep1","11n");
*/

int positivep1(
  void *environment)
  {
   DATA_OBJECT temp;

   /*==================================*/
   /* Check for exactly one argument. */
   /*==================================*/

   if (EnvArgCountCheck(environment,"positivep1",EXACTLY,1) == -1)
     { return(FALSE); }

   /*=========================================================*/
   /* Get the first argument using the ArgTypeCheck function. */
   /*=========================================================*/

   if (EnvArgTypeCheck(environment,"positivep1",1,INTEGER_OR_FLOAT,&temp) == 0)
     { return(FALSE); }

   /*=====================================*/
   /* Determine if the value is positive. */
   /*=====================================*/

   if (GetType(temp) == INTEGER)
     { if (DOToLong(temp) <= 0L) return(FALSE); }
   else /* the type must be FLOAT */
     { if (DOToDouble(temp) <= 0.0) return(FALSE); }

   return(TRUE);
  }
```

This function also returns true if its first argument is a number greater than zero. It uses a symbolic return value.

```
#include "clips.h"

/*
Use EnvDefineFunction2(environment,"positivep2",'w',
                       PTIEF positivep2,"positivep2","11n");
*/

void *positivep2(
  void *environment)
  {
   DATA_OBJECT temp;

   /*==================================*/
   /* Check for exactly one argument. */
   /*==================================*/
```

```
  if (EnvArgCountCheck(environment,"positivep1",EXACTLY,1) == -1)
    { return(EnvFalseSymbol(environment)); }

  /*=========================================================*/
  /* Get the first argument using the ArgTypeCheck function. */
  /*=========================================================*/

  if (EnvArgTypeCheck(environment,"positivep1",1,INTEGER_OR_FLOAT,&temp) == 0)
    { return(EnvFalseSymbol(environment)); }

  /*====================================*/
  /* Determine if the value is positive. */
  /*====================================*/

  if (GetType(temp) == INTEGER)
    { if (DOToLong(temp) <= 0L) return(EnvFalseSymbol(environment)); }
  else /* the type must be FLOAT */
    { if (DOToDouble(temp) <= 0.0) return(EnvFalseSymbol(environment)); }

  return(EnvTrueSymbol(environment));
 }
```

3.3.3 Returning Fact and Instance Addresses

A user function may return a fact address or an instance address. The user should use **EnvDefineFunction** to declare their function as returning a fact address ('y') or an instance address type ('x'). The function should then either return a void pointer to a fact address or instance address. Returning NULL will create a reference to the dummy fact or dummy instance, which can be safely referenced by functions that expect a fact address or instance address, but are recognized as invalid by functions such as **fact-existp** and **instance-existp**.

The printed representation of a fact address is

```
<Fact-XXX>
```

where XXX is the fact-index of the fact.

The printed representation of an instance address is

```
<Instance-XXX>
```

where XXX is the name of the instance.

3.3.4 Returning External Addresses

A user function may return an external address. The user should use **EnvDefineFunction** to declare their function as returning an external address type ('a'). An external addresses can be added by calling the function **EnvAddExternalAddress** and using the value returned by this function as the return value of the user-defined function.

```
#include "clips.h"

void *EnvAddExternalAddress(environment,externalAddress,type);

void *environment;
void *externalAddress;
int type;
```

The value used for the type parameter should be the constant C_POINTER_EXTERNAL_ADDRESS.

Within CLIPS, the printed representation of an external address is

```
<Pointer-C-XXXXXXXX>
```

where XXXXXXXX is the external address. Note that it is up to the user to make sure that external addresses remain valid within CLIPS.

Example

This function uses the memory allocation function malloc to dynamically allocated 100 bytes of memory and then returns a pointer to the memory to CLIPS.

```
#include <stdlib.h>
#include "clips.h"

/*
   EnvDefineFunction2(environment,"malloc",'a',PTIEF CLIPSmalloc,
                                  "CLIPSmalloc","00");
*/

void *CLIPSmalloc(
  void *environment)
  {
   void *rv;

   rv = malloc(100);

   return EnvAddExternalAddress(environment,rv,C_POINTER_EXTERNAL_ADDRESS);
  }
```

Once defined, the function can be called within CLIPS:

```
CLIPS> (malloc)
<Pointer-C-0x7fbd915000f0>
CLIPS>
```

3.3.5 Returning Unknown Data Types

A user-defined function also may return values of an unknown type. The user must declare the function as returning type unknown; i.e., place a 'u' for data type in the call to **EnvDefineFunction**. The user function will be passed a pointer to a structure of type DATA_OBJECT (DATA_OBJECT_PTR) which should be modified to contain the return value. The user should set both the type and the value of the DATA_OBJECT. Note that the value of a DATA_OBJECT cannot be directly set to a double, long, or external address value (the functions **EnvAddLong**, **EnvAddDouble**, and **EnvAddExternalAddress** should be used in a manner similar to **EnvAddSymbol**). The actual return value of the user function is ignored.

```
#include "clips.h"

int            SetType(argument,type);
int            SetpType(&argument,type);

void          *SetValue(argument,value);
void          *SetpValue(&argument,value);

void          *EnvAddLong(environment,longValue);
void          *EnvAddDouble(environment,doubleValue);

void          *GetValue(argument);
void          *GetpValue(&argument);

const char    *ValueToString(value);
double         ValueToDouble(value);
long long      ValueToLong(value);
void          *ValueToExternalAddress(value);

void          *environment;
long long      longValue;
double         doubleValue;
void          *value;
int            type;
DATA_OBJECT    argument;
```

Macros **SetType** and **SetpType** can be used to set the type of a DATA_OBJECT or DATA_OBJECT_PTR respectively. The type parameter should be one of the following CLIPS defined constants (note that these are not strings): SYMBOL, STRING, INTEGER, FLOAT, EXTERNAL_ADDRESS, FACT_ADDRESS, INSTANCE_NAME, or INSTANCE_ADDRESS. Macros **SetValue** (for DATA_OBJECTs) and **SetpValue** (for DATA_OBJECT_PTRs) can be used to set the value of a DATA_OBJECT. The functions

EnvAddSymbol (for symbols, strings and instance names), **EnvAddLong** (for integers), and **EnvAddDouble** (for floats), and **EnvAddExternalAddress** (for external addresses) can be used to produce values that can be used with these macros (instance addresses can be used directly). Macros **GetValue** (for DATA_OBJECTs) and **GetpValue** (for DATA_OBJECT_PTRs) can be used to retrieve the value of a DATA_OBJECT. Note that the value for a fact address or an instance address can be retrieved directly using one of these macros. For other data types, the macros **ValueToString** (for symbols, strings, and instance names), **ValueToLong** (for integers), and **ValueToDouble** (for floats), and **ValueToExternalAddress** (for external addresses) can be used to convert the retrieved value from a DATA_OBJECT to a C object of type char *, long long , double, or void * respectively.

Example

This function "cubes" its argument returning either an integer or float depending upon the type of the original argument. It returns the symbol FALSE upon an error.

```
#include "clips.h"

/*
Use EnvDefineFunction2(environment,"cube",'u',PTIEF cube,"cube","11n");
*/

void cube(
  void *environment,
  DATA_OBJECT_PTR returnValuePtr)
  {
   void *value;
   long long longValue;
   double doubleValue;

   /*==================================*/
   /* Check for exactly one argument. */
   /*==================================*/

   if (EnvArgCountCheck(environment,"cube",EXACTLY,1) == -1)
     {
      SetpType(returnValuePtr,SYMBOL);
      SetpValue(returnValuePtr,EnvFalseSymbol(environment));
      return;
     }

   /*=========================================================*/
   /* Get the first argument using the ArgTypeCheck function. */
   /*=========================================================*/

   if (! EnvArgTypeCheck(environment,"cube",1,INTEGER_OR_FLOAT,returnValuePtr))
     {
      SetpType(returnValuePtr,SYMBOL);
      SetpValue(returnValuePtr,EnvFalseSymbol(environment));
      return;
```

```
  }

/*=========================================================*/
/* Cube the argument. Note that the return value DATA_OBJECT */
/* is used to retrieve the function's argument and return    */
/* the function's return value.                              */
/*=========================================================*/

if (GetpType(returnValuePtr) == INTEGER)
  {
   value = GetpValue(returnValuePtr);
   longValue = ValueToLong(value);
   value = EnvAddLong(environment,longValue * longValue * longValue);
  }
else /* the type must be FLOAT */
  {
   value = GetpValue(returnValuePtr);
   doubleValue = ValueToDouble(value);
   value = EnvAddDouble(environment,doubleValue * doubleValue * doubleValue);
  }

/*=====================================================*/
/* Set the value of the return DATA_OBJECT. The return */
/* type does not have to be changed since it will be   */
/* the same as the 1st argument to the function.       */
/*=====================================================*/

SetpValue(returnValuePtr,value);
return;
}
```

3.3.6 Returning Multifield Values

Multifield values can also be returned from an external function. When defining such an external function, the data type should be set to 'm' in the call to **DefineFunction**. Note that a multifield value can also be returned from a 'u' function, whereas only a multifield value should be returned from an 'm' function. As with returning unknown data types, the user function will be passed a pointer of type DATA_OBJECT_PTR which can be modified to set up a multifield value. The following macros and functions are useful for this purpose:

```
void          *EnvCreateMultifield(environment,size);
int            SetMFType(multifieldPtr,fieldPosition,type);
void          *SetMFValue(multifieldPtr,fieldPosition,value);
long           SetDOBegin(returnValue,fieldPosition);
long           SetpDOBegin(&returnValue,fieldPosition);
long           SetDOEnd(returnValue,fieldPosition);
long           SetpDOEnd(&returnValue,fieldPosition);
void           EnvSetMultifieldErrorValue(environment,&returnValue);

void          *environment;
DATA_OBJECT    returnValue;
```

```
   long           size;
   long           fieldPosition,
   int            type;
   void          *multifieldPtr;
   void          *value;
```

If a new multifield is to be created from an existing multifield, then the type and value of the existing multifield can be copied and the begin and end indices can be modified to obtain the appropriate subfields of the multifield value. If you wish to create a new multifield value that is not part of an existing multifield value, then use the function **EnvCreateMultifield**. Given an integer argument, this function will create a multifield value of the specified size with valid indices ranging from one to the given size (zero is a legitimate parameter to create a multifield value with no fields). The macros **SetMFType** and **SetMFValue** can be used to set the types and values of the fields of the newly created multifield value. Both macros accept as their first argument the value returned by **EnvCreateMultifield**. The second argument should be an integer representing the position of the multifield value to be set. The third argument is the same as the arguments used for **SetType** and **SetValue** macros.

Do *not* set the value or type of any field within a multifield value that has been returned to you by CLIPS. Use these macros only on multifield values created using the **EnvCreateMultifield** function.

The macros **SetDOBegin**, **SetpDOBegin**, **SetDOEnd**, **SetpDOEnd** can be used to assign values to the begin and end indices of a DATA_OBJECT or DATA_OBJECT_PTR containing a multifield value. These macros are useful for creating "new" multifield values by manipulating the indices of a currently existing multifield value. For example, a function that returns the first field of a multifield value could do so by setting the end index equal to the begin index (if the length of the multifield value was greater than zero).

The function **EnvSetMultifieldErrorValue** can be used to create a multifield value of length zero (which is useful to return as an error value). Its only parameter is a DATA_OBJECT_PTR which is appropriately modified to create a zero length multifield value.

Examples

The following example creates a multifield value with two fields, a word and a number:

```
#include "clips.h"

/*
Use EnvDefineFunction2(environment,"sample4",'m',PTIEF sample4,"sample4","00");
*/

void sample4(
  void *environment,
  DATA_OBJECT_PTR returnValuePtr)
```

```
{
 void *multifieldPtr;

 /*====================================*/
 /* Check for exactly zero arguments. */
 /*====================================*/

 if (EnvArgCountCheck(environment,"sample4",EXACTLY,0) == -1)
   {
    EnvSetMultifieldErrorValue(environment,returnValuePtr);
    return;
   }

 /*=======================================*/
 /* Create a multi-field value of length 2 */
 /*=======================================*/

 multifieldPtr = EnvCreateMultifield(environment,2);

 /*=============================================*/
 /* The first field in the multi-field value   */
 /* will be a SYMBOL. Its value will be        */
 /* "altitude".                                */
 /*=============================================*/

 SetMFType(multifieldPtr,1,SYMBOL);
 SetMFValue(multifieldPtr,1,EnvAddSymbol(environment,"altitude"));

 /*=============================================*/
 /* The second field in the multi-field value */
 /* will be a FLOAT. Its value will be 900.   */
 /*=============================================*/

 SetMFType(multifieldPtr,2,FLOAT);
 SetMFValue(multifieldPtr,2,EnvAddDouble(environment,900.0));

 /*=========================================================*/
 /* Assign the type and value to the return DATA_OBJECT. */
 /*=========================================================*/

 SetpType(returnValuePtr,MULTIFIELD);
 SetpValue(returnValuePtr,multifieldPtr);

 /*================================================*/
 /* The length of our multi-field value will be 2.  */
 /* Since we will create our own multi-field value  */
 /* the begin and end indexes to our function will  */
 /* be 1 and the length of the multi-field value    */
 /* respectively. If we are examining a multi-field */
 /* value, or using an existing multi-field value   */
 /* to create a new multi-field value, then the     */
 /* begin and end indexes may not correspond to 1   */
 /* and the length of the multi-field value.        */
```

```
   /*================================================*/

   SetpDOBegin(returnValuePtr,1);
   SetpDOEnd(returnValuePtr,2);

   return;
  }
```

The following example returns all but the first field of a multifield value:

```
#include "clips.h"

/*
Use EnvDefineFunction2(environment,"rest",'m',PTIEF rest,"rest","11m");
*/

void rest(
  void *environment,
  DATA_OBJECT_PTR returnValuePtr)
  {
   /*==================================*/
   /* Check for exactly one argument. */
   /*==================================*/

   if (EnvArgCountCheck(environment,"rest",EXACTLY,1) == -1)
     {
      EnvSetMultifieldErrorValue(environment,returnValuePtr);
      return;
     }

   /*=========================*/
   /* Check for a MULTIFIELD. */
   /*=========================*/

   if (EnvArgTypeCheck(environment,"rest",1,MULTIFIELD,returnValuePtr) == 0)
     {
      EnvSetMultifieldErrorValue(environment,returnValuePtr);
      return;
     }

   /*================================================*/
   /* Don't bother with a zero length multifield value. */
   /*================================================*/

   if (GetpDOBegin(returnValuePtr) > GetpDOEnd(returnValuePtr))
     { return; }

   /*==================================*/
   /* Increment the begin index by one. */
   /*==================================*/

   SetpDOBegin(returnValuePtr,GetpDOBegin(returnValuePtr) + 1);
  }
```

3.4 User-Defined Function Example

This section lists the steps needed to define and implement a user-defined function. The example given is somewhat trivial, but it demonstrates the point. The user function merely triples a number and returns the new value.

1) Copy all of the CLIPS source code file to the user directory.

2) Define the user function in a new file.

```
#include "clips.h"

double TripleNumber(
  void *environment)
  {
   return(3.0 * EnvRtnDouble(environment,1));
  }
```

The preceding function does the job just fine. The following function, however, accomplishes the same purpose while providing error handling on arguments and allowing either an integer or double return value.

```
#include "clips.h"

void TripleNumber(
  void *environment,
  DATA_OBJECT_PTR returnValuePtr)
  {
   void *value;
   long long longValue;
   double doubleValue;

   /*=============================================*/
   /* If illegal arguments are passed, return zero. */
   /*=============================================*/

   if (EnvArgCountCheck(environment,"triple",EXACTLY,1) == -1)
     {
      SetpType(returnValuePtr,INTEGER);
      SetpValue(returnValuePtr,EnvAddLong(environment,0LL));
      return;
     }

   if (! EnvArgTypeCheck(environment,"triple",1,INTEGER_OR_FLOAT,returnValuePtr))
     {
      SetpType(returnValuePtr,INTEGER);
      SetpValue(returnValuePtr,EnvAddLong(environment,0LL));
      return;
     }

   /*====================*/
```

```
   /* Triple the number. */
   /*====================*/

   if (GetpType(returnValuePtr) == INTEGER)
     {
      value = GetpValue(returnValuePtr);
      longValue = 3 * ValueToLong(value);
      SetpValue(returnValuePtr,EnvAddLong(environment,longValue));
     }
   else /* the type must be FLOAT */
     {
      value = GetpValue(returnValuePtr);
      doubleValue = 3.0 * ValueToDouble(value);
      SetpValue(returnValuePtr,EnvAddDouble(environment,doubleValue));
     }

   return;
  }
```

3) Define the constructs which use the new function in a new file (or in an existing constructs file). For example:

```
(deffacts init-data
   (data 34)
   (data 13.2))

(defrule get-data
   (data ?num)
   =>
   (printout t "Tripling " ?num crlf)
   (assert (new-value (triple ?num))))

(defrule get-new-value
   (new-value ?num)
   =>
   (printout t "Now equal to " ?num crlf))
```

4) Modify the CLIPS **userfunctions.c** file to include the new EnvUserFunctions definition.

```
void EnvUserFunctions(
  void *environment)
  {
   /* The following code is used with the second example */
   /* of the TripleFunction listed in step 2.            */

   extern void TripleNumber(void *,DATA_OBJECT_PTR);

   EnvDefineFunction2(environment,"triple",'u',PTIEF TripleNumber,
                      "TripleNumber","11n");

   /* Alternately, if the TripleFunction with a double return */
   /* value from step 2 was used, the following declaration   */
```

```
   /* and DefineFunction2 call should be used in place of the */
   /* one above.                                              */

   /*
   extern double TripleNumber(void *);

   EnvDefineFunction2(environment,"triple",'d',PTIEF TripleNumber,
                      "TripleNumber","11n");
   */
}
```

5) Compile the CLIPS files along with any files which contain user-defined functions.

6) Link all object code files.

7) Execute new CLIPS executable. Load the constructs file and test the new function.

Section 4:
Embedding CLIPS

CLIPS was designed to be embedded within other programs. When CLIPS is used as an embedded application, the user must provide a main program. Calls to CLIPS are made like any other subroutine. To embed CLIPS, add the following include statements to the user's main program file:

```
#include "clips.h"
```

Most of the embedded API function calls require an environment pointer argument. Each environment represents a single instance of the CLIPS engine which can load and run a program. A program must create at least one environment in order to make embedded API calls. In many cases, the program's main function will create a single environment to be used as the argument for all embedded API calls. In other cases, such as creating shared libraries or DLLs, new instances of environments will be created as they are needed. New environments can be created by calling the function **CreateEnvironment** (see section 9).

To create an embedded program, compile and link all of the user's code with all CLIPS files *except* **main.c**. If a library is being created, it may be necessary to use different link options or compile and link "wrapper" source code with the CLIPS source files. Otherwise, the embedded program must provide a replacement main function for the one normally provided by CLIPS.

When running CLIPS as an embedded program, many of the capabilities available in the interactive interface (in addition to others) are available through function calls. The functions are documented in the following sections. Prototypes for these functions can be included by using the **clips.h** header file.

4.1 Environment Functions

The following function calls control the CLIPS environment:

4.1.1 EnvAddClearFunction

```
int EnvAddClearFunction(environment,clearItemName,clearFunction,priority);

void        *environment;
const char  *clearItemName;
void       (*clearFunction)(void *);
int          priority;
```

```
void  clearFunction(environment);

void *environment;
```

Purpose: Adds a user defined function to the list of functions which are called when the CLIPS **clear** command is executed.

Arguments:
1) A generic pointer to an environment.
2) The name of the new clear item.
3) A pointer to the function which is to be called whenever a **clear** command is executed.
4) The priority of the clear item which determines the order in which clear items are called (higher priority items are called first). The values -2000 to 2000 are reserved for CLIPS system defined clear items and should not be used for user defined clear items.

Returns: Returns a zero value if the clear item could not be added, otherwise a non-zero value is returned.

4.1.2 EnvAddPeriodicFunction

```
int EnvAddPeriodicFunction(environment,periodicItemName,
                           periodicFunction,priority);

void        *environment;
const char  *periodicItemName;
void       (*periodicFunction)(void *);
int          priority;

void periodicFunction(environment);

void *environment;
```

Purpose: Adds a user defined function to the list of functions which are called periodically while CLIPS is executing. This ability was primarily included to allow interfaces to process events and update displays during CLIPS execution. Care should be taken not to use any operations in a periodic function which would affect CLIPS data structures constructively or destructively, i.e. CLIPS internals may be examined but not modified during a periodic function.

Arguments:
1) A generic pointer to an environment.
2) The name of the new periodic item.

3) A pointer to a function which is to be called periodically while CLIPS is executing.
4) The priority of the periodic item which determines the order in which periodic items are called (higher priority items are called first). The values -2000 to 2000 are reserved for CLIPS system defined periodic items and should not be used for user defined periodic items.

Returns: Returns a zero value if the periodic item could not be added, otherwise a non-zero value is returned.

4.1.3 EnvAddResetFunction

```
int EnvAddResetFunction(environment,resetItemName,resetFunction,priority);

void        *environment;
const char  *resetItemName;
void       (*resetFunction)(void *);
int          priority;

void resetFunction(environment);

void *environment;
```

Purpose: Adds a user defined function to the list of functions which are called when the CLIPS **reset** command is executed.

Arguments:
1) A generic pointer to an environment.
2) The name of the new reset item.
3) A pointer to the function which is to be called whenever a **reset** command is executed.
4) The priority of the reset item which determines the order in which reset items are called (higher priority items are called first). The values -2000 to 2000 are reserved for CLIPS system defined reset items and should not be used for user defined reset items.

Returns: Returns a zero value if the reset item could not be added, otherwise a non-zero value is returned.

4.1.4 EnvBatchStar

```
int EnvBatchStar(environment,fileName);

void       *environment;
```

```
const char *fileName;
```

Purpose: Evaluates the series of commands stored in the specified file without replacing standard input (the C equivalent of the CLIPS **batch*** command).

Arguments:
1) A generic pointer to an environment.
2) A string representing the name of the file.

Returns: Returns an integer; Zero if the file couldn't be opened or 1 if the file was opened.

Other: The **BatchStar** function is not available for use in run-time programs.

4.1.5 EnvBload

```
int EnvBload(environment,fileName);

void       *environment;
const char *fileName;
```

Purpose: Loads a binary image of constructs into the CLIPS data base (the C equivalent of the CLIPS **bload** command).

Arguments:
1) A generic pointer to an environment.
2) A string representing the name of the file.

Returns: Returns an integer; if zero, an error occurred. A positive one is returned upon success.

4.1.6 EnvBsave

```
int EnvBsave(environment,fileName);

void       *environment;
const char *fileName;
```

Purpose: Saves a binary image of constructs from the CLIPS data base (the C equivalent of the CLIPS **bsave** command).

Arguments:
1) A generic pointer to an environment.
2) A string representing the name of the file.

Returns: Returns an integer; if zero, an error occurred. A positive one is returned upon success.

4.1.7 EnvBuild

```
int EnvBuild(environment,constructString);

void       *environment;
const char *constructString;
```

Purpose: Allows a construct to be defined (the C equivalent of the CLIPS **build** command).

Arguments: 1) A generic pointer to an environment.
2) A string containing the construct to be added.

Returns: Returns an integer. 1 if the construct was successfully parsed, otherwise 0.

Other: The **Build** function is not available for use in run-time programs (since individual constructs can't be added or deleted).

4.1.8 EnvClear

```
void EnvClear(environment);

void *environment;
```

Purpose: Clears the CLIPS environment (the C equivalent of the CLIPS **clear** command).

Arguments: A generic pointer to an environment.

Returns: No meaningful return value.

Other: This function can trigger garbage collection.

4.1.9 EnvEval

```
int EnvEval(environment,expressionString,&outputValue);

void        *environment;
const char  *expressionString;
```

```
DATA_OBJECT  outputValue;
```

Purpose: Allows an expression to be evaluated (the C equivalent of the CLIPS **eval** command).

Arguments:
1) A generic pointer to an environment.
2) A string containing the expression to be evaluated.
3) A pointer to a DATA_OBJECT in which the function results are stored. See sections 3.2.3 and 3.2.4 for information on examining the value stored in a DATA_OBJECT.

Returns: Returns an integer. 1 if the expression was successfully evaluated, otherwise 0.

Other: The **Eval** function is not available for use in run-time programs.

4.1.10 EnvFunctionCall

```
int EnvFunctionCall(environment,functionName,arguments,&outputValue);

void        *environment;
const char  *functionName;
const char  *arguments;
DATA_OBJECT  outputValue;
```

Purpose: Allows CLIPS system functions, deffunctions and generic functions to be called from C.

Arguments:
1) A generic pointer to an environment.
2) The name of the system function, deffunction or generic function to be called.
3) A string containing any *constant* arguments separated by blanks (this argument can be NULL).
4) A pointer to a DATA_OBJECT in which the function results are stored. See sections 3.2.3 and 3.2.4 for information on examining the value stored in a DATA_OBJECT.

Returns: An integer; TRUE (1) if an error occurred while evaluating the function, otherwise FALSE (0).

Other: This function can trigger garbage collection.

Example

```
DATA_OBJECT outputValue;
```

```
void *environment;

EnvFunctionCall(environment,"+","1 2",&outputValue);
```

4.1.11 EnvGetAutoFloatDividend

```
int EnvGetAutoFloatDividend(environment);

void *environment;
```

Purpose: Returns the current value of the auto-float dividend behavior (the C equivalent of the CLIPS **get-auto-float-dividend** command).

Arguments: A generic pointer to an environment.

Returns: An integer; FALSE (0) if the behavior is disabled and TRUE (1) if the behavior is enabled.

4.1.12 EnvGetDynamicConstraintChecking

```
int EnvGetDynamicConstraintChecking(environment);

void *environment;
```

Purpose: Returns the current value of the dynamic constraint checking behavior (the C equivalent of the CLIPS **get-dynamic-constraint-checking** command).

Arguments: A generic pointer to an environment.

Returns: An integer; FALSE (0) if the behavior is disabled and TRUE (1) if the behavior is enabled.

4.1.13 EnvGetSequenceOperatorRecognition

```
int EnvGetSequenceOperatorRecognition(environment);

void *environment;
```

Purpose: Returns the current value of the sequence operator recognition behavior (the C equivalent of the CLIPS **get-sequence-operator-recognition** command).

Arguments: A generic pointer to an environment.

Returns: An integer; FALSE (0) if the behavior is disabled and TRUE (1) if the behavior is enabled.

4.1.14 EnvGetStaticConstraintChecking

```
int EnvGetStaticConstraintChecking(environment);

void *environment;
```

Purpose: Returns the current value of the static constraint checking behavior (the C equivalent of the CLIPS **get-static-constraint-checking** command).

Arguments: A generic pointer to an environment.

Returns: An integer; FALSE (0) if the behavior is disabled and TRUE (1) if the behavior is enabled.

4.1.15 InitializeEnvironment

```
void InitializeEnvironment();
```

Purpose: Initializes the CLIPS system. Must be called prior to any other CLIPS function call. NOTE: This function should be called only once.

Arguments: None.

Returns: No meaningful return value.

Other: Use of this function is deprecated. Embedded programs should use CreateEnvironment to create environments for use with embedded API calls.

4.1.16 EnvLoad

```
int EnvLoad(environment,fileName);

void       *environment;
const char *fileName;
```

Purpose: Loads a set of constructs into the CLIPS data base (the C equivalent of the CLIPS **load** command).

Arguments: 1) A generic pointer to an environment.
2) A string representing the name of the file.

Returns: Returns an integer; Zero if the file couldn't be opened, -1 if the file was opened but an error occurred while loading, and 1 if the file was opened an no errors occurred while loading. If syntactic errors are in the constructs, **Load** still will attempt to read the entire file and error notices will be sent to **werror**.

Other: The **load** function is not available for use in run-time programs (since individual constructs can't be added or deleted). To execute different sets of constructs, the switching feature must be used in a run-time program (see section 5 for more details).

4.1.17 EnvRemoveClearFunction

```
int EnvRemoveClearFunction(environment,clearItemName);

void       *environment;
const char *clearItemName;
```

Purpose: Removes a named function from the list of functions to be called during a **clear** command.

Arguments: 1) A generic pointer to an environment.
2) The name associated with the user-defined clear function. This is the same name that was used when the clear function was added with the function **AddClearFunction**.

Returns: Returns the integer value 1 if the named function was found and removed, otherwise 0 is returned.

4.1.18 EnvRemovePeriodicFunction

```
int EnvRemovePeriodicFunction(environment,periodicItemName);

void       *environment;
const char *periodicItemName;
```

Purpose: Removes a named function from the list of functions which are called periodically while CLIPS is executing.

Arguments: 1) A generic pointer to an environment.

2) The name associated with the user-defined periodic function. This is the same name that was used when the periodic function was added with the function **AddPeriodicFunction**.

Returns: Returns the integer value 1 if the named function was found and removed, otherwise 0 is returned.

4.1.19 EnvRemoveResetFunction

```
int EnvRemoveResetFunction(environment,resetItemName);

void       *environment;
const char *resetItemName;
```

Purpose: Removes a named function from the list of functions to be called during a **reset** command.

Arguments:
1) A generic pointer to an environment.
2) The name associated with the user-defined reset function. This is the same name that was used when the reset function was added with the function **AddResetFunction**.

Returns: Returns the integer value 1 if the named function was found and removed, otherwise 0 is returned.

4.1.20 EnvReset

```
void EnvReset(environment);

void *environment;
```

Purpose: Resets the CLIPS environment (the C equivalent of the CLIPS **reset** command).

Arguments: A generic pointer to an environment.

Returns: No meaningful return value.

Other: This function can trigger garbage collection.

4.1.21 EnvSave

```
int   EnvSave(environment,fileName);

void       *environment;
const char *fileName;
```

Purpose: Saves a set of constructs to the specified file (the C equivalent of the CLIPS **save** command).

Arguments:
1) A generic pointer to an environment.
2) A string representing the name of the file.

Returns: Returns an integer; if zero, an error occurred while opening the file. If non-zero no errors were detected while performing the save.

4.1.22 EnvSetAutoFloatDividend

```
int EnvSetAutoFloatDividend(environment,value);

void *environment;
int   value;
```

Purpose: Sets the auto-float dividend behavior (the C equivalent of the CLIPS **set-auto-float-dividend** command). When this behavior is enabled (by default) the dividend of the division function is automatically converted to a floating point number.

Arguments:
1) A generic pointer to an environment.
2) The new value for the behavior: TRUE (1) to enable the behavior and FALSE (0) to disable it.

Returns: Returns the old value for the behavior.

4.1.23 EnvSetDynamicConstraintChecking

```
int EnvSetDynamicConstraintChecking(environment,value);

void *environmenet;
int   value;
```

Purpose: Sets the value of the dynamic constraint checking behavior (the C equivalent of the CLIPS command **set-dynamic-constraint-checking**). When this behavior is disabled (FALSE by default), newly created data objects (such as deftemplate facts and instances)

do not have their slot values checked for constraint violations. When this behavior is enabled (TRUE), the slot values are checked for constraint violations. The return value for this function is the old value for the behavior.

Arguments:

1) A generic pointer to an environment.
2) The new value for the behavior: TRUE (1) to enable the behavior and FALSE (0) to disable it.

Returns: Returns the old value for the behavior.

4.1.24 EnvSetSequenceOperator Recognition

```
int EnvSetSequenceOperatorRecognition(environment,value);

void *environment;
int   value;
```

Purpose: Sets the sequence operator recognition behavior (the C equivalent of the CLIPS **set-sequence-operator-recognition** command). When this behavior is disabled (by default) multifield variables found in function calls are treated as a single argument. When this behaviour is enabled, multifield variables are expanded and passed as separate arguments in the function call.

Arguments:

1) A generic pointer to an environment.
2) The new value for the behavior: TRUE (1) to enable the behavior and FALSE (0) to disable it.

Returns: Returns the old value for the behavior.

4.1.25 EnvSetStaticConstraintChecking

```
int EnvSetStaticConstraintChecking(environment,value);

void *environment;
int   value;
```

Purpose: Sets the value of the static constraint checking behavior (the C equivalent of the CLIPS command **set-static-constraint-checking**). When this behavior is disabled (FALSE), constraint violations are not checked when function calls and constructs are parsed. When this behavior is enabled (TRUE by default), constraint violations

are checked when function calls and constructs are parsed. The return value for this function is the old value for the behavior.

Arguments: 1) A generic pointer to an environment.
2) The new value for the behavior: TRUE (1) to enable the behavior and FALSE (0) to disable it.

Returns: Returns the old value for the behavior.

4.2 Debugging Functions

The following function call controls the CLIPS debugging aids:

4.2.1 EnvDribbleActive

```
int EnvDribbleActive(environment);

void *environment;
```

Purpose: Determines if the storing of dribble information is active.

Arguments: A generic pointer to an environment.

Returns: Zero if dribbling is not active, non-zero otherwise.

4.2.2 EnvDribbleOff

```
int EnvDribbleOff(environment);

void *environment;
```

Purpose: Turns off the storing of dribble information (the C equivalent of the CLIPS **dribble-off** command).

Arguments: A generic pointer to an environment.

Returns: A zero if an error occurred closing the file; otherwise a one.

4.2.3 EnvDribbleOn

```
int EnvDribbleOn(environment,fileName);

void       *environment;
```

```
const char *fileName;
```

Purpose: Allows the dribble function of CLIPS to be turned on (the C equivalent of the CLIPS **dribble-on** command).

Arguments:
1) A generic pointer to an environment.
2) The name of the file in which to store dribble information. Only one dribble file (per environment) may be opened at a time.

Returns: A zero if an error occurred opening the file; otherwise a one.

4.2.4 EnvGetWatchItem

```
int EnvGetWatchItem(environment,item);

void       *environment;
const char *item;
```

Purpose: Returns the current value of a watch item.

Arguments:
1) A generic pointer to an environment.
2) The name of the watch item which should be one of the following strings: facts, rules, activations, focus, compilations, statistics, globals, instances, slots, messages, message-handlers, generic-functions, method, or deffunctions.

Returns: Returns 1 if the watch item is enabled, 0 if the watch item is disabled, and -1 if the watch item does not exist.

4.2.5 EnvUnwatch

```
int EnvUnwatch(environment,item);

void       *environment;
const char *item;
```

Purpose: Allows the tracing facilities of CLIPS to be deactivated (the C equivalent of the CLIPS **unwatch** command).

Arguments:
1) A generic pointer to an environment.
2) The item to be deactivated which should be one of the following strings: facts, rules, activations, focus, compilations, statistics, globals, deffunctions, instances, slots, messages,

message-handlers, generic-functions, methods, or all. If all is selected, all possible watch items will not be traced.

Arguments: The item to be deactivated which should be one of the following strings: facts, rules, activations, focus, compilations, statistics, globals, deffunctions, instances, slots, messages, message-handlers, generic-functions, methods, or all. If all is selected, all possible watch items will not be traced.

Returns: A one if the watch item was successfully set; otherwise a zero.

4.2.6 EnvWatch

```
int EnvWatch(environment,item);

void       *environment;
const char *item;
```

Purpose: Allows the tracing facilities of CLIPS to be activated (the C equivalent of the CLIPS **watch** command).

Arguments:
1) A generic pointer to an environment.
2) The item to be activated which should be one of the following strings: facts, rules, activations, focus, compilations, statistics, globals, deffunctions, instances, slots, messages, message-handlers, generic-functions, methods, or all. If all is selected, all possible watch items will not be traced.

Returns: A one if the watch item was successfully set; otherwise a zero.

4.3 Deftemplate Functions

The following function calls are used for manipulating deftemplates.

4.3.1 EnvDeftemplateModule

```
const char *EnvDeftemplateModule(environment,deftemplatePtr);

void *environment;
void *deftemplatePtr;
```

Purpose: Returns the module in which a deftemplate is defined (the C equivalent of the CLIPS **deftemplate-module** command).

Arguments: 1) A generic pointer to an environment.
2) A generic pointer to a deftemplate.

Returns: A string containing the name of the module in which the deftemplate is defined.

4.3.2 EnvDeftemplateSlotAllowedValues

```
void EnvDeftemplateSlotAllowedValues(environment,deftemplatePtr,
                                     slotName,&outputValue);

void        *environment;
void        *deftemplatePtr;
const char  *slotName;
DATA_OBJECT  outputValue;
```

Purpose: Groups the allowed-values for a slot into a multifield data object. This function is the C equivalent of the CLIPS **deftemplate-slot-allowed-values** function.

Arguments: 1) A generic pointer to an environment.
2) A generic pointer to a deftemplate data structure.
3) Name of the slot.
4) A pointer to a DATA_OBJECT in which the function results are stored. See sections 3.2.3 and 3.2.4 for information on examining the value stored in a DATA_OBJECT.

Returns: No meaningful return value.

4.3.3 EnvDeftemplateSlotCardinality

```
void EnvDeftemplateSlotCardinality(environment,deftemplatePtr,
                                   slotName,&outputValue);

void        *environment;
void        *deftemplatePtr;
const char  *slotName;
DATA_OBJECT  outputValue;
```

Purpose: Groups the cardinality information for a slot into a multifield data object. This function is the C equivalent of the CLIPS **deftemplate-slot-cardinality** function.

Arguments: 1) A generic pointer to an environment.

2) A generic pointer to a deftemplate data structure.
3) Name of the slot.
4) A pointer to a DATA_OBJECT in which the function results are stored. See sections 3.2.3 and 3.2.4 for information on examining the value stored in a DATA_OBJECT.

Returns: No meaningful return value.

4.3.4 EnvDeftemplateSlotDefaultP

```
int EnvDeftemplateSlotDefaultP(environment,deftemplatePtr,slotName);

void       *environment;
void       *deftemplatePtr,
const char *slotName;
```

Purpose: Determines if the specified slot has a default value. This function is the C equivalent of the CLIPS **deftemplate-slot-defaultp** function.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a deftemplate data structure.
3) The name of the slot.

Returns: One of the following defined integer constants:

NO_DEFAULT
STATIC_DEFAULT
DYNAMIC_DEFAULT

4.3.5 EnvDeftemplateSlotDefaultValue

```
void EnvDeftemplateSlotDefaultValue(environment,deftemplatePtr,slotName,&result);

void        *environment;
void        *defclassPtr;
const char  *slotName;
DATA_OBJECT  outputValue;
```

Purpose: Returns the default value in the data object. This function is the C equivalent of the CLIPS **deftemplate-slot-default-value** function.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a deftemplate data structure.
3) Name of the slot.

4) A pointer to a DATA_OBJECT in which the function results are stored. See sections 3.2.3 and 3.2.4 for information on examining the value stored in a DATA_OBJECT.

Returns: No meaningful return value.

4.3.6 EnvDeftemplateSlotExistP

```
int EnvDeftemplateSlotExistP(environment,deftemplatePtr,slotName);

void       *environment;
void       *deftemplatePtr,
const char *slotName;
```

Purpose: Determines if the specified slot exists. This function is the C equivalent of the CLIPS **deftemplate-slot-existp** function.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a deftemplate data structure.
3) The name of the slot.

Returns: An integer: If the slot is defined in the specified deftemplate, then 1 is returned, otherwise 0 is returned.

4.3.7 EnvDeftemplateSlotMultiP

```
int DeftemplateSlotMultiP(environment,deftemplatePtr,slotName);

void       *environment;
void       *deftemplatePtr,
const char *slotName;
```

Purpose: Determines if the specified slot is a multifield slot. This function is the C equivalent of the CLIPS **deftemplate-slot-multip** function.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a deftemplate data structure.
3) The name of the slot.

Returns: An integer: If the slot in the specified deftemplate is a multifield slot, then 1 is returned, otherwise 0 is returned.

4.3.8 EnvDeftemplateSlotNames

```
void EnvDeftemplateSlotNames(environment,deftemplatePtr,&slotNames);

void        *environment;
void        *deftemplatePtr;
DATA_OBJECT  slotNames;
```

Purpose: Retrieves the list of slot names associated with a deftemplate (the C equivalent of the CLIPS **deftemplate-slot-names** function).

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a deftemplate data structure.
3) A pointer to a DATA_OBJECT in which the function results are stored. See sections 3.2.3 and 3.2.4 for information on examining the value stored in a DATA_OBJECT. For implied deftemplates, a multifield value containing the single symbol *implied* is returned.

Returns: No meaningful value.

4.3.9 EnvDeftemplateSlotRange

```
void EnvDeftemplateSlotRange(environment,deftemplatePtr,slotName,&outputValue);

void        *environment;
void        *deftemplatePtr;
const char  *slotName;
DATA_OBJECT  outputValue;
```

Purpose: Groups the numeric range information for a slot into a multifield data object. This function is the C equivalent of the CLIPS **deftemplate-slot-range** function.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a deftemplate data structure.
3) Name of the slot.
4) A pointer to a DATA_OBJECT in which the function results are stored. See sections 3.2.3 and 3.2.4 for information on examining the value stored in a DATA_OBJECT.

Returns: No meaningful return value.

4.3.10 EnvDeftemplateSlotSingleP

```
int EnvDeftemplateSlotSingleP(environment,deftemplatePtr,slotName);

void       *environment;
void       *deftemplatePtr,
const char *slotName;
```

Purpose: Determines if the specified slot is a single-field slot. This function is the C equivalent of the CLIPS **deftemplate-slot-singlep** function.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a deftemplate data structure.
3) The name of the slot.

Returns: An integer: If the slot in the specified deftemplate is a single-field slot, then 1 is returned, otherwise 0 is returned.

4.3.11 EnvDeftemplateSlotTypes

```
void EnvDeftemplateSlotTypes(environment,deftemplatePtr,slotName,&outputValue);

void        *environment;
void        *deftemplatePtr;
const char  *slotName;
DATA_OBJECT  outputValue;
```

Purpose: Groups the names of the primitive data types allowed for a slot into a multifield data object. This function is the C equivalent of the CLIPS **deftemplate-slot-types** function.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a deftemplate data structure.
3) Name of the slot.
4) A pointer to a DATA_OBJECT in which the function results are stored. See sections 3.2.3 and 3.2.4 for information on examining the value stored in a DATA_OBJECT.

Returns: No meaningful return value.

4.3.12 EnvFindDeftemplate

```
void *EnvFindDeftemplate(environment,deftemplateName);

void       *environment;
const char *deftemplateName;
```

Purpose: Returns a generic pointer to a named deftemplate.

Arguments:
1) A generic pointer to an environment.
2) The name of the deftemplate to be found.

Returns: A generic pointer to the named deftemplate if it exists, otherwise NULL.

4.3.13 EnvGetDeftemplateList

```
void EnvGetDeftemplateList(environment,&outputValue,theModule);

void        *environment;
DATA_OBJECT  outputValue;
void        *theModule;
```

Purpose: Returns the list of deftemplates in the specified module as a multifield value in the returnValue DATA_OBJECT (the C equivalent of the CLIPS **get-deftemplate-list** function).

Arguments:
1) A generic pointer to an environment.
2) A pointer to a DATA_OBJECT in which the function results are stored. See sections 3.2.3 and 3.2.4 for information on examining the value stored in a DATA_OBJECT.
3) A generic pointer to the module from which the list will be extracted. A NULL pointer indicates that the list is to be extracted from al l modules.

Returns: No meaningful return value.

4.3.14 EnvGetDeftemplateName

```
const char *EnvGetDeftemplateName(environment,deftemplatePtr);

void *environment;
void *deftemplatePtr;
```

Purpose: Returns the name of a deftemplate.

Arguments: 1) A generic pointer to an environment.
2) A generic pointer to a deftemplate data structure.

Returns: A string containing the name of the deftemplate.

4.3.15 EnvGetDeftemplatePPForm

```
const char *GetDeftemplatePPForm(environment,deftemplatePtr);

void *environment;
void *deftemplatePtr;
```

Purpose: Returns the pretty print representation of a deftemplate.

Arguments: 1) A generic pointer to an environment.
2) A generic pointer to a deftemplate data structure.

Returns: A string containing the pretty print representation of the deftemplate (or the NULL pointer if no pretty print representation exists).

4.3.16 EnvGetDeftemplateWatch

```
unsigned EnvGetDeftemplateWatch(environment,deftemplatePtr);

void *environment;
void *deftemplatePtr;
```

Purpose: Indicates whether or not a particular deftemplate is being watched.

Arguments: 1) A generic pointer to an environment.
2) A generic pointer to a deftemplate data structure.

Returns: An integer; one (1) if the deftemplate is being watched, otherwise a zero (0).

4.3.17 EnvGetNextDeftemplate

```
void *EnvGetNextDeftemplate(environment,deftemplatePtr);

void *environment;
void *deftemplatePtr;
```

Purpose: Provides access to the list of deftemplates.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a deftemplate data structure (or NULL to get the first deftemplate).

Returns: A generic pointer to the first deftemplate in the list of deftemplates if *deftemplatePtr* is NULL, otherwise a generic pointer to the deftemplate immediately following *deftemplatePtr* in the list of deftemplates. If *deftemplatePtr* is the last deftemplate in the list of deftemplates, then NULL is returned.

4.3.18 EnvIsDeftemplateDeletable

```
int EnvIsDeftemplateDeletable(environment,deftemplatePtr);

void *environment;
void *deftemplatePtr;
```

Purpose: Indicates whether or not a particular deftemplate can be deleted.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a deftemplate data structure.

Returns: An integer; zero (0) if the deftemplate cannot be deleted, otherwise a one (1).

4.3.19 EnvListDeftemplates

```
void EnvListDeftemplates(environment,logicalName,theModule);

void       *environment;
const char *logicalName;
void       *theModule;
```

Purpose: Prints the list of deftemplates (the C equivalent of the CLIPS **list-deftemplates** command).

Arguments:
1) A generic pointer to an environment.
2) The logical name to which the listing output is sent.
3) A generic pointer to the module containing the deftemplates to be listed. A NULL pointer indicates that deftemplate in all modules should be listed.

Returns: No meaningful return value.

4.3.20 EnvSetDeftemplateWatch

```
void EnvSetDeftemplateWatch(environment,newState,deftemplatePtr);

void     *environment;
unsigned  newState;
void     *deftemplatePtr;
```

Purpose: Sets the facts watch item for a specific deftemplate.

Arguments:
1) A generic pointer to an environment.
2) The new facts watch state
3) A generic pointer to a deftemplate data structure.

Returns: No meaningful return value.

4.3.21 EnvUndeftemplate

```
int EnvUndeftemplate(environment,deftemplatePtr);

void *environment;
void *deftemplatePtr;
```

Purpose: Removes a deftemplate from CLIPS (the C equivalent of the CLIPS **undeftemplate** command).

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a deftemplate data structure. If the NULL pointer is used, then all deftemplates will be deleted.

Returns: An integer; zero (0) if the deftemplate could not be deleted, otherwise a one (1).

Other: This function can trigger garbage collection.

4.4 Fact Functions

The following function calls manipulate and display information about facts.

4.4.1 EnvAssert

```
void *EnvAssert(environment,factPtr);

void *environment;
void *factPtr;
```

Purpose: Adds a fact created using the function **CreateFact** to the fact-list. If the fact was asserted successfully, **Assert** will return a pointer to the fact. Otherwise, it will return NULL (i.e., the fact was already in the fact-list).

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to the fact created using **CreateFact**. The values of the fact should be initialized before calling **Assert**.

Returns: A generic pointer to a fact structure. If the fact was asserted successfully, **Assert** will return a generic pointer to the fact. Otherwise, it will return NULL (i.e., the fact was already in the fact-list).

Other: This function can trigger garbage collection.

Warning: If the return value from **Assert** is stored as part of a persistent data structure or in a static data area, then the function **IncrementFactCount** should be called to insure that the fact cannot be disposed while external references to the fact still exist.

4.4.2 EnvAssertString

```
void *EnvAssertString(environment,string);

void       *environment;
const char *string;
```

Purpose: Asserts a fact into the CLIPS fact-list (the C equivalent of the CLIPS **assert-string** command).

Arguments:
1) A generic pointer to an environment.
2) A string representing a single fact (expressed in either ordered or deftemplate format).

Returns: A generic pointer to a fact structure.

Other: This function can trigger garbage collection.

Warning: If the return value from **AssertString** is stored as part of a persistent data structure or in a static data area, then the function

IncrementFactCount should be called to insure that the fact cannot be disposed while external references to the fact still exist.

Examples

If the following deftemplate has been processed by CLIPS,

```
(deftemplate example
   (multislot v)
   (slot w (default 9))
   (slot x)
   (slot y)
   (multislot z))
```

then the following fact

```
(example (x 3) (y red) (z 1.5 b))
```

can be added to the fact-list using the function shown below.

```
void AddExampleFact1(
  void *environment)
  {
   EnvAssertString(environment,"(example (x 3) (y red) (z 1.5 b))");
  }
```

To construct a string based on variable data, use the C library function **sprintf** as shown following.

```
void VariableFactAssert(
  void *environment,
  int number,
  const char *status)
  {
   char tempBuffer[50];

   sprintf(tempBuffer,"(example (x %d) (y %s))",number,status);
   EnvAssertString(environment,tempBuffer);
  }
```

4.4.3 EnvAssignFactSlotDefaults

```
int EnvAssignFactSlotDefaults(environment,theFact);

void *environment;
void *theFact;
```

Purpose: Assigns default values to a fact.

Arguments: 1) A generic pointer to an environment.
2) A generic pointer to a fact data structure.

Returns: Boolean value. TRUE if the default values were successfully set, otherwise FALSE.

4.4.4 EnvCreateFact

```
void *EnvCreateFact(environment,deftemplatePtr);

void *environment;
void *deftemplatePtr;
```

Purpose: Function **CreateFact** returns a pointer to a fact structure with factSize fields. Once this fact structure is obtained, the fields of the fact can be given values by using **PutFactSlot** and **AssignFactSlotDefaults**. Function **Assert** should be called when the fact is ready to be asserted.

Arguments: 1) A generic pointer to an environment.
2) A generic pointer to a deftemplate data structure (which indicates the type of fact being created).

Returns: A generic pointer to a fact data structure.

Other: Use the **CreateFact** function to create a new fact and then the **PutFactSlot** function to set one or more slot values. The **AssignFactSlotDefaults** function is then used to assign default values for slots not set with the PutFactSlot function. Finally, the **Assert** function is called with the new fact.

Since **CreateFact** requires a generic deftemplate pointer, it is not possible to use it to create ordered facts unless the associated implied deftemplate has already been created. In cases where the implied deftemplate has not been created, the function **AssertString** can be used to create ordered facts.

This function allows individual fields of a fact to be assigned under programmer control. This is useful, for example, if a fact asserted from an external function needs to contain an external address or an instance address (since the function **AssertString** does not permit these data types). For most situations in which a fact needs to be

asserted, however, the **AssertString** function should be preferred (it is slighter slower than using the **CreateFact** and **Assert** functions, but it is much easier to use and less prone to being used incorrectly).

Example

This example demonstrates how to create this fact:

```
(example (x 3) (y red) (z 1.5 b))
```

Using this deftemplate:

```
(deftemplate example
   (multislot v)
   (slot w (default 9))
   (slot x)
   (slot y)
   (multislot z))
```

Replace main.c with the following code:

```
#include "clips.h"

void AddExampleFact2(void *);

int main()
  {
   void *theEnv;
   DATA_OBJECT rv;
   char *cs;

   theEnv = CreateEnvironment();

   cs =  "(deftemplate example"
         "   (multislot v)"
         "   (slot w (default 9))"
         "   (slot x)"
         "   (slot y)"
         "   (multislot z))";

   EnvBuild(theEnv,cs);

   AddExampleFact2(theEnv);

   EnvEval(theEnv,"(facts)",&rv);
  }

void AddExampleFact2(
```

```
void *environment)
{
 void *newFact;
 void *templatePtr;
 void *theMultifield;
 DATA_OBJECT theValue;

 /*=============================================================*/
 /* Disable garbage collection. It's only necessary to disable */
 /* garbage collection when calls are made into CLIPS from an  */
 /* embedding program. It's not necessary to do this when the  */
 /* the calls to user code are made by CLIPS (such as for      */
 /* user-defined functions) or in the case of this example,    */
 /* there are no calls to functions which can trigger garbage  */
 /* collection (such as Send or FunctionCall).                 */
 /*=============================================================*/

 //IncrementGCLocks(environment);

 /*==================*/
 /* Create the fact. */
 /*==================*/

 templatePtr = EnvFindDeftemplate(environment,"example");
 newFact = EnvCreateFact(environment,templatePtr);
 if (newFact == NULL) return;

 /*==============================*/
 /* Set the value of the x slot. */
 /*==============================*/

 theValue.type = INTEGER;
 theValue.value = EnvAddLong(environment,3);
 EnvPutFactSlot(environment,newFact,"x",&theValue);

 /*==============================*/
 /* Set the value of the y slot. */
 /*==============================*/

 theValue.type = SYMBOL;
 theValue.value = EnvAddSymbol(environment,"red");
 EnvPutFactSlot(environment,newFact,"y",&theValue);

 /*==============================*/
 /* Set the value of the z slot. */
 /*==============================*/

 theMultifield = EnvCreateMultifield(environment,2);
 SetMFType(theMultifield,1,FLOAT);
 SetMFValue(theMultifield,1,EnvAddDouble(environment,1.5));
 SetMFType(theMultifield,2,SYMBOL);
 SetMFValue(theMultifield,2,EnvAddSymbol(environment,"b"));
 SetDOBegin(theValue,1);
```

```
   SetDOEnd(theValue,2);

   theValue.type = MULTIFIELD;
   theValue.value = theMultifield;
   EnvPutFactSlot(environment,newFact,"z",&theValue);

   /*=================================*/
   /* Assign default values since all */
   /* slots were not initialized.     */
   /*=================================*/

   EnvAssignFactSlotDefaults(environment,newFact);

   /*=========================================================*/
   /* Enable garbage collection. Each call to IncrementGCLocks */
   /* should have a corresponding call to DecrementGCLocks.    */
   /*=========================================================*/

   //EnvDecrementGCLocks(environment);

   /*==================*/
   /* Assert the fact. */
   /*==================*/

   EnvAssert(environment,newFact);
  }
```

Compiling and running CLIPS will produce this output:

```
f-0     (initial-fact)
f-1     (example (v) (w 9) (x 3) (y red) (z 1.5 b))
For a total of 2 facts.
```

4.4.5 EnvDecrementFactCount

```
   void EnvDecrementFactCount(environment,factPtr);

   void *environment;
   void *factPtr;
```

Purpose: This function should *only* be called to reverse the effects of a previous call to **IncrementFactCount**. As long as an fact's count is greater than zero, the memory allocated to it cannot be released for other use.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a fact.

Returns: No meaningful return value.

4.4.6 EnvFactDeftemplate

```
void *EnvFactDeftemplate(environment,factPtr);

void *environment;
void *factPtr;
```

Purpose: Returns the deftemplate associated with a fact.

Arguments: A generic pointer to a fact data structure.

Returns: Returns a generic pointer to the deftemplate data structure associated with the fact.

4.4.7 EnvFactExistp

```
int EnvFactExistp(environment,factPtr);

void *environment;
void *factPtr;
```

Purpose: Indicates whether a fact is still in the fact-list or has been retracted (the C equivalent of the CLIPS **fact-existp** function).

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a fact data structure.

Returns: An integer; zero (0) if the fact is not in the fact-list, otherwise a one (1).

4.4.8 EnvFactIndex

```
long long EnvFactIndex(environment,factPtr);

void *environment;
void *factPtr;
```

Purpose: Returns the fact index of a fact (the C equivalent of the CLIPS **fact-index** command).

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a fact data structure.

Returns: A long integer (the fact-index of the fact).

4.4.9 EnvFacts

```
void EnvFacts(environment,logicalName,theModule,start,end,max);

void       *environment;
const char *logicalName;
void       *theModule;
long long   start;
long long   end;
long long   max;
```

Purpose: Prints the list of all facts currently in the fact-list (the C equivalent of the CLIPS **facts** command). Output is sent to the logical name **wdisplay**.

Arguments:

1) A generic pointer to an environment.
2) The logical name to which the listing output is sent.
3) A generic pointer to the module containing the facts to be listed (all facts visible to that module). A NULL pointer indicates that all facts in all modules should be listed.
4) The start index of the facts to be listed. Facts with indices less than this value are not listed. A value of -1 indicates that the argument is unspecified and should not restrict the facts printed.
5) The end index of the facts to be listed. Facts with indices greater than this value are not listed. A value of -1 indicates that the argument is unspecified and should not restrict the facts printed.
6) The maximum number of facts to be listed. Facts in excess of this limit are not listed. A value of -1 indicates that the argument is unspecified and should not restrict the facts printed.

Returns: No meaningful return value.

4.4.10 EnvFactSlotNames

```
void EnvFactSlotNames(environment,factPtr,&outputValue);

void        *environment;
void        *factPtr;
DATA_OBJECT  outputValue;
```

Purpose: Retrieves the list of slot names associated with a fact (the C equivalent of the CLIPS **fact-slot-names** function).

Arguments:

1) A generic pointer to an environment.
2) A generic pointer to a fact data structure.

3) A pointer to a DATA_OBJECT in which the function results are stored. See sections 3.2.3 and 3.2.4 for information on examining the value stored in a DATA_OBJECT. For ordered facts, a multifield value containing the single symbol *implied* is returned.

Returns: No meaningful value.

4.4.11 EnvGetFactDuplication

```
int EnvGetFactDuplication(environment);

void *environment;
```

Purpose: Returns the current value of the fact duplication behavior (the C equivalent of the CLIPS **get-fact-duplication** command).

Arguments: A generic pointer to an environment.

Returns: An integer; FALSE (0) if the behavior is disabled and TRUE (1) if the behavior is enabled.

4.4.12 EnvGetFactList

```
void EnvGetFactList(environment,&outputValue,theModule);

void        *environment;
DATA_OBJECT  outputValue;
void        *theModule;
```

Purpose: Returns the list of facts visible to the specified module as a multifield value in the returnValue DATA_OBJECT (the C equivalent of the CLIPS **get-fact-list** function).

Arguments:
1) A generic pointer to an environment.
2) A pointer to a DATA_OBJECT in which the function results are stored. See sections 3.2.3 and 3.2.4 for information on examining the value stored in a DATA_OBJECT.
3) A generic pointer to the module from which the list will be extracted. A NULL pointer indicates that the list is to be extracted from al l modules.

Returns: No meaningful return value.

4.4.13 EnvGetFactListChanged

```
int EnvGetFactListChanged(environment);

void *environment;
```

Purpose: Determines if any changes to the fact list have occurred. If this function returns a non-zero integer, it is the user's responsibility to call SetFactListChanged(0) to reset the internal flag. Otherwise, this function will continue to return non-zero even when no changes have occurred. This function is primarily used to determine when to update a display tracking the fact list.

Arguments: A generic pointer to an environment.

Returns: 0 if no changes to the fact list have occurred, non-zero otherwise.

4.4.14 EnvGetFactPPForm

```
void EnvGetFactPPForm(environment,buffer,bufferLength,factPtr);

void   *environment;
char   *buffer;
size_t  bufferLength;
void   *factPtr;
```

Purpose: Returns the pretty print representation of a fact in the caller's buffer.

Arguments:
1) A generic pointer to an environment.
2) A pointer to the caller's character buffer.
3) The maximum number of characters which could be stored in the caller's buffer (not including space for the terminating null character).
4) A generic pointer to a fact data structure.

Returns: No meaningful return value. The fact pretty print form is stored in the caller's buffer.

4.4.15 EnvGetFactSlot

```
int EnvGetFactSlot(environment,factPtr,slotName,&outputValue);

void        *environment;
void        *factPtr;
const char  *slotName;
```

```
DATA_OBJECT  outputValue;
```

Purpose: Retrieves a slot value from a fact.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a fact data structure.
3) The name of the slot to be retrieved (NULL should be used for the implied multifield slot of an implied deftemplate).
4) A pointer to a DATA_OBJECT in which the function results are stored. See sections 3.2.3 and 3.2.4 for information on examining the value stored in a DATA_OBJECT.

Returns: Boolean value. TRUE if the slot value was successfully retrieved, otherwise FALSE.

4.4.16 EnvGetNextFact

```
void *EnvGetNextFact(environment,factPtr);

void *environment;
void *factPtr;
```

Purpose: Provides access to the fact-list.

Arguments: A generic pointer to a fact data structure (or NULL to get the first fact in the fact-list).

Returns:
1) A generic pointer to an environment.
2) A generic pointer to the first fact in the fact-list if *factPtr* is NULL, otherwise a generic pointer to the fact immediately following *factPtr* in the fact-list. If *factPtr* is the last fact in the fact-list, then NULL is returned.

Other: Once this generic pointer to the fact structure is obtained, the fields of the fact can be examined by using the macros **GetMFType** and **GetMFValue**. The values of a fact obtained using this function should never be changed. See **CreateFact** for details on accessing deftemplate facts.

Warning: Do not call this function with a pointer to a fact that has been retracted. If the return value from **GetNextFact** is stored as part of a persistent data structure or in a static data area, then the function **IncrementFactCount** should be called to insure that the fact cannot be disposed while external references to the fact still exist.

4.4.17 EnvGetNextFactInTemplate

```
void *EnvGetNextFactInTemplate(environment,templatePtr,factPtr);

void *environment;
void *templatePtr;
void *factPtr;
```

Purpose: Provides access to the list of facts for a particular deftemplate.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a deftemplate.
3) A generic pointer to a fact data structure (or NULL to get the first fact from the deftemplate's fact-list).

Returns: A generic pointer to the first fact of the specified deftemplate if *factPtr* is NULL, otherwise a generic pointer to the next fact of the specified deftemplate immediately following *factPtr*. If *factPtr* is the last fact belonging to the deftemplate, then NULL is returned.

Other: Once this generic pointer to the fact structure is obtained, the fields of the fact can be examined by using the macros **GetMFType** and **GetMFValue**. The values of a fact obtained using this function should never be changed. See **CreateFact** for details on accessing deftemplate facts.

Warning: Do not call this function with a pointer to a fact that has been retracted. If the return value from **GetNextFactInTemplate** is stored as part of a persistent data structure or in a static data area, then the function **IncrementFactCount** should be called to insure that the fact cannot be disposed while external references to the fact still exist.

4.4.18 EnvIncrementFactCount

```
void EnvIncrementFactCount(environment,factPtr);

void *environment;
void *factPtr;
```

Purpose: This function should be called for each external copy of pointer to a fact to let CLIPS know that such an outstanding external reference exists. As long as an fact's count is greater than zero, CLIPS will not release its memory because there may be outstanding pointers to the fact. However, the fact can still be *functionally* retracted, i.e. the

fact will *appear* to no longer be in the fact-list. The fact address always can be safely *examined* using the fact access functions as long as the count for the fact is greater than zero. Retracting an already retracted fact will have no effect, however, the function **AddFact** should not be called twice for the same pointer created using **CreateFact**. Note that this function only needs to be called if you are storing pointers to facts that may later be referenced by external code after the fact has been retracted.

Arguments: 1) A generic pointer to an environment.
2) A generic pointer to a fact.

Returns: No meaningful return value.

4.4.19 EnvLoadFacts

```
int EnvLoadFacts(environment,fileName);

void       *environment;
const char *fileName;
```

Purpose: Loads a set of facts into the CLIPS data base (the C equivalent of the CLIPS **load-facts** command).

Arguments: A string representing the name of the file.

Returns: Returns an integer; if zero, an error occurred while opening the file. If non-zero no errors were detected while performing the load.

4.4.20 EnvLoadFactsFromString

```
int   EnvLoadFactsFromString(environment,inputString,maximumPosition);

void       *environment;
const char *inputString;
long        maximumPosition;
```

Purpose: Loads a set of facts into the CLIPS data base using a string as the input source (in a manner similar to the CLIPS **load-facts** command).

Arguments: 1) A generic pointer to an environment.
2) A string containing the fact definitions to be loaded.

3) The maximum number of characters to be read from the string. A value of -1 indicates the entire string.

Returns: Returns an integer; if zero, an error occurred while processing the string.

4.4.21 EnvPPFact

```
void EnvPPFact(environment,factPtr,logicalName,ignoreDefaultFlag);

void       *environment;
void       *factPtr;
const char *logicalName;
int         ignoreDefaultsFlag;
```

Purpose: Displays a single fact (the C equivalent of the CLIPS **ppfact** command).

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a fact.
3) The logical name to which the listing output is sent.
4) The integer 1 to exclude slots from display where the current value is the same as the static default, otherwise the integer 0 to display all slots regardless of their current value.

Returns: No meaningful return value.

4.4.22 EnvPutFactSlot

```
int EnvPutFactSlot(environment,factPtr,slotName,&inputValue);

void        *environment;
void        *factPtr;
const char  *slotName;
DATA_OBJECT  inputValue;
```

Purpose: Sets the slot value of a fact.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a fact data structure.
3) The name of the slot to be set (NULL should be used for the implied multifield slot of an implied deftemplate).
4) A pointer to a DATA_OBJECT that contains the slot's new value. A multifield or implied multifield slot should only be passed a multifield value. A single field slot should only be

passed a single field value. See sections 3.3.5 and 3.3.6 for information on setting the value stored in a DATA_OBJECT.

Returns: Boolean value. TRUE if the slot value was successfully set, otherwise FALSE.

Warning: Do *not* use this function to change the slot value of a fact that has already been asserted. This function should only be used on facts created using **CreateFact**.

4.4.23 EnvRetract

```
int EnvRetract(environment,factPtr);

void *environment;
void *factPtr;
```

Purpose: Retracts a fact from the CLIPS fact-list (the C equivalent of the CLIPS **retract** command).

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a fact structure (usually captured as the return value from a call to **AssertString** or **Assert**). If the NULL pointer is used, then all facts will be retracted.

Returns: An integer; zero (0) if fact already has been retracted, otherwise a one (1).

Other: The caller of **RetractFact** is responsible for insuring that the fact passed as an argument is still valid. The functions **IncrementFactCount** and **DecrementFactCount** can be used to inform CLIPS whether a fact is still in use.

This function can trigger garbage collection.

4.4.24 EnvSaveFacts

```
int EnvSaveFacts(environment,fileName,saveScope);

void       *environment;
const char *fileName;
int         saveScope;
```

Purpose: Saves the facts in the fact-list to the specified file (the C equivalent of the CLIPS **save-facts** command).

Arguments:
1) A generic pointer to an environment.
2) A string representing the name of the file.
3) An integer constant representing the scope for the facts being saved which should be either LOCAL_SAVE or VISIBLE_SAVE.

Returns: Returns an integer; if zero, an error occurred while opening the file. If non-zero no errors were detected while performing the save.

4.4.25 EnvSetFactDuplication

```
int EnvSetFactDuplication(environment,value);

void *environment;
int   value;
```

Purpose: Sets the fact duplication behavior (the C equivalent of the CLIPS **set-fact-duplication** command). When this behavior is disabled (by default), asserting a duplicate of a fact already in the fact-list produces no effect. When enabled, the duplicate fact is asserted with a new fact-index.

Arguments:
1) A generic pointer to an environment.
2) The new value for the behavior: TRUE (1) to enable the behavior and FALSE (0) to disable it.

Returns: Returns the old value for the behavior.

4.4.26 EnvSetFactListChanged

```
void EnvSetFactListChanged(environment,changedFlag);

void *environment;
int   changedFlag;
```

Purpose: Sets the internal boolean flag which indicates when changes to the fact list have occurred. This function is normally used to reset the flag to zero after GetFactListChanged() returns non-zero.

Arguments:
1) A generic pointer to an environment.

2) An integer indicating whether changes in the fact list have occurred (non-zero) or not (0).

Returns: Nothing useful.

4.5 Deffacts Functions

The following function calls are used for manipulating deffacts.

4.5.1 EnvDeffactsModule

```
char *EnvDeffactsModule(environment,theDeffacts);

void *environment;
void *theDeffacts;
```

Purpose: Returns the module in which a deffacts is defined (the C equivalent of the CLIPS **deffacts-module** command).

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a deffacts data structure.

Returns: A string containing the name of the module in which the deffacts is defined.

4.5.2 EnvFindDeffacts

```
void *EnvFindDeffacts(environment,deffactsName);

void       *environment;
const char *deffactsName;
```

Purpose: Returns a generic pointer to a named deffacts.

Arguments:
1) A generic pointer to an environment.
2) The name of the deffacts to be found.

Returns: A generic pointer to the named deffacts if it exists, otherwise NULL.

4.5.3 EnvGetDeffactsList

```
void EnvGetDeffactsList(environment,&outputValue,theModule);

void        *environment;
DATA_OBJECT  outputValue;
void        *theModule;
```

Purpose: Returns the list of deffacts in the specified module as a multifield value in the returnValue DATA_OBJECT (the C equivalent of the CLIPS **get-deffacts-list** function).

Arguments:
1) A generic pointer to an environment.
2) A pointer to a DATA_OBJECT in which the function results are stored. See sections 3.2.3 and 3.2.4 for information on examining the value stored in a DATA_OBJECT.
3) A generic pointer to the module from which the list will be extracted. A NULL pointer indicates that the list is to be extracted from al l modules.

Returns: No meaningful return value.

4.5.4 EnvGetDeffactsName

```
const char *EnvGetDeffactsName(environment,deffactsPtr);

void *environment;
void *deffactsPtr;
```

Purpose: Returns the name of a deffacts.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a deffacts data structure.

Returns: A string containing the name of the deffacts.

4.5.5 EnvGetDeffactsPPForm

```
const char *EnvGetDeffactsPPForm(environment,deffactsPtr);

void *environment;
void *deffactsPtr;
```

Purpose: Returns the pretty print representation of a deffacts.

Arguments: 1) A generic pointer to an environment.
2) A generic pointer to a deffacts data structure.

Returns: A string containing the pretty print representation of the deffacts (or the NULL pointer if no pretty print representation exists).

4.5.6 EnvGetNextDeffacts

```
void *EnvGetNextDeffacts(environment,deffactsPtr);

void *environment;
void *deffactsPtr;
```

Purpose: Provides access to the list of deffacts.

Arguments: 1) A generic pointer to an environment.
2) A generic pointer to a deffacts data structure (or NULL to get the first deffacts).

Returns: A generic pointer to the first deffacts in the list of deffacts if *deffactsPtr* is NULL, otherwise a generic pointer to the deffacts immediately following *deffactsPtr* in the list of deffacts. If *deffactsPtr* is the last deffacts in the list of deffacts, then NULL is returned.

4.5.7 EnvIsDeffactsDeletable

```
int EnvIsDeffactsDeletable(environment,deffactsPtr);

void *environment;
void *deffactsPtr;
```

Purpose: Indicates whether or not a particular deffacts can be deleted.

Arguments: 1) A generic pointer to an environment.
2) A generic pointer to a deffacts data structure.

Returns: An integer; zero (0) if the deffacts cannot be deleted, otherwise a one (1).

4.5.8 EnvListDeffacts

```
void EnvListDeffacts(environment,logicalName,theModule);

void *environment;
char *logicalName;
void *theModule;
```

Purpose: Prints the list of deffacts (the C equivalent of the CLIPS **list-deffacts** command).

Arguments:
1) A generic pointer to an environment.
2) The logical name to which the listing output is sent.
3) A generic pointer to the module containing the deffacts to be listed. A NULL pointer indicates that deffacts in all modules should be listed.

Returns: No meaningful return value.

4.5.9 EnvUndeffacts

```
int EnvUndeffacts(environment,deffactsPtr);

void *environment;
void *deffactsPtr;
```

Purpose: Removes a deffacts construct from CLIPS (the C equivalent of the CLIPS **undeffacts** command).

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a deffacts data structure. If the NULL pointer is used, then all deffacts will be deleted.

Returns: An integer; zero (0) if the deffacts could not be deleted, otherwise a one (1).

Other: This function can trigger garbage collection.

4.6 Defrule Functions

The following function calls are used for manipulating defrules.

4.6.1 EnvDefruleHasBreakpoint

```
int EnvDefruleHasBreakpoint(environment,defrulePtr);

void *environment;
void *defrulePtr;
```

Purpose: Indicates whether or not a particular defrule has a breakpoint set.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a defrule data structure.

Returns: An integer; one (1) if a breakpoint exists for the rule, otherwise a zero (0).

4.6.2 EnvDefruleModule

```
const char *EnvDefruleModule(environment,theDefrule);

void *environment;
void *theDefrule;
```

Purpose: Returns the module in which a defrule is defined (the C equivalent of the CLIPS **defrule-module** command).

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a defrule data structure.

Returns: A string containing the name of the module in which the defrule is defined.

4.6.3 EnvFindDefrule

```
void *EnvFindDefrule(environment,defruleName);

void       *environment;
const char *defruleName;
```

Purpose: Returns a generic pointer to a named defrule.

Arguments:
1) A generic pointer to an environment.
2) The name of the defrule to be found.

Returns: A generic pointer to the named defrule if it exists, otherwise NULL.

4.6.4 EnvGetDefruleList

```
void EnvGetDefruleList(environment,&outputValue,theModule);

void        *environment;
DATA_OBJECT  outputValue;
void        *theModule;
```

Purpose: Returns the list of defrules in the specified module as a multifield value in the returnValue DATA_OBJECT (the C equivalent of the CLIPS **get-defrule-list** function)..

Arguments:
1) A generic pointer to an environment.
2) A pointer to a DATA_OBJECT in which the function results are stored. See sections 3.2.3 and 3.2.4 for information on examining the value stored in a DATA_OBJECT.
3) A generic pointer to the module from which the list will be extracted. A NULL pointer indicates that the list is to be extracted from al l modules.

Returns: No meaningful return value.

4.6.5 EnvGetDefruleName

```
const char *EnvGetDefruleName(environment,defrulePtr);

void *environment;
void *defrulePtr;
```

Purpose: Returns the name of a defrule.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a defrule data structure.

Returns: A string containing the name of the defrule.

4.6.6 EnvGetDefrulePPForm

```
const char *EnvGetDefrulePPForm(environment,defrulePtr);

void *environment;
void *defrulePtr;
```

Purpose: Returns the pretty print representation of a defrule.

Arguments: 1) A generic pointer to an environment.
2) A generic pointer to a defrule data structure.

Returns: A string containing the pretty print representation of the defrule (or the NULL pointer if no pretty print representation exists).

4.6.7 EnvGetDefruleWatchActivations

```
unsigned EnvGetDefruleWatchActivations(environment,defrulePtr);

void *environment;
void *defrulePtr;
```

Purpose: Indicates whether or not a particular defrule is being watched for activations.

Arguments: 1) A generic pointer to an environment.
2) A generic pointer to a defrule data structure.

Returns: An integer; one (1) if the defrule is being watched for activations, otherwise a zero (0).

4.6.8 EnvGetDefruleWatchFirings

```
unsigned EnvGetDefruleWatchFirings(environment,defrulePtr);

void *environment;
void *defrulePtr;
```

Purpose: Indicates whether or not a particular defrule is being watched for rule firings.

Arguments: 1) A generic pointer to an environment.
2) A generic pointer to a defrule data structure.

Returns: An integer; one (1) if the defrule is being watched for rule firings, otherwise a zero (0).

4.6.9 EnvGetIncrementalReset

```
int EnvGetIncrementalReset(environment);

void *environment;
```

Purpose: Returns the current value of the incremental reset behavior (the C equivalent of the CLIPS **get-incremental-reset** command).

Arguments: A generic pointer to an environment.

Returns: An integer; FALSE (0) if the behavior is disabled and TRUE (1) if the behavior is enabled.

4.6.10 EnvGetNextDefrule

```
void *EnvGetNextDefrule(environment,defrulePtr);

void *environment;
void *defrulePtr;
```

Purpose: Provides access to the list of defrules.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a defrule data structure (or NULL to get the first defrule).

Returns: A generic pointer to the first defrule in the list of defrules if *defrulePtr* is NULL, otherwise a generic pointer to the defrule immediately following *defrulePtr* in the list of defrules. If *defrulePtr* is the last defrule in the list of defrules, then NULL is returned.

4.6.11 EnvIsDefruleDeletable

```
int EnvIsDefruleDeletable(environment,defrulePtr);

void *environment;
void *defrulePtr;
```

Purpose: Indicates whether or not a particular defrule can be deleted.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a defrule data structure.

Returns: An integer; zero (0) if the defrule cannot be deleted, otherwise a one (1).

4.6.12 EnvListDefrules

```
void EnvListDefrules(environment,logicalName,theModule);

void       *environment;
const char *logicalName;
void       *theModule;
```

Purpose: Prints the list of defrules (the C equivalent of the CLIPS **list-defrules** command).

Arguments:
1) A generic pointer to an environment.
2) The logical name to which the listing output is sent.
3) A generic pointer to the module containing the defrules to be listed. A NULL pointer indicates that defrules in all modules should be listed.

Returns: No meaningful return value.

4.6.13 EnvMatches

```
void EnvMatches(environment,defrulePtr,verbosity,result);

void *environment;
void *defrulePtr;
int   verbosity;
DATA_OBJECT  result;
```

Purpose: Prints the partial matches and activations of a defrule (the C equivalent of the CLIPS **matches** command).

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a defrule data structure.
3) An integer indicating the amount of output that should be displayed – one of the following defined integer constants:

   VERBOSE
   SUCCINCT
   TERSE

4) A pointer to a DATA_OBJECT in which the function results are stored. See sections 3.2.3 and 3.2.4 for information on examining the value stored in a DATA_OBJECT. The symbol FALSE will be returned if the rule was not found, otherwise a multifield value with three inteer fields indicating the number of pattern matches, partial matches, and activations.

Returns: No meaningful return value.

4.6.14 EnvRefresh

```
int EnvRefresh(environment,defrulePtr);

void *environment;
void *defrulePtr;
```

Purpose: Refreshes a rule (the C equivalent of the CLIPS **refresh** command).

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a defrule data structure.

Returns: An integer; zero (0) if the rule was not found, otherwise a one (1).

4.6.15 EnvRemoveBreak

```
int EnvRemoveBreak(environment,defrulePtr);

void *environment;
void *defrulePtr;
```

Purpose: Removes a breakpoint for the specified defrule (the C equivalent of the CLIPS **remove-break** command).

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a defrule data structure.

Returns: An integer; zero (0) if a breakpoint did not exist for the rule, otherwise a one (1).

4.6.16 EnvSetBreak

```
void EnvSetBreak(environment,defrulePtr);

void *environment;
void *defrulePtr;
```

Purpose: Adds a breakpoint for the specified defrule (the C equivalent of the CLIPS **set-break** command).

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a defrule data structure.

Returns: No meaningful return value.

4.6.17 EnvSetDefruleWatchActivations

```
void EnvSetDefruleWatchActivations(environment,newState,defrulePtr);

void     *environment;
unsigned  newState;
void     *defrulePtr;
```

Purpose: Sets the activations watch item for a specific defrule.

Arguments:
1) A generic pointer to an environment.
2) The new activations watch state.
3) A generic pointer to a defrule data structure.

4.6.18 EnvSetDefruleWatchFirings

```
void EnvSetDefruleWatchFirings(environment,newState,defrulePtr);

void     *environment;
unsigned  newState;
void     *defrulePtr;
```

Purpose: Sets the rule firing watch item for a specific defrule.

Arguments:
1) A generic pointer to an environment.
2) The new rule firing watch state.
3) A generic pointer to a defrule data structure.

4.6.19 EnvSetIncrementalReset

```
int EnvSetIncrementalReset(environment,value);

void *environment;
int   value;
```

Purpose: Sets the incremental reset behavior. When this behavior is enabled (by default), newly defined rules are update based upon the current state of the fact-list. When disabled, newly defined rules are only updated by facts added after the rule is defined (the C equivalent of the CLIPS **set-incremental-reset** command).

Arguments:
1) A generic pointer to an environment.
2) The new value for the behavior: TRUE (1) to enable the behavior and FALSE (0) to disable it.

Returns: Returns the old value for the behavior.

4.6.20 EnvShowBreaks

```
void EnvShowBreaks(environment,logicalName,theModule);

void       *environment;
const char *logicalName;
void       *theModule;
```

Purpose: Prints the list of all rule breakpoints (the C equivalent of the CLIPS **show-breaks** command).

Arguments:
1) A generic pointer to an environment.
2) The logical name to which the listing output is sent.
3) A generic pointer to the module for which the breakpoints are to be listed. A NULL pointer indicates that the the breakpoints in all modules should be listed.

Returns: No meaningful return value.

4.6.21 EnvUndefrule

```
int EnvUndefrule(environment,defrulePtr);

void *environment;
void *defrulePtr;
```

Purpose: Removes a defrule from CLIPS (the C equivalent of the CLIPS **undefrule** command).

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a defrule data structure. If the NULL pointer is used, then all defrules will be deleted.

Returns: An integer; zero (0) if the defrule could not be deleted, otherwise a one (1).

Other: This function can trigger garbage collection.

4.7 Agenda Functions

The following function calls are used for manipulating the agenda.

4.7.1 EnvAddRunFunction

```
int EnvAddRunFunction(environment,runItemName,runFunction,priority);

void        *environment;
const char  *runItemName;
void       (*runFunction)(void *);
int          priority;

void runFunction(environment);

void *environment;
```

Purpose: Allows a user-defined function to be called after each rule firing. Such a feature is useful, for example, when bringing data in from some type of external device which does not operate in a synchronous manner. A user may define an external function which will be called by CLIPS after every rule is fired to check for the existence of new data.

Arguments:
1) A generic pointer to an environment.
2) The name associated with the user-defined run function. This name is used by the function **RemoveRunFunction**.
3) A pointer to the function which is to be called after every rule firing.
4) The priority of the run item which determines the order in which run items are called (higher priority items are called first). The

values -2000 to 2000 are reserved for CLIPS system defined run items and should not be used for user defined run items.

Returns: Returns a zero value if the run item could not be added, otherwise a non-zero value is returned.

Example

The following code is a simple example that prints a period after each rule firing:

```
#include "clips.h"

void PrintPeriod(void *);

int main()
  {
   void *theEnv;
   DATA_OBJECT rv;
   char *cs;

   theEnv = CreateEnvironment();

   cs =  "(defrule loop"
         "   ?f <- (loop)"
         "   =>"
         "   (retract ?f)"
         "   (assert (loop)))";

   EnvBuild(theEnv,cs);

   EnvAssertString(theEnv,"(loop)");

   EnvAddRunFunction(theEnv,"print-dot",PrintPeriod,0);

   EnvRun(theEnv,20);
  }

void PrintPeriod(
  void *environment)
  {
   EnvPrintRouter(environment,STDOUT,".");
  }
```

4.7.2 EnvAgenda

```
    void EnvAgenda(environment,logicalName,theModule)

    void       *environment;
    const char *logicalName;
    void       *theModule;
```

Purpose: Prints the list of rules currently on the agenda (the C equivalent of the CLIPS **agenda** command).

Arguments:
1) A generic pointer to an environment.
2) The logical name to which the listing output is sent.
3) A generic pointer to the module containing the agenda to be listed. A NULL pointer indicates that the agendas of all modules should be listed.

Returns: No meaningful return value.

4.7.3 EnvClearFocusStack

```
void EnvClearFocusStack(environment);

void *environment;
```

Purpose: Removes all modules from the focus stack (the C equivalent of the CLIPS **clear-focus-stack** command).

Arguments: A generic pointer to an environment.

Returns: No meaningful return value.

4.7.4 EnvDeleteActivation

```
int EnvDeleteActivation(environment,activationPtr);

void *environment;
void *activationPtr;
```

Purpose: Removes an activation from the agenda.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to an activation data structure. If the NULL pointer is used, then all activations will be deleted.

Returns: An integer; zero (0) if the activation could not be deleted, otherwise a one (1).

4.7.5 EnvFocus

```
void EnvFocus(environment,defmodulePtr);

void *environment;
void *defmodulePtr;
```

Purpose: Sets the current focus (the C equivalent of the CLIPS **focus** command).

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a defmodule data structure.

Returns: No meaningful value.

4.7.6 EnvGetActivationName

```
const char *EnvGetActivationName(environment,activationPtr);

void *environment;
void *activationPtr;
```

Purpose: Returns the name of the defrule from which the activation was generated.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to an activation data structure.

Returns: A string containing a defrule name.

4.7.7 EnvGetActivationPPForm

```
void EnvGetActivationPPForm(environment,buffer,bufferLength,activationPtr);

void   *environment;
char   *buffer;
size_t  bufferLength;
void   *activationPtr;
```

Purpose: Returns the pretty print representation of an agenda activation in the caller's buffer.

Arguments:
1) A generic pointer to an environment.
2) A pointer to the caller’s character buffer.

3) The maximum number of characters which could be stored in the caller's buffer (not including space for the terminating null character).
4) A generic pointer to an activation data structure.

4.7.8 EnvGetActivationSalience

```
int EnvGetActivationSalience(environment,activationPtr);

void *environment;
void *activationPtr;
```

Purpose: Returns the salience value associated with an activation. This salience value may be different from the the salience value of the defrule which generated the activation (due to dynamic salience).

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to an activation data structure.

Returns: The integer salience value of an activation.

4.7.9 EnvGetAgendaChanged

```
int EnvGetAgendaChanged(environment);

void *environment;
```

Purpose: Determines if any changes to the agenda of rule activations have occurred. If this function returns a non-zero integer, it is the user's responsibility to call SetAgendaChanged(0) to reset the internal flag. Otherwise, this function will continue to return non-zero even when no changes have occurred. This function is primarily used to determine when to update a display tracking rule activations.

Arguments: A generic pointer to an environment.

Returns: 0 if no changes to the agenda have occurred, non-zero otherwise.

4.7.10 EnvGetFocus

```
void *EnvGetFocus(environment);

void *environment;
```

Purpose: Returns the module associated with the current focus (the C equivalent of the CLIPS **get-focus** function).

Arguments: A generic pointer to an environment.

Returns: A generic pointer to a defmodule data structure (or NULL if the focus stack is empty).

4.7.11 EnvGetFocusStack

```
void EnvGetFocusStack(environment,&outputValue);

void        *environment;
DATA_OBJECT  outputValue;
```

Purpose: Returns the module names in the focus stack as a multifield value in the returnValue DATA_OBJECT (the C equivalent of the CLIPS **get-focus-stack** function).

Arguments:
1) A generic pointer to an environment.
2) A pointer to a DATA_OBJECT in which the function results are stored. See sections 3.2.3 and 3.2.4 for information on examining the value stored in a DATA_OBJECT.

4.7.12 EnvGetNextActivation

```
void *EnvGetNextActivation(environment,activationPtr);

void *environment;
void *activationPtr;
```

Purpose: Provides access to the list of activations on the agenda.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to an activation data structure (or NULL to get the first activation on the agenda).

Returns: A generic pointer to the first activation on the agenda if *activationPtr* is NULL, otherwise a generic pointer to the activation immediately following *activationPtr* on the agenda. If *activationPtr* is the last activation on the agenda, then NULL is returned.

4.7.13 EnvGetSalienceEvaluation

```
int EnvGetSalienceEvaluation(environment);

void *environment;
```

Purpose: Returns the current salience evaluation behavior (the C equivalent of the CLIPS **get-salience-evaluation** command).

Arguments: A generic pointer to an environment.

Returns: An integer (see SetSalienceEvaluation for the list of defined constants).

4.7.14 EnvGetStrategy

```
int EnvGetStrategy(environment);

void *environment;
```

Purpose: Returns the current conflict resolution strategy (the C equivalent of the CLIPS **get-strategy** command).

Arguments: A generic pointer to an environment.

Returns: An integer (see SetStrategy for the list of defined strategy constants).

4.7.15 EnvListFocusStack

```
void EnvListFocusStack(environment,logicalName);

void       *environment;
const char *logicalName;
```

Purpose: Prints the current focus stack (the C equivalent of the CLIPS **list-focus-stack** command).

Arguments:
1) A generic pointer to an environment.
2) The logical name to which the listing output is sent.

Returns: No meaningful return value.

4.7.16 EnvPopFocus

```
void *EnvPopFocus(environment);

void *environment;
```

Purpose: Removes the current focus from the focus stack and returns the module associated with that focus (the C equivalent of the CLIPS **pop-focus** function).

Arguments: A generic pointer to an environment.

Returns: A generic pointer to a defmodule data structure.

4.7.17 EnvRefreshAgenda

```
void EnvRefreshAgenda(environment,theModule);

void *environment;
void *theModule;
```

Purpose: Recomputes the salience values for all activations on the agenda and then reorders the agenda (the C equivalent of the CLIPS **refresh-agenda** command).

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to the module containing the agenda to be refreshed. A NULL pointer indicates that the agendas of all modules should be refreshed.

Returns: No meaningful return value.run

4.7.18 EnvRemoveRunFunction

```
int EnvRemoveRunFunction(environment,runItemName);

void *environment;
char *runItemName;
```

Purpose: Removes a named function from the list of functions to be called after every rule firing.

Arguments:
1) A generic pointer to an environment.

2) The name associated with the user-defined run function. This is the same name that was used when the run function was added with the function **AddRunFunction**.

Returns: Returns the integer value 1 if the named function was found and removed, otherwise 0 is returned.

4.7.19 EnvReorderAgenda

```
void EnvReorderAgenda(environment,theModule);

void *environment;
void *theModule;
```

Purpose: Reorders the agenda based on the current conflict resolution strategy and current activation saliences.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to the module containing the agenda to be reordered. A NULL pointer indicates that the agendas of all modules should be reordered.

Returns: No meaningful return value.

4.7.20 EnvRun

```
long long EnvRun(environment,runLimit);

void      *environment;
long long  runLimit;
```

Purpose: Allows rules to execute (the C equivalent of the CLIPS **run** command).

Arguments:
1) A generic pointer to an environment.
2) An integer which defines how many rules should fire before returning. If runLimit is a negative integer, rules will fire until the agenda is empty.

Returns: Returns an integer value; the number of rules that were fired.

4.7.21 EnvSetActivationSalience

```
int EnvSetActivationSalience(environment,activationPtr,newSalience);

void *environment;
void *activationPtr;
int   newSalience;
```

Purpose: Sets the salience value of an activation. The salience value of the defrule which generated the activation is unchanged.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to an activation data structure.
3) The new salience value (which is not restricted to the -10000 to +10000 range).

Returns: The old salience value of the activation.

Other: The function **ReorderAgenda** should be called after salience values have been changed to update the agenda.

4.7.22 EnvSetAgendaChanged

```
void EnvSetAgendaChanged(environment,changedFlag);

void *environment;
int   changedFlag;
```

Purpose: Sets the internal boolean flag which indicates when changes to the agenda of rule activations have occurred. This function is normally used to reset the flag to zero after GetAgendaChanged() returns non-zero.

Arguments:
1) A generic pointer to an environment.
2) An integer indicating whether changes in the agenda have occurred (non-zero) or not (0).

Returns: Nothing useful.

4.7.23 EnvSetSalienceEvaluation

```
int EnvSetSalienceEvaluation(environment,value);

void *environment;
int   value;
```

Purpose: Sets the salience evaluation behavior (the C equivalent of the CLIPS **set-salience-evaluation** command).

Arguments:
1) A generic pointer to an environment.
2) The new value for the behavior – one of the following defined integer constants:

   WHEN_DEFINED
   WHEN_ACTIVATED
   EVERY_CYCLE

Returns: Returns the old value for the behavior.

4.7.24 EnvSetStrategy

```
int EnvSetStrategy(environment,value);

void *environment;
int   value;
```

Purpose: Sets the conflict resolution strategy (the C equivalent of the CLIPS **set-strategy** command).

Arguments:
1) A generic pointer to an environment.
2) The new value for the behavior – one of the following defined integer constants:

   DEPTH_STRATEGY
   BREADTH_STRATEGY
   LEX_STRATEGY
   MEA_STRATEGY
   COMPLEXITY_STRATEGY
   SIMPLICITY_STRATEGY
   RANDOM_STRATEGY

Returns: Returns the old value for the strategy.

4.8 Defglobal Functions

The following function calls are used for manipulating defglobals.

4.8.1 EnvDefglobalModule

```
const char *EnvDefglobalModule(environment,theDefglobal);

void *environment;
void *theDefglobal;
```

Purpose: Returns the module in which a defglobal is defined (the C equivalent of the CLIPS **defglobal-module** command).

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a defglobal data structure.

Returns: A string containing the name of the module in which the defglobal is defined.

4.8.2 EnvFindDefglobal

```
void *EnvFindDefglobal(environment,globalName);

void       *environment;
const char *globalName;
```

Purpose: Returns a generic pointer to a named defglobal.

Arguments:
1) A generic pointer to an environment.
2) The name of the defglobal to be found (e.g. *x* for ?*x*).

Returns: A generic pointer to the named defglobal if it exists, otherwise NULL.

4.8.3 EnvGetDefglobalList

```
void EnvGetDefglobalList(environment,&outputValue,theModule);

void        *environment;
DATA_OBJECT  outputValue;
void        *theModule;
```

Purpose: Returns the list of defglobals in the specified module as a multifield value in the returnValue DATA_OBJECT (the C equivalent of the CLIPS **get-defglobal-list** function).

Arguments:
1) A generic pointer to an environment.

2) A pointer to a DATA_OBJECT in which the function results are stored. See sections 3.2.3 and 3.2.4 for information on examining the value stored in a DATA_OBJECT.
3) A generic pointer to the module from which the list will be extracted. A NULL pointer indicates that the list is to be extracted from al l modules.

Returns: No meaningful return value.

4.8.4 EnvGetDefglobalName

```
const char *EnvGetDefglobalName(environment,defglobalPtr);

void *environment;
void *defglobalPtr;
```

Purpose: Returns the name of a defglobal.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a defglobal data structure.

Returns: A string containing the name of the defglobal (e.g. *x* for ?*x*).

4.8.5 EnvGetDefglobalPPForm

```
const char *EnvGetDefglobalPPForm(environment,defglobalPtr);

void *environment;
void *defglobalPtr;
```

Purpose: Returns the pretty print representation of a defglobal.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a defglobal data structure.

Returns: A string containing the pretty print representation of the defglobal (or the NULL pointer if no pretty print representation exists).

4.8.6 EnvGetDefglobalValue

```
int   EnvGetDefglobalValue(environment,globalName,&outputValue);

void        *environment;
const char  *globalName;
```

```
DATA_OBJECT  outputValue;
```

Purpose: Returns the value of a defglobal.

Arguments:
1) A generic pointer to an environment.
2) The name of the global variable to be retrieved (e.g. *y* for ?*y*).
3) A pointer to a DATA_OBJECT in which the function results are stored. See sections 3.2.3 and 3.2.4 for information on examining the value stored in a DATA_OBJECT.

Returns: An integer; zero (0) if the defglobal was not found, otherwise a one (1). The DATA_OBJECT vPtr is assigned the current value of the defglobal.

4.8.7 EnvGetDefglobalValueForm

```
void  EnvGetDefglobalValueForm(environment,buffer,bufferLength,defglobalPtr);

void   *environment;
char   *buffer;
size_t  bufferLength;
void   *defglobalPtr;
```

Purpose: Returns a printed representation of a defglobal and its current value in the caller's buffer. For example,

```
?*x* = 5
```

Arguments:
1) A generic pointer to an environment.
2) A pointer to the caller's character buffer.
3) The maximum number of characters which could be stored in the caller's buffer (not including space for the terminating null character).
4) A generic pointer to a defglobal data structure.

4.8.8 EnvGetDefglobalWatch

```
unsigned EnvGetDefglobalWatch(environment,defglobalPtr);

void *environment;
void *defglobalPtr;
```

Purpose: Indicates whether or not a particular defglobal is being watched.

Arguments:
1) A generic pointer to an environment.

2) A generic pointer to a defglobal data structure.

Returns: An integer; one (1) if the defglobal is being watched, otherwise a zero (0).

4.8.9 EnvGetGlobalsChanged

```
int EnvGetGlobalsChanged(environment);

void *environment;
```

Purpose: Determines if any changes to global variables have occurred. If this function returns a non-zero integer, it is the user's responsibility to call SetGlobalsChanged(0) to reset the internal flag. Otherwise, this function will continue to return non-zero even when no changes have occurred. This function is primarily used to determine when to update a display tracking global variables.

Arguments: A generic pointer to an environment.

Returns: 0 if no changes to global variables have occurred, non-zero otherwise.

4.8.10 EnvGetNextDefglobal

```
void *EnvGetNextDefglobal(environment,defglobalPtr);

void *environment;
void *defglobalPtr;
```

Purpose: Provides access to the list of defglobals.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a defglobal data structure (or NULL to get the first defglobal).

Returns: A generic pointer to the first defglobal in the list of defglobals if *defglobalPtr* is NULL, otherwise a generic pointer to the defglobal immediately following *defglobalPtr* in the list of defglobals. If *defglobalPtr* is the last defglobal in the list of defglobals, then NULL is returned.

4.8.11 EnvGetResetGlobals

```
int EnvGetResetGlobals(environment);

void *environment;
```

Purpose: Returns the current value of the reset global variables behavior (the C equivalent of the CLIPS **get-reset-globals** command).

Arguments: A generic pointer to an environment.

Returns: An integer; FALSE (0) if globals are not reset and TRUE (1) if globals are reset.

4.8.12 EnvIsDefglobalDeletable

```
int EnvIsDefglobalDeletable(environment,defglobalPtr);

void *environment;
void *defglobalPtr;
```

Purpose: Indicates whether or not a particular defglobal can be deleted.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a defglobal data structure.

Returns: An integer; zero (0) if the defglobal cannot be deleted, otherwise a one (1).

4.8.13 EnvListDefglobals

```
void EnvListDefglobals(environment,logicalName,theModule);

void       *environment;
const char *logicalName;
void       *theModule;
```

Purpose: Prints the list of defglobals (the C equivalent of the CLIPS **list-defglobals** command).

Arguments:
1) A generic pointer to an environment.
2) The logical name to which the listing output is sent.
3) A generic pointer to the module containing the defglobals to be listed. A NULL pointer indicates that defglobals in all modules should be listed.

Returns: No meaningful return value.

4.8.14 EnvSetDefglobalValue

```
int EnvSetDefglobalValue(environment,globalName,&inputValue);

void        *environment;
const char  *globalName;
DATA_OBJECT  theValue;
```

Purpose: Sets the value of a defglobal.

Arguments:
1) A generic pointer to an environment.
2) The name of the global variable to be set (e.g. *y* for ?*y*).
3) A pointer to a DATA_OBJECT in which the new value is contained (see sections 3.2.3 and 3.3.5 for details on this data structure).

Returns: An integer; zero (0) if the defglobal was not found, otherwise a one (1).

Other: This function can trigger garbage collection.

4.8.15 EnvSetDefglobalWatch

```
void EnvSetDefglobalWatch(environment,newState,defglobalPtr);

void     *environment;
unsigned  newState;
void     *defglobalPtr;
```

Purpose: Sets the globals watch item for a specific defglobal.

Arguments:
1) A generic pointer to an environment.
2) The new globals watch state.
3) A generic pointer to a defglobal data structure.

4.8.16 EnvSetGlobalsChanged

```
void EnvSetGlobalsChanged(environment,changedFlag);

void *environment;
int   changedFlag;
```

Purpose: Sets the internal boolean flag which indicates when changes to global variables have occurred. This function is normally used to reset the flag to zero after GetGlobalsChanged() returns non-zero.

Arguments:
1) A generic pointer to an environment.
2) An integer indicating whether changes in global variables have occurred (non-zero) or not (0).

Returns: Nothing useful.

4.8.17 EnvSetResetGlobals

```
int   EnvSetResetGlobals(environment,value);

void *environment;
int   value;
```

Purpose: Sets the reset-globals behavior (the C equivalent of the CLIPS **set-reset-globals** command). When this behavior is enabled (by default) global variables are reset to their original values when the **reset** command is performed.

Arguments:
1) A generic pointer to an environment.
2) The new value for the behavior: TRUE (1) to enable the behavior and FALSE (0) to disable it.

Returns: Returns the old value for the behavior.

4.8.18 EnvShowDefglobals

```
void EnvShowDefglobals(environment,logicalName,theModule);

void       *environment;
const char *logicalName;
void       *theModule;
```

Purpose: Prints the list of defglobals and their current values (the C equivalent of the CLIPS **show-defglobals** command).

Arguments:
1) A generic pointer to an environment.
2) The logical name to which the listing output is sent.
3) A generic pointer to the module containing the defglobals to be displayed. A NULL pointer indicates that defglobals in all modules should be displayed.

Returns: No meaningful return value.

4.8.19 EnvUndefglobal

```
int EnvUndefglobal(environment,defglobalPtr);

void *environment;
void *defglobalPtr;
```

Purpose: Removes a defglobal from CLIPS (the C equivalent of the CLIPS **undefglobal** command).

Arguments: A generic pointer to a defglobal data structure. If the NULL pointer is used, then all defglobals will be deleted.

Returns:
1) A generic pointer to an environment.
2) An integer; zero (0) if the defglobal could not be deleted, otherwise a one (1).

Other: This function can trigger garbage collection.

4.9 Deffunction Functions

The following function calls are used for manipulating deffunctions.

4.9.1 EnvDeffunctionModule

```
const char *EnvDeffunctionModule(environment,theDeffunction);

void *environment;
void *theDeffunction;
```

Purpose: Returns the module in which a deffunction is defined (the C equivalent of the CLIPS **deffunction-module** command).

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a deffunction.

Returns: A string containing the name of the module in which the deffunction is defined.

4.9.2 EnvFindDeffunction

```
void *EnvFindDeffunction(environment,deffunctionName);

void       *environment;
const char *deffunctionName;
```

Purpose: Returns a generic pointer to a named deffunction.

Arguments:
1) A generic pointer to an environment.
2) The name of the deffunction to be found.

Returns: A generic pointer to the named deffunction if it exists, otherwise NULL.

4.9.3 EnvGetDeffunctionList

```
void EnvGetDeffunctionList(environment,&outputValue,theModule);

void        *environment;
DATA_OBJECT  outputValue;
void        *theModule;
```

Purpose: Returns the list of deffunctions in the specified module as a multifield value in the returnValue DATA_OBJECT (the C equivalent of the CLIPS **get-deffunction-list** function).

Arguments:
1) A generic pointer to an environment.
2) A pointer to a DATA_OBJECT in which the function results are stored. See sections 3.2.3 and 3.2.4 for information on examining the value stored in a DATA_OBJECT.
3) A generic pointer to the module from which the list will be extracted. A NULL pointer indicates that the list is to be extracted from al l modules.

Returns: No meaningful return value.

4.9.4 EnvGetDeffunctionName

```
const char *EnvGetDeffunctionName(environment,deffunctionPtr);

void *environment;
void *deffunctionPtr;
```

Purpose: Returns the name of a deffunction.

Arguments: 1) A generic pointer to an environment.
2) A generic pointer to a deffunction data structure.

Returns: A string containing the name of the deffunction.

4.9.5 EnvGetDeffunctionPPForm

```
const char *EnvGetDeffunctionPPForm(environment,deffunctionPtr);

void *environment;
void *deffunctionPtr;
```

Purpose: Returns the pretty print representation of a deffunction.

Arguments: 1) A generic pointer to an environment.
2) A generic pointer to a deffunction data structure.

Returns: A string containing the pretty print representation of the deffunction (or the NULL pointer if no pretty print representation exists).

4.9.6 EnvGetDeffunctionWatch

```
unsigned EnvGetDeffunctionWatch(environment,deffunctionPtr);

void *environment;
void *deffunctionPtr;
```

Purpose: Indicates whether or not a particular deffunction is being watched.

Arguments: 1) A generic pointer to an environment.
2) A generic pointer to a deffunction data structure.

Returns: An integer; one (1) if the deffunction is being watched, otherwise a zero (0).

4.9.7 EnvGetNextDeffunction

```
void *EnvGetNextDeffunction(environment,deffunctionPtr);

void *environment;
void *deffunctionPtr;
```

Purpose: Provides access to the list of deffunctions.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a deffunction data structure (or NULL to get the first deffunction).

Returns: A generic pointer to the first deffunction in the list of deffunctions if *deffunctionPtr* is NULL, otherwise a generic pointer to the deffunction immediately following *deffunctionPtr* in the list of deffunctions. If *deffunctionPtr* is the last deffunction in the list of deffunctions, then NULL is returned.

4.9.8 EnvIsDeffunctionDeletable

```
int EnvIsDeffunctionDeletable(environment,deffunctionPtr);

void *environment;
void *deffunctionPtr;
```

Purpose: Indicates whether or not a particular deffunction can be deleted.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a deffunction data structure.

Returns: An integer; zero (0) if the deffunction cannot be deleted, otherwise a one (1).

4.9.9 EnvListDeffunctions

```
void EnvListDeffunctions(environment,logicalName,theModule);

void       *environment;
const char *logicalName;
void       *theModule;
```

Purpose: Prints the list of deffunction (the C equivalent of the CLIPS **list-deffunctions** command).

Arguments:
1) A generic pointer to an environment.
2) The logical name to which the listing output is sent.
3) A generic pointer to the module containing the deffunctions to be listed. A NULL pointer indicates that deffunctions in all modules should be listed.

Returns: No meaningful return value.

4.9.10 EnvSetDeffunctionWatch

```
void EnvSetDeffunctionWatch(environment,newState,deffunctionPtr);

void     *environment;
unsigned  newState;
void     *deffunctionPtr;
```

Purpose: Sets the deffunctions watch item for a specific deffunction.

Arguments:
1) A generic pointer to an environment.
2) The new deffunctions watch state and a generic pointer to a deffunction data structure.

Returns: No meaningful return value.

4.9.11 EnvUndeffunction

```
int EnvUndeffunction(environment,deffunctionPtr);

void *environment;
void *deffunctionPtr;
```

Purpose: Removes a deffunction from CLIPS (the C equivalent of the CLIPS **undeffunction** command).

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to the deffunction (NULL means to delete all deffunctions).

Returns: An integer; zero (0) if the deffunction could not be deleted, otherwise a one (1).

Other: This function can trigger garbage collection.

4.10 Defgeneric Functions

The following function calls are used for manipulating generic functions.

4.10.1 EnvDefgenericModule

```
const char *EnvDefgenericModule(environment,theDefgeneric);

void *environment;
void *theDefgeneric;
```

Purpose: Returns the module in which a defgeneric is defined (the C equivalent of the CLIPS **defgeneric-module** command).

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a defgeneric data structure.

Returns: A string containing the name of the module in which the defgeneric is defined.

4.10.2 EnvFindDefgeneric

```
void *EnvFindDefgeneric(environment,defgenericName);

void       *environment;
const char *defgenericName;
```

Purpose: Returns a generic pointer to a named generic function.

Arguments:
1) A generic pointer to an environment.
2) The name of the generic to be found.

Returns: A generic pointer to the named generic function if it exists, otherwise NULL.

4.10.3 EnvGetDefgenericList

```
void EnvGetDefgenericList(environment,&outputValue,theModule);

void        *environment;
DATA_OBJECT  outputValue;
void        *theModule;
```

Purpose: Returns the list of defgenerics in the specified module as a multifield value in the returnValue DATA_OBJECT (the C equivalent of the CLIPS **get-defgeneric-list** function).

Arguments:
1) A generic pointer to an environment.
2) A pointer to a DATA_OBJECT in which the function results are stored. See sections 3.2.3 and 3.2.4 for information on examining the value stored in a DATA_OBJECT.
3) A generic pointer to the module from which the list will be extracted. A NULL pointer indicates that the list is to be extracted from al l modules.

Returns: No meaningful return value.

4.10.4 EnvGetDefgenericName

```
const char *EnvGetDefgenericName(environment,defgenericPtr);

void *environment;
void *defgenericPtr;
```

Purpose: Returns the name of a generic function.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a defgeneric data structure.

Returns: A string containing the name of the generic function.

4.10.5 EnvGetDefgenericPPForm

```
const char *EnvGetDefgenericPPForm(environment,defgenericPtr);

void *environment;
void *defgenericPtr;
```

Purpose: Returns the pretty print representation of a generic function.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a defgeneric data structure.

Returns: A string containing the pretty print representation of the generic function (or the NULL pointer if no pretty print representation exists).

4.10.6 EnvGetDefgenericWatch

```
unsigned EnvGetDefgenericWatch(environment,defgenericPtr);

void *environment;
void *defgenericPtr;
```

Purpose: Indicates whether or not a particular defgeneric is being watched.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a defgeneric data structure.

Returns: An integer; one (1) if the defgeneric is being watched, otherwise a zero (0).

4.10.7 EnvGetNextDefgeneric

```
void *EnvGetNextDefgeneric(environment,defgenericPtr);

void *environment;
void *defgenericPtr;
```

Purpose: Provides access to the list of generic functions.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a defgeneric data structure (or NULL to get the first generic function).

Returns: A generic pointer to the first generic function in the list of generic functions if *defgenericPtr* is NULL, otherwise a generic pointer to the generic function immediately following *defgenericPtr* in the list of generic functions. If *defgenericPtr* is the last generic function in the list of generic functions, then NULL is returned.

4.10.8 EnvIsDefgenericDeletable

```
int EnvIsDefgenericDeletable(environment,defgenericPtr);

void *environment;
void *defgenericPtr;
```

Purpose: Indicates whether or not a particular generic function and all its methods can be deleted.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a defgeneric data structure.

Returns: An integer: zero (0) if the generic function and all its methods cannot be deleted, otherwise a one (1).

4.10.9 EnvListDefgenerics

```
void EnvListDefgenerics(environment,logicalName,theModule);

void       *environment;
const char *logicalName;
void       *theModule;
```

Purpose: Prints the list of defgenerics (the C equivalent of the CLIPS **list-defgenerics** command).

Arguments:
1) A generic pointer to an environment.
2) The logical name to which the listing output is sent.
3) A generic pointer to the module containing the defgenerics to be listed. A NULL pointer indicates that defgenerics in all modules should be listed.

Returns: No meaningful return value.

4.10.10 EnvSetDefgenericWatch

```
void EnvSetDefgenericWatch(environment,newState,defgenericPtr);

void     *environment;
unsigned  newState;
void     *defgenericPtr;
```

Purpose: Sets the defgenerics watch item for a specific defgeneric.

Arguments:
1) A generic pointer to an environment.
2) The new generic-functions watch state.
3) A generic pointer to a defgeneric data structure.

4.10.11 EnvUndefgeneric

```
int EnvUndefgeneric(environment,defgenericPtr);

void *environment;
void *defgenericPtr;
```

Purpose: Removes a generic function and all its methods from CLIPS (the C equivalent of the CLIPS **undefgeneric** command).

Arguments:
1) A generic pointer to an environment.

2) A generic pointer to the generic function (NULL means to delete all generic functions).

Returns: An integer: zero (0) if the generic function and all its methods could not be deleted, otherwise a one (1).

Other: This function can trigger garbage collection.

4.11 Defmethod Functions

The following function calls are used for manipulating generic function methods.

4.11.1 EnvGetDefmethodDescription

```
void EnvGetDefmethodDescription(environment,buffer,bufferLength,
                                defgenericPtr,methodIndex);

void   *environment;
char   *buffer;
size_t  bufferLength;
void   *defgenericPtr;
long    methodIndex;
```

Purpose: Stores a synopsis of the method parameter restrictions in the caller's buffer.

Arguments:
1) A generic pointer to an environment.
2) A pointer to the caller's buffer.
3) The maximum number of characters which could be stored in the caller's buffer (not including space for the terminating null character).
4) A generic pointer to a defgeneric data structure.
5) The index of the generic function method.

Returns: No meaningful return value.

4.11.2 EnvGetDefmethodList

```
void EnvGetDefmethodList(environment,defgenericPtr,&outputValue);

void        *environment;
void        *defgenericPtr;
DATA_OBJECT  outputValue;
```

Purpose: Returns the list of currently defined defmethods for the specified defgeneric. This function is the C equivalent of the CLIPS **get-defmethod-list** command).

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to the defgeneric (NULL for all defgenerics).
3) A pointer to a DATA_OBJECT in which the function results are stored. See sections 3.2.3 and 3.2.4 for information on examining the value stored in a DATA_OBJECT.

Returns: No meaningful return value.

Notes: Note that the name and index for each defmethod are stored as pairs in the retrieved multifield value.

4.11.3 EnvGetDefmethodPPForm

```
const char *EnvGetDefmethodPPForm(environment,defgenericPtr,methodIndex);

void *environment;
void *defgenericPtr;
long  methodIndex;
```

Purpose: Returns the pretty print representation of a generic function method.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a defgeneric data structure.
3) The index of the generic function method.

Returns: A string containing the pretty print representation of the generic function method (or the NULL pointer if no pretty print representation exists).

4.11.4 EnvGetDefmethodWatch

```
unsigned EnvGetDefmethodWatch(environment,defgenericPtr,methodIndex);

void *environment;
void *defgenericPtr;
long  methodIndex
```

Purpose: Indicates whether or not a particular defmethod is being watched.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a defgeneric data structure.
3) The index of the generic function method.

Returns: An integer; one (1) if the defmethod is being watched, otherwise a zero (0).

4.11.5 EnvGetMethodRestrictions

```
void EnvGetMethodRestrictions(environment,defgenericPtr,methodIndex,&outputValue);

void        *environment;
void        *defgenericPtr;
long         methodIndex;
DATA_OBJECT  outputValue;
```

Purpose: Returns the restrictions for the specified method. This function is the C equivalent of the CLIPS **get-method-restrictions** function.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to the defgeneric (NULL for all defgenerics).
3) The index of the generic function method.
4) A pointer to a DATA_OBJECT in which the function results are stored. See sections 3.2.3 and 3.2.4 for information on examining the value stored in a DATA_OBJECT.

Returns: A multifield value containing the restrictions for the specified method (the description of the **get-method-restrictions** function in the Basic Programming Guide explains the meaning of the fields in the multifield value). The multifield functions described in section 3.2.4 can be used to retrieve the method restrictions from the list.

4.11.6 EnvGetNextDefmethod

```
unsigned EnvGetNextDefmethod(environment,defgenericPtr,methodIndex);

void *environment;
void *defgenericPtr;
long  methodIndex;
```

Purpose: Provides access to the list of methods for a particular generic function.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a defgeneric data structure.
3) The index of a generic function method (0 to get the first method of the generic function).

Returns: The index of the first method in the list of methods for the generic function if *methodIndex* is 0, otherwise the index of the method immediately following *methodIndex* in the list of methods for the generic function. If *methodIndex* is the last method in the list of methods for the generic function, then 0 is returned.

4.11.7 EnvIsDefmethodDeletable

```
int EnvIsDefmethodDeletable(environment,defgenericPtr,methodIndex);

void *environment;
void *defgenericPtr;
long  methodIndex;
```

Purpose: Indicates whether or not a particular generic function method can be deleted.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a defgeneric data structure.
3) The index of the generic function method.

Returns: An integer: zero (0) if the method cannot be deleted, otherwise a one (1).

4.11.8 EnvListDefmethods

```
void EnvListDefmethods(environment,logicalName,defgenericPtr);

void       *environment;
const char *logicalName;
void       *defgenericPtr;
```

Purpose: Prints the list of methods for a particular generic function (the C equivalent of the CLIPS **list-defmethods** command).

Arguments:
1) A generic pointer to an environment.
2) The logical name of the output destination to which tosend the method listing
3) A generic pointer to the generic function (NULL to list methods for all generic functions).

Returns: No meaningful return value.

4.11.9 EnvSetDefmethodWatch

```
void EnvSetDefmethodWatch(environment,newState,defgenericPtr,methodIndex);

void     *environment;
unsigned  newState;
void     *defgenericPtr;
long      methodIndex
```

Purpose: Sets the methods watch item for a specific defmethod.

Arguments:
1) A generic pointer to an environment.
2) The new methods watch state.
3) A generic pointer to a defgeneric data structure.
4) The index of the generic function method.

4.11.10 EnvUndefmethod

```
int EnvUndefmethod(environment,defgenericPtr,methodIndex);

void *environment;
void *defgenericPtr;
long  methodIndex;
```

Purpose: Removes a generic function method from CLIPS (the C equivalent of the CLIPS **undefmethod** command).

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a defgeneric data structure (NULL to delete all methods for all generic functions).
3) The index of the generic function method (0 to delete all methods of the generic function - must be 0 if *defgenericPtr* is NULL).

Returns: An integer: zero (0) if the method could not be deleted, otherwise a one (1).

Other: This function can trigger garbage collection.

4.12 Defclass Functions

The following function calls are used for manipulating defclasses.

4.12.1 EnvBrowseClasses

```
void EnvBrowseClasses(environment,logicalName,defclassPtr);

void       *environment;
const char *logicalName;
void       *defclassPtr;
```

Purpose: Prints a "graph" of all classes which inherit from the specified class. This function is the C equivalent of the CLIPS **browse-classes** command.

Arguments:
1) A generic pointer to an environment.
2) The logical name of the output destination to which to send the browse display.
3) A generic pointer to the class which is to be browsed.

Returns: No meaningful return value.

4.12.2 EnvClassAbstractP

```
int EnvClassAbstractP(environment,defclassPtr);

void *environment;
void *defclassPtr;
```

Purpose: Determines if a class is concrete or abstract, i.e. if a class can have direct instances or not. This function is the C equivalent of the CLIPS **class-abstractp** command.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to the defclass data structure.

Returns: The integer 1 if the class is abstract, or 0 if the class is concrete.

4.12.3 EnvClassReactiveP

```
int EnvClassReactiveP(environment,defclassPtr);

void *environment;
void *defclassPtr;
```

Purpose: Determines if a class is reactive or non-reactive, i.e. if objects of the class can match object patterns. This function is the C equivalent of the CLIPS **class-reactivep** command.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to the defclass data structure.

Returns: The integer 1 if the class is reactive, or 0 if the class is non-reactive.

4.12.4 EnvClassSlots

```
void EnvClassSlots(environment,defclassPtr,&outputValue,inheritFlag);

void        *environment;
void        *defclassPtr;
DATA_OBJECT  outputValue;
int          inheritFlag;
```

Purpose: Groups the names of slots of a class into a multifield data object. This function is the C equivalent of the CLIPS **class-slots** command.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to the defclass data structure.
3) A pointer to a DATA_OBJECT in which the function results are stored. See sections 3.2.3 and 3.2.4 for information on examining the value stored in a DATA_OBJECT.

4) The integer 1 to include inherited slots or 0 to only include explicitly defined slots.

Returns: No meaningful return value.

4.12.5 EnvClassSubclasses

```
void EnvClassSubclasses(environment,defclassPtr,&result,inheritFlag);

void        *environment;
void        *defclassPtr;
DATA_OBJECT  result;
int          inheritFlag;
```

Purpose: Groups the names of subclasses of a class into a multifield data object. This function is the C equivalent of the CLIPS **class-subclasses** command.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to the defclass data structure.
3) A Pointer to the data object in which to store the multifield. See sections 3.3.5 and 3.3.6 for information on setting the value stored in a DATA_OBJECT.
4) The integer 1 to include inherited subclasses or 0 to only include direct subclasses.

Returns: No meaningful return value.

4.12.6 EnvClassSuperclasses

```
void EnvClassSuperclasses(environment,defclassPtr,&outputValue,inheritFlag);

void        *environment;
void        *defclassPtr;
DATA_OBJECT  outputValue;
int          inheritFlag;
```

Purpose: Groups the names of superclasses of a class into a multifield data object. This function is the C equivalent of the CLIPS **class-superclasses** command.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to the defclass data structure.

3) A pointer to a DATA_OBJECT in which the function results are stored. See sections 3.2.3 and 3.2.4 for information on examining the value stored in a DATA_OBJECT.
4) The integer 1 to include inherited superclasses or 0 to only include direct superclasses.

Returns: No meaningful return value.

4.12.7 EnvDefclassModule

```
const char *EnvDefclassModule(environment,theDefclass);

void *environment;
void *theDefclass;
```

Purpose: Returns the module in which a defclass is defined (the C equivalent of the CLIPS **defclass-module** command).

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to the defclass data structure.

Returns: A string containing the name of the module in which the defclass is defined.

4.12.8 EnvDescribeClass

```
void EnvDescribeClass(environment,logicalName,defclassPtr);

void       *environment;
const char *logicalName;
void       *defclassPtr;
```

Purpose: Prints a summary of the specified class including: abstract/concrete behavior, slots and facets (direct and inherited) and recognized message-handlers (direct and inherited). This function is the C equivalent of the CLIPS **describe-class** command.

Arguments:
1) A generic pointer to an environment.
2) The logical name of the output destination to which to send the description.
3) A generic pointer to the class which is to be described.

Returns: No meaningful return value.

4.12.9 EnvFindDefclass

```
void *EnvFindDefclass(environment,defclassName);

void       *environment;
const char *defclassName;
```

Purpose: Returns a generic pointer to a named class.

Arguments:
1) A generic pointer to an environment.
2) The name of the class to be found.

Returns: A generic pointer to the named class if it exists, otherwise NULL.

4.12.10 EnvGetClassDefaultsMode

```
unsigned short EnvGetClassDefaultsMode(environment);

void *environment;
```

Purpose: Returns the current class defaults mode (the C equivalent of the CLIPS **get-class-defaults-mode** command).

Arguments: A generic pointer to an environment.

Returns: An integer (see SetClassDefaultsMode for the list of mode constants).

4.12.11 EnvGetDefclassList

```
void EnvGetDefclassList(environment,&outputValue,theModule);

void        *environment;
DATA_OBJECT  outputValue;
void        *theModule;
```

Purpose: Returns the list of defclasses in the specified module as a multifield value in the returnValue DATA_OBJECT (the C equivalent of the CLIPS **get-defclass-list** function).

Arguments:
1) A generic pointer to an environment.
2) A pointer to a DATA_OBJECT in which the function results are stored. See sections 3.2.3 and 3.2.4 for information on examining the value stored in a DATA_OBJECT.

3) A generic pointer to the module from which the list will be extracted. A NULL pointer indicates that the list is to be extracted from al l modules.

Returns: No meaningful return value.

4.12.12 EnvGetDefclassName

```
const char *EnvGetDefclassName(environment,defclassPtr);

void *environment;
void *defclassPtr;
```

Purpose: Returns the name of a class.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a defclass data structure.

Returns: A string containing the name of the class.

4.12.13 EnvGetDefclassPPForm

```
const char *EnvGetDefclassPPForm(environment,defclassPtr);

void *environment;
void *defclassPtr;
```

Purpose: Returns the pretty print representation of a class.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a defclass data structure.

Returns: A string containing the pretty print representation of the class (or the NULL pointer if no pretty print representation exists).

4.12.14 EnvGetDefclassWatchInstances

```
unsigned EnvGetDefclassWatchInstances(environment,defclassPtr);

void *environment;
void *defclassPtr;
```

Purpose: Indicates whether or not a particular defclass is being watched for instance creation and deletions.

Arguments: 1) A generic pointer to an environment.
2) A generic pointer to a defclass data structure.

Returns: An integer; one (1) if the defclass is being watched, otherwise a zero (0).

4.12.15 EnvGetDefclassWatchSlots

```
unsigned EnvGetDefclassWatchSlots(environment,defclassPtr);

void *environment;
void *defclassPtr;
```

Purpose: Indicates whether or not a particular defclass is being watched for slot changes.

Arguments: 1) A generic pointer to an environment.
2) A generic pointer to a defclass data structure.

Returns: An integer; one (1) if the defclass is being watched for slot changes, otherwise a zero (0).

4.12.16 EnvGetNextDefclass

```
void *EnvGetNextDefclass(environment,defclassPtr);

void *environment;
void *defclassPtr;
```

Purpose: Provides access to the list of classes.

Arguments: 1) A generic pointer to an environment.
2) A generic pointer to a defclass data structure (or NULL to get the first class).

Returns: A generic pointer to the first class in the list of classes if *defclassPtr* is NULL, otherwise a generic pointer to the class immediately following *defclassPtr* in the list of classes. If *defclassPtr* is the last class in the list of classes, then NULL is returned.

4.12.17 EnvIsDefclassDeletable

```
int EnvIsDefclassDeletable(environment,defclassPtr);

void *environment;
void *defclassPtr;
```

Purpose: Indicates whether or not a particular class and all its subclasses can be deleted.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a defclass data structure.

Returns: An integer; zero (0) if the class cannot be deleted, otherwise a one (1).

4.12.18 EnvListDefclasses

```
void EnvListDefclasses(environment,logicalName,theModule);

void       *environment;
const char *logicalName;
void       *theModule;
```

Purpose: Prints the list of defclasses (the C equivalent of the CLIPS **list-defclasses** command).

Arguments:
1) A generic pointer to an environment.
2) The logical name to which the listing output is sent.
3) A generic pointer to the module containing the defclasses to be listed. A NULL pointer indicates that defclasses in all modules should be listed.

Returns: No meaningful return value.

4.12.19 EnvSetClassDefaultsMode

```
unsigned short EnvSetClassDefaultsMode(environment,value);

void           *environment;
unsigned short  value;
```

Purpose: Sets the current class defaults mode (the C equivalent of the CLIPS **set-class-defaults-mode** command).

Arguments:
1) A generic pointer to an environment.
2) The new value for the mode – one of the following defined integer constants:

   CONVENIENCE_MODE
   CONSERVATION_MODE

Returns: Returns the old value for the mode.

4.12.20 EnvSetDefclassWatchInstances

```
void EnvSetDefclassWatchInstances(environment,newState,defclassPtr);

void     *environment;
unsigned  newState;
void     *defclassPtr;
```

Purpose: Sets the instances watch item for a specific defclass.

Arguments:
1) A generic pointer to an environment.
2) The new instances watch state and a generic pointer to a defclass data structure.

4.12.21 EnvSetDefclassWatchSlots

```
void EnvSetDefclassWatchSlots(environment,newState,defclassPtr);

void     *environment;
unsigned  newState;
void     *defclassPtr;
```

Purpose: Sets the slots watch item for a specific defclass.

Arguments:
1) A generic pointer to an environment.
2) The new slots watch state and a generic pointer to a defclass data structure.

4.12.22 EnvSlotAllowedClasses

```
void EnvSlotAllowedClasses(environment,defclassPtr,slotName,&outputValue);

void        *environment;
void        *defclassPtr;
const char  *slotName;
DATA_OBJECT  outputValue;
```

Purpose: Groups the allowed-classes for a slot into a multifield data object. This function is the C equivalent of the CLIPS **slot-allowed-classes** function.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a defclass data structure.
3) Name of the slot.
4) A pointer to a DATA_OBJECT in which the function results are stored. See sections 3.2.3 and 3.2.4 for information on examining the value stored in a DATA_OBJECT.

Returns: No meaningful return value.

4.12.23 EnvSlotAllowedValues

```
void EnvSlotAllowedValues(environment,defclassPtr,slotName,&outputValue);

void        *environment;
void        *defclassPtr;
const char  *slotName;
DATA_OBJECT  outputValue;
```

Purpose: Groups the allowed-values for a slot into a multifield data object. This function is the C equivalent of the CLIPS **slot-allowed-values** function.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a defclass data structure.
3) Name of the slot.
4) A pointer to a DATA_OBJECT in which the function results are stored. See sections 3.2.3 and 3.2.4 for information on examining the value stored in a DATA_OBJECT.

Returns: No meaningful return value.

4.12.24 EnvSlotCardinality

```
void EnvSlotCardinality(environment,defclassPtr,slotName,&outputValue);

void        *environment;
void        *defclassPtr;
const char  *slotName;
DATA_OBJECT  outputValue;
```

Purpose: Groups the cardinality information for a slot into a multifield data object. This function is the C equivalent of the CLIPS **slot-cardinality** function.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to the class.
3) Name of the slot.
4) A pointer to a DATA_OBJECT in which the function results are stored. See sections 3.2.3 and 3.2.4 for information on examining the value stored in a DATA_OBJECT.

Returns: No meaningful return value.

4.12.25 EnvSlotDefaultValue

```
void EnvSlotDefaultValue(environment,defclassPtr,slotName,&outputValue);

void        *environment;
void        *defclassPtr;
const char  *slotName;
DATA_OBJECT  outputValue;
```

Purpose: Returns the default value in the data object. This function is the C equivalent of the CLIPS **slot-default-value** function.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to the class.
3) Name of the slot.
4) A pointer to a DATA_OBJECT in which the function results are stored. See sections 3.2.3 and 3.2.4 for information on examining the value stored in a DATA_OBJECT.

Returns: No meaningful return value.

4.12.26 EnvSlotDirectAccessP

```
int EnvSlotDirectAccessP(environment,defclassPtr,slotName);

void       *environment;
void       *defclassPtr,
const char *slotName;
```

Purpose: Determines if the specified slot is directly accessible.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a defclass data structure.
3) The name of the slot.

Returns: An integer: 1 if the slot is directly accessible, otherwise 0.

4.12.27 EnvSlotExistP

```
int EnvSlotExistP(environment,defclassPtr,slotName,inheritFlag);

void       *environment;
void       *defclassPtr,
const char *slotName;
int         inheritFlag;
```

Purpose: Determines if the specified slot exists.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a defclass data structure.
3) The name of the slot.

Returns: An integer: If inheritFlag is 0 and the slot is directly defined in the specified class, then 1 is returned, otherwise 0 is returned. If inheritFlag is 1 and the slot is defined either in the specified class or an inherited class, then 1 is returned, otherwise 0 is returned.

4.12.28 EnvSlotFacets

```
void EnvSlotFacets(environment,defclassPtr,slotName,&outputValue);

void        *environment;
void        *defclassPtr;
const char  *slotName;
DATA_OBJECT  outputValue;
```

Purpose: Groups the facet values of a class slot into a multifield data object. This function is the C equivalent of the CLIPS **slot-facets** command. See section 10.8.1.11 in the Basic Programming Guide for more detail.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to the class.
3) The name of the slot.

4) A pointer to a DATA_OBJECT in which the function results are stored. See sections 3.2.3 and 3.2.4 for information on examining the value stored in a DATA_OBJECT.

Returns: No meaningful return value.

4.12.29 EnvSlotInitableP

```
int EnvSlotInitableP(environment,defclassPtr,slotName);

void       *environment;
void       *defclassPtr,
const char *slotName;
```

Purpose: Determines if the specified slot is initable.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a defclass data structure.
3) The name of the slot.

Returns: An integer: 1 if the slot is initable, otherwise 0.

4.12.30 EnvSlotPublicP

```
int EnvSlotPublicP(environment,defclassPtr,slotName);

void       *environment;
void       *defclassPtr,
const char *slotName;
```

Purpose: Determines if the specified slot is public.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a defclass data structure.
3) The name of the slot.

Returns: An integer: 1 if the slot is public, otherwise 0.

4.12.31 EnvSlotRange

```
void EnvSlotRange(environment,defclassPtr,slotName,&outputValue);

void        *environment;
void        *defclassPtr;
const char  *slotName;
```

```
DATA_OBJECT  outputValue;
```

Purpose: Groups the numeric range information for a slot into a multifield data object. This function is the C equivalent of the CLIPS **slot-range** function.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a defclass data structure.
3) Name of the slot.
4) A pointer to a DATA_OBJECT in which the function results are stored. See sections 3.2.3 and 3.2.4 for information on examining the value stored in a DATA_OBJECT.

Returns: No meaningful return value.

4.12.32 EnvSlotSources

```
void EnvSlotSources(environment,defclassPtr,slotName,&outputValue);

void        *environment;
void        *defclassPtr;
const char  *slotName;
DATA_OBJECT  outputValue;
```

Purpose: Groups the names of the class sources of a slot into a multifield data object. This function is the C equivalent of the CLIPS **slot-sources** command. See section 10.8.1.12 in the Basic Programming Guide for more detail.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a defclass data structure.
3) Name of the slot.
4) A pointer to a DATA_OBJECT in which the function results are stored. See sections 3.2.3 and 3.2.4 for information on examining the value stored in a DATA_OBJECT.

Returns: No meaningful return value.

4.12.33 EnvSlotTypes

```
void EnvSlotTypes(environment,defclassPtr,slotName,&outputValue);

void        *environment;
void        *defclassPtr;
const char  *slotName;
```

```
DATA_OBJECT  outputValue;
```

Purpose: Groups the names of the primitive data types allowed for a slot into a multifield data object. This function is the C equivalent of the CLIPS **slot-types** function.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a defclass data structure.
3) The name of the slot.
4) A pointer to a DATA_OBJECT in which the function results are stored. See sections 3.2.3 and 3.2.4 for information on examining the value stored in a DATA_OBJECT.

Returns: No meaningful return value.

4.12.34 EnvSlotWritableP

```
int EnvSlotWritableP(environment,defclassPtr,slotName);

void       *environment;
void       *defclassPtr,
const char *slotName;
```

Purpose: Determines if the specified slot is writable.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a defclass data structure.
3) The name of the slot.

Returns: An integer: 1 if the slot is writable, otherwise 0.

4.12.35 EnvSubclassP

```
int EnvSubclassP(environment,defclassPtr1,defclassPtr2);

void *environment;
void *defclassPtr1;
void *defclassPtr2;
```

Purpose: Determines if a class is a subclass of another class.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a defclass data structure.
3) A generic pointer to a defclass data structure.

Returns: An integer: 1 if the first class is a subclass of the second class.

4.12.36 EnvSuperclassP

```
int EnvSuperclassP(environment,defclassPtr1,defclassPtr2);

void *environment;
void *defclassPtr1;
void *defclassPtr2;
```

Purpose: Determines if a class is a superclass of another class.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a defclass data structure.
3) A generic pointer to a defclass data structure.

Returns: An integer: 1 if the first class is a superclass of the second class.

4.12.37 EnvUndefclass

```
int EnvUndefclass(environment,defclassPtr);

void *environment;
void *defclassPtr;
```

Purpose: Removes a class and all its subclasses from CLIPS (the C equivalent of the CLIPS **undefclass** command).

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a defclass data structure.

Returns: An integer; zero (0) if the class could not be deleted, otherwise a one (1).

Other: This function can trigger garbage collection.

4.13 Instance Functions

The following function calls are used for manipulating instances.

4.13.1 EnvBinaryLoadInstances

```
long EnvBinaryLoadInstances(environment,fileName);

void       *environment;
const char *fileName;
```

Purpose: Loads a set of instances from a binary file into the CLIPS data base (the C equivalent of the CLIPS **bload-instances** command).

Arguments:
1) A generic pointer to an environment.
2) A string representing the name of the binary file.

Returns: Returns the number of instances restored or -1 if the file could not be accessed.

4.13.2 EnvBinarySaveInstances

```
long EnvBinarySaveInstances(environment,fileName,saveCode);

void       *environment;
const char *fileName;
int         saveCode;
```

Purpose: Saves the instances in the system to the specified binary file (the C equivalent of the CLIPS **bsave-instances** command).

Arguments:
1) A generic pointer to an environment.
2) A string representing the name of the binary file.
3) An integer flag indicating whether to save local (current module only) or visible instances. Use either the constant LOCAL_SAVE or VISIBLE_SAVE.

Returns: Returns the number of instances saved.

4.13.3 EnvCreateRawInstance

```
void *EnvCreateRawInstance(environment,defclassPtr,instanceName);

void       *environment;
```

```
void       *defclassPtr;
const char *instanceName;
```

Purpose: Creates an empty instance with the specified name of the specified class. No slot overrides or class default initializations are performed for the instance.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to the class of the new instance.
3) The name of the new instance.

Returns: A generic pointer to the new instance, NULL on errors.

Warning: This function bypasses message-passing.

4.13.4 EnvDecrementInstanceCount

```
void EnvDecrementInstanceCount(environment,instancePtr);

void *environment;
void *instancePtr;
```

Purpose: This function should *only* be called to reverse the effects of a previous call to IncrementInstanceCount(). As long as an instance's count is greater than zero, the memory allocated to it cannot be released for other use.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to the instance.

Returns: No meaningful return value.

4.13.5 EnvDeleteInstance

```
int EnvDeleteInstance(environment,instancePtr);

void *environment;
void *instancePtr;
```

Purpose: Deletes the specified instance(s).

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to the instance to be deleted. If the pointer is NULL, all instances in the system are deleted.

Returns: Non-zero if successful, 0 otherwise.

Other: This function can trigger garbage collection.

Warning: This function bypasses message-passing.

4.13.6 EnvDirectGetSlot

```
void EnvDirectGetSlot(environment,instancePtr,slotName,&outputValue);

void        *environment;
void        *instancePtr;
const char  *slotName;
DATA_OBJECT  outputValue;
```

Purpose: Stores the value of the specified slot of the specified instance in the caller's buffer (the C equivalent of the CLIPS **dynamic-get** function).

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to the instance.
3) The name of the slot.
4) A pointer to a DATA_OBJECT in which the function results are stored. See sections 3.2.3 and 3.2.4 for information on examining the value stored in a DATA_OBJECT.

Returns: No meaningful return value.

Warning: This function bypasses message-passing.

4.13.7 EnvDirectPutSlot

```
int EnvDirectPutSlot(environment,instancePtr,slotName,&inputValue);

void        *environment;
void        *instancePtr;
const char  *slotName;
DATA_OBJECT  inputValue;
```

Purpose: Stores a value in the specified slot of the specified instance (the C equivalent of the CLIPS **dynamic-put** function).

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to the instance.

3) The name of the slot.
4) The caller's buffer containing the new value (an error is generated if this value is NULL). See sections 3.3.5 and 3.3.6 for information on setting the value stored in a DATA_OBJECT.

Returns: Returns an integer; if zero, an error occurred while setting the slot. If non-zero, no errors occurred.

Other: This function can trigger garbage collection.

Warning: This function bypasses message-passing.

4.13.8 EnvFindInstance

```
void *EnvFindInstance(environment,theModule,instanceName,searchImports);

void       *environment;
void       *theModule;
const char *instanceName;
unsigned    searchImports;
```

Purpose: Returns the address of the specified instance.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to the module to be searched (NULL to search the current module).
3) The name of the instance (should not include a module specifier).
4) A boolean flag indicating whether imported modules should also be searched: TRUE to search imported modules, otherwise FALSE.

Returns: A generic pointer to the instance, NULL if the instance does not exist.

4.13.9 EnvGetInstanceClass

```
void *EnvGetInstanceClass(environment,instancePtr);

void *environment;
void *instancePtr;
```

Purpose: Determines the class of an instance.

Arguments: 1) A generic pointer to an environment.
2) A generic pointer to an instance.

Returns: A generic pointer to the class of the instance.

4.13.10 EnvGetInstanceName

```
const char *EnvGetInstanceName(environment,instancePtr);

void *environment;
void *instancePtr;
```

Purpose: Determines the name of an instance.

Arguments: 1) A generic pointer to an environment.
2) A generic pointer to an instance.

Returns: The name of the instance.

4.13.11 EnvGetInstancePPForm

```
void EnvGetInstancePPForm(environment,buffer,bufferLength,instancePtr);

void   *environment;
char   *buffer;
size_t  bufferLength;
void   *instancePtr;
```

Purpose: Returns the pretty print representation of an instance in the caller's buffer.

Arguments: 1) A generic pointer to an environment.
2) A pointer to the caller’s character buffer.
3) The maximum number of characters which could be stored in the caller’s buffer (not including space for the terminating null character).
4) A generic pointer to an instance.

Returns: No meaningful return value. The instance pretty print form is stored in the caller's buffer.

4.13.12 EnvGetInstancesChanged

```
int EnvGetInstancesChanged(environment);

void *environment;
```

Purpose: Determines if any changes to instances of user-defined instances have occurred, e.g. instance creations/deletions or slot value changes. If this function returns a non-zero integer, it is the user's responsibility to call SetInstancesChanged(0) to reset the internal flag. Otherwise, this function will continue to return non-zero even when no changes have occurred. This function is primarily used to determine when to update a display tracking instances.

Arguments: A generic pointer to an environment.

Returns: 0 if no changes to instances of user-defined classes have occurred, non-zero otherwise.

4.13.13 EnvGetNextInstance

```
void *EnvGetNextInstance(environment,instancePtr);

void *environment;
void *instancePtr;
```

Purpose: Provides access to the list of instances.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to an instance (or NULL to get the first instance in the list).

Returns: A generic pointer to the first instance in the list of instances if *instancePtr* is NULL, otherwise a pointer to the instance immediately following *instancePtr* in the list. If *instancePtr* is the last instance in the list, then NULL is returned.

4.13.14 EnvGetNextInstanceInClass

```
void *EnvGetNextInstanceInClass(environment,defclassPtr,instancePtr);

void *environment;
void *defclassPtr;
void *instancePtr;
```

Purpose: Provides access to the list of instances for a particular class.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a defclass data structure.
3) A generic pointer to an instance (or NULL to get the first instance in the specified class).

Returns: A generic pointer to the first instance in the list of instances for the specified class if *instancePtr* is NULL, otherwise a pointer to the instance immediately following *instancePtr* in the list. If *instancePtr* is the last instance in the class, then NULL is returned.

4.13.15 EnvGetNextInstanceInClassAndSubclasses

```
void *EnvGetNextInstanceInClassAndSubclasses(environment,defclassPtr,instancePtr,
                                             &iterationData);

void         *environment;
void        **defclassPtr;
void         *instancePtr;
DATA_OBJECT   iterationData;
```

Purpose: Provides access to the list of instances for a particular class and its subclasses.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a generic pointer to a class.
3) A generic pointer to an instance (or NULL to get the first instance in the specified class).
4) A pointer to a DATA_OBJECT in which instance iteration is stored. No initialization of this argument is required and the values stored in this argument are not intended for examination by the calling function.

Returns: A generic pointer to the first instance in the list of instances for the specified class and its subclasses if *instancePtr* is NULL, otherwise a pointer to the instance immediately following *instancePtr* in the list or the next instance in a subclass of the class. If *instancePtr* is the last instance in the class and all its subclasses, then NULL is returned.

As the subclasses of the specified class are iterated through to find instances, the value stored in defclassPtr is updated to indicate the class of the instance returned by this function.

Example

```
#include "clips.h"

int main()
  {
   void *theEnv;
   DATA_OBJECT iterate;
   void *theInstance;
   void *theClass;

   theEnv = CreateEnvironment();

   EnvBuild(theEnv,"(defclass A (is-a USER))");
   EnvBuild(theEnv,"(defclass B (is-a USER))");
   EnvMakeInstance(theEnv,"(a1 of A)");
   EnvMakeInstance(theEnv,"(a2 of A)");
   EnvMakeInstance(theEnv,"(b1 of B)");
   EnvMakeInstance(theEnv,"(b2 of B)");

   theClass = EnvFindDefclass(theEnv,"USER");

   for (theInstance = EnvGetNextInstanceInClassAndSubclasses(theEnv,&theClass,
                                                             NULL,&iterate);
        theInstance != NULL;
        theInstance = EnvGetNextInstanceInClassAndSubclasses(theEnv,&theClass,
                                                             theInstance,&iterate))
     {
      EnvPrintRouter(theEnv,WDISPLAY,EnvGetInstanceName(theEnv,theInstance));
      EnvPrintRouter(theEnv,WDISPLAY,"\n");
     }
  }
```

The output when running this example is:

```
initial-object
a1
a2
b1
b2
```

4.13.16 EnvIncrementInstanceCount

```
void EnvIncrementInstanceCount(environment,instancePtr);

void *environment;
void *instancePtr;
```

Purpose: This function should be called for each external copy of an instance address to let CLIPS know that such an outstanding external reference exists. As long as an instance's count is greater than zero,

CLIPS will not release its memory because there may be outstanding pointers to the instance. However, the instance can still be *functionally* deleted, i.e. the instance will *appear* to no longer be in the system. The instance address always can be safely passed to instance access functions as long as the count for the instance is greater than zero. These functions will recognize when an instance has been functionally deleted.

Arguments:

1) A generic pointer to an environment.
2) A generic pointer to the instance.

Returns:

No meaningful return value.

Example

```
#include "clips.h"

/*===========*/
/* Incorrect */
/*===========*/

void InstanceReferenceExampleIncorrect(
  void *theEnv)
  {
   void *myInstancePtr;

   myInstancePtr = EnvFindInstance(theEnv,NULL,"my-instance",TRUE);

   /*=========================================*/
   /* Instance my-instance could be potentially */
   /* deleted during the run.                 */
   /*=========================================*/

   EnvRun(theEnv,-1L);

   /*=========================================*/
   /* This next function call could dereference */
   /* a dangling pointer and cause a crash.     */
   /*=========================================*/

   EnvDeleteInstance(theEnv,myInstancePtr);
  }

/*=========*/
/* Correct */
/*=========*/

void InstanceReferenceExampleCorrect(
  void *theEnv)
  {
```

```
  void *myInstancePtr;

  myInstancePtr = EnvFindInstance(theEnv,NULL,"my-instance",TRUE);

  /*=====================================================*/
  /* The instance is correctly marked so that a dangling */
  /* pointer cannot be created during the run.           */
  /*=====================================================*/

  EnvIncrementInstanceCount(theEnv,myInstancePtr);
  EnvRun(theEnv,-1L);
  EnvDecrementInstanceCount(theEnv,myInstancePtr);

  /*=========================================================*/
  /* The instance can now be safely deleted using the pointer. */
  /*=========================================================*/

  EnvDeleteInstance(theEnv,myInstancePtr);
 }
```

4.13.17 EnvInstances

```
void EnvInstances(environment,logicalName,modulePtr,className,subclassFlag);

void       *environment;
const char *logicalName;
void       *defmodulePtr;
const char *className;
int         subclassFlag;
```

Purpose: Prints the list of all direct instances of a specified class currently in the system (the C equivalent of the CLIPS **instances** command).

Arguments:

1) A generic pointer to an environment.
2) The logical name to which output is sent.
3) A generic pointer to a defmodule data structure (NULL indicates to list all instances of all classes in all modules—the third and fourth arguments are ignored).
4) The name of the class for which to list instances (NULL indicates to list all instances of all classes in the specified module—the fourth argument is ignored).
5) A flag indicating whether or not to list recursively direct instances of subclasses of the named class in the specified module. 0 indicates no, and any other value indicates yes.

Returns: No meaningful return value.

4.13.18 EnvLoadInstances

```
long EnvLoadInstances(environment,fileName);

void       *environment;
const char *fileName;
```

Purpose: Loads a set of instances into the CLIPS data base (the C equivalent of the CLIPS **load-instances** command).

Arguments:
1) A generic pointer to an environment.
2) A string representing the name of the file.

Returns: Returns the number of instances loaded or -1 if the file could not be accessed.

4.13.19 EnvLoadInstancesFromString

```
long   EnvLoadInstancesFromString(environment,inputString,maximumPosition);

void       *environment;
const char *inputString;
int         maximumPosition;
```

Purpose: Loads a set of instances into the CLIPS data base using a string as the input source (in a manner similar to the CLIPS **load-instances** command).

Arguments:
1) A generic pointer to an environment.
2) A string containing the instance definitions.
3) The maximum number of characters to be read from the string. A value of -1 indicates the entire string.

Returns: Returns the number of instances loaded or -1 if there were problems using the string as an input source.

4.13.20 EnvMakeInstance

```
void *EnvMakeInstance(environment,makeCommand);

void       *environment;
const char *makeCommand;
```

Purpose: Creates and initializes an instance of a user-defined class (the C equivalent of the CLIPS **make-instance** function).

Arguments:
1) A generic pointer to an environment.
2) A string containing a **make-instance** command in the format below:

   (<instance-name> of <class-name> <slot-override>*)
   <slot-override> :== (<slot-name> <constant>*)

Returns: A generic pointer to the new instance, NULL on errors.

Other: This function can trigger garbage collection.

Example

```
#include "clips.h"

int main()
  {
   void *theEnv;
   DATA_OBJECT rv;
   void *theInstance;

   theEnv = CreateEnvironment();

   EnvBuild(theEnv,"(defclass boy (is-a USER) (slot age))");
   EnvMakeInstance(theEnv,"(henry of boy (age 8))");
   EnvEval(theEnv,"(instances)",&rv);
  }
```

Running this code produces the following output:

```
[initial-object] of INITIAL-OBJECT
[henry] of boy
For a total of 2 instances.
```

4.13.21 EnvRestoreInstances

```
long EnvRestoreInstances(environment,fileName);

void       *environment;
const char *fileName;
```

Purpose: Loads a set of instances into the CLIPS data base (the C equivalent of the CLIPS **restore-instances** command).

Arguments:
1) A generic pointer to an environment.
2) A string representing the name of the file.

Returns: Returns the number of instances restored or -1 if the file could not be accessed.

4.13.22 EnvRestoreInstancesFromString

```
long   EnvRestoreInstancesFromString(environment,inputString,maximumPosition);

void       *environment;
const char *inputString;
int         maximumPosition;
```

Purpose: Loads a set of instances into the CLIPS data base using a string as the input source (in a manner similar to the CLIPS **restore-instances** command).

Arguments:
1) A generic pointer to an environment.
2) A string containing the instance definitions.
3) The maximum number of characters to be read from the string. A value of -1 indicates the entire string.

Returns: Returns the number of instances loaded or -1 if there were problems using the string as an input source.

4.13.23 EnvSaveInstances

```
long EnvSaveInstances(environment,fileName,saveCode);

void       *environment;
const char *fileName;
int         saveCode;
```

Purpose: Saves the instances in the system to the specified file (the C equivalent of the CLIPS **save-instances** command).

Arguments:
1) A generic pointer to an environment.
2) A string representing the name of the file.
3) An integer flag indicating whether to save local (current module only) or visible instances. Use either the constant LOCAL_SAVE or VISIBLE_SAVE.

Returns: Returns the number of instances saved.

4.13.24 EnvSend

```
void EnvSend(environment,&instanceInputValue,msg,msgArgs,&outputValue);

void        *environment;
DATA_OBJECT  instanceInputValue;
DATA_OBJECT  outputValue;
const char  *msg;
const char  *msgArgs;
```

Purpose: Message-passing from C Sends a message with the specified arguments to the specified object and stores the result in the caller's buffer (the C equivalent of the CLIPS **send** function).

Arguments:
1) A generic pointer to an environment.
2) A data value holding the object (instance, symbol, float, etc.) which will receive the message.
3) The message.
4) A string containing any *constant* arguments separated by blanks (this argument can be NULL).
5) Caller's buffer for storing the result of the message. See sections 3.2.3 and 3.2.4 for information on getting the value stored in a DATA_OBJECT.

Returns: No meaningful return value.

Other: This function can trigger garbage collection.

Example

```
#include "clips.h"

int main()
  {
   void *theEnv;
   void *myInstancePtr;
   char *cs;
   DATA_OBJECT insdata, rtn;

   theEnv = CreateEnvironment();

   EnvBuild(theEnv,"(defclass MY-CLASS (is-a USER))");

   // Note the use of escape characters to embed quotation marks.
   // (defmessage-handler MY-CLASS my-msg (?x ?y ?z)
   //    (printout t ?x " " ?y " " ?z crlf))

   cs =  "(defmessage-handler MY-CLASS my-msg (?x ?y ?z)"
         "   (printout t ?x \" \" ?y \" \" ?z crlf))";
```

```
  EnvBuild(theEnv,cs);

  myInstancePtr = EnvMakeInstance(theEnv,"(my-instance of MY-CLASS)");
  SetType(insdata,INSTANCE_ADDRESS);
  SetValue(insdata,myInstancePtr);
  EnvSend(theEnv,&insdata,"my-msg","1 abc 3",&rtn);
 }
```

4.13.25 EnvSetInstancesChanged

```
void EnvSetInstancesChanged(environment,changedFlag);

void *environment;
int   changedFlag;
```

Purpose: Sets the internal boolean flag which indicates when changes to instances of user-defined classes have occurred. This function is normally used to reset the flag to zero after GetInstancesChanged() returns non-zero.

Arguments:
1) A generic pointer to an environment.
2) An integer indicating whether changes in instances of user-defined classes have occurred (non-zero) or not (0).

Returns: Nothing useful.

4.13.26 EnvUnmakeInstance

```
int EnvUnmakeInstance(environment,instancePtr);

void *environment;
void *instancePtr;
```

Purpose: This function is equivalent to DeleteInstance except that it uses message-passing instead of directly deleting the instance(s).

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to the instance to be deleted. If the pointer is NULL, all instances in the system are deleted.

Returns: Non-zero if successful, 0 otherwise.

Other: This function can trigger garbage collection.

4.13.27 EnvValidInstanceAddress

```
int EnvValidInstanceAddress(environment,instancePtr);

void *environment;
void *instancePtr;
```

Purpose: Determines if an instance referenced by an address still exists. See the description of IncrementInstanceCount.

Arguments:
1) A generic pointer to an environment.
2) The address of the instance.

Returns: The integer 1 if the instance still exists, 0 otherwise.

4.14 Defmessage-handler Functions

The following function calls are used for manipulating defmessage-handlers.

4.14.1 EnvFindDefmessageHandler

```
unsigned EnvFindDefmessageHandler(environment,defclassPtr,
                                                  handlerName,handlerType);

void       *environment;
void       *defclassPtr,
const char *handlerName;
const char *handlerType;
```

Purpose: Returns an index to the specified message-handler within the list of handlers for a particular class.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to the class to which the handler is attached.
3) The name of the handler.
4) The type of the handler: around, before, primary or after.

Returns: An index to the specified handler if it exists, otherwise 0.

4.14.2 EnvGetDefmessageHandlerList

```
void EnvGetDefmessageHandlerList(environment,defclassPtr,&outputValue,
                                 includeInheritedp);

void        *environment;
void        *defclassPtr;
DATA_OBJECT  outputValue;
int          includeInheritedp;
```

Purpose: Returns the list of currently defined defmessage-handlers for the specified class. This function is the C equivalent of the CLIPS **get-defmessage-handler-list** command).

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to the class (NULL for all classes).
3) A pointer to a DATA_OBJECT in which the function results are stored. See sections 3.2.3 and 3.2.4 for information on examining the value stored in a DATA_OBJECT.
4) An integer flag indicating whether to list inherited handlers (TRUE to list them or FALSE to not list them).

Returns: No meaningful value. The second argument to this function is set to a multifield value containing the list of defmessage-handler constructs for the specified class. The multifield functions described in section 3.2.4 can be used to retrieve the defmessage-handler class, name, and type from the list. Note that the class, name, and type for each defmessage-handler are stored as triplets in the return multifield value.

4.14.3 EnvGetDefmessageHandlerName

```
const char *EnvGetDefmessageHandlerName(environment,defclassPtr,handlerIndex);

void *environment;
void *defclassPtr;
int   handlerIndex;
```

Purpose: Returns the name of a message-handler.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a defclass data structure.
3) The index of a message-handler.

Returns: A string containing the name of the message-handler.

4.14.4 EnvGetDefmessageHandlerPPForm

```
const char *EnvGetDefmessageHandlerPPForm(environment,defclassPtr,handlerIndex);

void *environment;
void *defclassPtr;
int   handlerIndex;
```

Purpose: Returns the pretty print representation of a message-handler.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a defclass data structure.
3) The index of a message-handler.

Returns: A string containing the pretty print representation of the message-handler (or the NULL pointer if no pretty print representation exists).

4.14.5 EnvGetDefmessageHandlerType

```
const char *EnvGetDefmessageHandlerType(environment,defclassPtr,handlerIndex);

void *environment;
void *defclassPtr;
int   handlerIndex;
```

Purpose: Returns the type (around, before, primary or after) of a message-handler.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a defclass data structure.
3) The index of a message-handler.

Returns: A string containing the type of the message-handler.

4.14.6 EnvGetDefmessageHandlerWatch

```
unsigned EnvGetDefmessageHandlerWatch(environment,defclassPtr,handlerIndex);

void *environment;
void *defclassPtr;
int   handlerIndex;
```

Purpose: Indicates whether or not a particular defmessage-handler is being watched.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a defclass data structure and the index of the message-handler.

Returns: An integer; one (1) if the defmessage-handler is being watched, otherwise a zero (0).

4.14.7 EnvGetNextDefmessageHandler

```
unsigned EnvGetNextDefmessageHandler(environment,defclassPtr,handlerIndex);

void *environment;
void *defclassPtr;
int   handlerIndex;
```

Purpose: Provides access to the list of message-handlers.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a defclass data structure.
3) An index to a particular message-handler for the class (or 0 to get the first message-handler).

Returns: An index to the first handler in the list of handlers if *handlerIndex* is 0, otherwise an index to the handler immediately following *handlerIndex* in the list of handlers for the class. If *handlerIndex* is the last handler in the list of handlers for the class, then 0 is returned.

4.14.8 EnvIsDefmessageHandlerDeletable

```
int EnvIsDefmessageHandlerDeletable(environment,defclassPtr,handlerIndex);

void *environment;
void *defclassPtr;
int   handlerIndex;
```

Purpose: Indicates whether or not a particular message-handler can be deleted.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a defclass data structure.
3) The index of a message-handler.

Returns: An integer; zero (0) if the message-handler cannot be deleted, otherwise a one (1).

4.14.9 EnvListDefmessageHandlers

```
void EnvListDefmessageHandlers(environment,logicalName,defclassPtr,
                               includeInheritedp);

void       *environment;
const char *logicalName;
void       *defclassPtr;
int         includeInheritedp;
```

Purpose: Prints the list of message-handlers for the specified class. This function is the C equivalent of the CLIPS **list-defmessage-handlers** command).

Arguments:
1) A generic pointer to an environment.
2) The logical name to which the listing output is sent.
3) A generic pointer to the class (NULL for all classes).
4) An integer flag indicating whether to list inherited handlers (TRUE to list them or FALSE to not list them).

Returns: No meaningful return value.

4.14.10 EnvPreviewSend

```
void EnvPreviewSend(environment,logicalName,defclassPtr,messageName);

void       *environment;
const char *logicalName;
void       *defclassPtr;
const char *messageName;
```

Purpose: Prints a list of all applicable message-handlers for a message sent to an instance of a particular class (the C equivalent of the CLIPS **preview-send** command). Output is sent to the logical name **wdisplay**.

Arguments:
1) A generic pointer to an environment.
2) The logical name to which output is sent.
3) A generic pointer to the class.
4) The message name.

Returns: No meaningful return value.

4.14.11 EnvSetDefmessageHandlerWatch

```
void EnvSetDefmessageHandlerWatch(environment,newState,defclassPtr,
                                  handlerIndex);

void *environment;
int   newState;
void *defclassPtr;
int   handlerIndex;
```

Purpose: Sets the message-handlers watch item for a specific defmessage-handler.

Arguments:
1) A generic pointer to an environment.
2) The new message-handlers watch state.
3) A generic pointer to a defclass data structure.
4) The index of the message-handler.

4.14.12 EnvUndefmessageHandler

```
int EnvUndefmessageHandler(environment,defclassPtr,handlerIndex);

void *environment;
void *defclassPtr;
int   handlerIndex;
```

Purpose: Removes a message-handler from CLIPS (similar but *not* equivalent to the CLIPS **undefmessage-handler** command - see WildDeleteHandler).

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a defclass data structure (NULL to delete all message-handlers in all classes).
3) The index of the message-handler (0 to delete all message-handlers in the class - must be 0 if *defclassPtr* is NULL).

Returns: An integer; zero (0) if the message-handler could not be deleted, otherwise a one (1).

Other: This function can trigger garbage collection.

4.15 Definstances Functions

The following function calls are used for manipulating definstances.

4.15.1 EnvDefinstancesModule

```
const char *EnvDefinstancesModule(environment,theDefinstances);

void *environment;
void *theDefinstances;
```

Purpose: Returns the module in which a definstances is defined (the C equivalent of the CLIPS **definstances-module** command).

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a definstances.

Returns: A string containing the name of the module in which the definstances is defined.

4.15.2 EnvFindDefinstances

```
void *EnvFindDefinstances(environment,definstancesName);

void *environment;
char *definstancesName;
```

Purpose: Returns a generic pointer to a named definstances.

Arguments:
1) A generic pointer to an environment.
2) The name of the definstances to be found.

Returns: A generic pointer to the named definstances if it exists, otherwise NULL.

4.15.3 EnvGetDefinstancesList

```
void EnvGetDefinstancesList(environment,&outputValue,theModule);

void        *environment;
DATA_OBJECT  outputValue;
void        *theModule;
```

Purpose: Returns the list of definstances in the specified module as a multifield value in the returnValue DATA_OBJECT (the C equivalent of the CLIPS **get-definstances-list** function).

Arguments:
1) A generic pointer to an environment.

2) A pointer to a DATA_OBJECT in which the function results are stored. See sections 3.2.3 and 3.2.4 for information on examining the value stored in a DATA_OBJECT.
3) A generic pointer to the module from which the list will be extracted. A NULL pointer indicates that the list is to be extracted from al l modules.

Returns: No meaningful return value.

4.15.4 EnvGetDefinstancesName

```
const char *EnvGetDefinstancesName(environment,definstancesPtr);

void *environment;
void *definstancesPtr;
```

Purpose: Returns the name of a definstances.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a definstances data structure.

Returns: A string containing the name of the definstances.

4.15.5 EnvGetDefinstancesPPForm

```
const char *EnvGetDefinstancesPPForm(environment,definstancesPtr);

void *environment;
void *definstancesPtr;
```

Purpose: Returns the pretty print representation of a definstances.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a definstances data structure.

Returns: A string containing the pretty print representation of the definstances (or the NULL pointer if no pretty print representation exists).

4.15.6 EnvGetNextDefinstances

```
void *EnvGetNextDefinstances(environment,definstancesPtr);

void *environment;
```

```
void *definstancesPtr;
```

Purpose: Provides access to the list of definstances.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a definstances data structure (or NULL to get the first definstances).

Returns: A generic pointer to the first definstances in the list of definstances if *definstancesPtr* is NULL, otherwise a generic pointer to the definstances immediately following *definstancesPtr* in the list of definstances. If *definstancesPtr* is the last definstances in the list of definstances, then NULL is returned.

4.15.7 EnvIsDefinstancesDeletable

```
int EnvIsDefinstancesDeletable(environment,definstancesPtr);

void *environment;
void *definstancesPtr;
```

Purpose: Indicates whether or not a particular class definstances can be deleted.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a definstances data structure.

Returns: An integer; zero (0) if the definstances cannot be deleted, otherwise a one (1).

4.15.8 EnvListDefinstances

```
void EnvListDefinstances(environment,logicalName,theModule);

void *environment;
char *logicalName;
void *theModule;
```

Purpose: Prints the list of definstances (the C equivalent of the CLIPS **list-definstances** command).

Arguments:
1) A generic pointer to an environment.
2) The logical name to which the listing output is sent.

3) A generic pointer to the module containing the definstances to be listed. A NULL pointer indicates that definstances in all modules should be listed.

Returns: No meaningful return value.

4.15.9 EnvUndefinstances

```
int EnvUndefinstances(environment,definstancesPtr);

void *environment;
void *definstancesPtr;
```

Purpose: Removes a definstances from CLIPS (the C equivalent of the CLIPS **undefinstances** command).

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a definstances data structure.

Returns: An integer; zero (0) if the definstances could not be deleted, otherwise a one (1).

Other: This function can trigger garbage collection.

4.16 Defmodule Functions

The following function calls are used for manipulating defmodules.

4.16.1 EnvFindDefmodule

```
void *EnvFindDefmodule(environment,defmoduleName);

void       *environment;
const char *defmoduleName;
```

Purpose: Returns a generic pointer to a named defmodule.

Arguments:
1) A generic pointer to an environment.
2) The name of the defmodule to be found.

Returns: A generic pointer to the named defmodule if it exists, otherwise NULL.

4.16.2 EnvGetCurrentModule

```
void *EnvGetCurrentModule(environment);

void *environment;
```

Purpose: Returns the current module (the C equivalent of the CLIPS **get-current-module** function).

Arguments: A generic pointer to an environment.

Returns: A generic pointer to the generic defmodule data structure that is the current module.

4.16.3 EnvGetDefmoduleList

```
void EnvGetDefmoduleList(environment,&outputValue);

void        *environment;
DATA_OBJECT  outputValue;
```

Purpose: Returns the list of defmodules as a multifield value in the returnValue DATA_OBJECT (the C equivalent of the CLIPS **get-defmodule-list** function).

Arguments:
1) A generic pointer to an environment.
2) A pointer to a DATA_OBJECT in which the function results are stored. See sections 3.2.3 and 3.2.4 for information on examining the value stored in a DATA_OBJECT.

Returns: No meaningful return value.

4.16.4 EnvGetDefmoduleName

```
const char *EnvGetDefmoduleName(environment,defmodulePtr);

void *environment;
void *defmodulePtr;
```

Purpose: Returns the name of a defmodule.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a defmodule data structure.

Returns: A string containing the name of the defmodule.

4.16.5 EnvGetDefmodulePPForm

```
const char *EnvGetDefmodulePPForm(environment,defmodulePtr);

void *environment;
void *defmodulePtr;
```

Purpose: Returns the pretty print representation of a defmodule.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a defmodule data structure.

Returns: A string containing the pretty print representation of the defmodule (or the NULL pointer if no pretty print representation exists).

4.16.6 EnvGetNextDefmodule

```
void *EnvGetNextDefmodule(environment,defmodulePtr);

void *environment;
void *defmodulePtr;
```

Purpose: Provides access to the list of defmodules.

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a defmodule data structure (or NULL to get the first defmodule).

Returns: A generic pointer to the first defmodule in the list of defmodules if *defmodulePtr* is NULL, otherwise a generic pointer to the defmodule immediately following *defmodulePtr* in the list of defmodules. If *defmodulePtr* is the last defmodule in the list of defmodules, then NULL is returned.

4.16.7 EnvListDefmodules

```
void EnvListDefmodules(environment,logicalName);

void       *environment;
const char *logicalName;
```

Purpose: Prints the list of defmodules (the C equivalent of the CLIPS **list-defmodules** command).

Arguments:
1) A generic pointer to an environment.

2) The logical name to which the listing output is sent.

Returns: No meaningful return value.

4.16.8 EnvSetCurrentModule

```
void *EnvSetCurrentModule(environment,defmodulePtr);

void *environment;
void *defmodulePtr;
```

Purpose: Sets the current module to the specified module (the C equivalent of the CLIPS **set-current-module** function).

Arguments:
1) A generic pointer to an environment.
2) A generic pointer to a defmodule data structure.

Returns: A generic pointer to the previous current defmodule data structure.

4.17 Embedded Application Examples

4.17.1 User-Defined Functions

This section lists the steps needed to define and use an embedded CLIPS application. The example given is the same system used in section 3.4, now set up to run as an embedded application.

1) Copy all of the CLIPS source code file to the user directory.

2) Define the user function (TripleNumber) and a new main routine in a new file. These could go in separate files if desired. For this example, they will all be included in a single file.

```
#include "clips.h"

int main()
  {
   void *theEnv;

   theEnv = CreateEnvironment();
   EnvLoad(theEnv,"constructs.clp");
   EnvReset(theEnv);
   EnvRun(theEnv,-1L);
  }

void TripleNumber(
```

```
  void *theEnv,
  DATA_OBJECT_PTR returnValuePtr)
  {
   void      *value;
   long long  longValue;
   double     doubleValue;

   /*=============================================*/
   /* If illegal arguments are passed, return zero. */
   /*=============================================*/

   if (EnvArgCountCheck(theEnv,"triple",EXACTLY,1) == -1)
     {
      SetpType(returnValuePtr,INTEGER);
      SetpValue(returnValuePtr,EnvAddLong(theEnv,0LL));
      return;
     }

   if (! EnvArgTypeCheck(theEnv,"triple",1,INTEGER_OR_FLOAT,returnValuePtr))
     {
      SetpType(returnValuePtr,INTEGER);
      SetpValue(returnValuePtr,EnvAddLong(theEnv,0LL));
      return;
     }

   /*====================*/
   /* Triple the number. */
   /*====================*/

   if (GetpType(returnValuePtr) == INTEGER)
     {
      value = GetpValue(returnValuePtr);
      longValue = 3 * ValueToLong(value);
      SetpValue(returnValuePtr,EnvAddLong(theEnv,longValue));
     }
   else /* the type must be FLOAT */
     {
      value = GetpValue(returnValuePtr);
      doubleValue = 3.0 * ValueToDouble(value);
      SetpValue(returnValuePtr,EnvAddDouble(theEnv,doubleValue));
     }

   return;
  }
```

3) Modify EnvUserFunctions in the CLIPS **userfunctions.c** file.

```
void EnvUserFunctions(
  void *environment)
  {
   extern void TripleNumber(void *,DATA_OBJECT_PTR);

   EnvDefineFunction2(environment,"triple",'u',PTIEF TripleNumber,
```

```
                        "TripleNumber","11n");
  }
```

4) Define constructs which use the new function in a file called **constructs.clp** (or any file; just be sure the call to **Load** loads all necessary constructs prior to execution).

```
(deffacts init-data
   (data 34)
   (data 13.2))

(defrule get-data
   (data ?num)
   =>
   (printout t "Tripling " ?num crlf)
   (assert (new-value (triple ?num))))

(defrule get-new-value
   (new-value ?num)
   =>
   (printout t crlf "Now equal to " ?num crlf))
```

5) Compile all CLIPS files, *except* **main.c**, along with all user files.

6) Link all object code files.

7) Execute new CLIPS executable.

4.17.2 Manipulating Objects and Calling CLIPS Functions

This section lists the steps needed to define and use an embedded CLIPS application. The example illustrates how to call deffunctions and generic functions as well as manipulate objects from C.

1) Copy all of the CLIPS source code file to the user directory.

2) Define a new main routine in a new file.

```
#include <stdio.h>
#include "clips.h"

int main()
  {
   void *theEnv;
   void *c1,*c2,*c3;
   DATA_OBJECT insdata,result;
   char numbuf[20];

   theEnv = CreateEnvironment();
```

```
/*=========================================================*/
/* Load the classes, message-handlers, generic functions */
/* and generic functions necessary for handling complex  */
/* numbers.                                              */
/*=========================================================*/

EnvLoad(theEnv,"complex.clp");

/*=========================================================*/
/* Create two complex numbers. Message-passing is used to  */
/* create the first instance c1, but c2 is created and has */
/* its slots set directly.                                 */
/*=========================================================*/

c1 = EnvMakeInstance(theEnv,"(c1 of COMPLEX (real 1) (imag 10))");
c2 = EnvCreateRawInstance(theEnv,EnvFindDefclass(theEnv,"COMPLEX"),"c2");

result.type = INTEGER;
result.value = EnvAddLong(theEnv,3LL);
EnvDirectPutSlot(theEnv,c2,"real",&result);

result.type = INTEGER;
result.value = EnvAddLong(theEnv,-7LL);
EnvDirectPutSlot(theEnv,c2,"imag",&result);

/*=========================================================*/
/* Call the function '+' which has been overloaded to handle */
/* complex numbers.  The result of the complex addition is   */
/* stored in a new instance of the COMPLEX class.            */
/*=========================================================*/

EnvFunctionCall(theEnv,"+","[c1] [c2]",&result);
c3 = EnvFindInstance(theEnv,NULL,DOToString(result),TRUE);

/*=========================================================*/
/* Print out a summary of the complex addition using the */
/* "print" and "magnitude" messages to get information   */
/* about the three complex numbers.                      */
/*=========================================================*/

EnvPrintRouter(theEnv,STDOUT,"The addition of\n\n");

SetType(insdata,INSTANCE_ADDRESS);
SetValue(insdata,c1);
EnvSend(theEnv,&insdata,"print",NULL,&result);

EnvPrintRouter(theEnv,STDOUT,"\nand\n\n");

SetType(insdata,INSTANCE_ADDRESS);
SetValue(insdata,c2);
EnvSend(theEnv,&insdata,"print",NULL,&result);
```

```
   EnvPrintRouter(theEnv,STDOUT,"\nis\n\n");

   SetType(insdata,INSTANCE_ADDRESS);
   SetValue(insdata,c3);
   EnvSend(theEnv,&insdata,"print",NULL,&result);

   EnvPrintRouter(theEnv,STDOUT,"\nand the resulting magnitude is\n\n");

   SetType(insdata,INSTANCE_ADDRESS);
   SetValue(insdata,c3);
   EnvSend(theEnv,&insdata,"magnitude",NULL,&result);
   sprintf(numbuf,"%lf\n",DOToDouble(result));
   EnvPrintRouter(theEnv,STDOUT,numbuf);
  }
```

3) Define constructs which use the new function in a file called **complex.clp** (or any file; just be sure the call to **EnvLoad** loads all necessary constructs prior to execution).

```
(defclass COMPLEX (is-a USER)
   (role concrete)
   (slot real (create-accessor read-write))
   (slot imag (create-accessor read-write)))

(defmethod + ((?a COMPLEX) (?b COMPLEX))
   (make-instance of COMPLEX
      (real (+ (send ?a get-real) (send ?b get-real)))
      (imag (+ (send ?a get-imag) (send ?b get-imag)))))

(defmessage-handler COMPLEX magnitude ()
   (sqrt (+ (** ?self:real 2) (** ?self:imag 2))))
```

4) Compile all CLIPS files, *except* **main.c**, along with all user files.

5) Link all object code files.

6) Execute new CLIPS executable.

Section 5:
Creating a CLIPS Run-time Program

5.1 Compiling the Constructs

This section describes the procedure for creating a CLIPS run-time module. A run-time program compiles all of the constructs (defrule, deffacts, deftemplate, etc.) into a single executable and reduces the size of the executable image. A run-time program will not run any faster than a program loaded using the **load** or **bload** commands. The **constructs-to-c** command used to generate a run-time program creates files containing the C data structures that would dynamically be allocated if the **load** or **bload** command was used. With the exception of some initialization routines, the **constructs-to-c** command does not generate any executable code. The primary benefits of creating a run-time program are: applications can be delivered as a single executable file; loading constructs as part of an executable is faster than loading them from an text or binary file; the CLIPS portion of the run-time program is smaller because the code needed to parse constructs can be discarded; and less memory is required to represent your program's constructs since memory for them is statically rather than dynamically allocated.

Creating a run-time module can be achieved with the following steps:

1) Start CLIPS and load in all of the constructs that will constitute a run-time module. Call the **constructs-to-c** command using the following syntax:

   (constructs-to-c <file-name> <id> [<target-path> [<max-elements>]])

   where <file-name> is a string or a symbol, <id> is an integer, <target-path> is a string or symbol, and the <max-elements> is an integer. For example, if the construct file loaded was named "expert.clp", the conversion command might be

   (constructs-to-c exp 1)

   This command would store the converted constructs in several output files ("exp1_1.c", "exp1_2.c", ... , "exp7_1.c") and use a module id of 1 for this collection of constructs. The use of the module id will be discussed in greater detail later. Once the conversion is complete, exit CLIPS. For large systems, this output may be *very* large (> 200K). If <target-path> is specified, it is prepended to the name of the file when it is created, allowing target directory to be specified for the generated files. For example, specifying the target path Temp\ on a Unix system would place the generated files in the directory Temp (assuming that it already exists).

It is possible to limit the size of the generated files by using the <max-elements> argument. This argument indicates the maximum number of structures which may be placed in a single array stored in a file. Where possible, if this number is exceeded new files will be created to store additional information. This feature is useful for compilers that may place a limitation on the size of a file that may be compiled.

Note that the .c extension is added by CLIPS. When giving the file name prefix, users should consider the maximum number of characters their system allows in a file name. For example, under MS-DOS, only eight characters are allowed in the file name. For very large systems, it is possible for CLIPS to add up to 5 characters to the file name prefix. Therefore, for system which allow only 8 character file names, the prefix should be no more than 3 characters.

Constraint information associated with constructs is not saved to the C files generated by the **constructs-to-c** command unless dynamic constraint checking is enabled (using the **set-dynamic-constraint-checking** command).

2) Set the RUN_TIME setup flag in the **setup.h** header file to 1 and compile all of the c files just generated.

3) Modify the **main.c** module for embedded operation. Unless the user has other specific uses, the argc and argv arguments to the main function should be eliminated. Also do *not* call the **CommandLoop** or **RerouteStdin** functions which are normally called from the **main** function of a command line version of CLIPS. Do *not* define any functions in **EnvUserFunctions** functions. These functions are not called during initialization. All of the function definitions have already been compiled in the 'C' constructs code. In order for your run-time program to be loaded, a function must be called to initialize the constructs module. This function is defined in the 'C' constructs code, and its name is dependent upon the id used when translating the constructs to 'C' code. The name of the function is **InitCImage_<id>** where <id> is the integer used as the construct module <id>. In the example above, the function name would be **InitCImage_1**. The return value of this function is a pointer to an environment (see section 9) which was created and initialized to contain your run-time program. This initialization steps probably would be followed by any user initialization, then by a reset and run. Finally, when you are finished with a run-time module, you can call **DestroyEnvironment** to remove it. An example **main.c** file would be

```
#include <stdio.h>
#include "clips.h"

main()
  {
   void *theEnv;
   extern void *InitCImage_1();
```

```
  theEnv = InitCImage_1();
      •
      •                          /* Any user Initialization */
      •
  EnvReset(theEnv);
  EnvRun(theEnv,-1);
      •
      •                          /* Any other code */
      •
  DestroyEnvironment(theEnv);
 }
```

4) Recompile all of the CLIPS source code (the RUN_TIME flag should still be 1). This causes several modifications in the CLIPS code. The run-time CLIPS module does not have the capability to load new constructs. Do NOT change any other compiler flags! Because of the time involved in recompiling CLIPS, it may be appropriate to recompile the run-time version of CLIPS into a separate library from the full version of CLIPS.

5) Link all regular CLIPS modules together with any user-defined function modules and the 'C' construct modules. Make sure that any user-defined functions have global scope. *Do not* place the construct modules within a library for the purposes of linking (the regular CLIPS modules, however, can be placed in a library). Some linkers (most notably the VAX VMS linker) will not correctly resolve references to global data that is stored in a module consisting only of global data.

6) The run-time module which includes user constructs is now ready to run.

Note that individual constructs may not be added or removed in a run-time environment. Because of this, the **load** function is not available for use in run-time programs. The clear command will also not remove any constructs (although it will clear facts and instances). Use calls to the InitCImage_... functions to clear the environment and replace it with a new set of constructs. In addition, the **eval** and **build** functions do not work in a run-time environment.

Since new constructs can't be added, a run-time program can't dynamically load a deffacts or definstances construct. To dynamically load facts and/or instances in a run-time program, the CLIPS **load-facts** and **load-instances** functions or the C **LoadFacts** and **LoadInstances** functions should be used in place of deffacts and definstances constructs.

❖ Important Note

Each call to separate **InitCImage** functions creates a unique environment into which the run-time program is loaded. Only the first call to a given **InitCImage** function will create an environment containing the specified run-time program. Subsequent calls have no effect and a value of NULL is returned by the function. Once the **DestroyEnvironment** function has been

called to remove an environment created by an **InitCImage** call, there is no way to reload the run-time program.

Section 6:
Integrating CLIPS with Other Languages and Environments

CLIPS is developed in C and is most easily combined with user functions written in C. However, other languages can be used for user-defined functions, and CLIPS even may be embedded within a program written in another language.

6.1 Introduction

Three basic capabilities are needed for complete language mixing.

- A program in another language may be used as the main program.
- The C access functions to CLIPS can be called from the other language and have parameters passed to them.
- Functions written in the other language can be called by CLIPS and have parameters passed to them.

The integration of CLIPS (and C) with other languages requires an understanding of how each language passes parameters between routines. In general, interface functions will be needed to pass parameters from C to another language and from another language to C. The basic concepts of mixed language parameter passing are the same regardless of the language or machine. However, since every machine and operating system passes parameters differently, specific details (and code) may differ from machine to machine. To improve usability and to minimize the amount of recoding needed for each machine, interface packages can be developed which allow user routines to call the standard CLIPS embedded command functions. The details of passing information *from* external routines to CLIPS generally are handled inside of the interface package. To pass parameters from CLIPS *to* an external routine, users will have to write interface functions.

Section 7:
I/O Router System

The **I/O router** system provided in CLIPS is quite flexible and will allow a wide variety of interfaces to be developed and easily attached to CLIPS. The system is relatively easy to use and is explained fully in sections 7.1 through 7.4. The CLIPS I/O functions for using the router system are described in sections 7.5 and 7.6, and finally, in appendix C, some examples are included which show how I/O routing could be used for simple interfaces.

7.1 Introduction

The problem that originally inspired the idea of I/O routing will be considered as an introduction to I/O routing. Because CLIPS was designed with portability as a major goal, it was not possible to build a sophisticated user interface that would support many of the features found in the interfaces of commercial expert system building tools. A prototype was built of a semi-portable interface for CLIPS using the CURSES screen management package. Many problems were encountered during this effort involving both portability concerns and CLIPS internal features. For example, every statement in the source code which used the C print function, **printf**, for printing to the terminal had to be replaced by the CURSES function, **wprintw**, which would print to a window on the terminal. In addition to changing function call names, different types of I/O had to be directed to different windows. The tracing information was to be sent to one window, the command prompt was to appear in another window, and output from printout statements was to be sent to yet another window.

This prototype effort pointed out two major needs: First, the need for generic I/O functions that would remain the same regardless of whether I/O was directed to a standard terminal interface or to a more complex interface (such as windows); and second, the need to be able to specify different sources and destinations for I/O. I/O routing was designed in CLIPS to handle these needs. The concept of I/O routing will be further explained in the following sections.

7.2 Logical Names

One of the key concepts of I/O routing is the use of **logical names**. An analogy will be useful in explaining this concept. Consider the Acme company which has two computers: computers X and Y. The Acme company stores three data sets on these two computers: a personnel data set, an accounting data set, and a documentation data set. One of the employees, Joe, wishes to update the payroll information in the accounting data set. If the payroll information was located in directory A on computer Y, Joe's command would be

```
update Y:[A]payroll
```

If the data were moved to directory B on computer X, Joe's command would have to be changed to

```
update X:[B]payroll
```

To update the payroll file, Joe must know its location. If the file is moved, Joe must be informed of its new location to be able to update it. From Joe's point of view, he does not care where the file is located physically. He simply wants to be able to specify that he wants the information from the accounting data set. He would rather use a command like

```
update accounting:payroll
```

By using logical names, the information about where the accounting files are located physically can be hidden from Joe while still allowing him to access them. The locations of the files are equated with logical names as shown here.

```
accounting     =  X:[A]
documentation  =  X:[C]
personnel      =  Y:[B]
```

Now, if the files are moved, Joe does not have to be informed of their relocation so long as the logical names are updated. This is the power of using logical names. Joe does not have to be aware of the physical location of the files to access them; he only needs to be aware that accounting is the logical name for the location of the accounting data files. Logical names allow reference to an object without having to understand the details of the implementation of the reference.

In CLIPS, logical names are used to send I/O requests without having to know which device and/or function is handling the request. Consider the message that is printed in CLIPS when rule tracing is turned on and a rule has just fired. A typical message would be

```
FIRE    1 example-rule:  f-0
```

The routine that requests this message be printed should not have to know where the message is being sent. Different routines are required to print this message to a standard terminal, a window interface, or a printer. The tracing routine should be able to send this message to a logical name (for example, **trace-out**) and should not have to know if the device to which the message is being sent is a terminal or a printer. The logical name **trace-out** allows tracing information to be sent simply to "the place where tracing information is displayed." In short, logical names allow I/O requests to be sent to specific locations without having to specify the details of how the I/O request is to be handled.

Many functions in CLIPS make use of logical names. Both the **printout** and **format** functions require a logical name as their first argument. The **read** function can take a logical name as an optional argument. The **open** function causes the association of a logical name with a file, and the **close** function removes this association.

Several logical names are predefined by CLIPS and are used extensively throughout the system code. These are

Name	Description
stdin	The default for all user inputs. The **read** and **readline** functions read from **stdin** if **t** is specified as the logical name.
stdout	The default for all user outputs. The **format** and **printout** functions send output to **stdout** if **t** is specified as the logical name.
wprompt	The CLIPS prompt is sent to this logical name.
wdialog	All informational messages are sent to this logical name.
wdisplay	Requests to display CLIPS information, such as facts or rules, are sent to this logical name.
werror	All error messages are sent to this logical name.
wwarning	All warning messages are sent to this logical name.
wtrace	All watch information is sent to this logical name (with the exception of compilations which is sent to wdialog).

Within CLIPS code, these predefined logical names should be specified in lower case (and typically the only one you'll use is **t** and depending upon which function you're using this will be mapped to either **stdin** or **stdout**). Within C code, these logical names can be specified using constants that have been defined in upper case: STDIN, STDOUT, WPROMPT, WDIALOG, WERROR, WWARNING, and WTRACE.

7.3 Routers

The use of logical names solves two problems. Logical names make it easy to create generic I/O functions, and they allow the specification of different sources and destinations for I/O. The use of logical names allows CLIPS to ignore the specifics of an I/O request. However, such requests must still be specified at some level. I/O routers are provided to handle the specific details of a request.

A router consists of three components. The first component is a function which can determine whether the router can handle an I/O request for a given logical name. The router which recognizes I/O requests that are to be sent to the serial port may not recognize the same logical names as that which recognizes I/O requests that are to be sent to the terminal. On the other hand, two routers may recognize the same logical names. A router that keeps a log of a CLIPS session (a dribble file) may recognize the same logical names as that which handles I/O requests for the terminal.

The second component of a router is its priority. When CLIPS receives an I/O request, it begins to query each router to discover whether it can handle an I/O request. Routers with high priorities are queried before routers with low priorities. Priorities are very important when dealing with one or more routers that can each process the same I/O request. This is particularly true when a router is going to redefine the standard user interface. The router associated with the standard interface will handle the same I/O requests as the new router; but, if the new router is given a higher priority, the standard router will never receive any I/O requests. The new router will "intercept" all of the I/O requests. Priorities will be discussed in more detail in the next section.

The third component of a router consists of the functions which actually handle an I/O request. These include functions for printing strings, getting a character from an input buffer, returning a character to an input buffer, and a function to clean up (e.g., close files, remove windows) when CLIPS is exited.

7.4 Router Priorities

Each I/O router has a priority. Priority determines which routers are queried first when determining the router that will handle an I/O request. Routers with high priorities are queried before routers with low priorities. Priorities are assigned as integer values (the higher the integer, the higher the priority). Priorities are important because more than one router can handle an I/O request for a single logical name, and they enable the user to define a custom interface for CLIPS. For example, the user could build a custom router which handles all logical names normally handled by the default router associated with the standard interface. The user adds the custom router with a priority higher than the priority of the router for the standard interface. The custom router will then intercept all I/O requests intended for the standard interface and specially process those requests to the custom interface.

Once the router system sends an I/O request out to a router, it considers the request satisfied. If a router is going to share an I/O request (i.e., process it) then allow other routers to process the request also, that router must deactivate itself and call **PrintRouter** again. These types of routers should use a priority of either 30 or 40. An example is given in appendix C.2.

Priority	Router Description
50	Any router that uses “unique” logical names and does not want to share I/O with catch-all routers.
40	Any router that wants to grab standard I/O and is willing to share it with other routers. A dribble file is a good example of this type of router. The dribble file router needs to grab all output that normally would go to the terminal so it can be placed in the dribble file, but this same output also needs to be sent to the router which displays output on the terminal.
30	Any router that uses “unique” logical names and is willing to share I/O with catch-all routers.
20	Any router that wants to grab standard logical names and is not willing to share them with other routers.
10	This priority is used by a router which redefines the default user interface I/O router. Only one router should use this priority.
0	This priority is used by the default router for handling standard and file logical names. Other routers should not use this priority.

7.5 Internal I/O Functions

The following functions are called internally by CLIPS. These functions search the list of active routers and determine which router should handle an I/O request. Some routers may wish to deactivate themselves and call one of these functions to allow the next router to process an I/O request. Prototypes for these functions can be included by using the **clips.h** header file or the **router.h** header file.

7.5.1 EnvExitRouter

```
void EnvExitRouter(environment,exitCode);

void *environment;
int   exitCode;
```

Purpose: The function **EnvExitRouter** calls the exit function associated with each active router before exiting CLIPS.

Arguments: 1) A generic pointer to an environment.

2) The *exitCode* argument corresponds to the value that normally would be sent to the system **exit** function. Consult a C system manual for more details on the meaning of this argument.

Returns: No meaningful return value.

Info: The function **ExitRouter** calls the system function **exit** with the argument num after calling all exit functions associated with I/O routers.

7.5.2 EnvGetcRouter

```
int EnvGetcRouter(environment,logicalName);

void       *environment;
const char *logicalName;
```

Purpose: The function **EnvGetcRouter** queries all active routers until it finds a router that recognizes the logical name associated with this I/O request to get a character. It then calls the get character function associated with that router.

Arguments:
1) A generic pointer to an environment.
2) The logical name associated with the get character I/O request.

Returns: An integer; the ASCII code of the character.

Info: This function should be used by any user-defined function in place of **getc** to ensure that character input from the function can be received from a custom interface. On machines which default to unbuffered I/O, user code should be prepared to handle special characters like the backspace.

7.5.3 EnvPrintRouter

```
int EnvPrintRouter(environment,logicalName,str);

void       *environment;
const char *logicalName;
const char *str;
```

Purpose: The function **EnvPrintRouter** queries all active routers until it finds a router that recognizes the logical name associated with this

I/O request to print a string. It then calls the print function associated with that router.

Arguments:

1) A generic pointer to an environment.
2) The logical name associated with the location at which the string is to be printed.
3) The string that is to be printed.

Returns: Returns a non-zero value if the logical name is recognized, otherwise it returns zero.

Info: This function should be used by any user-defined function in place of **printf** to ensure that output from the function can be sent to a custom interface.

7.5.4 EnvUngetcRouter

```
int EnvUngetcRouter(environment,ch,logicalName);

void       *environment;
int         ch;
const char *logicalName;
```

Purpose: The function **EnvUngetcRouter** queries all active routers until it finds a router that recognizes the logical name associated with this I/O request. It then calls the ungetc function associated with that router.

Arguments:

1) A generic pointer to an environment.
2) The ASCII code of the character to be returned.
3) The logical name associated with the ungetc character I/O request.

Returns: Returns *ch* if successful, otherwise -1.

Info: This function should be used by any user-defined function in place of **UngetcRouter** to ensure that character input from the function can be received from a custom interface. As with **GetcRouter**, user code should be prepared to handle special characters like the backspace on machines with unbuffered I/O.

7.6 Router Handling Functions

The following functions are used for creating, deleting, and handling I/O routers. They are intended for use within user-defined functions. Prototypes for these functions can be included by using the **clips.h** header file or the **router.h** header file.

7.6.1 EnvActivateRouter

```
int EnvActivateRouter(environment,routerName);

void       *environment;
const char *routerName;
```

Purpose: The function **EnvActivateRouter** activates an existing I/O router. This router will be queried to see if it can handle an I/O request. Newly created routers do not have to be activated.

Arguments:
1) A generic pointer to an environment.
2) The name of the I/O router to be activated.

Returns: Returns a non-zero value if the logical name is recognized, otherwise it returns zero.

7.6.2 EnvAddRouter

```
int EnvAddRouter(environment,routerName,priority,queryFunction,printFunction,
                 getcFunction,ungetcFunction,exitFunction);

void *environment;
char *routerName;
int priority;
int (*queryFunction)(environment,logicalName);
int (*printFunction)(environment,logicalName,str);
int (*getcFunction)(environment,logicalName);
int (*ungetcFunction)(environment,ch,logicalName);
int (*exitFunction)(environment,exitCode);

const char *logicalName;
const char *str;
int   ch, exitCode;
```

Purpose: The function **EnvAddRouter** adds a new I/O router to the list of I/O routers.

Arguments:
1) A generic pointer to an environment.

2) The name of the I/O router. This name is used to reference the router by the other I/O router handling functions.
3) The priority of the I/O router. I/O routers are queried in descending order of priorities.
4) A pointer to the query function associated with this router. This query function should accept two arguments: a generic pointer to an environment and a logical name. The return value should be either TRUE (1) or FALSE (0) depending upon whether the router recognizes the logical name.
5) A pointer to the print function associated with this router. This print function should accept three arguments: a generic pointer to an environment, a logical name, and a character string. The return value of the print function is not meaningful.
6) A pointer to the get character function associated with this router. The get character function should accept two arguments: a generic pointer to an environment and a logical name. The return value of the get character function should be an integer which represents the character or end of file (EOF) read from the source represented by logical name.
7) A pointer to the ungetc character function associated with this router. The ungetc character function accepts three arguments: a generic pointer to an environment, a logical name, and a character. The return value of the unget character function should be an integer which represents the character which was passed to it as an argument if the ungetc is successful or end of file (EOF) is the ungetc is not successful.
8) A pointer to the exit function associated with this router. The exit function should accept a two arguments: a generic pointer to an environment and the exit code represented by num.

Returns: Returns a zero value if the router could not be added, otherwise a non-zero value is returned.

Info: I/O routers are active upon being created. See the examples in appendix A for further information on how to use this function.

7.6.3 EnvDeactivateRouter

```
int EnvDeactivateRouter(environment,routerName);

void       *environment
const char *routerName;
```

Purpose: The function **EnvDeactivateRouter** deactivates an existing I/O router. This router will not be queried to see if it can handle an I/O request. The syntax of the **EnvDeactivateRouter** function is as follows.

Arguments:
1) A generic pointer to an environment.
2) The name of the I/O router to be deactivated.

Returns: Returns a non-zero value if the logical name is recognized, otherwise it returns zero.

7.6.4 EnvDeleteRouter

```
int DeleteRouter(environment,routerName);

void       *environment;
const char *routerName;
```

Purpose: The function **EnvDeleteRouter** removes an existing I/O router from the list of I/O routers.

Arguments:
1) A generic pointer to an environment.
2) The name of the I/O router to be deleted.

Returns: Returns a non-zero value if the logical name is recognized, otherwise it returns zero.

7.6.5 EnvAddRouterWithContext

```
int EnvAddRouterWithContext(environment,routerName,priority,queryFunction,
                            printFunction, getcFunction,ungetcFunction,
                            exitFunction,context);

void *environment;
char *routerName;
int priority;
int (*queryFunction)(environment,logicalName);
int (*printFunction)(environment,logicalName,str);
int (*getcFunction)(environment,logicalName);
int (*ungetcFunction)(environment,ch,logicalName);
int (*exitFunction)(environment,exitCode);
void *context;

const char *logicalName;
const char *str;
int   ch, exitCode;
```

Purpose: The function **EnvAddRouterWithContext** adds a new I/O router to the list of I/O routers with a user defined context.

Arguments:

1) A generic pointer to an environment.
2) The name of the I/O router. This name is used to reference the router by the other I/O router handling functions.
3) The priority of the I/O router. I/O routers are queried in descending order of priorities.
4) A pointer to the query function associated with this router. This query function should accept two arguments: a generic pointer to an environment and a logical name. The return value should be either TRUE (1) or FALSE (0) depending upon whether the router recognizes the logical name.
5) A pointer to the print function associated with this router. This print function should accept three arguments: a generic pointer to an environment, a logical name, and a character string. The return value of the print function is not meaningful.
6) A pointer to the get character function associated with this router. The get character function should accept two arguments: a generic pointer to an environment and a logical name. The return value of the get character function should be an integer which represents the character or end of file (EOF) read from the source represented by logical name.
7) A pointer to the ungetc character function associated with this router. The ungetc character function accepts three arguments: a generic pointer to an environment, a logical name, and a character. The return value of the unget character function should be an integer which represents the character which was passed to it as an argument if the ungetc is successful or end of file (EOF) is the ungetc is not successful.
8) A pointer to the exit function associated with this router. The exit function should accept a two arguments: a generic pointer to an environment and the exit code represented by num.
9) A generic pointer to data that will be associated with this router when its query, getc, ungetc, print, or exit functions are invoked.

Returns: Returns a zero value if the router could not be added, otherwise a non-zero value is returned.

Info: I/O routers are active upon being created. See the examples in appendix A for further information on how to use this function.

Notes: Use this function to create multiple instances of the same router that needs context specific information.

7.6.6 GetEnvironmentRouterContext

```
void *GetEnvironmentRouterContext(environment);

void       *environment;
const char *routerName;
```

Purpose: The function **GetEnvironmentRouterContext** returns the context associated with the currently executing router.

Arguments: A generic pointer to an environment.

Returns: Returns the void pointer context that was associated with the currently executing router when it was defined using **EnvAddRouterWithContext**.

Section 8:
Memory Management

Efficient use of memory is a very important aspect of an expert system tool. Expert systems are highly memory intensive and require comparatively large amounts of memory. To optimize both storage and processing speed, CLIPS does much of its own memory management. Section 8.1 describes the basic memory management scheme used in CLIPS. Section 8.2 describes some functions that may be used to monitor/ control memory usage.

8.1 How CLIPS Uses Memory

The CLIPS internal data structures used to represent constructs and other data entities require the allocation of dynamic memory to create and execute. Memory can also be released as these data structures are no longer needed and are removed. All requests, either to allocate memory or to free memory, are routed through the CLIPS memory management functions. These functions request memory from the operating system and store previously used memory for reuse. By providing its own memory management, CLIPS is able to reduce the number of **malloc** calls to the operating system. This is very important since **malloc** calls are handled differently on each machine, and some implementations of **malloc** are very inefficient.

When new memory is needed by any CLIPS function, CLIPS first checks its own data buffers for a pointer to a free structure of the type requested. If one is found, the stored pointer is returned. Otherwise, a call is made to **malloc** for the proper amount of data and a new pointer is returned.

When a data structure is no longer needed, CLIPS saves the pointer to that memory against the next request for a structure of that type. Memory actually is released to the operating system only under limited circumstances. If a **malloc** call in a CLIPS function returns NULL, *all* free memory internally stored by CLIPS is released to the operating system and the **malloc** call is tried again. If the malloc call can still not be satisfied, the OutOfMemory function will be called. The default implementation of this function will print an error message and terminate execution of CLIPS. Users also may also force memory to be released to the operating system using the **EnvReleaseMem** function.

CLIPS uses the generic C function **malloc** to request memory. The generic CLIPS memory allocation and deallocation functions are stored in the **memalloc.c** file and are called **genalloc** and **genfree**. The call to **malloc** and **free** in these functions could be replaced if alternate memory management routines are desired.

Extensive effort has gone into making CLIPS garbage free. Theoretically, if an application can fit into the available memory on a machine, CLIPS should be able to run it forever. Of course, user-defined functions that use dynamic memory may affect this.

8.2 Standard Memory Functions

CLIPS currently provides a few functions that can be used to monitor and control memory usage. Prototypes for these functions can be included by using the **clips.h** header file or the **memalloc.h** header file.

8.2.1 EnvGetConserveMemory

```
int EnvGetConserveMemory(environment);

void *environment;
```

Purpose: Returns the current value of the conserve memory behavior.

Arguments: A generic pointer to an environment.

Returns: An integer; FALSE (0) if the behavior is disabled and TRUE (1) if the behavior is enabled.

8.2.2 EnvMemRequests

```
long int EnvMemRequests(environment);

void *environment;
```

Purpose: The function **EnvMemRequests** will return the number of times CLIPS has requested memory from the operating system (the C equivalent of the CLIPS **mem-requests** command).

Arguments: A generic pointer to an environment.

Returns: A long integer representing the number of requests CLIPS has made.

Other: When used in conjunction with **EnvMemoryUsed**, the user can estimate the number of bytes CLIPS requests per call to **malloc**.

8.2.3 EnvMemUsed

```
long int EnvMemUsed(environment);

void *environment;
```

Purpose: The function **EnvMemUsed** will return the number of bytes CLIPS has currently in use or has held for later use (the C equivalent of the CLIPS **mem-used** command).

Arguments: None.

Returns: A long integer representing the number of bytes requested.

Other: The number of bytes used does not include any overhead for memory management or data creation. It does include all free memory being held by CLIPS for later use; therefore, it is not a completely accurate measure of the amount of memory actually used to store or process information. It is used primarily as a minimum indication.

8.2.4 EnvReleaseMem

```
long int EnvReleaseMem(environment, howMuch);

void     *environment;
long int  howMuch;
```

Purpose: The function **EnvReleaseMem** will cause all free memory, or a specified amount, being held by CLIPS to be returned to the operating system (the C equivalent of the CLIPS **release-mem** command).

Arguments:
1) A generic pointer to an environment.
2) The number of bytes to be released. If this argument is -1, all memory will be released; otherwise, the specified number of bytes will be released.

Returns: A long integer representing the actual amount of memory freed to the operating system.

Other: This function can be useful if a user-defined function requires memory but cannot get any from a **malloc** call. However, it should be used carefully. Excessive calls to **EnvReleaseMemory** will

cause CLIPS to call **malloc** more often, which can reduce the performance of CLIPS.

8.2.5 EnvSetConserveMemory

```
int   EnvSetConserveMemory(environment,value);

void *environment;
int   value;
```

Purpose: The function **EnvSetConserveMemory** allows a user to turn on or off the saving of pretty print information. Normally, this information is saved. If constructs are never going to be pretty printed or saved, a significant amount of memory can be saved by not keeping the pretty print representation.

Arguments:
1) A generic pointer to an environment.
2) A boolean value: FALSE (0) to keep pretty print information for newly loaded constructs and TRUE (1) to not keep this information for newly loaded constructs.

Returns: Returns the old value for the behavior.

Other: This function can save considerable memory space. It should be turned on before loading any constructs. It can be turned on or off as many times as desired. Constructs loaded while this is turned off can be displayed only by reloading the construct, even if the option is turned on subsequently.

8.2.6 EnvSetOutOfMemoryFunction

```
int (*EnvSetOutOfMemoryFunction(environment,outOfMemoryFunction))();

int (*outOfMemoryFunction)(environment,size);

void *environment;
size_t size;
```

Purpose: Allows the user to specify a function to be called when CLIPS cannot satisfy a memory request.

Arguments:
1) A generic pointer to an environment.
2) A pointer to the function to be called when CLIPS cannot satisfy a memory request. This function is passed the size of the

memory request which could not be satisfied and a pointer to the environment. It should return a non-zero value if CLIPS should not attempt to allocate the memory again (and exit because of lack of available memory) or a zero value if CLIPS should attempt to allocate memory again.

Returns: Returns a pointer to the previously called out of memory function.

Other: Because the out of memory function can be called repeatedly for a single memory request, any user-defined out of memory function should return zero only if it has released memory.

Section 9:
Environments

CLIPS provides the ability to create multiple environments into which programs can be loaded and run. Each environment maintains its own set of data structures and can be run independently of the other environments. In many cases, the program's main function will create a single environment to be used as the argument for all embedded API calls. In other cases, such as creating shared libraries or DLLs, new instances of environments will be created as they are needed.

9.1 Creating and Destroying Environments

Environments are created using the **CreateEnvironment** function. The return value of the **CreateEnvironment** function is an anonymous (void *) pointer to an **environmentData** data structure. This pointer should be used for the embedded API function calls require an environment pointer argument.

If you have integrated code with CLIPS and use multiple concurrent environments, any functions or extensions which use global data should allocate this data for each environment by using the **AllocateEnvironmentData** function, otherwise one environment may overwrite the data used by another environment.

Once you are done with an environment, it can be deleted with the **DestroyEnvironment** function call. This will deallocate all memory associated with that environment.

The following is an of example of a main program which makes use of multiple environments:

```
#include "clips.h"

int main()
  {
   void *theEnv1, *theEnv2;

   theEnv1 = CreateEnvironment();
   theEnv2 = CreateEnvironment();

   EnvLoad(theEnv1,"program1.clp");
   EnvLoad(theEnv2,"program2.clp");

   EnvReset(theEnv1);
   EnvReset(theEnv2);

   EnvRun(theEnv1,-1);
```

```
   EnvRun(theEnv2,-1);

   DestroyEnvironment(theEnv1);
   DestroyEnvironment(theEnv2);
  }
```

9.2 Standard Environment Functions

The following functions are used to create and manipulate environments. Prototypes for these functions can be included by using the **clips.h** header file or the **envrnmnt.h** header file.

9.2.1 AddEnvironmentCleanupFunction

```
int AddEnvironmentCleanupFunction(theEnv,theName,theFunction,priority);
struct environmentData *theEnv;
char *theName;
void (*)(void *theFunction);
int priority;

void theFunction(void *);
```

Purpose: Adds a cleanup function that is called when an environment is destroyed.

Arguments:

1) A generic pointer to an environment data structure.
2) The name associated with the environment cleanup function.
3) A pointer to the environment cleanup function which is to be called when the environment is deleted. When called, the function is passed a generic pointer to the environment being destroyed.
4) The priority of the environment cleanup function which determines the order in which cleanup functions are called (higher priority items are called first). The values -2000 to 2000 are reserved for CLIPS system defined run items and should not be used for user defined run items.

Returns: Boolean value. TRUE if the cleanup function was successfully added, otherwise FALSE.

Other: Environment cleanup functions created using this function are called after all the cleanup functions associated with environment data created using **AllocateEnvironmentData** have been called.

9.2.2 AllocateEnvironmentData

```
int AllocateEnvironmentData(theEnv,position,size,cleanupFunction);
void *theEnv;
unsigned int position;
unsigned long size;
void (*)(void *cleanupFunction);

void cleanupFunction(void *);
```

Purpose: Allocates environment specific data of the specified size.

Arguments:

1) A generic pointer to an environment data structure.
2) The integer position index used to reference the data. The constant USER_ENVIRONMENT_DATA indicates the lowest index available for user defined data.
3) The amount of environment data that needs to be allocated.
4) A pointer to a cleanup function that is called when the environment is destroyed. When called, the function is passed a generic pointer to the environment being destroyed. CLIPS automatically handles the allocation and deallocation of the base environment data created by this function (the amount of data specified by the *size* argument). You do not need to supply a cleanup function for this purpose and can supply NULL as this argument. If your base environment data contains pointers to memory that you allocate, then you need to either supply a cleanup function as this argument or add a cleanup function using **AddEnvironmentCleanupFunction**.

Returns: Boolean value. TRUE if the environment data was successfully allocated, otherwise FALSE.

Other: Environment cleanup functions specified using this function are called in ascending order of the position indices. If the deallocation of your environment data has order dependencies, you can either assign the position indices appropriately to achieve the proper order or you can use the **AddEnvironmentCleanupFunction** function to more explicitly specify the order in which your environment data must be deallocated.

9.2.3 CreateEnvironment

```
void *CreateEnvironment();
```

Purpose: Creates an environment and initializes it.

Arguments: None.

Returns: A generic pointer to an environment data structure. NULL is returned in the event of an error.

9.2.4 DestroyEnvironment

```
int DestroyEnvironment(theEnv);
void *theEnv;
```

Purpose: Destroys the specified environment deallocating all memory associated with it.

Arguments: A generic pointer to an environment data structure.

Returns: Boolean value. TRUE if the environment was successfully destroyed, otherwise FALSE.

Other: You should not call this function to destroy an an environment that is currently executing.

9.2.5 GetEnvironmentData

```
void *GetEnvironmentData(theEnv,position);
void *theEnv;
unsigned int position;
```

Purpose: Returns a generic pointer to the environment data associated with the specified position index.

Arguments:
1) A generic pointer to an environment data structure.
2) An unsigned integer; the position index of the desired environment data.

Returns: A generic pointer; the environment data associated with the specified position index.

9.3 Allocating Environment Data

If your user-defined functions (or other extensions) make use of global data that could differ for each environment, you should allocate this data with the **AllocateEnvironmentData** function. A

call to this function has four arguments. The first is a generic pointer to the environment to which the data is being added.

The second argument is the integer position index. This is the value that you will pass in to the **GetEnvironmentData** function to retrieve the allocated environment data. This position index must be unique and if you attempt to use an index that has already been allocated, then the call to **AllocateEnvironmentData** will fail returning FALSE. To avoid collisions with environment positions predefined by CLIPS, use the macro constant USER_ENVIRONMENT_DATA as the base index for any position indices you define. This constant will always be greater than the largest predefined position index used by CLIPS. The maximum number of environment position indices is specified by the macro constant MAXIMUM_ENVIRONMENT_POSITIONS found in the envrnmnt.h header file. A call to **AllocateEnvironmentData** will fail if the position index is greater than or equal this value. If this happens, you can simply increase the value of this macro constant to provide more environment positions.

The third argument is an integer indicating the size of the environment data that needs to be allocated. Typically you'll define a struct containing the various values you want stored in the environment data and use the sizeof operator to pass in the size of the struct to the function. When an environment is created directly using **CreateEnvironment**, CLIPS automatically allocates memory of the size specified, initializes the memory to contain all zeroes, and stores the memory in the environment position associated with position index. When the environment is destroyed using **DestroyEnvironment**, CLIPS automatically deallocates the memory originally allocated for each environment data position. If the environment data contains pointers to memory that you allocate, it is your responsibility to deallocate this memory. You can do this by either specifying a cleanup function as the fourth argument in your **AllocateEnvironmentData** call or by adding a cleanup function using the **AddEnvironmentCleanupFunction** function.

The fourth argument is a pointer to a cleanup function. If this argument is not NULL, then the cleanup function associated with this environment position is called whenever an environment is deallocated using the **DestroyEnvironment** function. The cleanup functions are called in ascending order of the position indices.

As an example of allocating environment data, we'll look at a **get-index** function that returns an integer index starting with one and increasing by one each time it is called. For example:

```
CLIPS> (get-index)
1
CLIPS> (get-index)
2
CLIPS> (get-index)
3
CLIPS>
```

Each environment will need global data to store the current value of the index. The C source code that implements the environment data first needs to specify the position index and specify a data structure for storing the data:

```
#define INDEX_DATA USER_ENVIRONMENT_DATA + 0

struct indexData
  {
   long index;
  };

#define IndexData(theEnv) \
   ((struct indexData *) GetEnvironmentData(theEnv,INDEX_DATA))
```

First, the position index GET_INDEX_DATA is defined as USER_ENVIRONMENT_DATA with an offset of zero. If you were to define additional environment data, the offset would be increased each time by one to get to the next available position. Next, the *indexData* struct is defined. This struct contains a single member, *index*, which will use to store the next value returned by the **get-index** function. Finally, the IndexData macro is defined which merely provides a convenient mechanism for access to the environment data.

The next step in the C source code is to add the initialization code to the **EnvUserFunctions** function:

```
void EnvUserFunctions(
  void *theEnv)
  {
   if (! AllocateEnvironmentData(theEnv,INDEX_DATA,
                                 sizeof(struct indexData),NULL))
     {
      printf("Error allocating environment data for INDEX_DATA\n");
      exit(EXIT_FAILURE);
     }

   IndexData(theEnv)->index = 1;

   EnvDefineFunction2(theEnv,"get-index",'l',PTIEF GetIndex, "GetIndex",
                      "00");
  }
```

First, the call to **AllocateEnvironmentData** is made. If this fails, then an error message is printed and a call to **exit** is made to terminate the program. Otherwise, the *index* member of the environment data is initialized to one. If a starting value of zero was desired, it would not be necessary to perform any initialization since the value of *index* is automatically initialized to zero when the environment data is initialized. Finally, **EnvDefineFunction2** is called to register the **get-index** function.

The last piece of the C source code is the **GetIndex** C function which implements the **get-index** function:

```
long GetIndex(
  void *theEnv)
  {
   if (EnvArgCountCheck(theEnv,"get-index",EXACTLY,0) == -1)
     { return(0); }

   return(IndexData(theEnv)->index++);
  }
```

This function is fairly straightforward. A generic pointer to the current environment is passed to the function since it was registered using **EnvDefineFunction2**. First a check for the correct number of arguments is made and then a call to the **IndexData** macro is made to retrieve the *index* member of struct which is the return value. Use of the ++ operator increments the current value of the *index* member before the function returns.

Section 10:
CLIPS Java Native Interface

This section describes the CLIPS Java Native Inteface (CLIPSJNI) and the examples demonstrating the integration of CLIPS with a Swing interface. The examples have been tested using Java version 1.8.0_31 running on Mac OS X 10.10.2 and Windows 7.

10.1 CLIPSJNI Directory Structure

In order to use CLIPSJNI, you must obtain the source code by downloading the CLIPSJNI zip file from the Files page on the CLIPS SourceForge web page (see appendix A for the SourceForge URL). Once downloaded, you must then extract the contents of the file by right clicking on it and selecting the "Extract All…" menu item.

When unzipped the CLIPSJNI project file contains the following directory structure:

```
CLIPSJNI
    examples
        AnimalDemo
            resources
        AutoDemo
            resources
        SudokuDemo
            resources
        WineDemo
            resources
    java-src
        CLIPSJNI
    library-src
```

If you are using the CLIPSJNI on Mac OS X, then the native CLIPS library is already contained in the top level CLIPSJNI directory.

On Windows, it is necessary to verify that the correct DLL is installed. By default, the DLL for 64-bit Windows is used as the CLIPSJNI.dll file in the top level of the CLIPSJNI directory. If running CLIPSJNI with 32-bit Windows, delete the existing CLIPSJNI.dll file, then make a copy of the CLIPSJNI32.dll file and rename it to CLIPSJNI.dll.

On other systems, you must created a native library using the source files contained in the library-src directory before you can utilize the CLIPSJNI.

The CLIPSJNI jar file is also contained in the top level CLIPSJNI directory. The source files used to create the jar file are contained in the java-src directory.

10.2 Running CLIPSJNI in Command Line Mode

You can invoke the command line mode of CLIPS through CLIPSJNI to interactively enter commands while running within a Java environment.

On Windows 7, launch the Command Prompt application (select Start > All Programs > Accessories > Command Prompt). Set the directory to the CLIPSJNI top level directory (using the cd command).

On Mac OS X, launch the Terminal application (located in the Applications/Utilities directory). Set the directory to the CLIPSJNI top level directory (using the cd command).

From the CLIPSJNI directory, enter the following command:

```
java -cp CLIPSJNI.jar CLIPSJNI.Environment
```

The CLIPS banner and command prompt should appear:

```
         CLIPS (6.30 3/17/15)
CLIPS>
```

10.3 Running the Swing Demo Programs

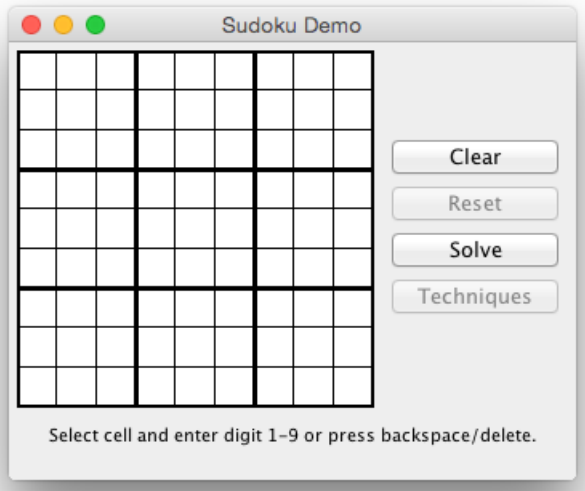
The Swing CLIPSJNI demonstration programs can be run on Windows 7 or Mac OS X using the precompiled native libraries in the CLIPSJNI top level directory. On other systems a native library must first be created before the programs can be run.

10.3.1 Running the Demo Programs on Mac OS X

Launch the Terminal application (located in the Applications/Utilities directory). Set the directory to the CLIPSJNI/examples/SudokuDemo directory (using the cd command). To run the Sudoku demo, enter the following command:

```
java -cp .:../../CLIPSJNI.jar -Djava.library.path=../.. SudokuDemo
```

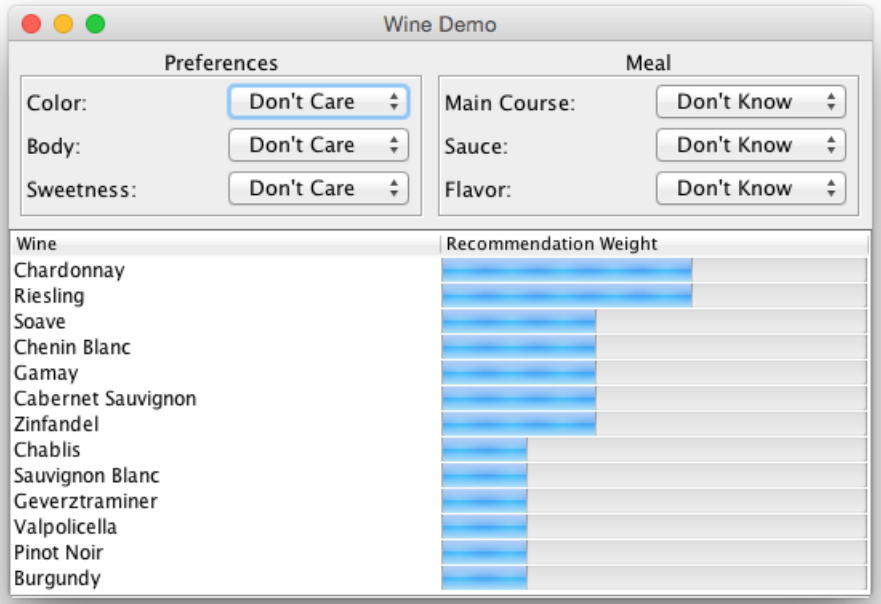
The Sudoku Demo window should appear:



To run the Wine demo, set the directory to the CLIPSJNI/examples/WineDemo directory and enter the following command:

```
java -cp .:../../CLIPSJNI.jar -Djava.library.path=../.. WineDemo
```

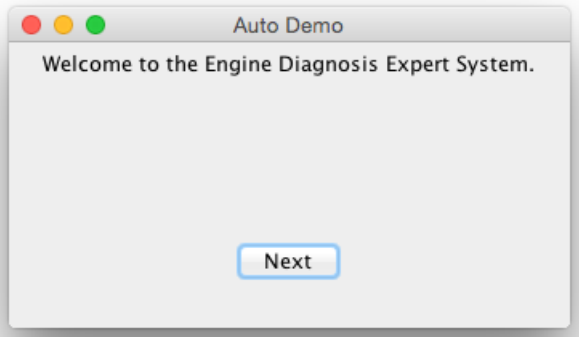
The Wine Demo window should appear:



To run the Auto demo, set the directory to the CLIPSJNI/examples/AutoDemo directory and enter the following command:

```
java -cp .:../../CLIPSJNI.jar -Djava.library.path=../.. AutoDemo
```

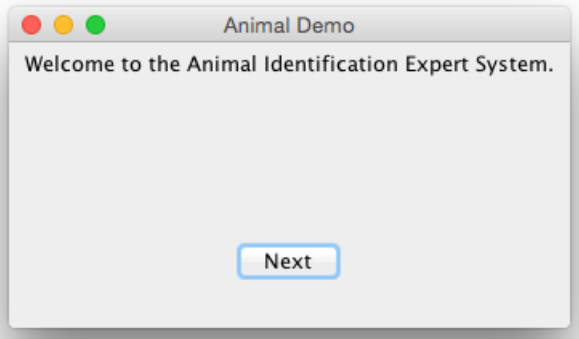
The Auto Demo window should appear:



To run the Animal demo, set the directory to the CLIPSJNI/examples/AnimalDemo directory and enter the following command:

```
java -cp .:../../CLIPSJNI.jar -Djava.library.path=../.. AnimalDemo
```

The Animal Demo window should appear:

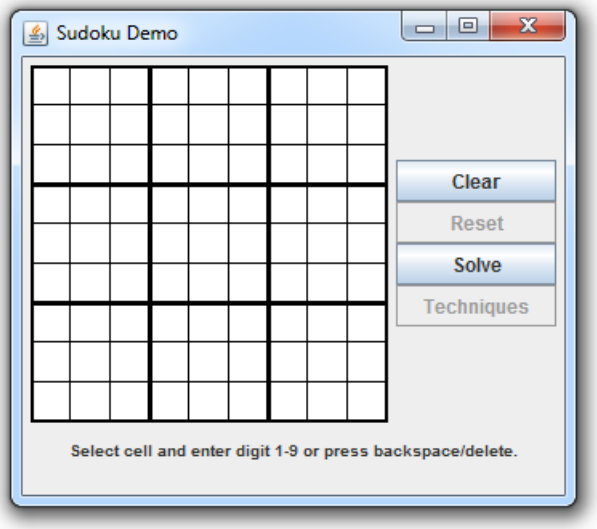


10.3.2 Running the Demo Programs on Windows 7

Launch the Command Prompt application (select Start > All Programs > Accessories > Command Prompt). Set the directory to the CLIPSJNI/examples/SudokuDemo directory (using the cd command). To run the Sudoku demo, enter the following command:

```
java -cp .;../../CLIPSJNI.jar -Djava.library.path=../.. SudokuDemo
```

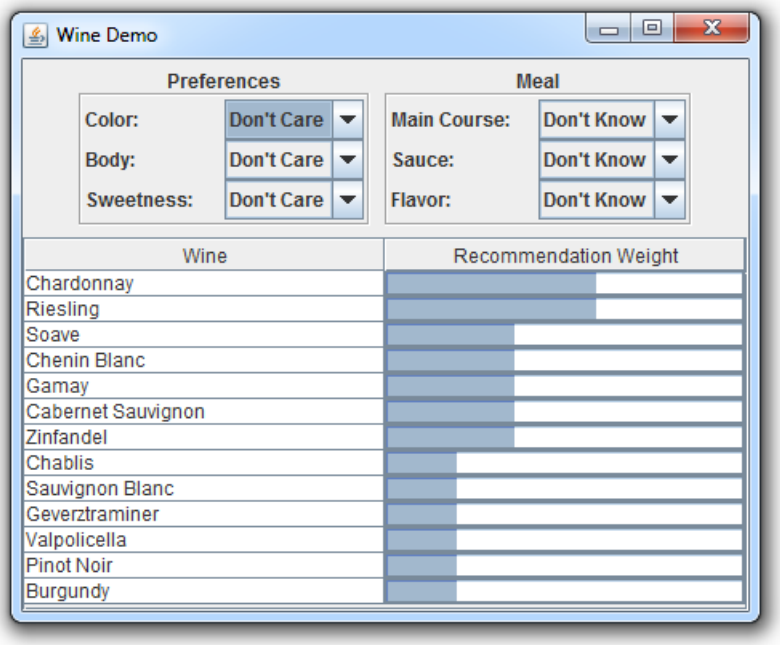
The Sudoku Demo window should appear:



To run the Wine demo, set the directory to the CLIPSJNI/examples/WineDemo directory and enter the following command:

```
java -cp .;../../CLIPSJNI.jar -Djava.library.path=../.. WineDemo
```

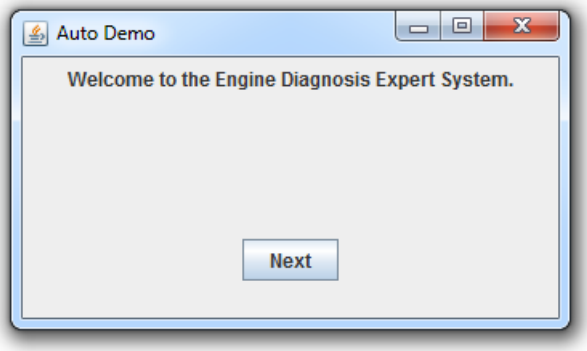
The Wine Demo window should appear:



To run the Auto demo, set the directory to the CLIPSJNI/examples/WineDemo directory and enter the following command:

```
java -cp .;../../CLIPSJNI.jar -Djava.library.path=../.. AutoDemo
```

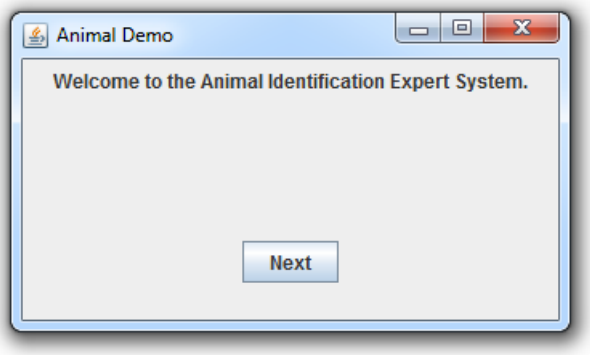
The Auto Demo window should appear:



To run the Animal demo, set the directory to the CLIPSJNI/examples/AnimalDemo directory and enter the following command:

```
java -cp .;../../CLIPSJNI.jar -Djava.library.path=../.. AnimalDemo
```

The Animal Demo window should appear:



10.4 Creating the CLIPSJNI JAR File

If you wish to add new functionality to the CLIPSJNI package, such as new Java methods which may call existing or new native functions, it is necessary to recreate the CLIPSJNI jar file. The CLIPSJNI distribution already contains the precompiled CLIPSJNI jar file in the top level CLIPSJNI directory, so if you are not adding new functionality to the CLIPSJNI package, you do not need to recreate the jar file (unless you want to create a jar file using a Java version prior to version 1.8.0).

To create the jar file, first open a terminal window where you can enter Java tool commands. On Mac OS X, launch the Terminal application (located in the Applications/Utilities directory). On Windows 7, launch the Command Prompt application (select Start > All Programs > Accessories > Command Prompt).

Using the appropriate commands (cd on Mac OS X and Windows 7), set the current directory to CLIPSJNI/java-src , then enter the following command to compile the CLIPSJNI java source:

```
javac CLIPSJNI/*.java
```

Once compiled, enter the following command to place the class files in a jar file:

```
jar -cf CLIPSJNI.jar CLIPSJNI/*.class
```

Once the CLIPSJNI.jar file is created, move it from the CLIPSJNI/java-src directory to the top level CLIPSJNI directory.

If you are adding new native functions to the CLIPSJNI package, it is also necessary to create the JNI header file which will be used to compile the native library. While you are still in the CLIPSJNI/java-src directory, enter the following command:

```
javah -jni CLIPSJNI.Environment
```

This command creates a file named CLIPSJNI_Environment.h which must be moved from the CLIPSJNI/java-src directory to the CLIPSJNI/library-src directory.

10.5 Creating the CLIPSJNI Native Library

The CLIPSJNI distribution already contains a precompiled universal library for Mac OS X, libCLIPSJNI.jnilib, and for Windows, CLIPSJNI.dll, in the top level CLIPSJNI directory. It is necessary to create a native library only if you are using the CLIPSJNI with an operating system other than Mac OS X or Windows. You must also create the native library if you want to add new functionality to the CLIPSJNI package by adding additional native functions. The steps for creating a native library varies between operating systems, so some research may be necessary to determine how to create one for your operating system.

10.5.1 Creating the Native Library on Mac OS X

Launch the Terminal application (located in the Applications/Utilities directory). Set the directory to the CLIPSJNI/lbrary-src directory (using the cd command).

To create a universal native library that can run on both Intel 32 and 64 bit architectures, enter the following command:

```
make -f makefile.mac
```

Once you have create the native library, copy the libCLIPSJNI.jnilib file from the CLIPSJNI/library-src to the top level CLIPSJNI directory.

10.5.2 Creating the Native Library on Windows 7

The following steps assume you have Microsoft Visual Studio 2013 installed. First, launch the Command Prompt application (select Start > All Programs > Accessories > Command Prompt).

Next, execute the script that sets up the environment variables for the appropriate target machine. For example, the vcvars64.bat batch file in the directory “Program Files (x86)/Microsoft Visual Studio 12.0/VC/bin/amd64”.

Set the directory to the CLIPSJNI/lbrary-src directory (using the cd command).

To create the native library DLL, enter the following command:

```
nmake -f makefile.win
```

Once you have create the native library, copy the CLIPSJNI.dll file from the CLIPSJNI/library-src to the top level CLIPSJNI directory.

10.5.3 Creating the Native Library On Other Systems

The file makefile.linux is intended to generate a native library for Linux systems using Java version 1.8.0. It can be invoked using the following command:

```
make -f makefile.linux
```

The shared library generated by this makefile is libCLIPSJNI.so. You will likely need to change the directory paths in the makefile to the appropriate location for your Java installation.

10.6 Recompiling the Swing Demo Programs

If you want to make modification to the Swing Demo programs and recompile them, you can use one of the following commands to do so (assuming you are in the appropriate directory for the example and the CLIPSJNI.jar file is present in the top level CLIPSJNI directory):

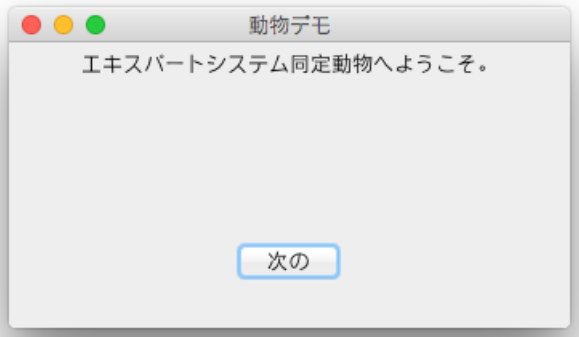
```
javac -classpath ../../CLIPSJNI.jar SudokuDemo.java

javac -classpath ../../CLIPSJNI.jar WineDemo.java

javac -classpath ../../CLIPSJNI.jar AutoDemo.java

javac -classpath ../../CLIPSJNI.jar AnimalDemo.java
```

10.7 Internationalizing the Swing Demo Programs

The Swing Demo Programs have been designed for internationalization. Several software generated example translations have been provided including Japanese (language code ja), Russian (language code ru), Spanish (language code es), and Arabic (language code ar). To make use of one of the translations, specify the language code when starting the demonstration program. For example, to run the Animal Demo in Japanese on Mac OS X, use the following command:

```
java -cp .:../../CLIPSJNI.jar -Djava.library.path=../.. -Duser.language=ja AnimalDemo
```

The welcome screen for the program should appear in Japanese rather than English:



It may be necessary to install additional fonts to view some languages. On Mac OS X, you can see which languages are supported by launching System Preferences and clicking the Language & Region icon. On Windows 7, you can see which languages are supported by launching Control Panel and selecting the Keyboards and Languages tab from Region and Language Options.

To create translations for other languages, first determine the two character language code for the target language. Make a copy in the resources directory of the ASCII English properties file for the demo program and save it as a UTF-8 encoded file including the language code in the name and using the .source extension. A list of language code is available at http://www.mathguide.de/info/tools/languagecode.html. For example, to create a Greek translation file for the Wine Demo, create the UTF-8 encoded WineResources_el.source file from the ASCII WineResources.properties file. Note that this step requires that you to do more than just duplicate the property file and rename it. You need to use a text editor that allows you to change the encoding from ASCII to UTF-8.

Once you've created the translation source file, edit the values for the properties keys and replaced the English text following each = symbol with the appropriate translation. When you have completed the translation, use the Java native2ascii utility to create an ASCII text file from the source file. For example, to create a Greek translation for the Wine Demo program, you'd use the following command:

```
native2ascii -encoding UTF-8 WineResources_el.source WineResources_el.properties
```

Note that the properties file for languages containing non-ASCII characters will contain Unicode escape sequences and is therefore more difficult to read (assuming of course that you can read the language in the original source file). This is the reason that two files are used for creating the translation. The UTF-8 source file is encoded so that you can read and edit the translation and the ASCII properties file is encoded in the format expected for use with Java internationalization features.

Section 11:
Microsoft Windows Integration

This section describes various techniques for integrating CLIPS and creating executables when using Microsoft Windows. The examples in this section have been tested running on Windows 7 Home Premium 32-bit Operating System with Visual C++ 2010 Express and Windows 7 Professional 64-bit Operating System with Visual Studio 2013.

11.1 Installing the Source Code

In order to run the integration examples, you must install the source code by downloading the clips_windows_projects_630.zip file (see appendix A for information on obtaining CLIPS). Once downloaded, you must then extract the contents of the file by right clicking on it and selecting the "Extract All…" menu item. Drag the *Projects* directory into the directory you'll be using for development. In addition to the source code specific to the Windows projects, the core CLIPS source code is also included, so there is no need to download this code separately.

11.2 Building the CLIPS Libraries and ExecutableS

The Windows integration source code includes six projects for building libraries and executables. They are:

- CLIPSStatic
- CLIPSDOS
- CLIPSJNI
- CLIPSIDE
- CLIPSDynamic
- CLIPSWrapper

CLIPSStatic is a starter project that demonstrates how to build a CLIPS C++ library that is statically linked with an executable. CLIPSDOS is a project that creates a DOS command-line version of CLIPS. CLIPSJNI is a starter project that demonstrates how to build a CLIPS library for use with the Java Native Interface. CLIPSIDE is a project that creates the CLIPS Integrated Development Environment (that is described in greater detail in the CLIPS Interfaces Guide). CLIPSDynamic is a starter project that demonstrates how to build a CLIPS Dynamic Link Library (DLL) that is dynamically linked with an executable. CLIPSWrapper is a C++ "wrapper" library that simplifies the use of the CLIPS DLL.

Unless you want to make changes to the executables or libraries, there is no need to build them. Windows executables are available through a separate installer and precompiled libraries are available in the Project\Libraries directory of the Windows source code.

11.2.1 Building the Projects Using Microsoft Visual C++ 2010 Express

Navigate to the *Projects\CLIPS_MVC_2010* directory. Open the file CLIPS.sln by double clicking on it or right click on it and select the *Open* menu item. After the file opens in Visual C++, select *Configuration Manager…* from the *Build* menu. Select the Configuration (Debug or Release) for each project and then click the *Close* button. Select the *Build Solution* menu item from the *Build* menu. When compilation is complete, the CLIPSIDE and CLIPSDOS executables will be in the corresponding <Platform>\<Configuration> directory of the *Projects\CLIPS_MVC_2010\Executables* directory and the library/DLL files will be in the *Projects\Libraries* directory.

To compile projects individually, right click on the project name in the *Solution Explorer* pane and select the *Build* menu item.

The CLIPSJNI project assumes that Java SE Development SE 8u31 is installed on your computer and that the Java header files are contained in the directories C:\Program Files\Java\jdk1.8.0_31\include and C:\Program Files\Java\jdk1.8.0_31\include\win32. To change the directory setting for the location of the headers files, right click on the CLIPSJNI project and select the *Properties* menu item. In the tree view control, open the *Configuration Properties* and *C/C++* branches, then select the *General* leaf item. Edit the value in the *Additional Include Directories* editable text box to include the appropriate directory for the Java include files.

11.2.2 Building the Projects Using Microsoft Visual Studio 2013

Navigate to the *Projects\CLIPS_MVS_2013* directory. Open the file CLIPS.sln by double clicking on it (or right click on it and select the *Open* menu item). After the file opens in Visual Studio, select *Configuration Manager…* from the *Build* menu. Select the Configuration (Debug or Release) and the Platform (Win32 or x64) for each project and then click the *Close* button. Select the *Build Solution* menu item from the *Build* menu. When compilation is complete, the CLIPSIDE and CLIPSDOS executables will be in the corresponding <Platform>\<Configuration> directory of the *Projects\CLIPS_MVS_2013\Executables* directory and the library/DLL files will be in the *Projects\Libraries* directory.

To compile projects individually, right click on the project name in the *Solution Explorer* pane and select the *Build* menu item.

The CLIPSJNI project assumes that Java SE Development SE 8u31 is installed on your computer and that the Java header files are contained in the directories C:\Program

Files\Java\jdk1.8.0_31\include and C:\Program Files\Java\jdk1.8.0_31\include\win32. To change the directory setting for the location of the headers files, right click on the CLIPSJNI project and select the *Properties* menu item. In the tree view control, open the *Configuration Properties* and *C/C++* branches, then select the *General* leaf item. Edit the value in the *Additional Include Directories* editable text box to include the appropriate directory for the Java include files.

11.3 Running the Library Examples

The Windows integration source code includes four projects that demonstrate the use of the static and dynamic libraries from Section 11.2. They are:

- ExplicitDLLExample
- ImplicitDLLExample
- SimpleLibExample
- WrappedDLLExample

The ExplicitDLLExample project demonstrates how to dynamically load the CLIPS DLL (either CLIPSDynamic32.dll or CLIPSDynamic64.dll depending upon the platform chosen). The example code explicitly loads the DLL using the *LoadLibrary* system call and then locates the exported functions using the *GetProcAddress* system call. The ImplicitDLLExample project demonstrates how to statically load the CLIPS DLL. The example code links with the DLL import library (either CLIPSDynamic32.lib or CLIPSDynamic64.lib) which handles the task of loading the DLL and locating the exported functions. The SimpleLibExample project demonstrates how to statically load the CLIPS C++ library (either CLIPSStatic32.lib or CLIPSStatic64.lib). The C++ class *CLIPSCPPEnv* is used to provide a C++ wrapper to the CLIPS API. The WrappedDLLExample demonstrates the use of a C++ wrapper to simplify the use of the DLL. The example code used in this project is identical to the code used with the SimpleLibExample project.

In order for the DLL examples to work properly, the directory containing the DLL must be on the system search path. To set the path, open the *Control Panel* from the *Start* menu and double click on the *System* control panel. Select *Advanced systems settings* and then click the *Environment Variables* button. In the *User variables* list box (or the *System variables* list box if you want the DLL accessible to all users) select the *path* variable and then click the *Edit* button. Add the directory containing the DLL to the path (which typically would be the full path to the Projects\Libraries directory).

11.2.1 Building the Examples Using Microsoft Visual C++ 2010 Express

Navigate to the *Projects\Examples_MVC_2010* directory. Open the file Examples.sln by double clicking on it (or right click on it and select the *Open* menu item). After the file opens in Visual C++, select *Configuration Manager…* from the *Build* menu. Select the Configuration (Debug or

Release) for each project and then click the *Close* button. Note that the configuration chosen should match the configuration of the libraries/DLLs in the *Projects\Libraries* directory. Select the *Build Solution* menu item from the *Build* menu. When compilation is complete, the example executables will be in the corresponding <Platform>\<Configuration> directory of the *Projects\Examples_MVC_2010\Executables* directory.

To compile projects individually, right click on the project name in the *Solution Explorer* pane and select the *Build* menu item.

11.2.2 Building the Examples Using Microsoft Visual Studio 2013

Navigate to the *Projects\Examples_MVS_2013* directory. Open the file Examples.sln by double clicking on it (or right click on it and select the *Open* menu item). After the file opens in Visual Studio, select *Configuration Manager…* from the *Build* menu. Select the Configuration (Debug or Release) and the Platform (Win32 or x64) for each project and then click the *Close* button. Note that the configuration chosen should match the configuration of the libraries/DLLs in the *Projects\Libraries* directory. Select the *Build Solution* menu item from the *Build* menu. When compilation is complete, the example executables will be in the corresponding <Platform>\<Configuration> directory of the *Projects\Examples_MVC_2013\Executables* directory.

To compile projects individually, right click on the project name in the *Solution Explorer* pane and select the *Build* menu item.

Appendix A:
Support Information

A.1 Questions and Information

The URL for the CLIPS Web page is http://clipsrules.sourceforge.net.

Questions regarding CLIPS can be posted to one of several online forums including the CLIPS Expert System Group, http://groups.google.com/group/CLIPSESG/, the SourceForge CLIPS Forums, http://sourceforge.net/forum/?group_id=215471, and Stack Overflow, http://stackoverflow.com/questions/tagged/clips.

Inquiries related to the use or installation of CLIPS can be sent via electronic mail to clipssupport@secretsocietysoftware.com.

A.2 Documentation

The CLIPS Reference Manuals and User's Guide are available in Portable Document Format (PDF) at http://clipsrules.sourceforge.net/OnlineDocs.html.

Expert Systems: Principles and Programming, 4th Edition, by Giarratano and Riley comes with a CD-ROM containing CLIPS 6.22 executables (DOS, Windows XP, and Mac OS), documentation, and source code. The first half of the book is theory oriented and the second half covers rule-based, procedural, and object-oriented programming using CLIPS.

A.3 CLIPS Source Code and Executables

CLIPS executables and source code are available on the SourceForge web site at http://sourceforge.net/projects/clipsrules/files.

Appendix B:
Update Release Notes

The following sections denote the changes and bug fixes for CLIPS versions 6.2, 6.21, 6.22, 6.23, 6.24, and 6.30.

B.1 Version 6.30

- **CLIPS Java Native Interface** – The **CLIPSJNI** project contains libraries and example programs demonstrating the integration of CLIPS with Java. See section 10 for more information.

- **Windows Integration Examples** – Several example projects are available demonstrating the creation of C++ wrapper classes, static libraries, and dynamic link libraries. See section 11 for more information.

- **External Function 64-bit Interface** - Several functions and macros have been modified to support "long long" integers:

  DOToLong (see section 3.2.3)
  DOPToLong (see section 3.2.3)
  EnvAddLong (see section 3.3.5)
  EnvFacts (see section 4.4.9)
  EnvFactIndex (see section 4.4.8)
  EnvDefineFunction 'g' argument (see section 3.1)
  EnvRtnLong (see section 3.2.2)
  EnvRun (see section 4.7.20)
  ValueToLong (see section 3.3.5)

- **Compiler Directives** – The **ENVIRONMENT_API_ONLY** flag has been removed. The **EX_MATH** flag has been renamed to the **EXTENDED_MATH_FUNCTIONS** flag. The **BASIC_IO** and **EXT_IO** flags have been combined into the **IO_FUNCTIONS** flag. The preprocessor definition flags in setup.h are now conditionally defined only if they are undefined (which allows you to define the flags from a makefile or project without editing setup.h). The **MAC_MCW**, **WIN_MCW**, and **WIN_BTC** flags have been removed. The associated compilers are no longer supported.

- **Environment Globals** – The **ALLOW_ENVIRONMENT_GLOBALS** flag now defaults to 0. The use of functions previously enabled by this flag is deprecated. These functions

include **GetCurrentEnvironment**, **GetEnvironmentByIndex**, **GetEnvironmentIndex**, **InitializeEnvironment**, **SetCurrentEnvironment**, and **SetCurrentEnviromentByIndex**.

- **Garbage Collection** – The mechanism used for garbage collection has been modified. See section 1.4 for information for more details. The EnvDecrementGCLocks function now performs garbage collection if the lock count is reduced to 0.

- **MicroEMACS Editor** – The built-in MicroEMACS editor is no longer supported. The associated source files and **EMACS_EDITOR** compiler directive flag have been removed.

- **Block Memory** – Block memory allocation is no longer supported. The associated source code and **BLOCK_MEMORY** compiler directive flag have been removed.

- **Help Functions** – The **help** and **help-path** funtions are no longer supported. The **HELP_FUNCTIONS** compiler directive flag has been removed.

- **External Addresses** – The method of retrieving and returning an external address from a user defined function has changed (see sections 3.2.3, 3.3.4, and 3.3.5).

- **Deleted Source Files** – The following source files have been removed:

  cmptblty.h
  ed.h
  edbasic.c
  edmain.c
  edmisc.c
  edstruct.c
  edterm.c

- **Logical Name Defintions** – Added logical name constants **STDIN** and **STDOUT** that can be used in place of the strings "stdin" and "stdout".

- **Command and Function Changes** - The following commands and functions have been changed:

  - **constructs-to-c** (see section 5.1). A target directory path can be specified for the files generated by this command.

  - **EnvMatches** (see section 4.6.13). The number of parameter for this function has changed.

  - **EnvSaveFacts** (see section 4.4.24). The number of parameter for this function has changed.

- **EnvSaveInstances** (see section 4.13.23). The number of parameter for this function has changed.

- **EnvBinarySaveInstances** (see section 4.13.2). The number of parameter for this function has changed.

- **EnvDefineFunction** (see section 3.1). New return types ‘g’ (long long integer) and ‘y’ (fact address) have been added.

- **EnvReleaseMem** (see section 8.2.4). The number of parameter for this function has changed.

B.2 Version 6.24

- **External Function Interface** - Several new functions have been added including:

  DeftemplateSlotAllowedValues
  DeftemplateSlotCardinality
  DeftemplateSlotDefaultP
  DeftemplateSlotDefaultValue
  DeftemplateSlotExistP
  DeftemplateSlotMultiP
  DeftemplateSlotNames
  DeftemplateSlotRange
  DeftemplateSlotSingleP
  DeftemplateSlotTypes
  PPFact
  SlotAllowedClasses
  SlotDefaultValue

- **C++ Compilation Errors** – Corrected a few compiler errors that occurred when compiling the CLIPS source files as C++ files.

- **Macro Redefinition** – Changed the internal macro definition of BOOLEAN to intBool to avoid conflict with the Cocoa/Obective C definitions.

- **Compiler Directives** – The following flags have been removed:

  AUXILARY_MESSAGE_HANDLERS
  CONFLICT_RESOLUTION_STRATEGIES
  DYNAMIC_SALIENCE

IMPERATIVE_MESSAGE_HANDLERS
IMPERATIVE_METHODS
INCREMENTAL_RESET
INSTANCE_PATTERN_MATCHING
LOGICAL_DEPENDENCIES
SHORT_LINK_NAMES

- **New Source Files** – New source files have been added:

  userfunctions.c

- **Deleted Source Files** – The following source files have been removed:

  shrtlnkn.h

B.3 Version 6.23

- **FalseSymbol and TrueSymbol Changes** – The FalseSymbol and TrueSymbol constants were not defined as specified in the *Advanced Programming Guide*. These constants have have now been defined as macros so that their corresponding environment companion functions (EnvFalseSymbol and EnvTrueSymbol) could be defined.

- **Run-time Program Bug Fix** – Files created by the constructs-to-c function for use in a run-time program generate compilation errors.

- **External Function Interface** - A new function has been added:

  GetNextFactInTemplate

- **Compiler Directives** – The FACT_SET_QUERIES flag has been added.

- **New Source Files** – New source files have been added:

  factqpsr.c
  factqpsr.h
  factqury.c
  factqury.h

B.4 Version 6.22

- **Function and Macro Corrections** – The following functions and macros were corrected to accept the correct number of arguments as specified in the *Advanced Programming Guide*:

Agenda
BatchStar
EnvGetActivationSalience
EnvBatchStar
EnvFactDeftemplate
EnvFactExistp
EnvFactList
EnvFactSlotNames
EnvGetNextInstanceInClassAndSubclasses
EnvLoadInstancesFromString
EnvRestoreInstancesFromString
EnvSetOutOfMemoryFunction
FactDeftemplate
FactExistp
FactList
FactSlotNames
GetNextInstanceInClassAndSubclasses
LoadInstancesFromString
RestoreInstancesFromString
SetOutOfMemoryFunction

B.5 Version 6.21

- **Introduction** – Added information on thread\concurrency and garbage collection issues.

- **External Function Interface** - Several new functions have been added including:

  DeallocateEnvironmentData
  DecrementGCLocks
  FactDeftemplate
  GetEnvironmentByIndex
  IncrementGCLocks

B.6 Version 6.2

- **Environments** – It is now possible in an embedded application to create multiple environments into which programs can be loaded.

- **External Function Interface** - Several new functions have been added including:

  GetClassDefaultsMode

SetClassDefaultsMode

- **Run-time Programs** – Support for environments requires some changes in code for loading run-time programs.

- **Compiler Directives** – Two new flags have been added: ENVIRONMENT_API_ONLY and ALLOW_ENVIRONMENT_GLOBALS.

- **New Source Files** – New source files have been added:

  envrnmnt.c
  envrnmnt.h

- **Deleted Source Files** – The following source files have been removed:

  extobj.h

Appendix C:
I/O Router Examples

The following examples demonstrate the use of the I/O router system. These examples show the necessary C code for implementing the basic capabilities described.

C.1 Dribble System

Write the necessary functions that will divert all tracing information to the trace file named "trace.txt".

```
/*
First of all, we need a file pointer to the dribble file which will  contain the
tracing information.
*/

#include <stdio.h>
#include "clips.h"

static FILE *TraceFP = NULL;

/*
We want to recognize any output that is sent to the logical name "wtrace" because all
tracing information is sent to this logical name.  The recognizer function for our
router is defined below.
*/

int FindTrace(
  void *environment,
  const char *logicalName)
  {
   if (strcmp(logicalName,WTRACE) == 0) return(TRUE);

   return(FALSE);
  }

/*
We now need to define a function which will print the tracing in¬formation to our
trace file.  The print function for our router is defined below.
*/

int PrintTrace(
  void *environment,
  const char *logicalName,
  const char *str)
  {
   fprintf(TraceFP,"%s",str);
```

```
   return 0;
  }

/*
When we exit CLIPS the trace file needs to be closed.  The exit function for our
router is defined below.
*/

int ExitTrace(
  void *environment,
  int exitCode)                                    /* unused */
  {
   fclose(TraceFP);
   return 0;
  }

/*
There is no need to define a get character or ungetc character function since this
router does not handle input.

A function to turn the trace mode on needs to be defined.  This function will check
if the trace file has already been opened.  If the file is already open, then nothing
will happen.  Otherwise, the trace file will be opened and the trace router will be
creat¬ed.  This new router will intercept tracing information intended for the user
interface and send it to the trace file.  The trace on function is defined below.
*/

int TraceOn(
  void *environment)
  {
   if (TraceFP == NULL)
     {
      TraceFP = fopen("trace.txt","w");
      if (TraceFP == NULL) return(FALSE);
     }
   else
     { return(FALSE); }

   EnvAddRouter(environment,
                "trace",                  /* Router name     */
                20,                       /* Priority        */
                FindTrace,                /* Query function  */
                PrintTrace,               /* Print function  */
                NULL,                     /* Getc function   */
                NULL,                     /* Ungetc function */
                ExitTrace);               /* Exit function   */

   return(TRUE);
  }

/*
```

```
A function to turn the trace mode off needs to be defined.  This function will check
if the trace file is already closed.  If the file is already closed, then nothing
will happen.  Otherwise, the trace router will be deleted and the trace file will be
closed.  The trace off function is defined below.
*/

int TraceOff(
  void *environment)
  {
   if (TraceFP != NULL)
     {
      EnvDeleteRouter(environment,"trace");

      if (fclose(TraceFP) == 0)
        {
         TraceFP = NULL;
         return(TRUE);
        }
     }

   TraceFP = NULL;
   return(FALSE);
  }

/*
Now add the definitions for these functions to the EnvUserFunctions function in file
"userfunctions.c".
*/

extern int TraceOn(void *), TraceOff(void *);

EnvDefineFunction(environment,"tron",'b',TraceOn, "TraceOn");
EnvDefineFunction(environment,"troff",'b',TraceOff, "TraceOff");

/*
Compile and link the appropriate files. The trace functions should now be accessible
within CLIPS as external functions. For Example
   CLIPS> (tron)
   CLIPS> (watch facts)
   CLIPS> (assert (example))
       ●
       ●
       ●
   CLIPS> (troff)
*/
```

C.2 Better Dribble System

Modify example 1 so the tracing information is sent to the terminal as well as to the trace dribble file.

```
/*
This example requires a modification of the PrintTrace function.  After the trace
string is printed to the file, the trace router must be deactivated.  The trace
string can then be sent through the PrintRouter function so that the next router in
line can handle the output.  After this is done, then the trace router can be
reactivated.
*/

int PrintTrace(
  void *environment,
  const char *logicalName,
  const char *str)
  {
   fprintf(TraceFP,"%s",str);
   EnvDeactivateRouter(environment,"trace");
   EnvPrintRouter(environment,logicalName,str);
   EnvActivateRouter(environment,"trace");
   return 0;
  }

/*
The TraceOn function must also be modified.  The priority of the router should be 40
instead of 20 since the router passes the output along to other routers.
*/

int TraceOn(
  void *environment)
  {
   if (TraceFP == NULL)
     {
      TraceFP = fopen("trace.txt","w");
      if (TraceFP == NULL) return(FALSE);
     }
   else
     { return(FALSE); }

   EnvAddRouter(environment,
                "trace",                  /* Router name     */
                40,                       /* Priority        */
                FindTrace,                /* Query function  */
                PrintTrace,               /* Print function  */
                NULL,                     /* Getc function   */
                NULL,                     /* Ungetc function */
                ExitTrace);               /* Exit function   */

   return(TRUE);
  }
```

C.3 Batch System

Write the necessary functions that will allow batch input from the file "batch.txt" to the CLIPS top-level interface.

```
/*
First of all, we need a file pointer to the batch file which will contain the batch
command information.
*/

#include <stdio.h>
#include "clips.h"

static FILE *BatchFP = NULL;

/*
We want to recognize any input requested from the logical name "stdin" because all
user input is received from this logical name.  The recognizer function for our
router is defined below.
*/

int FindMybatch(
  void *environment,
  const char *logicalName)
  {
   if (strcmp(logicalName,STDIN) == 0) return(TRUE);

   return(FALSE);
  }

/*
We now need to define a function which will get and unget characters from our batch
file.  The get and ungetc character functions for our router are defined below.
*/

static char BatchBuffer[80];
static int BatchLocation = 0;

int GetcMybatch(
  void *environment,
  const char *logicalName)
  {
   int rv;

   rv = getc(BatchFP);

   if (rv == EOF)
     {
      EnvDeleteRouter(environment,"mybatch");
      fclose(BatchFP);
      return(EnvGetcRouter(environment,logicalName));
     }
```

```
   BatchBuffer[BatchLocation] = (char) rv;
   BatchLocation++;
   BatchBuffer[BatchLocation] = EOS;

   if ((rv == '\n')  || (rv == '\r'))
     {
      EnvPrintRouter(environment,WPROMPT,BatchBuffer);
      BatchLocation = 0;
     }

   return(rv);
  }

int UngetcMybatch(
  void *environment,
  int ch,
  const char *logicalName)   /* unused */
  {
   if (BatchLocation > 0) BatchLocation--;
   BatchBuffer[BatchLocation] = EOS;
   return(ungetc(ch,BatchFP));
  }

/*
When we exit CLIPS the batch file needs to be closed.  The exit function for our
router is defined below.
*/

int ExitMybatch(
  void *environment,
  int exitCode)   /* unused */
  {
   fclose(BatchFP);
   return 0;
  }

/*
There is no need to define a print function since this router does not handle output
except for echoing the command line.
Now we define a function that turns the batch mode on.
*/

int MybatchOn(
  void *environment)
  {
   BatchFP = fopen("batch.txt","r");

   if (BatchFP == NULL) return(FALSE);

   EnvAddRouter(environment,
                "mybatch",               /* Router name     */
                20,                      /* Priority        */
                FindMybatch,             /* Query function  */
```

```
                NULL,                   /* Print function  */
                GetcMybatch,            /* Getc function   */
                UngetcMybatch,          /* Ungetc function */
                ExitMybatch);           /* Exit function   */

   return(TRUE);
  }

/*
Now add the definition for this function to the UserFunctions function in file
"userfunctions.c".
*/

extern int MybatchOn(void *);

EnvDefineFunction(environment,"mybatch",'b',MybatchOn, "MybatchOn");

/*
Compile and link the appropriate files. The batch function should now be accessible
within CLIPS as external function. For Example, create the file batch.txt with the
following content:

   (+ 2 3)
   (* 4 5)

Launch CLIPS and enter a (mybatch) command:

   CLIPS> (mybatch)
   TRUE
   CLIPS> (+ 2 3)
   5
   CLIPS> (* 4 5)
   20
   CLIPS>
*/
```

C.4 Simple Window System

Write the necessary functions using CURSES (a screen management function available in UNIX) that will allow a top/bottom split screen interface. Output sent to the logical name **top** will be printed in the upper window. All other screen I/O should go to the lower window. (NOTE: Use of CURSES may require linking with special libraries. On UNIX systems try using –lcurses when linking.)

```
/*
First of all, we need some pointers to the windows and a flag to indicate that the
windows have been initialized.
*/

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

```
#include "clips.h"

WINDOW *LowerWindow, *UpperWindow;
int WindowInitialized = FALSE;

/*
We want to intercept any I/O requests that the standard interface would handle. In
addition, we also need to handle requests for the logical name top. The recognizer
function for our router is defined below.
*/

int FindScreen(
  void *environment,
  const char *logicalName)
  {
   if ((strcmp(logicalName,STDOUT) == 0) ||
       (strcmp(logicalName,STDIN) == 0) ||
       (strcmp(logicalName,WPROMPT) == 0) ||
       (strcmp(logicalName,WDISPLAY) == 0) ||
       (strcmp(logicalName,WDIALOG) == 0) ||
       (strcmp(logicalName,WERROR) == 0) ||
       (strcmp(logicalName,WWARNING) == 0) ||
       (strcmp(logicalName,WTRACE) == 0) ||
       (strcmp(logicalName,"top") == 0) )
     { return(TRUE); }

   return(FALSE);
  }

/*
We now need to define a function which will print strings to the two windows. The
print function for our router is defined below.
*/

int PrintScreen(
  void *environment,
  const char *logicalName,
  const char *str)
  {
   if (strcmp(logicalName,"top") == 0)
     {
      wprintw(UpperWindow,"%s",str);
      wrefresh(UpperWindow);
     }
   else
     {
      wprintw(LowerWindow,"%s",str);
      wrefresh(LowerWindow);
     }

   return 0;
  }
```

```
/*
We now need to define a function which will get and unget characters from the lower
window. CURSES uses unbuffered input so we will simulate buffered input for CLIPS.
The get and ungetc character functions for our router are defined below.
*/

static int UseSave = FALSE;
static int SaveChar;
static int SendReturn = TRUE;

static char StrBuff[80] = {'\0'};
static int  CharLocation = 0;

int GetcScreen(
  void *environment,
  const char *logicalName)
  {
   int rv;

   if (UseSave == TRUE)
     {
      UseSave = FALSE;
      return(SaveChar);
     }

   if (StrBuff[CharLocation] == '\0')
     {
      if (SendReturn == FALSE)
        {
         SendReturn = TRUE;
         return('\n');
        }

      wgetstr(LowerWindow,&StrBuff[0]);
      CharLocation = 0;
     }

   rv = StrBuff[CharLocation];
   if (rv == '\0') return('\n');
   CharLocation++;
   SendReturn = FALSE;
   return(rv);
  }

int UngetcScreen(
  void *environment,
  int ch,
  const char *logicalName)
  {
   UseSave = TRUE;
   SaveChar = ch;
   return(ch);
  }
```

```
/*
When we exit CLIPS CURSES needs to be deactivated. The exit function for our router
is defined below.
*/

int ExitScreen(
  void *environment,
  int num)           /* unused */
  {
   endwin();
   return 0;
  }

/*
Now define a function that turns the screen mode on.
*/

int ScreenOn(
  void *environment)
  {
   int halfLines, i;

   /* Has initialization already occurred? */

   if (WindowInitialized == TRUE) return(FALSE);
   else WindowInitialized = TRUE;

   /* Reroute I/O and initialize CURSES. */

   initscr();
   echo();

   EnvAddRouter(environment,
                "screen",                 /* Router name     */
                10,                       /* Priority        */
                FindScreen,               /* Query function  */
                PrintScreen,              /* Print function  */
                GetcScreen,               /* Getc function   */
                UngetcScreen,             /* Ungetc function */
                ExitScreen);              /* Exit function   */

   /* Create the two windows. */

   halfLines = LINES / 2;
   UpperWindow = newwin(halfLines,COLS,0,0);
   LowerWindow = newwin(halfLines - 1,COLS,halfLines + 1,0);

   /* Both windows should be scrollable. */

   scrollok(UpperWindow,TRUE);
   scrollok(LowerWindow,TRUE);
```

```
   /* Separate the two windows with a line. */

   for (i = 0 ; i < COLS ; i++)
     { mvaddch(halfLines,i,'-'); }
   refresh();

   wclear(UpperWindow);
   wclear(LowerWindow);
   wmove(LowerWindow, 0,0);

   return(TRUE);
  }

/*
Now define a function that turns the screen mode off.
*/

int ScreenOff(
  void *environment)
  {
   /* Is CURSES already deactivated? */

   if (WindowInitialized == FALSE) return(FALSE);

   WindowInitialized = FALSE;

   /* Remove I/O rerouting and deactivate CURSES. */

   EnvDeleteRouter(environment,"screen");
   endwin();

   return(TRUE);
  }

/*
Now add the definitions for these functions to the UserFunctions function in file
"userfunctions.c".
*/

extern int ScreenOn(void *), ScreenOff(void *);

EnvDefineFunction(environment,"screen-on",'b',ScreenOn, "ScreenOn");
EnvDefineFunction(environment,"screen-off",'b',ScreenOff, "ScreenOff");

/*
Compile and link the appropriate files. The screen functions should now be accessible
within CLIPS as external functions. For Example
   CLIPS> (screen-on)
        •
        •
        •
   CLIPS> (screen-off)
*/
```

Index