Reference Manual

Volume II
Advanced Programming Guide

Version 6.40 Beta

August 21st 2018

CLIPS Advanced Programming Guide

Version 6.40 Beta August 21st 2018

CONTENTS

License Information

Preface

About CLIPS

Developed at NASA's Johnson Space Center from 1985 to 1996, the 'C' Language Integrated Production System (CLIPS) is a rule-based programming language useful for creating expert systems and other programs where a heuristic solution is easier to implement and maintain than an algorithmic solution. Written in C for portability, CLIPS can be installed and used on a wide variety of platforms. Since 1996, CLIPS has been available as public domain software.

CLIPS Version 6.4

Version 6.4 of CLIPS includes three major enhancements: a redesigned C Application Programming Interface (API); wrapper classes and example programs for .NET and Java; and Integrated Development Environments (IDEs) with Unicode support for Windows and Java. 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 several volumes:
  - *Volume I - The Basic Programming Guide* provides information on the CLIPS programming language.
  - *Volume II - The Advanced Programming Guide* provides information on compiling CLIPS and use of the C Application Programming Interfaces.
  - *Volume III - The Interfaces Guide* provides information on the CLIPS Integrated Development Environments, wrapper classes, and example programs.
- The *CLIPS User's Guide* provides an introduction to CLIPS and rule-based programming.

Section 1:
Introduction

This manual is the *Advanced Programming Guide* for CLIPS. It describes the Application Programmer Interface (API) that allows users to integrate CLIPS programs with code written in C. It is written with the assumption that the user has a basic understanding of both CLIPS and C. It is advised that users complete the *Basic Programming Guide* before reading this manual.

Section 2 of this document describes how to install and tailor CLIPS to meet specific needs. Section 3 describes the core API needed to embed CLIPS within a simple C program. Section 4 describes the API allowing the creation of constructs and the execution of commands and functions. Section 5 describes the API that allows a C program to keep persistent references to data structures subject to garbage collection. Section 6 describes the API for creating CLIPS primitive data values. Section 7 describes the API for creating Facts and Instances. Section 8 describes the API for adding user-defined functions. Section 9 describes the API for the I/O router system used by CLIPS for processing input and output requests. Section 10 describes the environment API which allows multiple expert systems to be loaded and run concurrently. Section 11 describes how to create run-time CLIPS programs which allow constructs to be save as C data structures which can be compiled and linked with CLIPS. Section 12 describes the additional APIs that are available for interacting with and retrieving information from CLIPS.

1.1 C++ Compatibility

The CLIPS source code can be compiled using either an ANSI C or C++ compiler. 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 compilers allow you to specify the whether a file should be compiled as C or C++ code based on the file extension. Other compilers 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). For compilers that exclusively use the file extension to determine whether the file should be compiled as a C or C++ code, it's necessary to change the .c extension of the CLIPS source files to a .cpp extension.

1.2 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 10, 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 Integrated Development Environments (IDEs) 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.

Section 2:
Installing and Tailoring CLIPS

This section describes how to install and tailor CLIPS to meet specific needs. Instructions are included for creating a console executable by compiling the portable core CLIPS source files. For instructions on compiling the Windows, macOS, and Java Integrated Development Environments for CLIPS, see the *Utilities and Interfaces Guide*.

2.1 Installing CLIPS

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

Testing of CLIPS 6.40 included the following software environments:

- Windows 10 Home Premium 32-bit and Windows 7 Professional 64-bit Operating Systems with Visual Studio Community 2017.

- MacOS High Sierra 10.13 using Xcode 9.4.

- Ubuntu 16.04 LTS using gcc 5.4.0; Debian GNU/Linux 9.1 with gcc 6.3; Fedora 26 with gcc 7.1.1; Linux Mint 18 with gcc 5.4.0; and CentOS Linux 7 with gcc 4.8.5.

CLIPS is designed for portability and should run on any operating system which supports an ANSI C or C++ compiler. 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	dfinscmp.h	immthpsr.h	prcdrpsr.h
analysis.h	drive.h	incrrset.h	prdctfun.h
argacces.h	emathfun.h	inherpsr.h	prntutil.h
bload.h	engine.h	inscom.h	proflfun.h
bmathfun.h	entities.h	insfile.h	reorder.h
bsave.h	envrnbld.h	insfun.h	reteutil.h
classcom.h	envrnmnt.h	insmngr.h	retract.h
classexm.h	evaluatn.h	insmoddp.h	router.h

classfun.h
classinf.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
dfinsbin.h
expressn.h
exprnbin.h
exprnops.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
factprt.h
factqpsr.h
factqury.h
factrete.h
factrhs.h
filecom.h
filertr.h
fileutil.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
insmult.h
inspsr.h
insquery.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
rulebin.h
rulebld.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
usrsetup.h
utility.h
watch.h

agenda.c
analysis.c
argacces.c
bload.c
bmathfun.c
bsave.c
drive.c
emathfun.c
engine.c
envrnbld.c
envrnmnt.c
evaluatn.c
immthpsr.c
incrrset.c
inherpsr.c
inscom.c
insfile.c
insfun.c
prcdrpsr.c
prdctfun.c
prntutil.c
proflfun.c
reorder.c
reteutil.c

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

In addition to these core files, the Integrated Development Environments require additional files for compilation. See the *Utilities and Interfaces Guide* for details on compiling the IDEs.

2) **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). 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 –DLINUX will work on many compilers to set the preprocessor definition of LINUX to 1.

3) **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 UserFunctions definition in userfunctions.c to reflect the user's functions (see section 3 for more on user-defined functions).

4) **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 Makefiles

The makefiles 'makefile.win' and 'makefile' are provided with the core source code to create executables and static libraries for Windows, MacOS, and Linux. The makefiles can be used to create either release or debug versions of the executables/libraries and to compile the code as C or C++.

Using the Windows Makefile

The following steps assume you have Microsoft Visual Studio Community 2017 installed. First, launch the Command Prompt application from the Start menu by selecting *Visual Studio 2017* and then either *VS2017 x64 Native Tools Command Prompt* or *VS2017 x86 Native Tools Command Prompt*. Next, use the cd command to change the current directory to the one

containing the core CLIPS source code and makefiles. To compile CLIPS as C code without debugging information, use the command

```
nmake -f makefile.win
```

or

```
nmake -f makefile.win BUILD=RELEASE
```

To compile CLIPS as C++ code without debugging information, use the following command:

```
nmake -f makefile.win BUILD=RELEASE_CPP
```

To compile CLIPS as C code with debugging information, use the following command:

```
nmake -f makefile.win BUILD=DEBUG
```

To compile CLIPS as C++ code with debugging information, use the following command:

```
nmake -f makefile.win BUILD=DEBUG_CPP
```

When compilation is complete, the executable file clips.exe and the static library file clips.lib will be created in the source directory.

Before rebuilding the executable and library with a different BUILD variable value, the clean action should be run:

```
nmake -f makefile.win clean
```

Using the macOS and Linux Makefile

First, launch the Terminal application. Use the cd command to change the current directory to the one containing the core CLIPS source code and makefiles. To compile CLIPS as C code without debugging information, use the command

```
make
```

or

```
make release
```

To compile CLIPS as C++ code without debugging information, use the following command:

```
make release_cpp
```

To compile CLIPS as C code with debugging information, use the following command:

```
make debug
```

To compile CLIPS as C++ code with debugging information, use the following command:

```
make debug_cpp
```

When compilation is complete, the executable file clips and the static library file libclips.a will be created in the source directory.

Before rebuilding the executable and library with a different configuration, the clean action should be run:

```
make clean
```

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 11). 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.

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.

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 **close**, **format**, **get-char**, **open**, **print**, **println**, **printout**, **put-char**, **read**, **readline**, **read-number**, **rename**, **remove**, and **set-locale**. 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 **delete$**, **delete-member$**, **explode$**, **first$**, **foreach**, **implode$**, **insert$**, **member$**, **nth$**, **progn$**, **replace$**, **replace-member$**, **rest$**, **subseq$**, and **subsetp**. The functions **create$**, **expand$**, and **length$** are 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. 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. See section 11 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 **build**, **eval**, **lowcase**, **string-to-**

field, **str-cat**, **str-compare**, **str-index**, **str-length**, **sub-string**, **sym-cat**, and **upcase**. 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**, **get-region**, **print-region**, and **toss** 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. This is off in the standard CLIPS executable.

Section 3:
Core Functions

The core functions can be used to embed CLIPS within a simple C program for situations where the user interacts with CLIPS through a text-only computer interface and there is no need for the C program to retrieve information from CLIPS. This removes the need for users to have to interact in any way with the CLIPS command prompt.

3.1 Creating and Destroying Environments

3.1.1 CreateEnvironment

```
Environment *CreateEnvironment();
```

The function **CreateEnvironment** creates and initializes a CLIPS environment. A pointer of type **Environment *** is returned to identify the target for other functions which operate on environments. If any error occurs, a null pointer is returned.

3.1.2 DestroyEnvironment

```
bool DestroyEnvironment(
   Environment *env);
```

The function **DestroyEnvironment** deallocates all memory associated with an environment. Parameter **env** is a pointer to a previously created environment. This function returns true if successful; otherwise, it returns false. It should not be called to destroy an environment that is currently executing.

3.2 Loading Constructs

3.2.1 Clear

```
bool Clear(
   Environment *env);
```

The function **Clear** is the C equivalent of the CLIPS **clear** command. Parameter **env** is a pointer to a previously created environment. This function removes all constructs and associated data from the specified environment. It returns true if successful; otherwise, it returns false.

3.2.2 Load

```
LoadError Load(
   Environment *env,
   const char *fileName);

typedef enum
  {
   LE_NO_ERROR,
   LE_OPEN_FILE_ERROR,
   LE_PARSING_ERROR,
  } LoadError;
```

The function **Load** is the C equivalent of the CLIPS **load** command. Parameter **env** is a pointer to a previously created environment; and parameter **fileName** is a full or partial path string to an ASCII or UTF-8 text file containing CLIPS constructs. This function returns LE_OPEN_FILE_ERROR if an error occurred opening the file; LE_PARSING_ERROR if errors occurred while parsing constructs contained in the file; and LE_NO_ERROR if no errors occurred.

3.3 Creating and Removing Facts and Instances

3.3.1 AssertString

```
Fact *AssertString(
   Environment *env,
   const char *str);

AssertStringError GetAssertStringError(
   Environment *env);

typedef enum
  {
   ASE_NO_ERROR,
   ASE_NULL_POINTER_ERROR,
   ASE_PARSING_ERROR,
   ASE_COULD_NOT_ASSERT_ERROR,
   ASE_RULE_NETWORK_ERROR
  } AssertStringError;
```

The function **AssertString** is the C equivalent of the CLIPS **assert-string** command. Parameter **env** is a pointer to a previously created environment; and parameter **str** is a pointer to a character array containing the text representation of an ordered or deftemplate fact. An

example ordered fact string is "(colors red green blue)". An example deftemplate fact string is "(person (name Fred Jones) (age 37))". A pointer of type **Fact *** is returned if a fact is successfully created or already exists; otherwise, a null pointer is returned. If the return value from **AssertString** is persistently stored in a variable or data structure for later reference, then the function **RetainFact** should be called to insure that the reference remains valid even if the fact has been retracted.

The function **GetAssertStringError** returns the error code for the last fact assertion. The value ASE_NO_ERROR indicates no error occurred; the value ASE_NULL_POINTER_ERROR indicates the **str** parameter was NULL; the value ASE_PARSING_ERROR indicates an error was encountered parsing the **str** parameter; the value ASE_COULD_NOT_ASSERT indicates the fact could not be asserted (such as when pattern matching of a fact or instance is already occurring); and the value ASE_RULE_NETWORK_ERROR indicates an error occurred while the assertion was being processed in the rule network.

3.3.2 MakeInstance

```
Instance *MakeInstance(
   Environment *env,
   const char *str);

typedef enum
  {
   MIE_NO_ERROR,
   MIE_NULL_POINTER_ERROR,
   MIE_PARSING_ERROR,
   MIE_COULD_NOT_CREATE_ERROR,
   MIE_RULE_NETWORK_ERROR
  } MakeInstanceError;

MakeInstanceError GetMakeInstanceError(
   Environment *env);
```

The function **MakeInstance** is the C equivalent of the CLIPS **make-instance** function. Parameter **env** is a pointer to a previously created environment; and parameter **str** is a pointer to a character array containing the text representation of an instance. Example instances strings are "([p1] of POINT)" and "(of POINT (x 1) (y 1))". Unlike the CLIPS **make-instance** function, slot overrides in the **instanceString** parameter to the function **MakeInstance** are restricted to constants; function calls are not permitted. This function returns a pointer of type **Instance *** if an instance is successfully created; otherwise, a null pointer it returned. If the return value from **MakeInstance** is persistently stored in a variable or data structure for later reference, then the function **RetainInstance** should be called to insure that the reference remains valid even if the instance has been deleted.

The function **GetMakeInstanceError** returns the error code for the last **MakeInstance** call. The value MIE_NO_ERROR indicates no error occurred; the value MIE_NULL_POINTER_ERROR indicates the **str** parameter was NULL; the value MIE_PARSING_ERROR indicates an error was encountered parsing the **str** parameter; the value MIE_COULD_NOT_CREATE indicates the instance could not be created (such as when pattern matching of a fact or instance is already occurring); and the value MIE_RULE_NETWORK_ERROR indicates an error occurred while the instance was being processed in the rule network.

3.3.3 Retract

```
RetractError Retract(
   Fact *f);

typedef enum
  {
   RE_NO_ERROR,
   RE_NULL_POINTER_ERROR,
   RE_COULD_NOT_RETRACT_ERROR,
   RE_RULE_NETWORK_ERROR
  } RetractError;
```

The function **Retract** is the C equivalent of the CLIPS **retract** command. Parameter **f** is the fact to be retracted. This function returns RE_NO_ERROR if the fact is successfully retracted; otherwise it returns RE_NULL_POINTER_ERROR if parameter **f** is NULL, RE_COULD_NOT_RETRACT_ERROR if the fact could not be retracted (such as when pattern matching of a fact or instance is already occurring), or RE_RULE_NETWORK_ERROR if an error occurs while the retraction is being processed in the rule network.

The caller of **Retract** is responsible for insuring that the fact passed as an argument is still valid. If a persistent reference to this fact was previously created using **RetainFact**, the function **ReleaseFact** should be called to remove that reference.

3.3.4 UnmakeInstance

```
UnmakeInstanceError UnmakeInstance(
   Instance *i);

typedef enum
  {
   UIE_NO_ERROR,
   UIE_NULL_POINTER_ERROR,
   UIE_COULD_NOT_DELETE_ERROR,
```

```
   UIE_DELETED_ERROR,
   UIE_RULE_NETWORK_ERROR
  } UnmakeInstanceError;
```

The function **UnmakeInstance** is the C equivalent of the CLIPS **unmake-instance** command. Parameter **i** is the instance to be deleted using message-passing. This function returns UIE_NO_ERROR if the instance is successfully deleted; otherwise it returns UIE_NULL_POINTER_ERROR if parameter **i** is NULL, UIE_COULD_NOT_DELETE_ERROR if the instance could not be deleted (such as when pattern matching of a fact or instance is already occurring), UIE_DELETED_ERROR if the instance has already been deleted, or UIE_RULE_NETWORK_ERROR if an error occurs while the deletion is being processed in the rule network.

The caller of **UnmakeInstance** is responsible for insuring that the instance passed as an argument is still valid. If a persistent reference to this instance was previously created using **RetainInstancee**, the function **ReleaseInstance** should be called to remove that reference.

3.4 Executing Rules

3.4.1 Reset

```
void Reset(
   Environment *env);
```

The function **Reset** is the C equivalent of the CLIPS **reset** command. Parameter **env** is a pointer to a previously created environment. This function removes all facts and instances; creates facts and instances defined in deffacts and definstances constructs; and resets the values of global variables in the specified environment.

3.4.2 Run

```
long long Run(
   Environment *env,
   long long limit);
```

The function **Run** is the C equivalent of the CLIPS **run** command. Parameter **env** is a pointer to a previously created environment; and parameter **limit** parameter is the maximum number of rules that will fire before the function returns. If the **limit** parameter value is negative, rules will fire until the agenda is empty. The return value of this function is the number of rules that were fired.

3.5 Debugging

3.5.1 DribbleOn and DribbleOff

```
bool DribbleOn(
   Environment *env,
   const char *fileName);

bool DribbleOff(
   Environment *env);
```

The function **DribbleOn** is the C equivalent of the CLIPS **dribble-on** command. Parameter **env** is a pointer to a previously created environment; and parameter **fileName** is a full or partial path string to the dribble file to be created. This function returns true if the dribble file is successfully opened; otherwise, it returns false.

The function **DribbleOff** is the C equivalent of the CLIPS **dribble-off** command. Parameter **env** is a pointer to a previously created environment. This function returns true if the dribble file is successfully closed; otherwise, it returns false.

3.5.2 Watch and Unwatch

```
void Watch(
   Environment *env,
   WatchItem item);

void Unwatch(
   Environment *env,
   WatchItem item);

typedef enum
  {
   ALL,
   FACTS,
   INSTANCES,
   SLOTS,
   RULES,
   ACTIVATIONS,
   MESSAGES,
   MESSAGE_HANDLERS,
   GENERIC_FUNCTIONS,
   METHODS,
   DEFFUNCTIONS,
```

```
   COMPILATIONS,
   STATISTICS,
   GLOBALS,
   FOCUS
  } WatchItem;
```

The function **Watch** is the C equivalent of the CLIPS **watch** command. The function **Unwatch** is the C equivalent of the CLIPS **unwatch** command. Parameter **env** is a pointer to a previously created environment; and parameter **item** is one of the specified **WatchItem** enumeration values to be enabled (for watch) or disabled (for unwatch). If the **ALL** enumeration value is specified, then all watch items will be enabled (for watch) or disabled (for unwatch).

3.6 Examples

3.6.1 Hello World

This example demonstrates how to load and run rules from a C program. The following output shows how you would typically perform this task from the CLIPS command prompt:

```
CLIPS> (load "hello.clp")
*
TRUE
CLIPS> (reset)
CLIPS> (run)
Hello World!
CLIPS>
```

To achieve the same result from a C program, first create a text file named *hello.clp* with the following contents:

```
(defrule hello
   =>
   (println "Hello World!"))
```

Next, change the contents of the main.c source file to the following:

```
#include "clips.h"

int main()
  {
   Environment *env;

   env = CreateEnvironment();

   // The file hello.clp must be in the same directory
   // as the CLIPS executable or you must specify the
```

```
   // full directory path as part of the file name.

   Load(env,"hello.clp");

   Reset(env);
   Run(env,-1);

   DestroyEnvironment(env);
  }
```

Finally, recompile the CLIPS source code to create an executable.

The following output will be produced when the program is run:

```
Hello World!
```

3.6.2 Debugging

This example demonstrates how to generate and capture debugging information from a C program. The following output shows how you would typically perform this task from the CLIPS command prompt:

```
CLIPS> (load sort.clp)
%*
TRUE
CLIPS> (watch facts)
CLIPS> (watch rules)
CLIPS> (watch activations)
CLIPS> (dribble-on "sort.dbg")
TRUE
CLIPS> (reset)
CLIPS> (assert (list (numbers 61 31 27 48)))
==> f-1     (list (numbers 61 31 27 48))
==> Activation 0      sort: f-1
==> Activation 0      sort: f-1
<Fact-1>
CLIPS> (run)
FIRE    1 sort: f-1
<== f-1     (list (numbers 61 31 27 48))
<== Activation 0      sort: f-1
==> f-1     (list (numbers 31 61 27 48))
==> Activation 0      sort: f-1
FIRE    2 sort: f-1
<== f-1     (list (numbers 31 61 27 48))
==> f-1     (list (numbers 31 27 61 48))
==> Activation 0      sort: f-1
==> Activation 0      sort: f-1
FIRE    3 sort: f-1
<== f-1     (list (numbers 31 27 61 48))
<== Activation 0      sort: f-1
```

```
==> f-1     (list (numbers 27 31 61 48))
==> Activation 0      sort: f-1
FIRE    4 sort: f-1
<== f-1     (list (numbers 27 31 61 48))
==> f-1     (list (numbers 27 31 48 61))
CLIPS> (dribble-off)
TRUE
CLIPS>
```

To achieve the same result from a C program, first create a text file named *sort.clp* with the following contents:

```
(deftemplate list
   (multislot numbers))

(defrule sort
   ?f <- (list (numbers $?b ?x ?y&:(> ?x ?y) $?e))
   =>
   (modify ?f (numbers ?b ?y ?x ?e)))
```

Next, change the contents of the main.c source file to the following:

```
#include "clips.h"

int main()
  {
   Environment *env;

   env = CreateEnvironment();

   Load(env,"sort.clp");

   Watch(env,FACTS);
   Watch(env,RULES);
   Watch(env,ACTIVATIONS);

   DribbleOn(env,"sort.dbg");

   Reset(env);

   AssertString(env,"(list (numbers 61 31 27 48))");

   Run(env,-1);

   DribbleOff(env);

   DestroyEnvironment(env);
  }
```

Finally, recompile the CLIPS source code to create an executable.

The following output will be produced when the program is run:

```
==> f-1     (list (numbers 61 31 27 48))
==> Activation 0      sort: f-1
==> Activation 0      sort: f-1
FIRE    1 sort: f-1
<== f-1     (list (numbers 61 31 27 48))
<== Activation 0      sort: f-1
==> f-2     (list (numbers 31 61 27 48))
==> Activation 0      sort: f-2
FIRE    2 sort: f-2
<== f-2     (list (numbers 31 61 27 48))
==> f-3     (list (numbers 31 27 61 48))
==> Activation 0      sort: f-3
==> Activation 0      sort: f-3
FIRE    3 sort: f-3
<== f-3     (list (numbers 31 27 61 48))
<== Activation 0      sort: f-3
==> f-4     (list (numbers 27 31 61 48))
==> Activation 0      sort: f-4
FIRE    4 sort: f-4
<== f-4     (list (numbers 27 31 61 48))
==> f-5     (list (numbers 27 31 48 61))
```

The file *sort.dbg* will contain the same output that is printed to the screen.

Section 4:
Calling Functions and Building Constructs

The **Eval** function provides a mechanism for executing functions and commands in CLIPS and returning a value from CLIPS back to C. This is useful for executing commands from C in a similar manner to using the CLIPS command prompt. In conjunction with the fact and instance query functions, it is also a useful mechanism for retrieving the results of a CLIPS program.

The **Build** function provides a mechanism for defining individual constructs in a similar manner to using the CLIPS command prompt. Much like entering constructs at the command prompt, this functionality is primarily useful in examples and tutorials.

Since many CLIPS functions (including **Eval** and **Build**) take string arguments that may need to be created dynamically, CLIPS provides **StringBuilder** functions that automate the allocation, construction, and resizing of strings as character data is appended.

4.1 CLIPS Primitive Values

CLIPS wraps the underlying C representation of its primitive data types within C structure types that share a common **header** field containing type information. This allows CLIPS primitive values to be passed as a single pointer that can be examined for type to determine the C primitive type stored in the structure.

There are nine C types for used to represent CLIPS primitive values: **TypeHeader** for any CLIPS primitive value; **CLIPSLexeme** for symbols, strings, and instance names, **CLIPSFloat** for floats; **CLIPSInteger** for integers; **CLIPSVoid** for void; **Fact** for facts; **Instance** for instances; **CLIPSExternalAddress** for external addresses; and **Multifield** for multifields.

4.1.1 TypeHeader

The C **TypeHeader** type is used to store the CLIPS primitive value type.

```
typedef struct typeHeader
  {
   unsigned short type;
  } TypeHeader;
```

The integer stored in the **type** field is one of the following predefined constants:

```
EXTERNAL_ADDRESS_TYPE
FACT_ADDRESS_TYPE
FLOAT_TYPE
INSTANCE_ADDRESS_TYPE
INSTANCE_NAME_TYPE
INTEGER_TYPE
MULTIFIELD_TYPE
STRING_TYPE
SYMBOL_TYPE
VOID_TYPE
```

4.1.2 CLIPSValue

The C **CLIPSValue** type encapsulates all of the CLIPS primitive types. Functions returning primitive values from CLIPS to C have parameters of type **CLIPSValue *** . The return value of the function is stored in the **CLIPSValue** structure allocated by caller. The **header** field of the **CLIPSValue** union can be examined to determine the CLIPS primitive value type and then the appropriate field from the **CLIPSValue** union can be examined to retrieve the C representation of the type.

```
typedef struct clipsValue
  {
   union
     {
      void *value;
      TypeHeader *header;
      CLIPSLexeme *lexemeValue;
      CLIPSFloat *floatValue;
      CLIPSInteger *integerValue;
      CLIPSVoid *voidValue;
      Fact *factValue;
      Instance *instanceValue;
      Multifield *multifieldValue;
      CLIPSExternalAddress *externalAddressValue;
     };
  } CLIPSValue;
```

4.1.3 Symbol, Strings, and Instance Names

The C **CLIPSLexeme** type is used to represent CLIPS symbol, string, and instance name primitive types.

```
typedef struct clipsLexeme
  {
   TypeHeader header;
   const char *contents;
```

```
  } CLIPSLexeme;
```

The **contents** field of the **CLIPSLexeme** contains the C string associated with the CLIPS primitive value. This value should not be changed by user code.

4.1.4 Integers

The C **CLIPSInteger** type is used to represent CLIPS integers.

```
typedef struct clipsInteger
  {
   TypeHeader header;
   long long contents;
  } CLIPSInteger;
```

The **contents** field of the **CLIPSInteger** contains the C long long associated with the CLIPS primitive value. This value should not be changed by user code.

4.1.5 Floats

The C **CLIPSFloat** type is used to represent CLIPS floats.

```
typedef struct clipsFloat
  {
   TypeHeader header;
   double contents;
  } CLIPSInteger;
```

The **contents** field of the **CLIPSFloat** contains the C double associated with the CLIPS primitive value. This value should not be changed by user code.

4.1.6 Multifields

The C **Multifield** type is used to represent CLIPS multifields.

```
typedef struct multifield
  {
   TypeHeader header;
   size_t length;
   CLIPSValue *contents;
  } Multifield;
```

The **length** field contains the number of CLIPS primitive values contained in the **Multifield** type. The **contents** field is a pointer to an array containing a number of CLIPValue structs that is specified by the length field.

4.1.7 Void

The **CLIPSVoid** type is used to represent the CLIPS void value.

```
typedef struct clipsVoid
  {
   TypeHeader header;
  } CLIPSVoid;
```

4.1.8 External Address

The **CLIPSExternalAddress** struct is used to represent CLIPS external addresses.

```
typedef struct clipsExternalAddress
  {
   TypeHeader header;
   void *contents;
  } CLIPSExternalAddress;
```

The **contents** field of the **CLIPSExternalAddress** contains the external address associated with the CLIPS primitive value. This value should not be changed by user code.

4.2 Eval and Build

```
EvalError Eval(
   Environment *env,
   const char *str,
   CLIPSValue *cv);

BuildError Build(
   Environment *env,
   const char *str);

typedef enum
  {
   EE_NO_ERROR,
   EE_PARSING_ERROR,
   EE_PROCESSING_ERROR
  } EvalError;

typedef enum
  {
   BE_NO_ERROR,
   BE_COULD_NOT_BUILD_ERROR,
```

```
   BE_CONSTRUCT_NOT_FOUND_ERROR,
   BE_PARSING_ERROR,
  } BuildError;
```

The function **Eval** is the C equivalent of the CLIPS **eval** command. The function **Build** is the C equivalent of the CLIPS **build** command. For both functions, the **env** parameter is a pointer to a previously created environment. The **str** parameter for the **Eval** function is a string containing a CLIPS command or function call; and for the **Build** function is a string containing a construct definition. If the **cv** parameter value for the **Eval** function is not a null pointer, then the return value of the CLIPS command or function call is stored in the **CLIPSValue** structure allocated by the caller and referenced by the pointer.

If no errors occur, the **Eval** function returns EE_NO_ERROR and the **Build** function returns BE_NO_ERROR. If a syntax error is encountered while parsing, the **Eval** function returns EE_PARSING_ERROR and the **Build** function returns BE_PARSING_ERROR. If the **Build** function does not recognize the construct name following the opening left parenthesis, it returns BE_CONSTRUCT_NOT_FOUND_ERROR. If constructs cannot be added (such as when pattern matching is active), the **Build** function returns BE_COULD_NOT_BUILD_ERROR. If an error occurs while executing the parsed expression, the **Eval** function returns EE_PROCESSING_ERROR.

4.3 FunctionCallBuilder Functions

The CLIPS **FunctionCallBuilder** functions provide a mechanism for dynamically calling CLIPS functions. It can be used in place of the simpler **Eval** function when arguments that cannot be represented as strings (such as fact and instance pointers) must be passed to a function. A **FunctionCallBuilder** is created using the **CreateFunctionCallBuilder** function. Arguments can be appended to the **FunctionCallBuilder** using the **SBAppend...** functions. A function can then be evaluated with the assigned arguments by calling the **FCBCall** function. To call a function with different parameters, the **FCBReset** function can be called to reset the **FunctionCallBuilder** to its initial state. Once it is no longer needed, the **FunctionCallBuilder** can be deallocated using the **FCBDispose** function.

```
FunctionCallBuilder *CreateFunctionCallBuilder(
  Environment *env,
  size_t capacity);

FunctionCallBuilderError FCBCall(
  FunctionCallBuilder *fcb,
  const char *functionName,
  CLIPSValue *cv);
```

```
void FCBReset(
  FunctionCallBuilder *fcb);

void FCBDispose(
  FunctionCallBuilder *fcb);

void FCBAppend(
  FunctionCallBuilder *mb,
  CLIPSValue *value);

void FCBAppendUDFValue(
  FunctionCallBuilder *fcb,
  UDFValue *);

void FCBAppendInteger(
  FunctionCallBuilder *fcb,
  long long value);

void FCBAppendFloat(
  FunctionCallBuilder *fcb,
  double value);

void FCBAppendSymbol(
  FunctionCallBuilder *fcb,
  const char *value);

void FCBAppendString(
  FunctionCallBuilder *fcb,
  const char *value);

void FCBAppendInstanceName(
  FunctionCallBuilder *fcb,
  const char *value);

void FCBAppendCLIPSInteger(
  FunctionCallBuilder *fcb,
  CLIPSInteger *value);

void FCBAppendCLIPSFloat(
  FunctionCallBuilder *fcb,
  CLIPSFloat *value);

void FCBAppendCLIPSLexeme(
  FunctionCallBuilder *fcb,
  CLIPSLexeme *value);
```

```
void FCBAppendFact(
  FunctionCallBuilder *fcb,
  Fact *value);

void FCBAppendInstance(
  FunctionCallBuilder *fcb,
  CLIPSValue *value);

void FCBAppendMultifield(
  FunctionCallBuilder *fcb,
  Multifield *value);

void FCBAppendCLIPSExternalAddress(
  FunctionCallBuilder *fcb,
  CLIPSExternalAddress *value);
```

The function **CreateFunctionCallBuilder** creates and initializes a value of type **FunctionCallBuilder**. Parameter **env** is a pointer to a previously created environment; and parameter **capacity** is the initial size of the array used by the **FunctionCallBuilder** for storing the function call arguments. The initial size does not limit the maximum number arguments that can be passed to a function. The capacity of the **FunctionCallBuilder** will be increased if the number of arguments becomes larger than the initial capacity. If successful, this function returns a pointer of type **FunctionCallBuilder *** to identify the target of other functions accepting a **FunctionCallBuilder** parameter; if any error occurs, a null pointer is returned.

The function **FCBCall** executes the function specified by the **functionName** parameter with the arguments that were previously appended to the **FunctionCallBuilder** specified by parameter **fcb**. If the parameter **cv** is not a NULL pointer, the return value of the executed function will be stored in the specified **CLIPSValue** structure.

If no errors occur, the **FCBCall** function returns FCBE_NO_ERROR. If an error occurred, the value FCBE_NULL_POINTER_ERROR indicates the **fcb** or **functionName** parameter was NULL; the value FCBE_FUNCTION_NOT_FOUND_ERROR indicates a function, deffunction, or generic function could not be found with the name specified by the **functionName** parameter; the value FCBE_INVALID_FUNCTION_ERROR indicates the function or command has a specialized parser (such as the **assert** command) and cannot be invoked; the value FCBE_ARGUMENT_COUNT_ERROR indicates the function was passed the incorrect number of arguments; the value FCBE_ARGUMENT_TYPE_ERROR indicates the function was passed an argument with an invalid type; and the value ASE_PROCESSING_ERROR indicates an error occurred while the function was being evaluated.

The function **FCBReset** resets the **FunctionCallBuilder** specified by parameter **fcb** to its initial capacity. Any arguments previously appended are removed.

The function **FCBDispose** deallocates all memory associated with previously allocated **FunctionCallBuilder** specified by parameter **fcb**.

The functions **FCBAppend**, **FCBAppendUDFValue**, **FCBAppendInteger**, **FCBAppendFloat**, **FCBAppendSymbol**, **FCBAppendString**, **FCBAppendInstanceName**, **FCBAppendCLIPSInteger**, **FCBAppendCLIPSFloat**, **FCBAppendCLIPSLexeme**, **FCBAppendFact**, **FCBAppendInstance**, **FCBAppendMultifield**, and **FCBAppendCLIPSExternalAddress** append the parameter **value** to the end of the argments being created by the **FunctionCallBuilder** specified by parameter **fcb**.

4.4 StringBuilder Functions

The CLIPS **StringBuilder** functions provide a mechanism for dynamically creating strings of varying length, automatically resizing the **content** output string as character data is added. A StringBuilder is created using the **CreateStringBuilder** function. Character data can then be append to the StringBuilder using the **SBAddChar** and **SBAppend** functions. To build additional strings, the **SBReset** function can be called to reset the **StringBuilder** to its initial state. Once it is no longer needed, the StringBuilder can be deallocated using the **SBDispose** function.

The **StringBuilder** type definition with public fields is:

```
typedef struct stringBuilder
  {
   char *contents;
   size_t length;
  } StringBuilder;
```

The **contents** field of the **StringBuilder** type is a pointer to a character array containing all of the characters that have been appended by calls to the **SBAddChar** and **SBAppend** functions. The value of the **contents** field can change if appending to the **StringBuilder** exceeds the current capacity, so it is recommended to always directly retrieve the **contents** field from the **StringBuilder** pointer. Use the **SBCopy** function to create a copy of the **contents** field if desired; otherwise, the **contents** field should never be directly modified.

The **length** field of the **StringBuilder** type contains the number of characters in the **contents** field not including the null character at the end of the string.

4.4.1 CreateStringBuilder

```
StringBuilder *CreateStringBuilder(
   Environment *env,
   size_t capacity);
```

The function **CreateStringBuilder** creates and initializes a value of type **StringBuilder**. Parameter **env** is a pointer to a previously created environment; and parameter **capacity** is the initial size of the character array used by the **StringBuilder** for constructing strings. The initial size does not limit the maximum size of the **contents** string. The capacity of the StringBuilder will be increased if the string size becomes larger than the initial capacity. If successful, this function returns a pointer of type **StringBuilder *** to identify the target of other functions accepting a **StringBuilder** parameter; if any error occurs, a null pointer is returned.

4.4.2 SBAddChar

```
void SBAddChar(
   StringBuilder *sb,
   int c);
```

The function **SBAddChar** appends the single character specified by parameter **c** to the **contents** string of the previously allocated **StringBuilder** specified by parameter **sb**. If the **c** parameter value is a backspace, then the last character of the **contents** string is removed.

4.4.3 SBAppend Functions

```
void SBAppend(
   StringBuilder *sb,
   const char *value);

void SBAppendInteger(
   StringBuilder *sb,
   long long value);

void SBAppendFloat(
   StringBuilder *sb,
   double value);
```

The **SBAppend** functions append the parameter **value** to the **contents** string of the previously allocated **StringBuilder** specified by parameter **sb**.

4.4.4 SBCopy

```
char *SBCopy(
   StringBuilder *sb);
```

The function **SBCopy** returns a copy of **contents** string of the previously allocated **StringBuffer** specified by parameter **sb**. The memory allocated for this string will not be freed by CLIPS, so it is necessary for the user's code to call the **free** C library function to deallocate the memory once it is no longer needed.

4.4.5 SBDispose

```
void SBDispose(
   StringBuilder *sb);
```

The function **SBDispose** deallocates all memory associated with previously allocated **StringBuilder** specified by parameter **sb**.

4.4.6 SBReset

```
void SBReset(
   StringBuilder *sb);
```

The function **SBReset** resets the **StringBuilder** specified by parameter **sb** to its initial capacity. The **contents** string is set to an empty string and the **length** of the **StringBuilder** is set to 0.

4.5 Examples

4.5.1 Debugging Revisited

This example reimplements the debugging example from section 3.6.2 but uses the **Build** function for adding constructs and the **Eval** function for issuing commands.

```
#include "clips.h"

int main()
  {
   Environment *env;

   env = CreateEnvironment();

   Build(env,"(deftemplate list"
             "   (multislot numbers))");
```

```
   Build(env,"(defrule sort"
             "   ?f <- (list (numbers $?b ?x ?y&:(> ?x ?y) $?e))"
             "   =>"
             "   (modify ?f (numbers ?b ?y ?x ?e)))");

   Eval(env,"(watch facts)",NULL);
   Eval(env,"(watch rules)",NULL);
   Eval(env,"(watch activations)",NULL);

   Eval(env,"(dribble-on sort.dbg)",NULL);

   Eval(env,"(reset)",NULL);

   Eval(env,"(assert (list (numbers 61 31 27 48)))",NULL);

   Eval(env,"(run)",NULL);

   Eval(env,"(dribble-off)",NULL);

   DestroyEnvironment(env);
  }
```

4.5.2 String Builder Function Call

This example illustrates using the **StringBuilder** and **Eval** functions to construct and evaluate a function call. The **PrintString** and **PrintCLIPSValue Router** functions (described in Section 9) are used to print the value and type of each field in the multifield return value.

```
#include "clips.h"

int main()
  {
   Environment *env;
   StringBuilder *sb;
   CLIPSValue cv;
   char *fullName;

   // Create an Environment
   // and StringBuilder.

   env = CreateEnvironment();
   sb = CreateStringBuilder(env,512);

   // Get the first name.

   Write(env,"First Name: ");
   Eval(env,"(read)",&cv);
   if (cv.header->type == SYMBOL_TYPE)
     { SBAppend(sb,cv.lexemeValue->contents); }
   else
```

```
      { SBAppend(sb,"John"); }

    SBAppend(sb," ");

    // Get the last name.

    Write(env,"Last Name: ");
    Eval(env,"(read)",&cv);
    if (cv.header->type == SYMBOL_TYPE)
      { SBAppend(sb,cv.lexemeValue->contents); }
    else
      { SBAppend(sb,"Doe"); }

    // Get a copy of the full name
    // constructed by the StringBuilder.

    fullName = SBCopy(sb);

    // Create a function call to convert
    // the full name to upper case.

    SBReset(sb);
    SBAppend(sb,"(upcase \"");
    SBAppend(sb,fullName);
    SBAppend(sb,"\")");

    // Evaluate the function call
    // and print the results.

    Eval(env,sb->contents,&cv);
    Write(env,"Result is ");
    Writeln(env,cv.lexemeValue->contents);

    // Free the fullName

    free(fullName);

    // Dispose of the StringBuilder
    // and the Environment.

    SBDispose(sb);
    DestroyEnvironment(env);
   }
```

The resulting output (with input in bold) is:

```
First Name: Sally
Last Name: Jones
Result is SALLY JONES
```

4.5.3 Multifield Iteration

This example illustrates iteration over the values contained in a **Multifield**.

```
#include "clips.h"

int main()
  {
   Environment *env;
   StringBuilder *sb;
   CLIPSValue cv;

   // Create an Environment
   // and StringBuilder.

   env = CreateEnvironment();
   sb = CreateStringBuilder(env,512);

   // Call the CLIPS readline function to
   // capture a list of values in a string.

   Write(env,"Enter a list of values: ");

   Eval(env,"(readline)",&cv);

   // Call the CLIPS create$ function to generate
   // a multifield value from the string.

   SBAppend(sb,"(create$ ");
   SBAppend(sb,cv.lexemeValue->contents);
   SBAppend(sb,")");

   Eval(env,sb->contents,&cv);

   // Iterate over each value in the
   // multifield and print its type.

   for (size_t i = 0; i < cv.multifieldValue->length; i++)
     {
      WriteCLIPSValue(env,STDOUT,&cv.multifieldValue->contents[i]);

      switch(cv.multifieldValue->contents[i].header->type)
        {
         case INTEGER_TYPE:
           Write(env," is an integer\n");
           break;

         case FLOAT_TYPE:
           Write(env," is a float\n");
           break;

         case STRING_TYPE:
           Write(env," is a string\n");
```

```
            break;

          case SYMBOL_TYPE:
            Write(env," is a symbol\n");
            break;

          case INSTANCE_NAME_TYPE:
            Write(env," is an instance name\n");
            break;
         }
      }

    // Dispose of the StringBuilder
    // and the Environment.

    SBDispose(sb);
    DestroyEnvironment(env);
   }
```

The resulting output (with input in bold) is:

```
Enter a list of values: a "b" [c] 1.2 3
a is a symbol
"b" is a string
[c] is an instance name
1.2 is a float
3 is an integer
```

4.5.4 Fact Query

This example illustrates how to use the **StringBuilder** and **Eval** functions to dynamically construct and assert a fact, retrieve values from CLIPS function calls, and use a fact query to retrieve a slot value from a fact after rules have executed.

The first section of code for this example (that includes the function **CreateNumbers**) demonstrates the construction of a **list** fact with a **numbers** slot containing zero or more integers. The number of integers to be created is specified by the **howMany** parameter.

```
#include "clips.h"

void CreateNumbers(Environment *,StringBuilder *,int);
void PrintNumbers(Environment *);

void CreateNumbers(
  Environment *env,
  StringBuilder *sb,
  int howMany)
  {
   CLIPSValue cv;
```

```
   // Append the opening parentheses
   // for the fact and the slot.

   SBAppend(sb,"(list (numbers");

   // Loop adding the specified
   // number of random integers

   for (int i = 0; i < howMany; i++)
     {
      // Generate a random number in the
      // range 0 - 99. Convert the integer
      // on the CLIPS side to a symbol.

      Eval(env,"(sym-cat (random 0 99))",&cv);

      // Add the string value of the
      // integer to the slot.

      SBAppend(sb," ");
      SBAppend(sb,cv.lexemeValue->contents);
     }

   // Append the closing parentheses
   // for the slot and the fact.

   SBAppend(sb,"))");

   // Assert the fact.

   Watch(env,FACTS);
   AssertString(env,sb->contents);
   Unwatch(env,FACTS);

   // Clear the StringBuilder.

   SBReset(sb);
  }
```

The next section of code (that includes the **PrintNumbers** function) demonstrates how to use a query function to retrieve a slot value from a fact; and how to iterate through the contents of a multifield value. The **Write** and **WriteCLIPSValue Router** functions (described in Section 9) are used to print the slot value.

```
void PrintNumbers(
  Environment *env)
  {
   CLIPSValue cv;

   // This do-for-fact query call will find the first list
   // fact -- there should just be one -- and return the
```

```
    // value of the numbers slot in the variable cv.

    Eval(env,"(do-for-fact ((?f list)) TRUE ?f:numbers)",&cv);

    // The numbers slot should be a multifield value.

    if (cv.header->type == MULTIFIELD_TYPE)
      {
       Write(env,"Sorted list is (");

       // Iterate over each value in the
       // multifield and print it.

       for (size_t i = 0; i < cv.multifieldValue->length; i++)
         {
          if (i != 0) Write(env," ");
          WriteCLIPSValue(env,STDOUT,&cv.multifieldValue->contents[i]);
         }

       Write(env,")\n");
      }
  }
```

The final section of code (that includes the **main** function) generates a list of random numbers (using the **CreateNumbers** function), sorts them (using the **sort** rule), prints the sorted numbers (using the **PrintNumbers** function), and then repeats the process a second time.

```
int main()
  {
   Environment *env;
   StringBuilder *sb;

   // Create an Environment
   // and StringBuilder.

   env = CreateEnvironment();
   sb = CreateStringBuilder(env,512);

   // Seed the random number generator so that
   // different numeric values are generated
   // each time the program is run.

   Eval(env,"(seed (integer (time)))",NULL);

   // Create the sorting constructs

   Build(env,"(deftemplate list"
             "   (multislot numbers))");

   Build(env,"(defrule sort"
             "   ?f <- (list (numbers $?b ?x ?y&:(> ?x ?y) $?e))"
             "   =>"
```

```
            "   (modify ?f (numbers ?b ?y ?x ?e)))");

   // Create a list, sort it,
   // and print the results.

   Reset(env);
   CreateNumbers(env,sb,5);
   Run(env,-1);
   PrintNumbers(env);

   // Create another list, sort
   // it, and print the results.

   Reset(env);
   CreateNumbers(env,sb,7);
   Run(env,-1);
   PrintNumbers(env);

   // Dispose of the StringBuilder
   // and the Environment.

   SBDispose(sb);
   DestroyEnvironment(env);
  }
```

The resulting output is:

```
==> f-1     (list (numbers 43 76 3 55 87))
Sorted list is (3 43 55 76 87)
==> f-1     (list (numbers 73 17 7 84 68 54 67))
Sorted list is (7 17 54 67 68 73 84)
```

Note that the sorted integers displayed will vary since the generated random integers are dependent on the implementation of the C **rand** library function as well as the seeding of the random number generator using the current time.

4.5.5 Function Call Builder

This example illustrates using the **FunctionCallBuilder** functions to send an instance a print message.

```
#include "clips.h"

int main()
  {
   Environment *env;
   FunctionCallBuilder *fcb;
   Instance *ins;

   env = CreateEnvironment();
```

```
   Build(env,"(defclass POINT (is-a USER) (slot x) (slot y))");
   ins = MakeInstance(env,"([p1] of POINT (x 3) (y 4))");

   fcb = CreateFunctionCallBuilder(env,2);

   FCBAppendInstance(fcb,ins);
   FCBAppendSymbol(fcb,"print");
   FCBCall(fcb,"send",NULL);

   FCBDispose(fcb);

   DestroyEnvironment(env);
  }
```

The resulting output is:

```
[p1] of POINT
(x 3)
(y 4)
```

Section 5:
Garbage Collection

5.1 Introduction

CLIPS primitive values (including those which have counterparts to C primitive values such as integer, floats, and strings) are represented using data structures. As a CLIPS program executes, it allocates memory for primitive values dynamically (such as when facts/instances are created or functions are evaluated). CLIPS automatically tracks references to these primitive values so that they can be deallocated once there are no longer any outstanding references to them. 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**.

If you use one of the interactive CLIPS executables, all garbage collection is handled automatically for you including garbage created when entering commands and by constructs which execute code (such as defrules and deffunctions).

Embedded applications, however, can generate garbage and trigger garbage collection when invoking certain API calls to CLIPS, so it is necessary to follow some guidelines when using the APIs to allow CLIPS to safely garbage collect data that is no longer needed and to prevent primitive values that are referenced by user code from being garbage collected. First, functions which can cause CLIPS code to be executed (such as Clear, Load, Reset, Run, Send, and Eval) can trigger garbage collection. Second, a primitive value returned through an API call (such as Eval) is not subject to garbage collection until a subsequent API call triggering garbage collection is invoked. Third, use the Retain API functions to create an outstanding reference to a primitive value and the Release API functions to remove an outstanding reference.

The following code illustrates the first two guidelines:

```
#include "clips.h"

int main()
  {
   Environment *env;
   CLIPSValue cv;
   CLIPSLexeme *sym1, *sym2;

   env = CreateEnvironment();

   Eval(env,"(sym-cat abc def)",&cv);
   sym1 = cv.lexemeValue;

   // Safe to refer to sym1 here. */
```

```
   Eval(env,"(sym-cat ghi jkl)",&cv);
   sym2 = cv.lexemeValue;

   // Not safe to refer to sym1 here.
   // Safe to refer to sym2 here.
  }
```

The first call to **Eval** triggers garbage collection, but since no data has been returned yet to the embedding program this does not cause any problems. The **lexemeValue** field of the **CLIPSLexeme** returned in the variable **cv** is assigned to the variable **sym1**. The **contents** field of this variable can be safely referenced because the returned value was excluded from garbage collection.

The second call to **Eval** also triggers garbage collection. In this case, however, the value returned by the prior call to **Eval** will be garbage collected as a result. Therefore it is not safe to reference the value stored in the variable **sym1** after this point. This is a problem if, for example, you want to compare the **contents** fields of variables **sym1** and **sym2**.

For float and integer primitive values, the **contents** field of the **CLIPSInteger** or **CLIPSFloat** structure can be directly copied to a variable if the value needs to be preserved, however for primitive types this problem can be corrected by using the Retain/Release APIs to inform CLIPS about values that should not be garbage collected. For example:

```
#include "clips.h"

int main()
  {
   Environment *env;
   CLIPSValue cv;
   CLIPSLexeme *sym1, *sym2;

   env = CreateEnvironment();

   Eval(env,"(sym-cat abc def)",&cv);
   sym1 = cv.lexemeValue;
   RetainLexeme(env,sym1);

   // Safe to refer to sym1 here. */

   Eval(env,"(sym-cat ghi jkl)",&cv);
   sym2 = cv.lexemeValue;

   // Safe to refer to sym1 here.
   // Safe to refer to sym2 here.

   ReleaseLexeme(env,sym1);

   // Not safe to refer to sym1 here.
   // Safe to refer to sym2 here.
  }
```

In this case, the **RetainLexeme** function is called to prevent the result of the first **Eval** call from being garbaged collected when the second **Eval** call is made. That result is protected until the call to **ReleaseLexeme** is made.

5.2 Retain and Release Functions

CLIPS provides numerous function for retaining and releasing primitive values. A primitive value can be retained multiple times and will not be garbage collected until a corresponding number of release function calls have been made.

```
void Retain(
   Environment *env,
   TypeHeader *value);

void RetainCV(
   Environment *env,
   CLIPSValue *value);

void RetainUDFV(
   Environment *env,
   UDFValue *value);

void RetainFact(
   Environment *env,
   Fact *value);

void RetainInstance(
   Environment *env,
   Instance *value);

void RetainMultifield(
   Environment *env,
   Multifield *value);

void RetainLexeme(
   Environment *env,
   CLIPSLexeme *value);

void RetainFloat(
   Environment *env,
   CLIPSFloat *value);

void RetainInteger(
```

```
void Release(
   Environment *env,
   TypeHeader *value);

void ReleaseCV(
   Environment *env,
   CLIPSValue *value);

void ReleaseUDFV(
   Environment *env,
   UDFValue *value);

void ReleaseFact(
   Environment *env,
   Fact *value);

void ReleaseInstance(
   Environment *env,
   Instance *value);

void ReleaseMultifield(
   Environment *env,
   Multifield *value);

void ReleaseLexeme(
   Environment *env,
   CLIPSLexeme *value);

void ReleaseFloat(
   Environment *env,
   CLIPSFLoat *value);

void ReleaseInteger(
```

```
   Environment *env,                          Environment *env,
   CLIPSInteger *value);                      CLIPSInteger *value);
```

5.3 Example

This example demonstrates how to retain a fact so that a subsequent call to retract the fact will produce the correct result is the fact is retracted by a rule.

```
#include "clips.h"

int main()
  {
   Environment *env;
   Fact *f1, *f2;

   env = CreateEnvironment();

   Build(env,"(deftemplate list"
             "   (multislot numbers))");

   Build(env,"(defrule sort"
             "   ?f <- (list (numbers $?b ?x ?y&:(> ?x ?y) $?e))"
             "   =>"
             "   (retract ?f) "
             "   (assert (list (numbers ?b ?y ?x ?e))))");

   // Create and retain two facts.
   // The first requires sorting.
   // The second does not require sorting.

   f1 = AssertString(env,"(list (numbers 61 31 27 48))");
   RetainFact(f1);

   f2 = AssertString(env,"(list (numbers 13 19 88 99))");
   RetainFact(f2);

   // Display facts before and after
   // the sort rule is executed.

   Eval(env,"(facts)",NULL);

   Run(env,-1);

   Eval(env,"(facts)",NULL);

   // Release and retract both facts.

   ReleaseFact(f1);
   Retract(f1);

   ReleaseFact(f2);
```

```
   Retract(f2);

   // Display remaining facts.
   // Fact f-1 had already been retracted by the sort rule.
   // Fact f-2 was retracted by the "Retract(f2);" call.

   Eval(env,"(facts)",NULL);

   DestroyEnvironment(env);
  }
```

The resulting output is:

```
f-1     (list (numbers 61 31 27 48))
f-2     (list (numbers 13 19 88 99))
For a total of 2 facts.
f-2     (list (numbers 13 19 88 99))
f-6     (list (numbers 27 31 48 61))
For a total of 2 facts.
f-6     (list (numbers 27 31 48 61))
For a total of 1 fact.
```

If the fact stored in the variable **f1** had not been retained, the memory allocated for that fact could have been reallocated to store another fact (such as f-6). In that case, the "Retract(f1);" function call would have retracted the wrong fact rather than recognizing that the fact f-1 had already been retracted.

Section 6:
Creating Primitive Values

Section 4 demonstrated how to examine primitive values returned by CLIPS. This section documents the API for dynamically creating primitive values.

6.1 Primitive Creation Functions

CLIPS uses hash tables to store all integer, float, symbol, string, and instance name primitives. These hash tables are used to prevent the duplication of primitive values referenced multiple times. Attempting to create one of these primitives values that already exists returns a pointer to the existing data structure for that primitive value.

6.1.1 Creating CLIPS Symbol, Strings, and Instance Names

```
CLIPSLexeme *CreateSymbol(
   Environment *env,
   const char *str);

CLIPSLexeme *CreateString(
   Environment *env,
   const char *str);

CLIPSLexeme *CreateInstanceName(
   Environment *env,
   const char *str);

CLIPSLexeme *CreateBoolean(
   Environment *env,
   bool b);

CLIPSLexeme *FalseSymbol(
   Environment *env);

CLIPSLexeme *TrueSymbol(
   Environment *env);
```

The functions **CreateSymbol**, **CreateString**, and **CreateInstanceName** create primitive values with **type** field values of SYMBOL_TYPE, STRING_TYPE, and INSTANCE_NAME_TYPE respectively. Parameter **env** is a pointer to a previously created

environment; and parameter **str** is a pointer to a character array containing the text that will be assigned to the **contents** field of the CLIPS symbol, string, or instance name being created. The return value of these functions is a pointer to a **CLIPSLexeme** type.

The function **CreateBoolean** creates a primitive value with **type** field value of SYMBOL_TYPE. Parameter **env** is a pointer to a previously created environment. If parameter **b** is true, a pointer to the **CLIPSLexeme** for the symbol TRUE is returned; otherwise a pointer to the **CLIPSLexeme** for the symbol FALSE is returned.

The function **FalseSymbol** returns a pointer to the **CLIPSLexeme** for the symbol FALSE. The function **TrueSymbol** returns a pointer to the **CLIPSLexeme** for the symbol TRUE.

6.1.2 Creating CLIPS Integers

```
CLIPSInteger *CreateInteger(
   Environment *env,
   long long ll);
```

The function **CreateInteger** creates a primitive value with a **type** field value of INTEGER_TYPE. Parameter **env** is a pointer to a previously created environment; and parameter **ll** is the C integer value that will be assigned to the **contents** field of the CLIPS integer being created. The return value of this function is a pointer to a **CLIPSInteger** type.

6.1.3 Creating CLIPS Floats

```
CLIPSFloat *CreateFloat(
   Environment *theEnv,
   double dbl);
```

The function **CreateFloat** creates a primitive value with a **type** field value of FLOAT_TYPE. Parameter **env** is a pointer to a previously created environment; and parameter **dbl** is the C double value that will be assigned to the **contents** field of the CLIPS float being created. The return value of this function is a pointer to a **CLIPSFloat** type.

6.1.4 Creating Multifields

```
Multifield *EmptyMultifield(
   Environment *env);

Multifield *StringToMultifield(
   Environment *env,
   const char *str);
```

```
MultifieldBuilder *CreateMultifieldBuilder(
  Environment *env,
  size_t capacity);

Multifield *MBCreate(
  MultifieldBuilder *mb);

void MBReset(
  MultifieldBuilder *mb);

void MBDispose(
  MultifieldBuilder *mb);

void MBAppend(
  MultifieldBuilder *mb,
  CLIPSValue *value);

void MBAppendUDFValue(
  MultifieldBuilder *mb,
  UDFValue *);

void MBAppendInteger(
  MultifieldBuilder *mb,
  long long value);

void MBAppendFloat(
  MultifieldBuilder *mb,
  double value);

void MBAppendSymbol(
  MultifieldBuilder *mb,
  const char *value);

void MBAppendString(
  MultifieldBuilder *mb,
  const char *value);

void MBAppendInstanceName(
  MultifieldBuilder *mb,
  const char *value);

void MBAppendCLIPSInteger(
  MultifieldBuilder *mb,
  CLIPSInteger *value);
```

```
void MBAppendCLIPSFloat(
  MultifieldBuilder *mb,
  CLIPSFloat *value);

void MBAppendCLIPSLexeme(
  MultifieldBuilder *mb,
  CLIPSLexeme *value);

void MBAppendFact(
  MultifieldBuilder *mb,
  Fact *value);

void MBAppendInstance(
  MultifieldBuilder *mb,
  CLIPSValue *value);

void MBAppendMultifield(
  MultifieldBuilder *mb,
  Multifield *value);

void MBAppendCLIPSExternalAddress(
  MultifieldBuilder *mb,
  CLIPSExternalAddress *value);
```

The function **EmptyMultifield** returns a multifield primitive value of length 0.

The function **StringToMultifield** parses and creates a **Multifield** value from the values contained in the parameter **str**. For example, if the **str** parameter value is "1 4.5 c", a multifield with three values—the integer **1**, the float **4.5**, and the symbol **c**—will be created.

The function **CreateMultifieldBuilder** creates and initializes a value of type **MultifieldBuilder**. Parameter **env** is a pointer to a previously created environment; and parameter **capacity** is the initial size of the array used by the **MultifieldBuilder** for constructing multifields. The initial size does not limit the maximum size of the multifield that can be created. The capacity of the **MultifieldBuilder** will be increased if the multifield size becomes larger than the initial capacity. If successful, this function returns a pointer of type **MultifieldBuilder *** to identify the target of other functions accepting a **MultifieldBuilder** parameter; if any error occurs, a null pointer is returned.

The function **MBCreate** creates and returns a **Multifield** based on values appended to the **MultifieldBuilder** specified by parameter **mb**. The length of the **MultifieldBuilder** is reset to 0 after this function is called.

The function **MBReset** resets the **MultifieldBuilder** specified by parameter **mb** to its initial capacity. Any values previously appended are removed and the **length** of the **MultifieldBuilder** is set to 0.

The function **MBDispose** deallocates all memory associated with previously allocated **MultifieldBuilder** specified by parameter **mb**.

The functions **MBAppend**, **MBAppendUDFValue**, **MBAppendInteger**, **MBAppendFloat**, **MBAppendSymbol**, **MBAppendString**, **MBAppendInstanceName**, **MBAppendCLIPSInteger**, **MBAppendCLIPSFloat**, **MBAppendCLIPSLexeme**, **MBAppendFact**, **MBAppendInstance**, **MBAppendMultifield**, and **MBAppendCLIPSExternalAddress** append the parameter **value** to the end of the multifield being created by the **MultifieldBuilder** specified by parameter **mb**. When appending a multifield value using **MBAppend**, **MBAppendUDFValue**, or **MBAppendMultifield**, each individual value within the multifield is appended to the multifield being created rather than the multifield being nested within the multifield being created.

6.1.5 The Void Value

```
CLIPSVoid *VoidConstant(
   Environment *env);
```

The function **VoidConstant** returns a pointer to the CLIPS void primitive value.

6.1.6 Creating External Addresses

```
CLIPSExternalAddress *CreateCExternalAddress(
   Environment *theEnv,
   void *ea);
```

Creates a CLIPS external address value from a C void pointer. Note that it is up to the user to make sure that external addresses remain valid within CLIPS.

6.2 Examples

6.2.1 StringToMultifield

This example illustrates how to create a multifield primitive value from a string.

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

```
int main()
  {
   Environment *env;
   Multifield *mf;

   env = CreateEnvironment();

   mf = StringToMultifield(env,"\"abc\" 3 4.5");

   Write(env,"Created multifield is ");
   WriteMultifield(env,STDOUT,mf);
   Write(env,"\n");

   DestroyEnvironment(env);
  }
```

The resulting output is:

```
Created multifield is ("abc" 3 4.5)
```

6.2.2 MultifieldBuilder

This example demonstrates how to create multifield primitive values using a **MultifieldBuilder**:

```
#include "clips.h"

int main()
  {
   Environment *env;
   MultifieldBuilder *mb;
   Multifield *mf;

   env = CreateEnvironment();

   mb = CreateMultifieldBuilder(env,10);

   MBAppendString(mb,"abc");
   MBAppendInt(mb,3);
   MBAppendFloat(mb,4.5);

   mf = MBCreate(mb);

   Write(env,"Created multifield is ");
   WriteMultifield(env,STDOUT,mf);
   Write(env,"\n");

   MBAppendSymbol(mb,"def");
   MBAppendInstanceName(mb,"i1");

   mf = MBCreate(mb);
```

```
   Write(env,"Created multifield is ");
   WriteMultifield(env,STDOUT,mf);
   Write(env,"\n");

   MBDispose(mb);

   DestroyEnvironment(env);
  }
```

The resulting output is:

```
Created multifield is ("abc" 3 4.5)
Created multifield is (def [i1])
```

Section 7:
Creating and Modifying Facts and Instances

This section documents the **FactBuilder**, **FactModifier**, **InstanceBuilder**, and **InstanceModifier** APIs.

7.1 FactBuilder Functions

The **FactBuilder** functions provide a mechanism for dynamically creating facts. A **FactBuilder** is created using the **CreateFactBuilder** function. The slot assignment functions described in section 7.5 are used to assign slot values to the fact being created. Once slots have been assigned, the **FBAssert** function can be used to assert the fact and then reset the **FactBuilder** to its initial state. Alternately, the **FBAbort** function can be called to cancel the creation of the current fact and reset the **FactBuilder** to its initial state. The **FBSetDeftemplate** function can be called to initialize the **FactBuilder** to create facts of a different deftemplate type. Once it is no longer needed, the **FactBuilder** can be deallocated using the **FBDispose** function.

The **FactBuilder** type definition is:

```
typedef struct factBuilder
  {
  } FactBuilder;
```

The prototypes for the **FactBuilder** functions are:

```
FactBuilder *CreateFactBuilder(
   Environment *env,
   const char *name);

Fact *FBAssert(
   FactBuilder *fb);

void FBDispose(
   FactBuilder *fb);
```

```
FactBuilderError FBSetDeftemplate(
   FactBuilder *fb,
   const char *name);

void FBAbort(
   FactBuilder *fb);

FactBuilderError FBError (
   Environment *env);

typedef enum
  {
   FBE_NO_ERROR,
   FBE_NULL_POINTER_ERROR,
   FBE_DEFTEMPLATE_NOT_FOUND_ERROR,
   FBE_IMPLIED_DEFTEMPLATE_ERROR,
   FBE_COULD_NOT_ASSERT_ERROR,
   FBE_RULE_NETWORK_ERROR
  } FactBuilderError;
```

The function **CreateFactBuilder** allocates and initializes a struct of type **FactBuilder**. Parameter **env** is a pointer to a previously created environment; and parameter **name** is the name of the deftemplate that will be created by the fact builder. Only deftemplates that have been explicitly defined can by used with a **FactBuilder**. The **name** parameter can be NULL in which case the **FBSetDeftemplate** function must be used to assign the deftemplate before slot values can be assigned and facts can be asserted.

If successful, this function returns a pointer to the created **FactBuilder**; otherwise, it returns a null pointer. The error code for the function call be retrieved using the **FBError** function. The value FBE_NO_ERROR indicates no error occurred; the value FBE_DEFTEMPLATE_NOT_FOUND_ERROR indicates the specified deftemplate is either not in scope in the current module or does not exist; and the value FBE_IMPLIED_DEFTEMPLATE_ERROR indicates the specified deftemplate was not explicitly defined.

The function **FBAssert** asserts the fact based on slot assignments made to the fact builder specified by parameter **fb**. Slots which have not been explicitly assigned a value are set to their default value. If successful, this function returns a pointer to the asserted **Fact**; otherwise it returns a null pointer. Slot assignments are discarded after the fact is asserted, so slot values need to be reassigned if the fact builder is used to build another fact. The error code for the function call be retrieved using the **FBError** function. The value FBE_NO_ERROR indicates no error occurred; the value FBE_NULL_POINTER_ERROR indicates the **FactBuilder** does not have an associated deftemplate; the value FBE_COULD_NOT_ASSERT_ERROR indicates the fact could not be asserted (such as when pattern matching of a fact or instance is already occurring);

and the value FBE_RULE_NETWORK_ERROR indicates an error occurred while the assertion was being processed in the rule network.

The function **FBDispose** deallocates the memory associated with the **FactBuilder** specified by parameter **fb**.

The function **FBSetDeftemplate** changes the type of fact created by the fact builder to the deftemplate specified by the parameter **name**. Any slot values that have been assigned to the builder are discarded. This function returns FBE_NO_ERROR if the deftemplate is successfully set; otherwise it returns FBE_NULL_POINTER_ERROR if parameter **fb** is NULL, FBE_DEFTEMPLATE_NOT_FOUND_ERROR if the deftemplate is not in scope in the current module or does not exist, and FBE_IMPLIED_DEFTEMPLATE_ERROR if the deftemplate was not explicitly defined.

The function **FBAbort** discards the slot value assignments that have been made for the fact builder specified by parameter **fb**.

7.2 FactModifier Functions

The **FactModifier** functions provide a mechanism for dynamically modifying facts. A **FactModifier** is created using the **CreateFactModifier** function. The slot assignment functions described in section 7.5 are used to assign slot values to the fact being modified. Once slots have been assigned, the **FMModify** function can be used to modify the fact and then reset the **FactModifier** to its initial state. Alternately, the **FMAbort** function can be called to cancel the modification of the current fact and reset the **FactModifier** to its initial state. The **FMSetFact** function can be called to initialize the **FactModifier** to modify a different fact. Once it is no longer needed, the **FactModifier** can be deallocated using the **FMDispose** function.

The **FactModifier** type definition is:

```
typedef struct factModifier
  {
  } FactModifier;
```

The prototypes for the **FactModifier** functions are:

```
FactModifier *CreateFactModifier(
   Environment *env,
   Fact *f);
```

```
Fact *FMModify(
   FactModifier *fm);

void FMDispose(
   FactModifier *fm);

bool FMSetFact(
   FactModifier *fm,
   Fact *f);

void FMAbort(
   FactModifier *fm);

FactModifierError FMError (
   Environment *env);

typedef enum
  {
   FME_NO_ERROR,
   FME_NULL_POINTER_ERROR,
   FME_RETRACTED_ERROR,
   FME_IMPLIED_DEFTEMPLATE_ERROR,
   FME_COULD_NOT_MODIFY_ERROR,
   FME_RULE_NETWORK_ERROR
  } FactModifierError;
```

The function **CreateFactModifier** allocates and initializes a struct of type **FactModifier**. Parameter **env** is a pointer to a previously created environment; and parameter **f** is a pointer to the **Fact** to be modified. The **f** parameter can be NULL in which case the **FMSetFact** function must be used to assign the fact before slot values can be assigned and the fact modified.

If successful, this function returns a pointer to the created **FactModifier**; otherwise, it returns a null pointer. The error code for the function call be retrieved using the **FMError** function. The value FME_NO_ERROR indicates no error occurred; the value FME_RETRACTED_ERROR indicates the specified fact to be modified has been retracted; and the value FBE_IMPLIED_DEFTEMPLATE_ERROR indicates the specified fact is associated with a deftemplate that was not explicitly defined.

The function **FMModify** modifies the fact based on slot assignments made to the fact modifier specified by parameter **fm**. If successful, this function returns a pointer to the modified **Fact**; otherwise it returns a null pointer. Slot assignments are discarded after the fact is asserted, so slot values need to be reassigned if the fact builder is used to modify another fact. The error code for the function call be retrieved using the **FMError** function. The value FME_NO_ERROR indicates no error occurred; the value FME_NULL_POINTER_ERROR indicates the

FactModifier does not have an associated fact; the value FME_RETRACTED indicates the fact is retracted and cannot be modified; the value FME_COULD_NOT_MODIFY indicates the fact could not be modified (such as when pattern matching of a fact or instance is already occurring); and the value FME_RULE_NETWORK_ERROR indicates an error occurred while the modification was being processed in the rule network.

The function **FMDispose** deallocates the memory associated with the **FactModifier** specified by parameter **fm**.

The function **FMSetFact** changes the fact being modified to the value specified by parameter **f**. Any slot values that have been assigned to the modifier are discarded. This function returns FME_NO_ERROR if the fact is successfully set; otherwise it returns FME_NULL_POINTER_ERROR if parameter **fm** is NULL, FME_RETRACTED_ERROR if the fact has been retracted and cannot be modified, and FME_IMPLIED_DEFTEMPLATE_ERROR if the specified fact is associated with a deftemplate that was not explicitly defined.

The function **FMAbort** discards the slot value assignments that have been made for the fact modifier specified by parameter **fm**.

7.3 InstanceBuilder Functions

The **InstanceBuilder** functions provide a mechanism for dynamically creating instances. An **InstanceBuilder** is created using the **CreateInstanceBuilder** function. The slot assignment functions described in section 7.5 are used to assign slot values to the instance being created. Once slots have been assigned, the **IBMake** function can be used to create the instance and then reset the **InstanceBuilder** to its initial state. Alternately, the **IBAbort** function can be called to cancel the creation of the current instance and reset the **InstanceBuilder** to its initial state. The **FBSetDefclass** function can be called to initialize the **InstanceBuilder** to create instances of a different defclass type. Once it is no longer needed, the **InstanceBuilder** can be deallocated using the **IBDispose** function.

The **InstanceBuilder** type definition is:

```
typedef struct instanceBuilder
  {
  } InstanceBuilder;
```

The prototypes for the **InstanceBuilder** functions are:

```
InstanceBuilder *CreateInstanceBuilder(
   Environment *env,
   const char *name);
```

```
Instance *IBMake(
   InstanceBuilder *ib,
   const char *name);

void IBDispose(
   InstanceBuilder *ib);

InstanceBuilderError IBSetDefclass(
   InstanceBuilder *ib,
   const char *name);

void IBAbort(
   InstanceBuilder *ib);

InstanceBuilderError IBError (
   Environment *env);

typedef enum
  {
   IBE_NO_ERROR,
   IBE_NULL_POINTER_ERROR,
   IBE_DEFCLASS_NOT_FOUND_ERROR,
   IBE_COULD_NOT_CREATE_ERROR,
   IBE_PROCESSING_ERROR
  } InstanceBuilderError;
```

The function **CreateInstanceBuilder** allocates and initializes a struct of type **InstanceBuilder**. Parameter **env** is a pointer to a previously created environment; and parameter **name** is the name of the instance defclass that will be created by the instance builder. The **name** parameter can be NULL in which case the **FBSetDefclass** function must be used to assign the defclass before slot values can be assigned and instances can be created.

If successful, this function returns a pointer to the created **InstanceBuilder**; otherwise, it returns a null pointer. The error code for the function call be retrieved using the **IBError** function. The value IBE_NO_ERROR indicates no error occurred and the value IBE_DEFCLASS_NOT_FOUND_ERROR indicates the specified defclass is either not in scope in the current module or does not exist.

The function **IBMake** creates the instance based on slot assignments made to the instance builder specified by parameter **ib**. Slots which have not been explicitly assigned a value are set to their default value. If the parameter **name** is a null pointer, then an instance name is generated for the newly created instance; otherwise, the **name** parameter value is used as the instance name. If successful, this function returns a pointer to the created **Instance**; otherwise it returns a

null pointer. Slot assignments are discarded after the instance is created, so slot values need to be reassigned if the instance builder is used to build another instance. The error code for the function call be retrieved using the **IBError** function. The value IBE_NO_ERROR indicates no error occurred; the value IBE_NULL_POINTER_ERROR indicates the **InstanceBuilder** does not have an associated defclass; the value IBE_COULD_NOT_CREATE indicates the instance could not be created (such as when pattern matching of a fact or instance is already occurring); and the value IBE_RULE_NETWORK_ERROR indicates an error occurred while the instance was being processed in the rule network

The function **IBDispose** deallocates the memory associated with the **InstanceBuilder** specified by parameter **ib**.

The function **IBSetDefclass** changes the type of instance created by the instance builder to the defclass specified by the parameter **name**. Any slot values that have been assigned to the builder are discarded. This function returns IBE_NO_ERROR if the defclass is successfully set; otherwise it returns IBE_NULL_POINTER_ERROR if parameter **ib** is NULL, and IBE_DEFCLASS_NOT_FOUND_ERROR if the defclass is not in scope in the current module or does not exist.

The function **IBAbort** discards the slot value assignments that have been made for the instance builder specified by parameter **ib**.

7.4 InstanceModifier Functions

The **InstanceModifier** functions provide a mechanism for dynamically modifying instances. An **InstanceModifier** is created using the **CreateInstanceModifier** function. The slot assignment functions described in section 7.5 are used to assign slot values to the instance being modified. Once slots have been assigned, the **IMModify** function can be used to modify the instance and then reset the **InstanceModifier** to its initial state. Alternately, the **IMAbort** function can be called to cancel the modification of the current instance and reset the **InstanceModifier** to its initial state. The **IMSetInstance** function can be called to initialize the **InstanceModifier** to modify a different instance. Once it is no longer needed, the **InstanceModifier** can be deallocated using the **IMDispose** function.

The **InstanceModifier** type definition is:

```
typedef struct instanceModifier
  {
  } InstanceModifier;
```

The prototypes for the **InstanceModifier** functions are:

```
InstanceModifier *CreateInstanceModifier(
   Environment *env,
   Instance *i);

Instance *IMModify(
   InstanceModifier *im);

void IMDispose(
   InstanceModifier *im);

InstanceModifierError IMSetInstance(
   InstanceModifier *im,
   Instance *i);

void IMAbort(
   InstanceModifier *im);

InstanceModifierError IMError (
   Environment *env);

typedef enum
  {
   IME_NO_ERROR,
   IME_NULL_POINTER_ERROR,
   IME_DELETED_ERROR,
   IME_COULD_NOT_MODIFY_ERROR,
   IME_RULE_NETWORK_ERROR
  } InstanceModifierError;
```

The function **CreateInstanceModifier** allocates and initializes a struct of type **InstanceModifier**. Parameter **env** is a pointer to a previously created environment; and parameter **i** is a pointer to the **Instance** to be modified. The **i** parameter can be NULL in which case the **IMSetInstance** function must be used to assign the instance before slot values can be assigned and instance modified.

If successful, this function returns a pointer to the created **InstanceModifier**; otherwise, it returns a null pointer. The error code for the function call be retrieved using the **IMError** function. The value IME_NO_ERROR indicates no error occurred and the value IME_DELETED_ERROR indicates the specified instance to be modified has been deleted.

The function **IMModify** modifies the instance based on slot assignments made to the instance modifier specified by parameter **im**. If successful, this function returns a pointer to the modified **Instance**; otherwise it returns a null pointer. Slot assignments are discarded after the instance is modified, so slot values need to be reassigned if the instance builder is used to modify a different instance. The error code for the function call be retrieved using the **IMError** function. The value

IME_NO_ERROR indicates no error occurred; the value IME_NULL_POINTER_ERROR indicates the **InstanceModifier** does not have an associated instance; the value IME_DELETED indicates the instance is deleted and cannot be modified; the value IME_COULD_NOT_MODIFY indicates the instance could not be modified (such as when pattern matching of a fact or instance is already occurring); and the value IME_RULE_NETWORK_ERROR indicates an error occurred while the modification was being processed in the rule network.

The function **IMDispose** deallocates the memory associated with the **InstanceModifier** specified by parameter **im**.

The function **IMSetInstance** changes the instance being modified to the value specified by parameter **i**. Any slot values that have been assigned to the modifier are discarded. This function returns IME_NO_ERROR if the instance is successfully set; otherwise it returns IME_NULL_POINTER_ERROR if parameter **im** is NULL or IME_DELETED_ERROR if the instance has been deleted and cannot be modified.

The function **IMAbort** discards the slot value assignments that have been made for the instance modifier specified by parameter **im**.

7.5 Slot Assignment Functions

The **FactBuilder**, **FactModifier**, **InstanceBuilder**, and **InstanceModifier** APIs each provide a set of functions for assigning slot values. The slot assignment functions return one of the following **PutSlotError** enumerations:

```
typedef enum
  {
   PSE_NO_ERROR,
   PSE_NULL_POINTER_ERROR,
   PSE_INVALID_TARGET_ERROR,
   PSE_SLOT_NOT_FOUND_ERROR,
   PSE_TYPE_ERROR,
   PSE_RANGE_ERROR,
   PSE_ALLOWED_VALUES_ERROR,
   PSE_CARDINALITY_ERROR,
   PSE_ALLOWED_CLASSES_ERROR
  } PutSlotError;
```

The PSE_NO_ERROR enumeration value indicates a successful slot assignment. The PSE_NULL_POINTER_ERROR enumeration value indicates that one of the function arguments was a NULL pointer. The PSE_INVALID_TARGET_ERROR enumeration value indicates that a fact/instance cannot be modified because the fact or instance pointer assigned to a FactModifier

or InstanceModifier has been deleted. The PSE_SLOT_NOT_FOUND_ERROR enumeration value indicates that fact or instance does not have the specified slot. The remaining enumeration values indicate that the specified slot value violates one of the constraints for the allowed types or values for the slot.

7.5.1 Assigning Generic Slot Values

```
PutSlotError FBPutSlot(
   FactBuilder *fb,
   const char *name,
   CLIPSValue *v);

PutSlotError FMPutSlot(
   FactModifier *fm,
   const char *name,
   CLIPSValue *v);
```

```
PutSlotError IBPutSlot(
   InstanceBuilder *ib,
   const char *name,
   CLIPSValue *v);

PutSlotError IMPutSlot(
   InstanceModifier *im,
   const char *name,
   CLIPSValue *v);
```

The function **FBPutSlot** sets the slot specified by the parameter **name** to the value specified by parameter **v** for the fact builder specified by parameter **fb**. This function returns true if the slot was successfully set; otherwise, it returns false.

The function **FMPutSlot** sets the slot specified by the parameter **name** to the value specified by parameter **v** for the fact modifier specified by parameter **fm**. This function returns true if the slot was successfully set; otherwise, it returns false.

The function **IBPutSlot** sets the slot specified by the parameter **name** to the value specified by parameter **v** for the instance builder specified by parameter **ib**. This function returns true if the slot was successfully set; otherwise, it returns false.

The function **IMPutSlot** sets the slot specified by the parameter **name** to the value specified by parameter **v** for the instance modifier specified by parameter **im**. This function returns true if the slot was successfully set; otherwise, it returns false.

7.5.2 Assigning Integer Slot Values

```
PutSlotError FBPutSlotCLIPSInteger(
   FactBuilder *fb,
   const char *name,
   CLIPSInteger *i);

PutSlotError FBPutSlotInteger(
   FactBuilder *fb,
   const char *name,
```

```
PutSlotError FMPutSlotCLIPSInteger(
   FactModifier *fm,
   const char *name,
   CLIPSInteger *i);

PutSlotError FMPutSlotInteger(
   FactModifier *fm,
   const char *name,
```

```
   long long i);

PutSlotError IBPutSlotCLIPSInteger(
   InstanceBuilder *ib,
   const char *name,
   CLIPSInteger *i);

PutSlotError IBPutSlotInteger(
   InstanceBuilder *ib,
   const char *name,
   long long i);
```

```
   long long i);

PutSlotError IMPutSlotCLIPSInteger(
   InstanceModifier *im,
   const char *name,
   CLIPSInteger *i);

PutSlotError IMPutSlotInteger(
   InstanceModifier *im,
   const char *name,
   long long i);
```

These functions assign an integer value in one of various forms to a **FactBuilder** (for parameter **fb**), **FactModifier** (for parameter **fm**), **InstanceBuilder** (for parameter ib), or **InstanceModifier** (for parameter **im**). Parameter **name** is the slot of either the fact or instance to be assigned. Parameter i is the value assigned to the slot: a **CLIPSInteger**, int, long, or long long value. These functions return true if the slot was successfully set; otherwise, they return false.

7.5.3 Assigning Float Slot Values

```
PutSlotError FBPutSlotCLIPSFloat(
   FactBuilder *fb,
   const char *name,
   CLIPSFloat *f);

PutSlotError FBPutSlotFloat(
   FactBuilder *fb,
   const char *name,
   double f);

PutSlotError IBPutSlotCLIPSFloat(
   InstanceBuilder *ib,
   const char *name,
   CLIPSFloat *f);

PutSlotError IBPutSlotFloat(
   InstanceBuilder *ib,
   const char *name,
   double f);
```

```
PutSlotError FMPutSlotCLIPSFloat(
   FactModifier *fm,
   const char *name,
   CLIPSFloat *f);

PutSlotError FMPutSlotFloat(
   FactModifier *fm,
   const char *name,
   double f);

PutSlotError IMPutSlotCLIPSFloat(
   InstanceModifier *im,
   const char *name,
   CLIPSFloat *f);

PutSlotError IMPutSlotFloat(
   InstanceModifier *im,
   const char *name,
   double f);
```

These functions assign a floating point value in one of various forms to a **FactBuilder** (for parameter **fb**), **FactModifier** (for parameter **fm**), **InstanceBuilder** (for parameter ib), or **InstanceModifier** (for parameter **im**). Parameter **name** is the slot of either the fact or instance

to be assigned. Parameter **f** is the value assigned to the slot: a **CLIPSFloat**, float, or double value. These functions return true if the slot was successfully set; otherwise, they return false.

7.5.4 Assigning Symbol, String, and Instance Name Slot Values

```
PutSlotError FBPutSlotCLIPSLexeme(
   FactBuilder *fb,
   const char *name,
   CLIPSLexeme *lex);

PutSlotError FBPutSlotInstanceName(
   FactBuilder *fb,
   const char *name,
   const char *lex);

PutSlotError FBPutSlotString(
   FactBuilder *fb,
   const char *name,
   const char *lex);

PutSlotError FBPutSlotSymbol(
   FactBuilder *fb,
   const char *name,
   const char *lex);

PutSlotError IBPutSlotCLIPSLexeme(
   InstanceBuilder *ib,
   const char *name,
   CLIPSLexeme *lex);

PutSlotError IBPutSlotInstanceName(
   InstanceBuilder *ib,
   const char *name,
   const char *lex);

PutSlotError IBPutSlotString(
   InstanceBuilder *ib,
   const char *name,
   const char *lex);

PutSlotError IBPutSlotSymbol(
   InstanceBuilder *ib,
   const char *name,
```

```
PutSlotError FMPutSlotCLIPSLexeme(
   FactModifier *fm,
   const char *name,
   CLIPSLexeme *lex);

PutSlotError FMPutSlotInstanceName(
   FactModifier *fm,
   const char *name,
   const char *lex);

PutSlotError FMPutSlotString(
   FactModifier *fm,
   const char *name,
   const char *lex);

PutSlotError FMPutSlotSymbol(
   FactModifier *fm,
   const char *name,
   const char *lex);

PutSlotError IMPutSlotCLIPSLexeme(
   InstanceModifier *im,
   const char *name,
   CLIPSLexeme *lex);

PutSlotError IMPutSlotInstanceName(
   InstanceModifier *im,
   const char *name,
   const char *lex);

PutSlotError IMPutSlotString(
   InstanceModifier *im,
   const char *name,
   const char *lex);

PutSlotError IMPutSlotSymbol(
   InstanceModifier *im,
   const char *name,
```

```
   const char *lex);                          const char *lex);
```

These functions assign a lexeme value (symbol, string, or instance name) in one of various forms to a **FactBuilder** (for parameter **fb**), **FactModifier** (for parameter **fm**), **InstanceBuilder** (for parameter ib), or **InstanceModifier** (for parameter **im**). Parameter **name** is the slot of either the fact or instance to be assigned. Parameter **lex** is the value assigned to the slot: a **CLIPSLexeme** or C string. These functions return true if the slot was successfully set; otherwise, they return false.

7.5.5 Assigning Fact and Instance Values

```
PutSlotError FBPutSlotFact(               PutSlotError IBPutSlotFact(
   FactBuilder *fb,                         InstanceBuilder *ib,
   const char *name,                        const char *name,
   Fact *f);                                Fact *f);

PutSlotError FBPutSlotInstance(           PutSlotError IBPutSlotInstance(
   FactBuilder *fb,                         InstanceBuilder *ib,
   const char *name,                        const char *name,
   Instance *i);                            Instance *i);

PutSlotError FMPutSlotFact(               PutSlotError IMPutSlotFact(
   FactModifier *fm,                        InstanceModifier *im,
   const char *name,                        const char *name,
   Fact *f);                                Fact *f);

PutSlotError FMPutSlotInstance(           PutSlotError IMPutSlotInstance(
   FactModifier *fm,                        InstanceModifier *im,
   const char *name,                        const char *name,
   Instance *i);                            Instance *i);
```

These functions assign a fact or instance value to a **FactBuilder** (for parameter **fb**), **FactModifier** (for parameter **fm**), **InstanceBuilder** (for parameter ib), or **InstanceModifier** (for parameter **im**). Parameter **name** is the slot of either the fact or instance to be assigned. Parameter **f** is the **Fact** value assigned to the slot; or parameter **i** is the **Instance** value assigned to the slot. These functions return true if the slot was successfully set; otherwise, they return false.

7.5.6 Assigning Multifield and External Address Slot Values

```
PutSlotError FBPutSlotExternalAddress(    PutSlotError IBPutSlotExternalAddress(
   FactBuilder *fb,                          InstanceBuilder *ib,
   const char *name,                         const char *name,
```

```
   CLIPSExternalAddress *ea);

PutSlotError FBPutSlotMultifield(
   FactBuilder *fb,
   const char *name,
   Multifield *mf);

PutSlotError FMPutSlotExternalAddress(
   FactModifier *fm,
   const char *name,
   CLIPSExternalAddress *ea);

PutSlotError FMPutSlotMultifield(
   FactModifier *fm,
   const char *name,
   Multifield *mf);
```

```
   CLIPSExternalAddress *ea);

PutSlotError IBPutSlotMultifield(
   InstanceBuilder *ib,
   const char *name,
   Multifield *mf);

PutSlotError IMPutSlotExternalAddress(
   FactModifier *im,
   const char *name,
   CLIPSExternalAddress *ea);

PutSlotError IMPutSlotMultifield(
   InstanceModifier *im,
   const char *name,
   Multifield *mf);
```

These functions assign an external address or multifield value to a **FactBuilder** (for parameter **fb**), **FactModifier** (for parameter **fm**), **InstanceBuilder** (for parameter ib), or **InstanceModifier** (for parameter **im**). Parameter **name** is the slot of either the fact or instance to be assigned. Parameter **ea** is the **CLIPSExternalAddress** value assigned to the slot; or parameter **mf** is the **Multifield** value assigned to the slot. These functions return true if the slot was successfully set; otherwise, they return false.

7.6 Examples

7.6.1 FactBuilder

This example illustrates use of the **FactBuilder** API.

```
#include "clips.h"

int main()
  {
   Environment *theEnv;
   FactBuilder *theFB;
   CLIPSValue cv;

   theEnv = CreateEnvironment();

   Build(theEnv,"(deftemplate person"
                "   (slot name)"
                "   (slot gender)"
                "   (slot age)"
                "   (slot marital-status (default single))"
                "   (multislot hobbies))");
```

```
   theFB = CreateFactBuilder(theEnv,"person");

   // Technique #1

   FBPutSlotString(theFB,"name","Mary Sue Smith");
   FBPutSlotSymbol(theFB,"gender","female");
   FBPutSlotInt(theFB,"age",25);
   FBAssert(theFB);

   // Technique #2

   FBPutSlotCLIPSLexeme(theFB,"name",CreateString(theEnv,"Sam Jones"));
   FBPutSlotCLIPSLexeme(theFB,"gender",CreateSymbol(theEnv,"male"));
   FBPutSlotCLIPSInteger(theFB,"age",CreateInteger(theEnv,48));
   FBPutSlotCLIPSLexeme(theFB,"marital-status",CreateSymbol(theEnv,"married"));
   FBPutSlotMultifield(theFB,"hobbies",StringToMultifield(theEnv,"reading skiing"));
   FBAssert(theFB);

   // Technique #3

   cv.lexemeValue = CreateString(theEnv,"John Doe");
   FBPutSlot(theFB,"name",&cv);
   cv.lexemeValue = CreateSymbol(theEnv,"male");
   FBPutSlot(theFB,"gender",&cv);
   cv.integerValue = CreateInteger(theEnv,73);
   FBPutSlot(theFB,"age",&cv);
   cv.lexemeValue = CreateSymbol(theEnv,"widowed");
   FBPutSlot(theFB,"marital-status",&cv);
   cv.multifieldValue = StringToMultifield(theEnv,"gardening");
   FBPutSlot(theFB,"hobbies",&cv);
   FBAssert(theFB);

   FBDispose(theFB);

   Eval(theEnv,"(do-for-all-facts ((?f person)) TRUE (ppfact ?f))",NULL);

   DestroyEnvironment(theEnv);
  }
```

The resulting output is:

```
(person
   (name "Mary Sue Smith")
   (gender female)
   (age 25)
   (marital-status single)
   (hobbies))
(person
   (name "Sam Jones")
   (gender male)
   (age 48)
   (marital-status married)
```

```
   (hobbies reading skiing))
(person
   (name "John Doe")
   (gender male)
   (age 73)
   (marital-status widowed)
   (hobbies gardening))
```

7.6.2 FactModifier

This example illustrates use of the **FactModifier** API.

```
#include "clips.h"

int main()
  {
   Environment *theEnv;
   FactModifier *theFM;
   Fact *theFact;

   theEnv = CreateEnvironment();

   Build(theEnv,"(deftemplate print"
                "   (slot value))");

   Build(theEnv,"(defrule print"
                "   (print (value ?v))"
                "   =>"
                "   (println ?v))");

   theFact = AssertString(theEnv,"(print (value \"Beginning\"))");
   Run(theEnv,-1);

   theFM = CreateFactModifier(theEnv,theFact);

   FMPutSlotString(theFM,"value","Middle");
   FMModify(theFM);
   Run(theEnv,-1);

   FMPutSlotString(theFM,"value","End");
   FMModify(theFM);
   Run(theEnv,-1);

   FMDispose(theFM);

   DestroyEnvironment(theEnv);
  }
```

The resulting output is:

```
Beginning
```

```
Middle
End
```

7.6.3 Fact Modifier with Referenced Facts

This example illustrates use of the **FactBuilder** and **FactModifier** APIs to iteratively change a group of facts.

```
#include "clips.h"

int main()
  {
   Environment *theEnv;
   FactBuilder *theFB;
   FactModifier *theFM;
   CLIPSValue cv;
   Fact *sensors[2];
   long long sensorValues[2] = { 6, 5 };

   theEnv = CreateEnvironment();

   // Create a deftemplate and defrule

   Build(theEnv,"(deftemplate sensor"
                "   (slot id)"
                "   (multislot range)"
                "   (slot value))");

   Build(theEnv,"(defrule sensor-value-out-of-range"
                "   (sensor (id ?id) (value ?value) (range ?lower ?upper))"
                "   (test (or (< ?value ?lower) (> ?value ?upper)))"
                "   =>"
                "   (println ?id \" value \" ?value "
                "            \" out of range \" ?lower \" - \" ?upper))");

   // Watch changes to facts

   Watch(theEnv,FACTS);

   // Create the facts

   theFB = CreateFactBuilder(theEnv,"sensor");

   FBPutSlotSymbol(theFB,"id","sensor-1");
   FBPutSlotMultifield(theFB,"range",StringToMultifield(theEnv,"4 8"));
   FBPutSlotInteger(theFB,"value",sensorValues[0]);
   sensors[0] = FBAssert(theFB);
   RetainFact(sensors[0]);

   FBPutSlotSymbol(theFB,"id","sensor-2");
   FBPutSlotMultifield(theFB,"range",StringToMultifield(theEnv,"2 7"));
   FBPutSlotInteger(theFB,"value",sensorValues[1]);
```

```
   sensors[1] = FBAssert(theFB);
   RetainFact(sensors[1]);

   FBDispose(theFB);

   // Seed the random number generator

   Eval(theEnv,"(seed (integer (time)))",NULL);

   // Create the FactModifier

   theFM = CreateFactModifier(theEnv,NULL);

   // Loop through 4 cycles of changes

   for (int cycle = 1; cycle < 5; cycle++)
     {
      Write(theEnv,"Cycle #");
      WriteInteger(theEnv,STDOUT,cycle);
      Write(theEnv,"\n");

      // Loop through each sensor

      for (int s = 0; s < 2; s++)
        {
         Fact *oldValue = sensors[s];

         FMSetFact(theFM,oldValue);

         // Change the sensor value

         Eval(theEnv,"(random -3 3)",&cv);
         sensorValues[s] += cv.integerValue->contents;

         FMPutSlotInteger(theFM,"value",sensorValues[s]);

         sensors[s] = FMModify(theFM);

         // Retain the new fact and release the old fact

         RetainFact(sensors[s]);
         ReleaseFact(oldValue);
        }

      // Execute Rules

      Run(theEnv,-1);
     }

   // Dispose of the FactModifier

   FMDispose(theFM);
```

```
   DestroyEnvironment(theEnv);
  }
```

The resulting output is:

```
==> f-1     (sensor (id sensor-1) (range 4 8) (value 6))
==> f-2     (sensor (id sensor-2) (range 2 7) (value 5))
Cycle #1
<== f-1     (sensor ... (value 6))
==> f-1     (sensor ... (value 9))
<== f-2     (sensor ... (value 5))
==> f-2     (sensor ... (value 3))
sensor-1 value 9 out of range 4 - 8
Cycle #2
<== f-1     (sensor ... (value 9))
==> f-1     (sensor ... (value 10))
sensor-1 value 10 out of range 4 - 8
Cycle #3
<== f-1     (sensor ... (value 10))
==> f-1     (sensor ... (value 13))
<== f-2     (sensor ... (value 3))
==> f-2     (sensor ... (value 2))
sensor-1 value 13 out of range 4 - 8
Cycle #4
<== f-1     (sensor ... (value 13))
==> f-1     (sensor ... (value 16))
<== f-2     (sensor ... (value 2))
==> f-2     (sensor ... (value 1))
sensor-2 value 1 out of range 2 - 7
sensor-1 value 16 out of range 4 - 8
```

Note that the specific values will vary because of calls to the random function.

Section 8:
User Defined Functions

CLIPS provides a collection of system defined functions and commands for a variety of purposes. In addition, the deffunction construct can be used to create new functions and commands within a CLIPS program. In some cases, however, it is necessary to integrate functions written in C with the CLIPS C source code. This may be for performance reasons; to integrate existing code written in C; or to integrate a C library.

Functions written in C that are integrated with CLIPS using the protocols described in this section are referred to as User Defined Functions (UDFs) and can be used in the same manner as system defined functions and commands. In fact, the system defined functions and commands provided by CLIPS are integrated using the protocols described in this section. Note that while the word 'command' is typically used throughout this documentation to refer to a function that has no return value, the protocols used to implement functions and commands are the same.

This section describes the protocols for registering UDFs, passing arguments to them, and returning values from them. Prototypes for the functions listed in this section can be included by using the **clips.h** header file.

8.1 User Defined Function Types

The interface between a function reference in CLIPS code and the C code which implements the function is handled by creating a function of type **UserDefinedFunction**:

```
typedef void UserDefinedFunction(
   Environment *env,
   UDFContext *udfc,
   UDFValue *out);

typedef struct udfContext
  {
   void *context;
  } UDFContext;

typedef struct udfValue
  {
   union
     {
      void *value;
```

```
      TypeHeader const *header;
      CLIPSLexeme *lexemeValue;
      CLIPSFloat *floatValue;
      CLIPSInteger *integerValue;
      CLIPSVoid *voidValue;
      Multifield *multifieldValue;
      Fact *factValue;
      Instance *instanceValue;
      CLIPSExternalAddress *externalAddressValue;
     };
   size_t begin;
   size_t range;
  } UDFValue;
```

A **UserDefinedFunction** is passed three parameters: **env** is a pointer to the **Environment** in which the UDF is executed; **udfc** is a pointer to a **UDFContext**; and **out** is a pointer to a **UDFValue**.

The **UDFContext** type contains the public field **context**, a pointer to user data supplied when the UDF is registered, as well as several private fields used to track UDF argument requests (through the **udfc** parameter value).

The **UDFValue** type is used both for returning a value from a UDF (through the **out** parameter value) as well as requesting argument values passed to the UDF. The **UDFValue** type is similar to the **CLIPSValue** type, but also includes **begin** and **range** fields. These fields allow you to manipulate multifield values within a UDF without creating a new multifield. The **begin** field represents the starting position and the **range** field represents the number of values within a multifield. For example, if a UDFValue contained the multifield (a b c d), setting the **begin** field to 1 and the **range** field to 2 would change the UDFValue to the multifield (b c).

8.2 Registering User Defined Functions

```
AddUDFError AddUDF(
   Environment *env,
   const char *clipsName,
   const char *returnTypes,
   unsigned short minArgs,
   unsigned short maxArgs,
   const char *argTypes,
   UserDefinedFunction *cfp,
   const char *cName,
   void *context);
```

```
typedef enum
  {
   AUE_NO_ERROR,
   AUE_MIN_EXCEEDS_MAX,
   AUE_FUNCTION_NAME_IN_USE,
   AUE_INVALID_ARGUMENT_TYPE,
   AUE_INVALID_RETURN_TYPE
  } AddUDFError;
```

UDFs must be registered with CLIPS using the function **AddUDF** before they can be referenced from CLIPS code. Calls to **AddUDF** can be made in the function **UserFunctions** contained in the CLIPS **userfunctions.c** file. Within **UserFunctions**, a call should be made 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 **AddUDF** from their own initialization code—the only restrictions is that it must be called after CLIPS has been initialized and before the UDF is referenced.

Parameter **env** is a pointer to a previously defined environment; parameter **clipsName** is the name associated with the UDF when it is called from within CLIPS; parameter **returnsTypes** is a string containing character codes indicating the CLIPS types returned by the UDF; parameter **minArgs** is the minimum number of arguments that must be passed to the UDF; parameter **maxArgs** is the maximum number of arguments that may be passed to the UDF; parameter **argTypes** is a string containing one or more groups of character codes specifying the allowed types for arguments; parameter **cfp** is a pointer to a function of type **UserDefinedFunction** to be invoked by CLIPS; parameter **cName** is the name of the UDF as specified in the C source code; and parameter **context** is a user supplied pointer to data that is passed to the UDF when it is invoked through the **UDFContext** paramter. This function returns AUE_NO_ERROR if the UDF was successfully added; otherwise, one of the error codes AUE_MIN_EXCEEDS_MAX, AUE_FUNCTION_NAME_IN_USE, AUE_INVALID_ARGUMENT_TYPE, or AUE_INVALID_RETURN_TYPE is returned.

User-defined functions override 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.

If the **returnTypes** parameter value is a null pointer, then CLIPS assumes that the UDF can return any valid type. Specifying one or more type character codes, however, allows CLIPS to detect errors when the return value of a UDF is used as a parameter value to a function that specifies the types allowed for that parameter. The following codes are supported for return values and argument types:

Type Code	Type
b	Boolean
d	Double Precision Float

e	External Address
f	Fact Address
i	Instance Address
l	Long Long Integer
m	Multifield
n	Instance Name
s	String
y	Symbol
v	Void—No Return Value
*	Any Type

One or more characters can be specified. For example, "l" indicates the UDF returns an integer; "ld" indicates the UDF returns an integer or float; and "syn" indicates the UDF returns a symbol, string, or instance name.

The **minArgs** and **maxArgs** parameter values can be specified as the constant **UNBOUNDED** to indicate that there is no restriction on the minimum or maximum number of arguments.

If the **argTypes** parameter value is a null pointer, then there are no argument type restrictions. One or more character argument types can also be specified, separated by semicolons. The first type specified is the default type (used when no other type is specified for an argument), followed by types for specific arguments. For example, "ld" indicates that the default argument type is an integer or float; "ld;s" indicates that the default argument type is an integer or float, and the first argument must be a string; "*;;m" indicates that the default argument type is any type, and the second argument must be a multifield; ";sy;ld" indicates that the default argument type is any type, the first argument must be a string or symbol; and the second argument type must be an integer or float.

8.3 Passing Arguments from CLIPS to User Defined Functions

Unlike a C function call, CLIPS does immediately evaluate all of the arguments in a function call and directly pass the resulting values to the C function implementating the UDF. Instead arguments are evaluated and supplied when requested through argument access functions.

CLIPS will generate an error and terminate the invocation of a UDF before it is called if the incorrect number of arguments is supplied (either there are fewer arguments than the minimum specified or more arguments than the maximum specified).

Several access functions are provided to retrieve arguments:

```
unsigned UDFArgumentCount(
   UDFContext *udfc);
```

```
bool UDFFirstArgument(
   UDFContext *udfc,
   unsigned expectedType,
   UDFValue *out);

bool UDFNextArgument(
   UDFContext *udfc,
   unsigned expectedType,
   UDFValue *out);

bool UDFNthArgument(
   UDFContext *udfc,
   unsigned n,
   unsigned expectedType,
   UDFValue *out);

bool UDFHasNextArgument(
   UDFContext *udfc);

void UDFThrowError (
  UDFContext *udfc);

void SetErrorValue (
  Environment *theEnv,
  TypeHeader *theValue);
```

The function **UDFArgumentCount** returns the number of arguments passed to the UDF. At the point the UDF is invoked, the argument count has been verified to fall within the range specified by the minimum and maximum number of arguments specified in the call to **AddUDF**. Thus a UDF should only need to check the argument count if the minimum and maximum number of arguments are not the same.

The function **UDFFirstArgument** retrieves the first argument passed to the UDF. Parameter **udfc** is a pointer to the UDFContext; parameter **expectedType** is a bit field containing the expcted types for the argument; and parameter **out** is a pointer to a UDFValue in which the retrieved argument value is stored. This function returns true if the argument was successfully retrieved and is the expected type; otherwise, it returns false.

The function **UDFNextArgument** retrieves the argument following the previously retrieved argument (either from **UDFFirstArgument**, **UDFNextArgument**, or **UDFNthArgument**). It retrieves the first argument if no arguments have been previously retrieved. Parameter **udfc** is a pointer to the UDFContext; parameter **expectedType** is a bit field containing the expcted types for the argument; and parameter **out** is a pointer to a UDFValue in which the retrieved

argument value is stored. This function returns true if the argument was successfully retrieved and is the expected type; otherwise, it returns false.

The function **UDFNthArgument** retrieves a specific argument passed to the UDF. Parameter **udfc** is a pointer to the UDFContext; parameter **n** is the index of the argument to be retrieved (with indices starting at 1); parameter **expectedType** is a bit field containing the expcted types for the argument; and parameter **out** is a pointer to a UDFValue in which the retrieved argument value is stored. This function returns true if the argument was successfully retrieved and is the expected type; otherwise, it returns false.

The function **UDFHasNextArgument** returns true if there is an argument is available to be retrieved; otherwise, it returns false. The "next" argument is considered to be the first argument if no previous call to **UDFFirstArgument**, **UDFNextArgument**, or **UDFNthArgument** has been made; otherwise it is the next argument following the most recent call to one of those functions.

The function **UDFThrowError** can be used by a **UDF** to indicate that an error has occurred and execution should terminate.

The **SetErrorValue** function can be used to assign an error value which can be retrieved using the **get-error** and **set-error** CLIPS functions. This function can be used for situations where a function's return value can not be used to indicate an error and execution should not be terminated.

The functions **UDFFirstArgument**, **UDFNextArgument**, and **UDFNthArgument** use bit fields rather than character codes to indicate the types allowed for the **expectedType** parameter value. The following constants are defined for specifying bit codes:

Type Bit Code	Type
FLOAT_BIT	Float
INTEGER_BIT	Integer
SYMBOL_BIT	Symbol
STRING_BIT	String
MULTIFIELD_BIT	Multifield
EXTERNAL_ADDRESS_BIT	External Address
FACT_ADDRESS_BIT	Fact Address
INSTANCE_ADDRESS_BIT	Instance Address
INSTANCE_NAME_BIT	Instance Name
VOID_BIT	Void
BOOLEAN_BIT	Boolean
NUMBER_BITS	Float, Integer
LEXEME_BITS	Symbol, String
ADDRESS_BITS	External Address, Fact Address, Instance Address
INSTANCE_BITS	Instance Address, Instance Name

SINGLEFIELD_BITS	Number, Lexeme, Address, Instance Name
ANY_TYPE_BITS	Void, Singlefield, Multifield

These bit codes can be combined using the C | operator. For example, the following code indicates that an argument should either be an integer or symbol:

```
INTEGER_BIT | SYMBOL_BIT
```

8.4 Examples

8.4.1 Euler's Number

This example demonstrates returning a mathematical constant, Euler's number, from a user defined function.

The **AddUDF** function call required in **UserFunctions** specifies that the CLIPS function name is **e**; the return value type is a float; the UDF does not expect any arguments; and the C implementation of the UDF is the function **EulersNumber**.

```
void UserFunctions(
  Environment *env)
  {
   AddUDF(env,"e","d",0,0,NULL,EulersNumber,"EulersNumber",NULL);
  }
```

The implementation of the CLIPS function **e** in the C function **EulersNumber** uses the function **CreateFloat** to create the return value. The C library function **exp** is used to calculate the value for Euler's number.

```
#include <math.h>

void EulersNumber(
  Environment *env,
  UDFContext *udfc,
  UDFValue *out)
  {
   out->floatValue = CreateFloat(env,exp(1.0));
  }
```

After creating a new executable including the UDF code, the function **e** can be invoked within CLIPS.

```
CLIPS> (e)
2.71828182845905
CLIPS>
```

8.4.2 Week Days Multifield Constant

This example demonstrates returning a multifield constant, the weekdays, from a user defined function.

The **AddUDF** function call required in **UserFunctions** specifies that the CLIPS function name is **weekdays**; the return value type is a multifield value; the UDF does not expect any arguments; and the C implementation of the UDF is the function **Weekdays**.

```
void UserFunctions(
  Environment *env)
  {
   AddUDF(env,"weekdays","m",0,0,NULL,Weekdays,"Weekdays",NULL);
  }
```

The implementation of the CLIPS function **weekdays** in the C function **Weekdays** uses the function **StringToMultifield** to create the return value.

```
void Weekdays(
  Environment *env,
  UDFContext *udfc,
  UDFValue *out)
  {
   out->multifieldValue =
       StringToMultifield(env,"Monday Tuesday Wednesday Thursday Friday");
  }
```

After creating a new executable including the UDF code, the function **week-days** can be invoked within CLIPS.

```
CLIPS> (weekdays)
(Monday Tuesday Wednesday Thursday Friday)
CLIPS>
```

8.4.3 Cubing a Number

This example demonstrates a user defined function that cubes a numeric argument value and returns either an integer or float depending upon the type of the argument value.

The **AddUDF** function call required in **UserFunctions** specifies that the CLIPS function name is **cube**; the return value type is an integer or float value; the UDF expects one argument that must be an integer or a float; and the C implementation of the UDF is the function **Cube**.

```
void UserFunctions(
  Environment *env)
  {
```

```
   AddUDF(env,"cube","ld",1,1,"ld",Cube,"Cube",NULL);
  }
```

The implementation of the CLIPS function **cube** in the C function **Cube** uses the function **UDFFirstArgument** to retrieve the numeric argument passed to the function. If the argument value is an integer, the function **CreateInteger** is used to create the return value. If the argument value is a float, the function **CreateFloat** is used to create the return value.

```
void Cube(
  Environment *env,
  UDFContext *udfc,
  UDFValue *out)
  {
   UDFValue theArg;

   // Retrieve the first argument.

   if (! UDFFirstArgument(udfc,NUMBER_BITS,&theArg))
     { return; }

   // Cube the argument.

   if (theArg.header->type == INTEGER_TYPE)
     {
      long long integerValue = theArg.integerValue->contents;
      integerValue = integerValue * integerValue * integerValue;
      out->integerValue = CreateInteger(env,integerValue);
     }
   else /* the type must be FLOAT */
     {
      double floatValue = theArg.floatValue->contents;
      floatValue = floatValue * floatValue * floatValue;
      out->floatValue = CreateFloat(env,floatValue);
     }
  }
```

After creating a new executable including the UDF code, the function **cube** can be invoked within CLIPS.

```
CLIPS> (cube 3)
27
CLIPS> (cube 3.5)
42.875
CLIPS>
```

8.4.4 Positive Number Predicate

This example demonstrates a user defined function that returns a boolean value indicating whether a numeric argument value is positive.

The **AddUDF** function call required in **UserFunctions** specifies that the CLIPS function name is **positivep**; the return value type is a boolean value (either the symbol TRUE or FALSE); the UDF expects one argument that must be an integer or a float; and the C implementation of the UDF is the function **Positivep**.

```
void UserFunctions(
  Environment *env)
  {
   AddUDF(env,"positivep","b",1,1,"ld",Positivep,"Positivep",NULL);
  }
```

The implementation of the CLIPS function **positivep** in the C function **Positivep** uses the function **UDFFirstArgument** to retrieve the numeric argument passed to the function. The function **CreateBoolean** is used to create the return value.

```
void Positivep(
  Environment *env,
  UDFContext *udfc,
  UDFValue *out)
  {
   UDFValue theArg;
   bool b;

   // Retrieve the first argument.

   if (! UDFFirstArgument(udfc,NUMBER_BITS,&theArg))
     { return; }

   // Determine if the value is positive.

   if (theArg.header->type == INTEGER_TYPE)
     { b = (theArg.integerValue->contents > 0); }
   else /* the type must be FLOAT */
     { b = (theArg.floatValue->contents > 0.0); }

   out->lexemeValue = CreateBoolean(env,b);
  }
```

After creating a new executable including the UDF code, the function **positivep** can be invoked within CLIPS.

```
CLIPS> (positivep -3)
FALSE
CLIPS> (positivep 4.5)
TRUE
CLIPS>
```

8.4.5 Exclusive Or

This example demonstrates a user defined function that returns a boolean value indicating whether an odd number of its arguments values are true (exclusive or).

The **AddUDF** function call required in **UserFunctions** specifies that the CLIPS function name is **xor**; the return value type is a boolean value (either the symbol TRUE or FALSE); the UDF expects at least two arguments; and the C implementation of the UDF is the function **Xor**.

```
void UserFunctions(
  Environment *env)
  {
   AddUDF(env,"xor","b",2,UNBOUNDED,NULL,Xor,"Xor",NULL);
  }
```

The implementation of the CLIPS function **xor** in the C function **Xor** uses the functions **UDFHasNextArgument** and **UDFNextArgument** to retrieve the variable number of arguments passed to the function. Any value other than the symbol FALSE is considered to be "TRUE" by CLIPS, so when counting the number of argument values to be true, each argument is compared for inequality to the return value of the **FalseSymbol** function. Finally, the function **CreateBoolean** is used to create the return value.

```
void Xor(
  Environment *env,
  UDFContext *udfc,
  UDFValue *out)
  {
   UDFValue theArg;
   int trueCount = 0;

   while (UDFHasNextArgument(udfc))
     {
      UDFNextArgument(udfc,ANY_TYPE_BITS,&theArg);

      if (theArg.value != FalseSymbol(env))
        { trueCount++; }
     }

   out->lexemeValue = CreateBoolean(env,trueCount % 2);
  }
```

After creating a new executable including the UDF code, the function **xor** can be invoked within CLIPS.

```
CLIPS> (xor TRUE FALSE)
TRUE
CLIPS> (xor FALSE FALSE)
FALSE
CLIPS> (xor TRUE FALSE TRUE FALSE TRUE)
```

```
TRUE
CLIPS>
```

8.4.6 String Reversal

This example demonstrates a user defined function that reverses the characters in a CLIPS symbol, string, or instance name.

The **AddUDF** function call required in **UserFunctions** specifies that the CLIPS function name is **reverse**; the return value type is a string, symbol, or instance name; the UDF expects one argument that must be a string, symbol, or instance name; and the C implementation of the UDF is the function **Reverse**.

```
void UserFunctions(
  Environment *env)
  {
   AddUDF(env,"reverse","syn",1,1,"syn",Reverse,"Reverse",NULL);
  }
```

The implementation of the CLIPS function **reverse** in the C function **Reverse** uses the function **UDFFirstArgument** to retrieve the lexeme argument (string, symbol, or instance name) passed to the function. The function **genalloc** is used to allocate temporary memory for reversing the order of characters in the string. Depending upon the argument value type, the function **CreateString**, **CreateSymbol**, or **CreateInstanceName** is used to create the return value. Finally, the function **genfree** is used to deallocate the temporary memory.

```
void Reverse(
  Environment *env,
  UDFContext *udfc,
  UDFValue *out)
  {
   UDFValue theArg;
   const char *theString;
   char *tempString;
   size_t length, i;

   // Retrieve the first argument.

   if (! UDFFirstArgument(udfc,LEXEME_BITS | INSTANCE_NAME_BIT,&theArg))
     { return; }

   theString = theArg.lexemeValue->contents;

   // Allocate temporary space to store the reversed string.

   length = strlen(theString);
   tempString = (char *) genalloc(env,length + 1);

   // Reverse the string.
```

```
   for (i = 0; i < length; i++)
     { tempString[length - (i + 1)] = theString[i]; }

   tempString[length] = '\0';

   // Set the return value before deallocating
   // the temporary reversed string.

   switch (theArg.header->type)
     {
      case STRING_TYPE:
        out->lexemeValue = CreateString(env,tempString);
        break;

      case SYMBOL_TYPE:
        out->lexemeValue = CreateSymbol(env,tempString);
        break;

      case INSTANCE_NAME_TYPE:
        out->lexemeValue = CreateInstanceName(env,tempString);
        break;
     }

   // Deallocate temporary space

   genfree(env,tempString,length+1);
  }
```

After creating a new executable including the UDF code, the function **reverse** can be invoked within CLIPS.

```
CLIPS> (reverse abcd)
dcba
CLIPS> (reverse "xyz")
"zyx"
CLIPS> (reverse [ijk])
[kji]
CLIPS>
```

8.4.7 Reversing the Values in a Multifield Value

This example demonstrates a user defined function that creates a multifield value comprised from its arguments values, but in reverse order.

The **AddUDF** function call required in **UserFunctions** specifies that the CLIPS function name is **reverse$**; the return value type is a multifield value; the UDF expects any number of arguments; and the C implementation of the UDF is the function **ReverseMF**.

```
void UserFunctions(
```

```
   Environment *env)
   {
    AddUDF(env,"reverse$","m",0,UNBOUNDED,NULL,ReverseMF,"ReverseMF",NULL);
   }
```

The implementation of the CLIPS function **reverse$** in the C function **ReverseMF** uses the functions **UDFArgumentCount** and **UDFNthArgument** to retrieve arguments passed to the function in reverse order. A multifield builder is created using the function **CreateMultifieldBuilder**. Each function argument is appended to the multifield builder using the function **MBAppendUDFValue**. The return value is created using **MBCreate**. Finally, the multifield builder is deallocated using the function **MBDispose**.

```
 void ReverseMF(
   Environment *env,
   UDFContext *udfc,
   UDFValue *out)
   {
    UDFValue theArg;
    MultifieldBuilder *mb;
    unsigned argCount;

    // Create the multifield builder.

    mb = CreateMultifieldBuilder(env,20);

    // Iterate over the argument
    // values in reverse order.

    argCount = UDFArgumentCount(udfc);

    for (unsigned i = argCount; i != 0; i--)
      {
       // Append the Nth argument
       // to the multifield builder

       UDFNthArgument(udfc,i,ANY_TYPE_BITS,&theArg);
       MBAppendUDFValue(mb,&theArg);
      }

    // Create the return value.

    out->multifieldValue = MBCreate(mb);

    // Dispose of the multifield value.

    MBDispose(mb);
   }
```

After creating a new executable including the UDF code, the function **reverse$** can be invoked within CLIPS.

```
CLIPS> (reverse$ 1 2 3)
(3 2 1)
CLIPS> (reverse$ a 6 (create$ 6.3 5.4) "s")
("s" 6.3 5.4 6 a)
CLIPS>
```

8.4.8 Trimming a Multifield

This example demonstrates a user defined function that trims values from the beginning and end of a multifield value.

The **AddUDF** function call required in **UserFunctions** specifies that the CLIPS function name is **trim$**; the return value type is a multifield value; the UDF expects three arguments (the default type is an integer and the first argument must be a multiefield value); and the C implementation of the UDF is the function **Trim**.

```
void UserFunctions(
  Environment *env)
  {
   AddUDF(env,"trim$","m",3,3,"l;m",Trim,"Trim",NULL);
  }
```

The implementation of the CLIPS function **trim$** in the C function **Trim** uses the functions **UDFFirstArgument** and **UDFNextArgument** to retrieve arguments passed to the function. The first argument is a multifield value; and the second and third arguments are integers (the number of values to trim from the beginning and end of the multifield value). If the trim values are negative or exceed the number of values contained in the multifield, then the functions **PrintString** and **UDFThrowError** are used to indicate an error; otherwise the **begin** and **range** fields of the argument value stored in the return value parameter **out** are modified to trim the appropriate number of values from the beginning and end of the multifield.

```
void Trim(
  Environment *env,
  UDFContext *udfc,
  UDFValue *out)
  {
   long long front, back;
   UDFValue arg;

   // Retrieve the arguments.

   UDFFirstArgument(udfc,MULTIFIELD_BIT,out);

   UDFNextArgument(udfc,INTEGER_BIT,&arg);
   front = arg.integerValue->contents;

   UDFNextArgument(udfc,INTEGER_BIT,&arg);
   back = arg.integerValue->contents;
```

```
   // Detect errors.

   if ((front < 0) || (back < 0))
     {
      Writeln(env,"Trim$ indices cannot be negative.");
      UDFThrowError(udfc);
      return;
     }

   if ((front + back) > (long long) out->multifieldValue->length)
     {
      Writeln(env,"Trim$ exceeds length of multifield.");
      UDFThrowError(udfc);
      return;
     };

   // Adjust the begin and range.

   out->begin += (size_t) front;
   out->range -= (size_t) (front + back);
  }
```

After creating a new executable including the UDF code, the function **trim$** can be invoked within CLIPS.

```
CLIPS> (trim$ (create$ a b c d e f g) 1 2)
(b c d e)
CLIPS> (trim$ (create$ a b c) -1 2)
Trim$ indices cannot be negative.
(a b c)
CLIPS>
```

8.4.9 Removing Duplicates from a Multifield

This example demonstrates a user defined function that removes duplicate values from a multifield value.

The **AddUDF** function call required in **UserFunctions** specifies that the CLIPS function name is **compact$**; the return value type is a multifield value; the UDF expects one argument that must be a multifield value; and the C implementation of the UDF is the function **Compact**.

```
void UserFunctions(
  Environment *env)
  {
   AddUDF(env,"compact$","m",1,1,"m",Compact,"Compact",NULL);
  }
```

The implementation of the CLIPS function **compact$** in the C function **Compact** uses the function **UDFFirstArgument** to retrieve the multifield argument value passed to the function. A multifield builder is created using the function **CreateMultifieldBuilder**. The **begin** and **range** fields are used to iterate over the values of the multifield argument value. If the value is not contained within the multifield builder, then it is added using the **MBAppend** function. The return value of the function is created using the **MBCreate** function. Finally, the multifield builder is deallocated using the function **MBDispose**.

```
void Compact(
  Environment *env,
  UDFContext *udfc,
  UDFValue *out)
  {
   UDFValue arg;
   MultifieldBuilder *mb;
   size_t i, j;

   // Retrieve the argument.

   UDFFirstArgument(udfc,MULTIFIELD_BIT,&arg);

   // Create the multifield builder.

   mb = CreateMultifieldBuilder(env,20);

   // Iterate over each value in the multifield
   // and add it to the compacted multifield if
   // it is not already present.

   for (i = arg.begin; i < (arg.begin + arg.range); i++)
     {
      // Look for the value in the multifield builder.

      for (j = 0; j < mb->length; j++)
        {
         if (arg.multifieldValue->contents[i].value == mb->contents[j].value)
           { break; }
        }

      // If the value wasn't found, add it.

      if (j == mb->length)
        { MBAppend(mb,&arg.multifieldValue->contents[i]); }
     }

   // Create the return value.

   out->multifieldValue = MBCreate(mb);

   // Dispose of the multifield builder.

   MBDispose(mb);
```

```
   }
```

After creating a new executable including the UDF code, the function **compact$** can be invoked within CLIPS.

```
CLIPS> (compact$ (create$ a b c))
(a b c)
CLIPS> (compact$ (create$ a a b c b c d))
(a b c d)
CLIPS>
```

8.4.10 Prime Factors

This example demonstrates a user defined function that determines the prime factors of an integer.

The **AddUDF** function call required in **UserFunctions** specifies that the CLIPS function name is **prime-factors**; the return value type is an integer; the UDF expects one argument that must be an integer; and the C implementation of the UDF is the function **PrimeFactors**.

```
void UserFunctions(
  Environment *env)
  {
   AddUDF(env,"prime-factors","m",1,1,"l",PrimeFactors,"PrimeFactors",NULL);
  }
```

The implementation of the CLIPS function **prime-factors** in the C function **PrimeFactor** uses the function **UDFFirstArgument** to retrieve the integer argument value passed to the function. If the integer is less than 2, the return value is created using the function **EmptyMultifield** to indicate that there are no prime factors. Otherwise, a multifield builder is created using the function **CreateMultifieldBuilder**. A trial division algorithm is then used to determine the prime factors and, as each is found, it is added to the multifield builder using the **MBAppendInteger** function. The return value of the function is created using the **MBCreate** function. Finally, the multifield builder is deallocated using the function **MBDispose**.

```
#include <math.h>

void PrimeFactors(
   Environment *env,
   UDFContext *udfc,
   UDFValue *out)
   {
    UDFValue value;
    long long num, p, upper;
    MultifieldBuilder *mb;

    // Retrieve the integer argument.
```

```
   UDFFirstArgument(udfc,INTEGER_BIT,&value);
   num = value.integerValue->contents;

   // Integers less than 2 don't have
   // a prime factorization.

   if (num < 2)
     {
      out->multifieldValue = EmptyMultifield(env);
      return;
     }

   // Create the multifield builder.

   mb = CreateMultifieldBuilder(env,10);

   // Determine the prime factors.

   upper = (long long) sqrt(num);
   for (p = 2; p <= upper; p++)
     {
      if ((p * p) > num) break;

      while ((num % p) == 0)
        {
         MBAppendInteger(mb,p);
         num /= p;
        }
     }

   if (num > 1)
     { MBAppendInteger(mb,num); }

   // Set the return value.

   out->multifieldValue = MBCreate(mb);

   // Dispose of the multifield builder.

   MBDispose(mb);
  }
```

After creating a new executable including the UDF code, the function **prime-factor** can be invoked within CLIPS.

```
CLIPS> (prime-factors 1)
()
CLIPS> (prime-factors 3)
(3)
CLIPS> (prime-factors 128)
(2 2 2 2 2 2 2)
CLIPS> (prime-factors 5040)
```

```
(2 2 2 2 3 3 5 7)
CLIPS> (prime-factors 1257383)
(373 3371)
CLIPS> (prime-factors 6469693230)
(2 3 5 7 11 13 17 19 23 29)
CLIPS>
```

Section 9:
I/O Routers

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 9.1 through 9.4. The CLIPS I/O functions for using the router system are described in sections 9.5 and 9.6, and finally, in section 9.7, some examples are included which show how I/O routing could be used for simple interfaces.

9.1 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.

9.2 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-1
```

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.
stderr	All error messages are sent to this logical name.
stdwrn	All warning messages are sent to this logical name.

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, STDERR, and STDWRN.

9.3 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.

9.4 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 **WriteString** again. These types of routers should use a priority of either 30 or 40. An example is given in appendix 9.7.2.

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.

9.5 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.

9.5.1 ExitRouter

```
void ExitRouter(
   Environment *env,
   int code);
```

The function **ExitRouter** calls the exit function callback associated with each active router before exiting CLIPS. Parameter **env** is a pointer to a previously created environment; and parameter **code** is an integer passed to the callback as well as the system **exit** function once all callbacks have been executed. User code that detects an unrecoverable error should call this function rather than calling the system **exit** function so that routers have the opportunity to execute cleanup code.

9.5.2 Input

```
int ReadRouter(
   Environment *env,
   const char *logicalName);

int UnreadRouter(
   Environment *env,
```

```
   const char *logicalName,
   int ch);
```

The function **ReadRouter** queries all active routers to retrieve character input. This function should be used in place of **getc** to ensure that character input from the function can be received from a custom interface. Parameter **env** is a pointer to a previously created environment; and parameter **logicalName** is the query string that must be recognized by the router to be invoked to handle the I/O request. The get character function callback for that router is invoked and the return value of that callback is returned by this function.

The function **UnreadRouter** queries all active routers to push character input back into an input source. This function should be used in place of **ungetc** to ensure that character input works properly using a custom interface. Parameter **env** is a pointer to a previously created environment; parameter **logicalName** is the query string that must be recognized by the router to be invoked to handle the I/O request; and parameter **ch** is the character to be pushed back to the input source. The unget character function callback for that router is invoked. The return value for this function is the parameter value **ch** if the character is successfully pushed back to the input source; otherwise, -1 is returned.

9.5.3 Output

```
void Write(
   Environment *env,
   const char *str);

void WriteCLIPSValue(
   Environment *env,
   const char *logicalName,
   CLIPSValue *cv);

void WriteFloat(
   Environment *env,
   const char *logicalName,
   double d);

void WriteInteger(
   Environment *env,
   const char *logicalName,
   long long l);

void Writeln(
   Environment *env,
   const char *str);
```

```
void WriteString(
   Environment *env,
   const char *logicalName,
   const char *str);

void WriteMultifield(
   Environment *env,
   const char *logicalName,
   Multifield *mf);

void WriteUDFValue(
   Environment *env,
   const char *logicalName,
   UDFValue *udfv);
```

The functions **Write**, **WriteCLIPSValue**, **WriteFloat**, **WriteInteger**, **Writeln**, **WriteString**, **WriteMultifield**, and **WriteUDFValue** direct output to a router for display. Using these functions in place of **printf** ensures that output will be displayed appropriately whether CLIPS is run as part of a console application, integrated development environment, or custom interface. By default, output from these functions will use **printf** for display if no other routers are detected to handle the output request.

For all of these functions, parameter **env** is a pointer to a previously created environment. For all function except **Write** and **Writeln**, the parameter **logicalName** is the query string that must be recognized by a router to indicate it can handle the output request. All active routers are queried in order of their priority until the query function callback for a router returns true to indicate if handles the specified logical name. The **Write** and **Writeln** functions automatically sent output to the STDOUT logical name.

The remaining parameter for all of these functions is the value to be printed. The parameters **d**, **l**, and **str** are the C types double, long long, and a char pointer to a null-terminated string. Parameter **mf** is a pointer to a **Multifield**. Parameters **cv** and **udfv** are pointers to **CLIPSValue** and **UDFValue** types that have been allocated and populated with data by the caller. The **Writeln** function additionally prints a carriage return after printing the **str** parameter.

9.6 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.

9.6.1 Creating Routers

```
bool AddRouter(
   Environment *env,
   const char *name,
   int priority,
   RouterQueryFunction *queryCallback,
   RouterWriteFunction *writeCallback,
   RouterReadFunction *readCallback,
   RouterUnreadFunction *unreadCallback,
   RouterExitFunction *exitCallback,
   void *context);

typedef bool RouterQueryFunction(
   Environment *env,
   const char *logicalName,
   void *context);

typedef void RouterWriteFunction(
   Environment *env,
   const char *logicalName,
   const char *str,
   void *context);

typedef void RouterExitFunction(
   Environment *environment,
   int code,
   void *context);

typedef int RouterReadFunction(
   Environment *env,
   const char *logicalName,
   void *context);

typedef int RouterUnreadFunction(
   Environment *env,
   const char *logicalName,
   int ch,
   void *context);
```

The function **AddRouter** creates and activates a new router. Parameter **env** is a pointer to a previously created environment; parameter **name** is a string that uniquely identifies the router for removal using **DeleteRouter**; parameters **queryCallback**, **writeCallback**, **readCallback**, **unreadCallback**, and **exitCallback** are pointers to callback functions;

parameter **priority** is the priority of the router used to determine the order in which routers are queried (higher priority routers are queried first); and parameter **context** is a user supplied pointer to data that is passed to the router callback functions when they are invoked (a null pointer should be used if there is no data that needs to be passed to the router callback functions). The **queryCallback** parameter value must be a non-null function pointer, otherwise the other router callback functions will never be invoked. If the router does not handle output requests, the **writeCallback** parameter value should be a null pointer. If the router does not handle input requests, the **readCallback** and **unreadCallback** parameter values should be null pointers. This function returns true if the router was successfully added; otherwise, it returns false.

The **RouterQueryFunction** type has three parameters: **env** is a pointer to a previously created environment; **logicalName** is the logical name associated with the I/O request; and **context** is the user supplied data pointer provided when the router was created. This function should return true if the **logicalName** parameter value is recognized by this router; otherwise, it should return false.

The **RouterWriteFunction** type has four parameters: **env** is a pointer to a previously created environment; **logicalName** is the logical name associated with the I/O request; **str** is the null character terminated string to be printed; and **context** is the user supplied data pointer provided when the router was created.

The **RouterExitFunction** type has three parameters: **env** is a pointer to a previously created environment; **code** is the exit code value (either the value passed to the CLIPS **exit** command or the C **ExitRouter** function); and **context** is the user supplied data pointer provided when the router was created.

The **RouterReadFunction** type has three parameters: **env** is a pointer to a previously created environment; **logicalName** is the logical name associated with the I/O request; and **context** is the user supplied data pointer provided when the router was created. The return value of this function is an integer character code or -1 to indicate EOF (end of file).

The **RouterUnreadFunction** type has four parameters: **env** is a pointer to a previously created environment; **logicalName** is the logical name associated with the I/O request; **ch** is the character code to be pushed back into the router input source; and **context** is the user supplied data pointer provided when the router was created. The return value of this function should be the **ch** parameter value if the function successfully pushes the character code; otherwise, it should return -1 (EOF).

9.6.2 Deleting Routers

```
bool DeleteRouter(
   Environment *env,
   const char *name);
```

The function **DeleteRouter** removes previously created router. Parameter **env** is a pointer to a previously created environment; and parameter **name** is the string used to identify the router when it was added using **AddRouter**. The function returns true if the router was successfully deleted; otherwise, it returns false.

9.6.3 Activating and Deactivating Routers

```
bool ActivateRouter(
   Environment *env,
   const char *name);

bool DeactivateRouter(
   Environment *env,
   const char *name);
```

The function **ActivateRouter** activates the I/O router specified by parameter **name** (the string used to identify the router when it was created using **AddRouter**). The activated router will be queried to see if it can handle an I/O request. Newly created routers do not have to be activated. This function returns true if the router exists and was successfully activated; otherwise, false is returned.

The function **DeactivateRouter** deactivates the I/O router specified by parameter **name** (the string used to identify the router when it was created using **AddRouter**). The deactivated router will not be queried to see if it can handle an I/O request. This function returns true if the router exists and was successfully deactivated; otherwise, false is returned.

9.7 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.

9.7.1 Dribble System

Write the necessary functions that will divert all error information to the file named "error.txt".

```
/*
First of all, we need to create an environment data structure for storing a file
pointer to the dribble file which will contain the error information. The data
position is offset to prevent conflict with other examples in this document. We also
need to declare prototypes for the functions used in this example.
*/

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

```
#include "clips.h"

#define DRIBBLE_DATA USER_ENVIRONMENT_DATA + 1

struct dribbleData
  {
   FILE *traceFP;
  };

#define DribbleData(theEnv) \
   ((struct dribbleData *) GetEnvironmentData(theEnv,DRIBBLE_DATA))

bool QueryTraceCallback(Environment *,const char *,void *);
void WriteTraceCallback(Environment *,const char *,const char *,void *);
void ExitTraceCallback(Environment *environment,int,void *);
void TraceOn(Environment *,UDFContext *,UDFValue *);
void TraceOff(Environment *,UDFContext *,UDFValue *);

/*
We want to recognize any output that is sent to the logical name STDERR because all
tracing information is sent to this logical name.  The query function for our router
is defined below.
*/

bool QueryTraceCallback(
  Environment *environment,
  const char *logicalName,
  void *context)
  {
   if (strcmp(logicalName,STDERR) == 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. The context argument
is used to retrieve the FILE pointer that will be supplied when AddRouter is called.
*/

void WriteTraceCallback(
  Environment *environment,
  const char *logicalName,
  const char *str,
  void *context)
  {
   FILE *theFile = (FILE *) context;

   fprintf(theFile,"%s",str);
  }

/*
```

```
When we exit CLIPS the trace file needs to be closed. The exit function for our
router is defined below. The context argument is used to retrieve the FILE pointer
that will be supplied when AddRouter is called.
*/

void ExitTraceCallback(
  Environment *environment,
  int exitCode,
  void *context)
  {
   FILE *theFile = (FILE *) context;

   fclose(theFile);
  }

/*
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.
*/

void TraceOn(
  Environment *environment,
  UDFContext *context,
  UDFValue *returnValue)
  {
   if (DribbleData(environment)->traceFP == NULL)
     {
      DribbleData(environment)->traceFP = fopen("error.txt","w");
      if (DribbleData(environment)->traceFP == NULL)
        {
         returnValue->lexemeValue = environment->FalseSymbol;
         return;
        }
     }
   else
     {
      returnValue->lexemeValue = environment->FalseSymbol;
      return;
     }

   AddRouter(environment,
             "trace",                          /* Router name     */
             20,                               /* Priority        */
             QueryTraceCallback,               /* Query function  */
             WriteTraceCallback,               /* Write function  */
             NULL,                             /* Read function   */
             NULL,                             /* Unread function */
```

```
             ExitTraceCallback,                  /* Exit function   */
             DribbleData(environment)->traceFP); /* Context         */

   returnValue->lexemeValue = environment->TrueSymbol;
  }

/*
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.
*/

void TraceOff(
  Environment *environment,
  UDFContext *context,
  UDFValue *returnValue)
  {
   if (DribbleData(environment)->traceFP != NULL)
     {
      DeleteRouter(environment,"trace");

      if (fclose(DribbleData(environment)->traceFP) == 0)
        {
         DribbleData(environment)->traceFP = NULL;
         returnValue->lexemeValue = environment->TrueSymbol;
         return;
        }
     }

   DribbleData(environment)->traceFP = NULL;
   returnValue->lexemeValue = environment->FalseSymbol;
  }

/*
Now add the definitions for these functions to the UserFunctions function in file
"userfunctions.c".
*/

void UserFunctions(
  Environment *env)
  {
   if (! AllocateEnvironmentData(env,DRIBBLE_DATA,
                                 sizeof(struct dribbleData),NULL))
     {
      printf("Error allocating environment data for DRIBBLE_DATA\n");
      exit(EXIT_FAILURE);
     }

   AddUDF(env,"tron","b",0,0,NULL,TraceOn,"tron",NULL);
   AddUDF(env,"troff","b",0,0,NULL,TraceOff,"troff",NULL);
  }
```

```
/*
Compile and link the appropriate files. The trace functions should now be accessible
within CLIPS as external functions. For Example:

   CLIPS> (tron)
   TRUE
   CLIPS> (+ 2 3)
   5
   CLIPS> (* 3 a)
   CLIPS> (troff)
   TRUE
   CLIPS> (exit)

The file error.txt will now contain the following text:

   [ARGACCES5] Function * expected argument #2 to be of type integer or float
*/
```

9.7.2 Better Dribble System

Modify example 1 so the error information is sent to the terminal as well as to the dribble file.

```
/*
This example requires a modification of the WriteTraceCallback function. After the
error string is printed to the file, the trace router must be deactivated. The error
string can then be sent through the WriteString function so that the next router in
line can handle the output.  After this is done, then the trace router can be
reactivated.
*/

void WriteTraceCallback(
  Environment *environment,
  const char *logicalName,
  const char *str,
  void *context)
  {
   FILE *theFile = (FILE *) context;

   fprintf(theFile,"%s",str);
   DeactivateRouter(environment,"trace");
   WriteString(environment,logicalName,str);
   ActivateRouter(environment,"trace");
  }

/*
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.
*/

void TraceOn(
  Environment *environment,
  UDFContext *context,
```

```
  UDFValue *returnValue)
  {
   if (DribbleData(environment)->traceFP == NULL)
     {
      DribbleData(environment)->traceFP = fopen("error.txt","w");
      if (DribbleData(environment)->traceFP == NULL)
        {
         returnValue->lexemeValue = environment->FalseSymbol;
         return;
        }
     }
   else
     {
      returnValue->lexemeValue = environment->FalseSymbol;
      return;
     }

   AddRouter(environment,
             "trace",                            /* Router name     */
             40,                                 /* Priority        */
             QueryTraceCallback,                 /* Query function  */
             WriteTraceCallback,                 /* Write function  */
             NULL,                               /* Read function   */
             NULL,                               /* Unread function */
             ExitTraceCallback,                  /* Exit function   */
             DribbleData(environment)->traceFP); /* Context         */

   returnValue->lexemeValue = environment->TrueSymbol;
  }
```

9.7.3 Batch System

Write the necessary functions that will allow batch input from the file "batch.txt" to the CLIPS top-level interface. ***Note that this example only works in the console version of CLIPS***.

```
/*
First of all, we need a file pointer to the batch file which will contain the batch
command information.
*/

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

#define BATCH_DATA USER_ENVIRONMENT_DATA + 2

struct batchData
  {
   FILE *batchFP;
   StringBuilder *batchBuffer;
  };
```

```
#define BatchData(theEnv) \
   ((struct batchData *) GetEnvironmentData(theEnv,BATCH_DATA))

bool QueryMybatchCallback(Environment *,const char *,void *);
int ReadMybatchCallback(Environment *,const char *,void *);
int UnreadMybatchCallback(Environment *,const char *,int,void *);
void ExitMybatchCallback(Environment *environment,int,void *);
void MybatchOn(Environment *,UDFContext *,UDFValue *);

/*
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.
*/

bool QueryMybatchCallback(
  Environment *environment,
  const char *logicalName,
  void *context)
  {
   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.
*/

int ReadMybatchCallback(
  Environment *environment,
  const char *logicalName,
  void *context)
  {
   int rv;

   rv = getc(BatchData(environment)->batchFP);

   if (rv == EOF)
     {
      Write(environment,BatchData(environment)->batchBuffer->contents);
      SBDispose(BatchData(environment)->batchBuffer);
      BatchData(environment)->batchBuffer = NULL;
      DeleteRouter(environment,"mybatch");
      fclose(BatchData(environment)->batchFP);
      return ReadRouter(environment,logicalName);
     }

   SBAddChar(BatchData(environment)->batchBuffer,rv);

   if ((rv == '\n')  || (rv == '\r'))
     {
```

```
      Write(environment,BatchData(environment)->batchBuffer->contents);
      SBReset(BatchData(environment)->batchBuffer);
     }

   return rv;
  }

int UnreadMybatchCallback(
  Environment *environment,
  const char *logicalName,
  int ch,
  void *context)
  {
   SBAddChar(BatchData(environment)->batchBuffer,'\b');

   return ungetc(ch,BatchData(environment)->batchFP);
  }

/*
When we exit CLIPS the batch file needs to be closed.  The exit function for our
router is defined below.
*/

void ExitMybatchCallback(
  Environment *environment,
  int exitCode,
  void *context)
  {
   FILE *theFile = (FILE *) context;

   if (BatchData(environment)->batchBuffer != NULL)
     {
      SBDispose(BatchData(environment)->batchBuffer);
      BatchData(environment)->batchBuffer = NULL;
     }

   fclose(theFile);
  }

/*
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.
*/

void MybatchOn(
  Environment *environment,
  UDFContext *context,
  UDFValue *returnValue)
  {
   BatchData(environment)->batchFP = fopen("batch.txt","r");

   if (BatchData(environment)->batchFP == NULL)
```

```
     {
      returnValue->lexemeValue = environment->FalseSymbol;
      return;
     }

   if (BatchData(environment)->batchBuffer == NULL)
     { BatchData(environment)->batchBuffer = CreateStringBuilder(environment,80); }

   AddRouter(environment,
             "mybatch",                        /* Router name     */
             20,                               /* Priority        */
             QueryMybatchCallback,             /* Query function  */
             NULL,                             /* Write function  */
             ReadMybatchCallback,              /* Read function   */
             UnreadMybatchCallback,            /* Unread function */
             ExitMybatchCallback,              /* Exit function   */
             BatchData(environment)->batchFP); /* context */

   returnValue->lexemeValue = environment->TrueSymbol;
  }

/*
Now add the definition for this function to the UserFunctions function in file
"userfunctions.c".
*/

void UserFunctions(
  Environment *env)
  {
   if (! AllocateEnvironmentData(env,BATCH_DATA,
                                 sizeof(struct batchData),NULL))
     {
      printf("Error allocating environment data for BATCH_DATA\n");
      exit(EXIT_FAILURE);
     }

   AddUDF(env,"mybatch","b",0,0,NULL,MybatchOn,"MybatchOn",NULL);
  }

/*
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>
*/
```

9.7.4 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 <stdlib.h>
#include <curses.h>
#include "clips.h"

#define CURSES_DATA USER_ENVIRONMENT_DATA + 3

struct cursesData
  {
   WINDOW *lowerWindow, *upperWindow;
   bool windowsInitialized;
   bool useSave;
   int saveChar;
   bool sendReturn;
   char buffer[512];
   int charLocation;
  };

#define CursesData(theEnv) \
   ((struct cursesData *) GetEnvironmentData(theEnv,CURSES_DATA))

bool QueryScreenCallback(Environment *,const char *,void *);
void WriteScreenCallback(Environment *,const char *,const char *,void *);
int ReadScreenCallback(Environment *,const char *,void *);
int UnreadScreenCallback(Environment *,const char *,int,void *);
void ExitScreenCallback(Environment *environment,int,void *);
void ScreenOn(Environment *,UDFContext *,UDFValue *);
void ScreenOff(Environment *,UDFContext *,UDFValue *);

/*
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.
```

```
*/

bool QueryScreenCallback(
  Environment *environment,
  const char *logicalName,
  void *context)
  {
   if ((strcmp(logicalName,STDOUT) == 0) ||
       (strcmp(logicalName,STDIN) == 0) ||
       (strcmp(logicalName,STDERR) == 0) ||
       (strcmp(logicalName,STDWRN) == 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.
*/

void WriteScreenCallback(
  Environment *environment,
  const char *logicalName,
  const char *str,
  void *context)
  {
   struct cursesData *theData = (struct cursesData *) context;

   if (strcmp(logicalName,"top") == 0)
     {
      wprintw(theData->upperWindow,"%s",str);
      wrefresh(theData->upperWindow);
     }
   else
     {
      wprintw(theData->lowerWindow,"%s",str);
      wrefresh(theData->lowerWindow);
     }
  }

/*
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.
*/

int ReadScreenCallback(
  Environment *environment,
  const char *logicalName,
  void *context)
  {
```

```
   struct cursesData *theData = (struct cursesData *) context;
   int rv;

   if (theData->useSave)
     {
      theData->useSave = false;
      return theData->saveChar;
     }

   if (theData->buffer[theData->charLocation] == '\0')
     {
      if (theData->sendReturn == false)
        {
         theData->sendReturn = true;
         return '\n';
        }

      wgetnstr(theData->lowerWindow,&theData->buffer[0],511);
      theData->charLocation = 0;
     }

   rv = theData->buffer[theData->charLocation];
   if (rv == '\0') return '\n';
   theData->charLocation++;
   theData->sendReturn = false;

   return rv;
  }

int UnreadScreenCallback(
  Environment *environment,
  const char *logicalName,
  int ch,
  void *context)
  {
   struct cursesData *theData = (struct cursesData *) context;

   theData->useSave = true;
   theData->saveChar = ch;

   return ch;
  }

/*
When we exit CLIPS CURSES needs to be deactivated. The exit function for our router
is defined below.
*/

void ExitScreenCallback(
  Environment *environment,
  int exitCode,
  void *context)
  {
```

```
   endwin();
  }

/*
Now define a function that turns the screen mode on.
*/

void ScreenOn(
  Environment *environment,
  UDFContext *context,
  UDFValue *returnValue)
  {
   int halfLines, i;

   /* Has initialization already occurred? */

   if (CursesData(environment)->windowsInitialized)
     {
      returnValue->lexemeValue = environment->FalseSymbol;
      return;
     }

   /* Reroute I/O and initialize CURSES. */

   initscr();
   echo();

   CursesData(environment)->windowsInitialized = true;
   CursesData(environment)->useSave = false;
   CursesData(environment)->sendReturn = true;
   CursesData(environment)->buffer[0] = '\0';
   CursesData(environment)->charLocation = 0;

   AddRouter(environment,
             "screen",                          /* Router name     */
             10,                                /* Priority        */
             QueryScreenCallback,               /* Query function  */
             WriteScreenCallback,               /* Write function  */
             ReadScreenCallback,                /* Read function   */
             UnreadScreenCallback,              /* Unread function */
             ExitScreenCallback,                /* Exit function   */
             CursesData(environment));          /* Context         */

   /* Create the two windows. */

   halfLines = LINES / 2;
   CursesData(environment)->upperWindow = newwin(halfLines,COLS,0,0);
   CursesData(environment)->lowerWindow = newwin(halfLines - 1,COLS,halfLines + 1,0);

   /* Both windows should be scrollable. */

   scrollok(CursesData(environment)->upperWindow,TRUE);
   scrollok(CursesData(environment)->lowerWindow,TRUE);
```

```
   /* Separate the two windows with a line. */

   for (i = 0 ; i < COLS ; i++)
     { mvaddch(halfLines,i,'-'); }
   refresh();

   wclear(CursesData(environment)->upperWindow);
   wclear(CursesData(environment)->lowerWindow);
   wmove(CursesData(environment)->lowerWindow,0,0);

   returnValue->lexemeValue = environment->TrueSymbol;
  }

/*
Now define a function that turns the screen mode off.
*/

void ScreenOff(
  Environment *environment,
  UDFContext *context,
  UDFValue *returnValue)
  {
   /* Is CURSES already deactivated? */

   if (CursesData(environment)->windowsInitialized == false)
     {
      returnValue->lexemeValue = environment->FalseSymbol;
      return;
     }

   CursesData(environment)->windowsInitialized = false;

   /* Remove I/O rerouting and deactivate CURSES. */

   DeleteRouter(environment,"screen");
   endwin();

   returnValue->lexemeValue = environment->TrueSymbol;
  }

/*
Now add the definitions for these functions to the UserFunctions function in file
"userfunctions.c".
*/

void UserFunctions(
  Environment *env)
  {
   if (! AllocateEnvironmentData(env,CURSES_DATA,
                                 sizeof(struct cursesData),NULL))
     {
      printf("Error allocating environment data for CURSES_DATA\n");
```

```
      exit(EXIT_FAILURE);
     }

   AddUDF(env,"screen-on","b",0,0,NULL,ScreenOn,"ScreenOn",NULL);
   AddUDF(env,"screen-off","b",0,0,NULL,ScreenOff,"ScreenOff",NULL);
  }

/*
Compile and link the appropriate files. The screen functions should now be accessible
within CLIPS as external functions. For Example
   CLIPS> (screen-on)
   CLIPS> (printout top "Hello World" crlf)
       •
       •
       •
   CLIPS> (screen-off)
*/
```

Section 10:
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.

10.1 Creating and Destroying Environments

Environments are created using the **CreateEnvironment** function. The return value of the **CreateEnvironment** function is pointer to an **Environment** 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()
  {
   Environment *theEnv1, *theEnv2;

   theEnv1 = CreateEnvironment();
   theEnv2 = CreateEnvironment();

   Load(theEnv1,"program1.clp");
   Load(theEnv2,"program2.clp");

   Reset(theEnv1);
   Reset(theEnv2);

   Run(theEnv1,-1);
   Run(theEnv2,-1);
```

```
    DestroyEnvironment(theEnv1);
    DestroyEnvironment(theEnv2);
   }
```

10.2 Environment Data Functions

User-defined functions (or other extensions) that make use of global data that could differ for each environment should allocate and retrieve this data using the environment data functions.

10.2.1 Allocating Environment Data

```
bool AllocateEnvironmentData(
   Environment *env,
   unsigned id,
   size_t size,
   EnvironmentCleanupFunction f);

bool AddEnvironmentCleanupFunction(
   Environment *env,
   const char *name,
   EnvironmentCleanupFunction f,
   int p);

typedef void EnvironmentCleanupFunction(
   Environment *environment);
```

The function **AllocateEnvironmentData** allocates memory for storage of data in an environment. Parameter **env** is a pointer to a previously created environment; parameter **id** is an integer that uniquely identifies the data for other functions which reference it; parameter **size** is the amount of memory allocated; and parameter **f** is a callback function invoked when an environment is destroyed. This function returns true if the environment data was successfully allocated; otherwise, it returns false.

The **id** parameter value must be unique; calls to **AllocateEnvironmentData** using a value that has already been allocated will fail. To avoid collisions with environment ids predefined by CLIPS, use the macro constant USER_ENVIRONMENT_DATA as the base index for any ids defined by user code.

For the **size** parameter, you'll typically you'll define a struct containing the various values to be stored in the environment data and use the sizeof operator to pass in the size of the struct to this function which will automatically allocate the specified amount of memory, initialize it to contain all zeroes, and then store the memory in the environment position associated with the **id**

parameter. Once the base storage has been allocated, additional allocation can be performed by user code. When the environment is destroyed, CLIPS automatically deallocates the amount of memory previously allocated for the base storage.
If the **f** parameter value is not a null pointer, then the specified callback function is invoked when the associated environment is destroyed. CLIPS automatically handles the allocation and deallocation of the base storage for environment data (the amount of data specified by the **size** parameter value). If the base storage includes pointers to memory allocated by user code, then this should be deallocated either by an **EnvironmentCleanupFunction** function specified by this function or the function **AddEnvironmentCleanupFunction**.

Environment cleanup functions specified using by the **AllocateEnvironmentData** function are called in ascending order of their **id** parameter value. If the deallocation of your environment data has order dependencies, you can either assign the ids 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.

The function **AddEnvironmentCleanupFunction** adds a callback function to the list of functions invoked when an environment is destroyed. Parameter **env** is a pointer to a previously created environment; parameter **name** is a string that uniquely identifies the callback; parameter **f** is a pointer to the callback function of type **EnvironmentCleanupFunction**; and parameter **p** is the priority of the callback function. The **priority** parameter determines the order in which the callback functions are invoked (higher priority items are called first); the values -2000 to 2000 are reserved for internal use by CLIPS. This function returns true if the callback function was successfully added; otherwise, it returns false.

Environment cleanup functions created using this function are called after all the cleanup functions associated with environment data created using **AllocateEnvironmentData** have been called.

10.2.2 Retrieving Environment Data

```
void *GetEnvironmentData(
   Environment *env,
   unsigned id);
```

The function **GetEnvironmentData** returns a pointer to the environment data associated with the identifier specified by parameter **id**.

10.2.3 Environment Data Example

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 **UserFunctions** function:

```
void UserFunctions(
  Environment *env)
  {
   if (! AllocateEnvironmentData(env,INDEX_DATA,
                                 sizeof(struct indexData),NULL))
     {
      Writeln(env,"Error allocating environment data for INDEX_DATA");
      ExitRouter(env,EXIT_FAILURE);
     }

   IndexData(env)->index = 1;

   AddUDF(env,"get-index","l",0,0,NULL,GetIndex,"GetIndex",NULL);
  }
```

First, the call to **AllocateEnvironmentData** is made. If this fails, then an error message is printed and a call to **ExitRouter** 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, **AddUDF** 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:

```
void GetIndex(Environment *,UDFContext *,UDFValue *);

void GetIndex(
  Environment *env,
  UDFContext *context,
  UDFValue *returnValue)
  {
   returnValue->integerValue = CreateInteger(env,IndexData(env)->index++);
  }
```

Section 11:
Creating a CLIPS Run-time Program

11.1 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.

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 **UserFunctions** 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()
  {
   Environment *env;
   extern Environment *InitCImage_1(void);

   env = InitCImage_1();
```

```
    •
    •                          /* Any user Initialization */
    •
 Reset(env);
 Run(env,-1);
    •
    •                          /* Any other code */
    •
 DestroyEnvironment(env);
}
```

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.

5) Link all regular CLIPS modules together with any user-defined function modules and the 'C' construct modules. Any user-defined functions must have global scope.

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 **build** function does 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 12:
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.

12.1 Environment Functions

The following function calls control the CLIPS environment:

12.1.1 LoadFromString

```
bool LoadFromString(
   Environment *env,
   const char *str,
   size_t length);
```

The function **LoadFromString** loads a set of constructs from a string input source (much like the **Load** function only using a string for input rather than a file). Parameter **env** is a pointer to a previously created environment; parameter **str** is a string containing constructs; and parameter **length** is the maximum number of characters to be read from the input string. If the **length** parameter value is SIZE_MAX, then the **str** parameter value must be terminated by a null character; otherwise, the **length** parameter value indicates the maximum number characters that will be read from the **str** parameter value. This function returns true if no error occurred while loading constructs; otherwise, it returns false.

12.1.2 Clear Callback Functions

```
bool AddClearFunction(
   Environment *env,
   const char *name,
   VoidCallFunction *f,
   int p,
   void *context);

bool RemoveClearFunction(
   Environment *env,
   const char *name);

typedef void VoidCallFunction(
   Environment *env,
   void *context);
```

The function **AddClearFunction** adds a callback function to the list of functions invoked when the CLIPS **clear** command is executed. Parameter **env** is a pointer to a previously created environment; parameter **name** is a string that uniquely identifies the callback for removal using **RemoveClearFunction**; parameter **f** is a pointer to the callback function of type **VoidCallFunction**; parameter **p** is the priority of the callback function; and parameter **context** is a user supplied pointer to data that is passed to the callback function when it is invoked (a null pointer should be used if there is no data that needs to be passed to the callback function). The **priority** parameter determines the order in which the callback functions are invoked (higher priority items are called first); the values -2000 to 2000 are reserved for internal use by CLIPS. This function returns true if the callback function was successfully added; otherwise, it returns false.

The function **RemoveClearFunction** removes a callback function from the list of functions invoked when the CLIPS **clear** command is executed. Parameter **env** is a pointer to a previously created environment; and parameter **name** is the string used to identify the callback when it was added using **AddClearFunction**. The function returns true if the callback was successfully removed; otherwise, it returns false.

12.1.3 Periodic Callback Functions

```
bool AddPeriodicFunction(
   Environment *env,
   const char *name
   VoidCallFunction *f,
   int p,
   void *context);

bool RemovePeriodicFunction(
   Environment *env,
   const char *name);

typedef void VoidCallFunction(
   Environment *env,
   void *context);
```

The function **AddPeriodicFunction** adds a callback function to the list of functions invoked periodically when CLIPS is executing. Among other possible uses, this functionality allows event processing during execution when CLIPS is embedded within an application that must periodically perform tasks. 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 callback.

Parameter **env** is a pointer to a previously created environment; parameter **name** is a string that uniquely identifies the callback for removal using **RemovePeriodicFunction**; parameter **f** is a pointer to the callback function of type **VoidCallFunction**; parameter **p** is the priority of the callback function; and parameter **context** is a user supplied pointer to data that is passed to the callback function when it is invoked (a null pointer should be used if there is no data that needs to be passed to the callback function). The **priority** parameter determines the order in which the callback functions are invoked (higher priority items are called first); the values -2000 to 2000 are reserved for internal use by CLIPS. This function returns true if the callback function was successfully added; otherwise, it returns false.

The function **RemovePeriodicFunction** removes a callback function from the list of functions invoked periodically when CLIPS is executing. Parameter **env** is a pointer to a previously created environment; and parameter **name** is the string used to identify the callback when it was added using **AddPeriodicFunction**. The function returns true if the callback was successfully removed; otherwise, it returns false.

12.1.4 Reset Callback Functions

```
bool AddResetFunction(
   Environment *env,
```

```
   const char *name,
   VoidCallFunction *f,
   int p,
   void *context);

bool RemoveResetFunction(
   Environment *env,
   const char *name);

typedef void VoidCallFunction(
   Environment *env,
   void *context);
```

The function **AddResetFunction** adds a callback function to the list of functions invoked when the CLIPS **reset** command is executed. Parameter **env** is a pointer to a previously created environment; parameter **name** is a string that uniquely identifies the callback for removal using **RemoveResetFunction**; parameter **f** is a pointer to the callback function of type **VoidCallFunction**; parameter **p** is the priority of the callback function; and parameter **context** is a user supplied pointer to data that is passed to the callback function when it is invoked (a null pointer should be used if there is no data that needs to be passed to the callback function). The **priority** parameter determines the order in which the callback functions are invoked (higher priority items are called first); the values -2000 to 2000 are reserved for internal use by CLIPS. This function returns true if the callback function was successfully added; otherwise, it returns false.

The function **RemoveResetFunction** removes a callback function from the list of functions invoked when the CLIPS **reset** command is executed. Parameter **env** is a pointer to a previously created environment; and parameter **name** is the string used to identify the callback when it was added using **AddResetFunction**. The function returns true if the callback was successfully removed; otherwise, it returns false.

12.1.5 File Operations

```
bool BatchStar(
   Environment *env
   const char *fileName);

bool Bload(
   Environment *env,
   const char *fileName);

bool Bsave(
   Environment *env,
   const char *fileName);
```

```
bool Save(
   Environment *env,
   const char *fileName);
```

The function **BatchStar** is the C equivalent of the CLIPS **batch*** command. The **env** parameter is a pointer to a previously created environment; the **filename** parameter is a full or partial path string to an ASCII or UTF-8 file containing CLIPS functions, commands, and constructs. This function returns true if the file was successfully opened; otherwise, it returns false.

The function **Bload** is the C equivalent of the CLIPS **bload** command. The **env** parameter is a pointer to a previously created environment; the **filename** parameter is a full or partial path string to a binary save file that was created using the C **Bsave** function or the CLIPS **bsave** command. This function returns true if the file was successfully opened; otherwise, it returns false.

The function **Bsave** is the C equivalent of the CLIPS **bsave** command. The **env** parameter is a pointer to a previously created environment; the **filename** parameter is a full or partial path string for the binary save file to be created. This function returns true if the file was successfully created; otherwise, it returns false.

The function **Save** is the C equivalent of the CLIPS **save** command. The **env** parameter is a pointer to a previously created environment; the **fileName** parameter is a full or partial path string for the text save file to be created. This function returns true if the file was successfully created; otherwise, it returns false.

12.1.6 Settings

```
bool GetDynamicConstraintChecking(
   Environment *env);

bool GetSequenceOperatorRecognition(
   Environment *env);

bool SetDynamicConstraintChecking(
   Environment *env,
   bool b);

bool SetSequenceOperatorRecognition(
   Environment *env,
   bool b);
```

The function **GetDynamicConstraintChecking** is the C equivalent of the CLIPS **get-dynamic-constraint-checking** command. The **env** parameter is a pointer to a

previously created environment. This function returns true if the dynamic constraint checking behavior is enabled; otherwise, it returns false.

The function **GetSequenceOperatorRecognition** is the C equivalent of the CLIPS **get-sequence-operator-recognition** command. Parameter **env** is a pointer to a previously created environment. This function returns true if the sequence operator recognition behavior is enabled; otherwise, it returns false.

The function **SetDynamicConstraintChecking** is the C equivalent of the CLIPS command **set-dynamic-constraint-checking**. The **env** parameter is a pointer to a previously created environment; the **b** parameter is the new setting for the behavior (either true to enable it or false to disable it). This function returns the old setting for the behavior.

The function **SetSequenceOperatorRecognition** is the C equivalent of the CLIPS **set-sequence-operator-recognition** command. Parameter **env** is a pointer to a previously created environment; and parameter **b** is the new setting for the behavior (either true to enable it or false to disable it). This function returns the old setting for the behavior.

12.2 Debugging Functions

The following function call controls the CLIPS debugging aids:

12.2.1 DribbleActive

```
bool DribbleActive(
   Environment *env);
```

The function **DribbleActive** returns true if the environment specified by parameter **env** has an active dribble file for capturing output; otherwise, it returns false.

12.2.2 GetWatchState and SetWatchState

```
bool GetWatchState(
   Environment *env,
   WatchItem item);

void SetWatchState(
   Environment *env,
   WatchItem item,
   bool b);

typedef enum
  {
```

```
   ALL,
   FACTS,
   INSTANCES,
   SLOTS,
   RULES,
   ACTIVATIONS,
   MESSAGES,
   MESSAGE_HANDLERS,
   GENERIC_FUNCTIONS,
   METHODS,
   DEFFUNCTIONS,
   COMPILATIONS,
   STATISTICS,
   GLOBALS,
   FOCUS
  } WatchItem;
```

The function **GetWatchState** returns the current state of the watch item specified by the parameter **item** in the environment specified by the parameter **env**: true if the watch item is enabled; otherwise, false. If ALL is specified for the parameter **item**, the return value of this function is undefined.

The function **SetWatchState** sets the state of the watch item specified by the parameter **item** in the environment specified by the parameter **env**. If parameter **b** is true, the watch item is enabled; otherwise, it is disabled. If ALL is specified for the parameter **item**, then all watch items are set to the state specified by parameter **b**.

12.3 Deftemplate Functions

The following function calls are used for manipulating deftemplates.

12.3.1 Search, Iteration, and Listing

```
Deftemplate *FindDeftemplate(
   Environment *env,
   const char *name);

Deftemplate *GetNextDeftemplate(
   Environment *env,
   Deftemplate *d);

void GetDeftemplateList(
   Environment *env,
```

```
   CLIPSValue *out,
   Defmodule *d);

void ListDeftemplates(
   Environment *env,
   const char *logicalName,
   Defmodule *d);
```

The function **FindDeftemplate** searches for the deftemplate specified by parameter **name** in the environment specified by parameter **env**. This function returns a pointer to the named deftemplate if it exists; otherwise, it returns a null pointer.

The function **GetNextDeftemplate** provides iteration support for the list of deftemplates in the current module. If parameter **d** is a null pointer, then a pointer to the first **Deftemplate** in the current module is returned by this function; otherwise, a pointer to the next **Deftemplate** following the **Deftemplate** specified by parameter **d** is returned. If parameter **d** is the last **Deftemplate** in the current module, a null pointer is returned.

The function **GetDeftemplateList** is the C equivalent of the CLIPS **get-deftemplate-list** function. Parameter **env** is a pointer to a previously created environment; parameter **out** is a pointer to a **CLIPSValue** allocated by the caller; and parameter **d** is a pointer to a **Defmodule**. The output of the function call—a multifield containing a list of deftemplate names—is stored in the **out** parameter value. If parameter **d** is a null pointer, then deftemplates in all modules will be included in parameter **out**; otherwise, only deftemplates in the specified module will be included.

The function **ListDeftemplates** is the C equivalent of the CLIPS **list-deftemplates** command). Parameter **env** is a pointer to a previously created environment; parameter **logicalName** is the router output destination; and parameter **d** is a pointer to a defmodule. If parameter **d** is a null pointer, then deftemplates in all modules will be listed; otherwise, only deftemplates in the specified module will be listed.

12.3.2 Attributes

```
const char *DeftemplateModule(
   Deftemplate *d);

const char *DeftemplateName(
   Deftemplate *d);

const char *DeftemplatePPForm(
   Deftemplate *d);
```

```
void DeftemplateSlotNames(
   Deftemplate *d,
   CLIPSValue *out);
```

The function **DeftemplateModule** is the C equivalent of the CLIPS **deftemplate-module** command. The return value of this function is the name of the module in which the deftemplate specified by parameter **d** is defined.

The function **DeftemplateName** returns the name of the deftemplate specified by the **d** parameter.

The function **DeftemplatePPForm** returns the text representation of the **Deftemplate** specified by the **d** parameter. The null pointer is returned if the text representation is not available.

The function **DeftemplateSlotNames** is the C equivalent of the CLIPS **deftemplate-slot-names** function. Parameter **d** is a pointer to a **Deftemplate**; and parameter **out** is a pointer to a **CLIPSValue** allocated by the caller. The output of the function call—a multifield containing the deftemplate's slot names—is stored in the **out** parameter value. For implied deftemplates, a multifield value containing the single symbol *implied* is returned.

12.3.3 Deletion

```
bool DeftemplateIsDeletable(
   Deftemplate *d);

bool Undeftemplate(
   Deftemplate *d,
   Environment *env);
```

The function **DeftemplateIsDeletable** returns true if the deftemplate specified by parameter **d** can be deleted; otherwise it returns false.

The **Undeftemplate** function is the C equivalent of the CLIPS **undeftemplate** command. It deletes the deftemplate specified by parameter **d**; or if parameter **d** is a null pointer, it deletes all deftemplates in the environment specified by parameter **env**. This function returns true if the deletion is successful; otherwise, it returns false.

12.3.4 Watching Deftemplate Facts

```
bool DeftemplateGetWatch(
   Deftemplate *d);
```

```
void DeftemplateSetWatch(
   Deftemplate *d,
   bool b);
```

The function **DeftemplateGetWatch** returns true if facts are being watched for the deftemplate specified by the **d** parameter value; otherwise, it returns false.

The function **DeftemplateSetWatch** sets the fact watch state for the deftemplate specified by the **d** parameter value to the value specified by the parameter **b**.

12.3.5 Slot Attributes

```
bool DeftemplateSlotAllowedValues(
   Deftemplate *d,
   const char *name,
   CLIPSValue *out);

bool DeftemplateSlotCardinality(
   Deftemplate *d,
   const char *name,
   CLIPSValue *out);

bool DeftemplateSlotRange(
   Deftemplate *d,
   const char *name,
   CLIPSValue *out);

bool DeftemplateSlotDefaultValue(
   Deftemplate *d,
   const char *name,
   CLIPSValue *out);

bool DeftemplateSlotTypes(
   Deftemplate *d,
   const char *name,
   CLIPSValue *out);
```

The function **DeftemplateSlotAllowedValues** is the C equivalent of the CLIPS **deftemplate-slot-allowed-values** function. The function **DeftemplateSlotCardinality** is the C equivalent of the CLIPS **deftemplate-slot-cardinality** function. The function **DeftemplateSlotRange** is the C equivalent of the CLIPS **deftemplate-slot-range** function. The function **DeftemplateSlotDefaultValue** is the C equivalent of the CLIPS **deftemplate-slot-default-value** function. The function **DeftemplateSlotTypes** is the C equivalent of the CLIPS **deftemplate-slot-types** function.

Parameter **d** is a pointer to a **Deftemplate**; parameter **name** specifies a valid slot name for the specified deftemplate; and parameter **out** is a pointer to a **CLIPSValue** allocated by the caller. The output of the function call—a multifield containing the attribute values—is stored in the **out** parameter value. These function return true if a valid slot name was specified and the output value is successfully set; otherwise, false is returned.

12.3.6 Slot Predicates

```
bool DeftemplateSlotExistP(
   Deftemplate *d,
   const char *name);

bool DeftemplateSlotMultiP(
   Deftemplate *d,
   const char *name);

bool DeftemplateSlotSingleP(
   Deftemplate *d,
   const char *name);

DefaultType DeftemplateSlotDefaultP(
   Deftemplate *deftemplatePtr,
   const char *slotName);

typedef enum
  {
   NO_DEFAULT,
   STATIC_DEFAULT,
   DYNAMIC_DEFAULT
  } DefaultType;
```

The function **DeftemplateSlotExistp** is the C equivalent of the CLIPS **deftemplate-slot-existp** function. Parameter **d** is a pointer to a **Deftemplate**; and parameter **name** specifies a slot name. This function returns true if specified slot exists; otherwise it returns false.

The function **DeftemplateSlotMultiP** is the C equivalent of the CLIPS **deftemplate-slot-multip** function. Parameter **d** is a pointer to a **Deftemplate**; and parameter **name** specifies a valid slot name. This function returns true if the specified slot is a multifield slot; otherwise it returns false.

The function **DeftemplateSlotSingleP** is the C equivalent of the CLIPS **deftemplate-slot-singlep** function. Parameter **d** is a pointer to a **Deftemplate**; and parameter **name** specifies a valid slot name. This function returns true if the specified slot is a single-field slot; otherwise it returns false.

The function **DeftemplateSlotDefaultP** is the C equivalent of the CLIPS **deftemplate-slot-defaultp** function. Parameter **d** is a pointer to a **Deftemplate**; and parameter **name** specifies a valid slot name. This function returns the **DefaultType** enumeration for the specified slot.

12.4 Fact Functions

The following function calls manipulate and display information about facts.

12.4.1 Iteration and Listing

```
Fact *GetNextFact(
   Environment *env,
   Fact *f);

Fact *GetNextFactInTemplate(
   Deftemplate *d,
   Fact *f);

void GetFactList(
   Environment *env,
   CLIPSValue *out,
   Defmodule *d);

void Facts(
   Environment *env,
   const char *logicalName,
   Defmodule *d,
   long long start,
   long long end,
   long long max);

void PPFact(
   Fact *f,
   const char *logicalName,
   bool ignoreDefaults);
```

The function **GetNextFact** provides iteration support for the list of facts in an environment. If parameter **f** is a null pointer, then a pointer to the first **Fact** in the environment specified by parameter **env** is returned by this function; otherwise, a pointer to the next **Fact** following the **Fact** specified by parameter **f** is returned. If parameter **f** is the last **Fact** in the specified environment, a null pointer is returned.

The function **GetNextFactInTemplate** provides iteration support for the list of facts belonging to a deftemplate. If parameter **f** is a null pointer, then a pointer to the first **Fact** for the deftemplate specified by parameter **d** is returned by this function; otherwise, a pointer to the next **Fact** of the specified deftemplate following the **Fact** specified by parameter **f** is returned. If parameter **f** is the last **Fact** for the specified deftemplate, a null pointer is returned.

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

The function **GetFactList** is the C equivalent of the CLIPS **get-fact-list** function. Parameter **env** is a pointer to a previously created environment; parameter **out** is a pointer to a **CLIPSValue** allocated by the caller; and parameter **d** is a pointer to a **Defmodule**. The output of the function call—a multifield containing a list of fact addresses—is stored in the **out** parameter value. If parameter **d** is a null pointer, then all facts in all modules will be included in parameter **out**; otherwise, only facts associated with deftemplates in the specified module will be included.

The function **Facts** is the C equivalent of the CLIPS **facts** command. Parameter env is a pointer to a previously created environment; parameter **logicalName** is the router output destination; parameter **d** is a pointer to a **Defmodule**; parameter **start** is the lower fact index range of facts to be listed; parameter **end** is the upper fact index range of facts to be listed; and parameter **max** is the maximum number of facts to be listed. If parameter **d** is a non-null pointer, then only facts visible to the specified module are printed; otherwise, all facts will be printed. A value of -1 for the **start**, **end**, or **max** parameter indicates the parameter is unspecified and should not restrict the facts that are listed.

The function **PPFact** is the C equivalent of the CLIPS **ppfact** command. Parameter **f** is a pointer to the **Fact** to be displayed; parameter **logicalName** is the router output destination; and parameter **ignoreDefaults** is a boolean flag indicating whether slots should be excluded from display if their current value is the same as their static default value.

12.4.2 Attributes

```
Deftemplate *FactDeftemplate(
   Fact *f);

long long FactIndex(
   Fact *f);

void FactPPForm(
   Fact *f,
```

```
   StringBuilder *sb,
   bool ignoreDefaults);

void FactSlotNames(
   Fact *f,
   CLIPSValue *out);

GetSlotError GetFactSlot(
   Fact *f,
   const char *name,
   CLIPSValue *out);

typedef enum
  {
   GSE_NO_ERROR,
   GSE_NULL_POINTER_ERROR,
   GSE_INVALID_TARGET_ERROR,
   GSE_SLOT_NOT_FOUND_ERROR,
  } GetSlotError;
```

The function **FactDeftemplateModule** returns a pointer to the **Deftemplate** associated with the **Fact** specified by parameter **f**.

The function **FactIndex** is the C equivalent of the CLIPS **fact-index** command. It returns the fact-index of the fact specified by parameter **f**.

The function **FactPPForm** stores the text representation of the **Fact** specified by the **f** parameter in the **StringBuilder** specified by parameter **sb**. The parameter **ignoreDefaults** is a boolean flag indicating whether slots should be excluded from display if their current value is the same as their static default value

The function **FactSlotNames** is the C equivalent of the CLIPS **fact-slot-names** function. Parameter **f** is a pointer to a **Fact**; and parameter **out** is a pointer to a **CLIPSValue** allocated by the caller. The output of the function call—a multifield containing the facts's slot names—is stored in the **out** parameter value. For ordered facts, a multifield value containing the single symbol *implied* is returned.

The function **GetFactSlot** retrieves the slot value specified by parameter **name** from the fact specified by parameter **f** and stores it in parameter **out**, a **CLIPSValue** allocated by the caller. For ordered facts—which have an implied multifield slot—a null pointer or the string "implied" should be used for the **name** parameter value.

The **GetFactSlot** function returns GSE_NO_ERROR if the slot value is successfully retrieved; otherwise it returns GSE_NULL_POINTER_ERROR if any of the function arguments are NULL pointers (except for the name parameter which may be NULL for an ordered fact),

GSE_INVALID_TARGET_ERROR if the fact specified by parameter **f** has been retracted, and GSE_SLOT_NOT_FOUND_ERROR if the fact does not have the specified slot.

12.4.3 Deletion

```
RetractError RetractAllFacts(
  Environment *env);

bool FactExistp(
   Fact *f);
```

The function **RetractAllFacts** retracts all of the facts in the environment specified by parameter **env**. It returns RE_NO_ERROR if all facts were successfully retracted. See the **Retract** command in section 3.3.3 for the list of error codes in the **RetractError** enumeration.

The function **FactExistp** is the C equivalent of the CLIPS **fact-existp** function. It returns true if the fact has not been retracted. The parameter **f** must be a **Fact** that has either not been retracted or has been retained (see section 5.2).

12.4.4 Loading and Saving Facts

```
bool LoadFacts(
   Environment *env,
   const char *fileName);

bool LoadFactsFromString(
   Environment *env,
   const char *str,
   size_t length);

bool SaveFacts(
   Environment *env,
   const char *filename,
   SaveScope scope);

typedef enum
  {
   LOCAL_SAVE,
   VISIBLE_SAVE
  } SaveScope;
```

The function **LoadFacts** is the C equivalent of the CLIPS **load-facts** command. Parameter **env** is a pointer to a previously created environment; and parameter **fileName** is a full or partial path string to an ASCII or UTF-8 file containing facts. This function returns true if no errors occurred while loading facts; otherwise, it returns false.

The function **LoadFactsFromString** loads a set of facts from a string input source (much like the **LoadFacts** function only using a string for input rather than a file). Parameter **env** is a pointer to a previously created environment; parameter **str** is a string containing facts; and parameter **length** is the maximum number of characters to be read from the input string. If the **length** parameter value is SIZE_MAX, then the **str** parameter value must be terminated by a null character; otherwise, the **length** parameter value indicates the maximum number characters that will be read from the **str** parameter value. This function returns true if no error occurred while loading facts; otherwise, it returns false.

The function **SaveFacts** is the C equivalent of the CLIPS **save-facts** command. Parameter **env** is a pointer to a previously created environment; parameter **fileName** is a full or partial path string to the fact save file that will be created; and parameter **scope** indicates whether all facts visible to the current module should be saved (VISIBLE_SAVE) or just those associated with deftemplates defined in the current module (LOCAL_SAVE). This function returns true if no errors occurred while saving facts; otherwise, it returns false.

12.4.5 Settings

```
bool GetFactDuplication(
   Environment *env);

bool SetFactDuplication(
   Environment *env,
   bool b);
```

The function **GetFactDuplication** is the C equivalent of the CLIPS **get-fact-duplication** command. The **env** parameter is a pointer to a previously created environment. This function returns the boolean value corresponding to the current setting.

The function **SetFactDuplication** is the C equivalent of the CLIPS **set-fact-duplication** command. The **env** parameter is a pointer to a previously created environment; the parameter **b** is the new setting for the behavior. This function returns the old setting for the behavior.

12.4.6 Detecting Changes to Facts

```
bool GetFactListChanged(
   Environment *env);
```

```
void SetFactListChanged(
   Environment *env,
   bool b);
```

The function **GetFactsChanged** returns true if changes to facts for the environment specified by parameter **env** have occurred (either assertions, retractions, or modifications); otherwise, it returns false. To track future changes, **SetFactsChanged** should reset the change tracking value to false.

The function **SetFactsChanged** sets the facts change tracking value for the environment specified by the parameter **env** to the value specified by the parameter **b**.

12.5 Deffacts Functions

The following function calls are used for manipulating deffacts.

12.5.1 Search, Iteration, and Listing

```
Deffacts *FindDeffacts(
   Environment *env,
   const char *name);

Deffacts *GetNextDeffacts(
   Environment *env,
   Deffacts *d);

void GetDeffactsList(
   Environment *env,
   CLIPSValue *out,
   Defmodule *d);

void ListDeffacts(
   Environment *env,
   const char *logicalName,
   Defmodule *d);
```

The function **FindDeffacts** searches for the deffacts specified by parameter **name** in the environment specified by parameter **env**. This function returns a pointer to the named deffacts if it exists; otherwise, it returns a null pointer.

The function **GetNextDeffacts** provides iteration support for the list of deffacts in the current module. If parameter **d** is a null pointer, then a pointer to the first **Deffacts** in the current module is returned by this function; otherwise, a pointer to the next **Deffacts** following the

Deffacts specified by parameter **d** is returned. If parameter **d** is the last **Deffacts** in the current module, a null pointer is returned.

The function **GetDeffactsList** is the C equivalent of the CLIPS **get-deffacts-list** function. Parameter **env** is a pointer to a previously created environment; parameter **out** is a pointer to a **CLIPSValue** allocated by the caller; and parameter **d** is a pointer to a **Defmodule**. The output of the function call—a multifield containing a list of deffacts names—is stored in the **out** parameter value. If parameter **d** is a null pointer, then deffacts in all modules will be included in parameter **out**; otherwise, only deffacts in the specified module will be included.

The function **ListDeffacts** is the C equivalent of the CLIPS **list-deffacts** command. Parameter **env** is a pointer to a previously created environment; parameter **logicalName** is the router output destination; and parameter **d** is a pointer to a defmodule. If parameter **d** is a null pointer, then deffacts in all modules will be listed; otherwise, only deffacts in the specified module will be listed.

12.5.2 Attributes

```
const char *DeffactsModule(
   Deffacts *d);

const char *DeffactsName(
   Deffacts *d);

const char *DeffactsPPForm(
   Deffacts *d);
```

The function **DeffactsModule** is the C equivalent of the CLIPS **deffacts-module** function. The return value of this function is the name of the module in which the deffacts specified by parameter **d** is defined.

The function **DeffactsName** returns the name of the deffacts specified by the **d** parameter.

The function **DeffactsPPForm** returns the text representation of the **Deffacts** specified by the **d** parameter. The null pointer is returned if the text representation is not available.

12.5.3 Deletion

```
bool DeffactsIsDeletable(
   Deffacts *d);

bool Undeffacts(
   Deffacts *d,
   Environment *env);
```

The function **DeffactsIsDeletable** returns true if the deffacts specified by parameter **d** can be deleted; otherwise it returns false.

The **Undeffacts** function is the C equivalent of the CLIPS **undeffacts** command. It deletes the deffacts specified by parameter **d**; or if parameter **d** is a null pointer, it deletes all deffacts in the environment specified by parameter **env**. This function returns true if the deletion is successful; otherwise, it returns false.

12.6 Defrule Functions

The following function calls are used for manipulating defrules.

12.6.1 Search, Iteration, and Listing

```
Defrule *FindDefrule(
   Environment *env,
   const char *name);

Defrule *GetNextDefrule(
   Environment *env,
   Defrule *d);

void GetDefruleList(
   Environment *env,
   CLIPSValue *out,
   Defmodule *d);

void ListDefrules(
   Environment *env,
   const char *logicalName,
   Defmodule *d);
```

The function **FindDefrule** searches for the defrule specified by the **name** parameter in the environment specified by the **env** parameter. This function returns a pointer to the named defrule if it exists; otherwise, it returns a null pointer.

The function **GetNextDefrule** provides iteration support for the list of defrules in the current module. If the **d** parameter value is a null pointer, then a pointer to the first **Defrule** in the current module is returned by this function; otherwise, the next **Defrule** following the **d** parameter value is returned. If the **d** parameter is the last **Defrule** in the current module, a null pointer is returned.

The function **GetDefruleList** is the C equivalent of the CLIPS **get-defrule-list** function. The **env** parameter is a pointer to a previously created environment; the **out** parameter is a pointer to a **CLIPSValue** allocated by the caller; and the **d** parameter is a pointer to a **Defmodule**. The output of the function call—a multifield containing a list of defrule names—is stored in the **out** parameter value. If the parameter **d** is a null pointer, then defrules in all modules will be included in the out parameter value; otherwise, only defrules in the specified module will be included.

The function **ListDefrules** is the C equivalent of the CLIPS **list-defrules** command. The **env** parameter is a pointer to a previously created environment; the **logicalName** parameter is the router output destination; and the **d** parameter is a pointer to a defmodule. If the parameter **d** is a null pointer, then defrules in all modules will be listed; otherwise, only defrules in the specified module will be listed.

12.6.2 Attributes

```
const char *DefruleModule(
   Defrule *d);

const char *DefruleName(
   Defrule *d);

const char *DefrulePPForm(
   Defrule *d);
```

The function **DefruleModule** is the C equivalent of the CLIPS **defrule-module** command). The return value of this function is the name of the module in which the **Defrule** specified by the **d** parameter is defined.

The function **DefruleName** returns the name of the **Defrule** specified by the **d** parameter.

The function **DefrulePPForm** returns the text representation of the **Defrule** specified by the **d** parameter. The null pointer is returned if the text representation is not available.

12.6.3 Deletion

```
bool DefruleIsDeletable(
   Defrule *d);

bool Undefrule(
   Defrule *d,
   Environment *env);
```

The function **DefruleIsDeletable** returns true if the **Defrule** specified by the **d** parameter value can be deleted; otherwise it returns false.

The function **Undefrule** is the C equivalent of the CLIPS **undefrule** command. It deletes the defrule specified by the **d** parameter; or if the **d** parameter is a null pointer, it deletes all defrules in the environment specified by the **env** parameter. The function returns true if the deletion was successful; otherwise, it returns false.

12.6.4 Watch Activations and Firings

```
bool DefruleGetWatchActivations(
   Defrule *d);

bool DefruleGetWatchFirings(
   Defrule *d);

void DefruleSetWatchActivations(
   Defrule *d,
   bool b);

void DefruleSetWatchFirings(
   Defrule *d,
   bool b);
```

The function **DefruleGetWatchActivations** returns true if rule activations are being watched for the defrule specified by the **d** parameter value; otherwise, it returns false.

The function **DefruleGetWatchFirings** returns true if rule firings are being watched for the **Defrule** specified by the **d** parameter; otherwise, it returns false.

The function **DefruleSetWatchActivations** sets the rule activations watch state for the defrule specified by the **d** parameter value to the value specified by the parameter **b**.

The function **DefruleSetWatchFirings** sets the rule firings watch state for the defrule specified by the **d** parameter value to the value specified by the parameter **b**.

12.6.5 Breakpoints

```
bool DefruleHasBreakpoint(
   Defrule *d);

bool RemoveBreak(
   Defrule *d);
```

```
void SetBreak(
   Defrule *d);

void ShowBreaks(
   Environment *env,
   const char *logicalName,
   Defmodule *d);
```

The function **DefruleHasBreakpoint** returns true if the defrule specified by the **d** parameter value has a breakpoint set; otherwise it returns false.

The function **RemoveBreak** is the C equivalent of the CLIPS **remove-break** command. It returns false if a breakpoint does not exist for the defrule specified by the **d** parameter value; otherwise, it removes the breakpoint and returns true;

The function **SetBreak** is the C equivalent of the CLIPS **set-break** command. It sets a breakpoint for the defrule specified by the **d** parameter value.

The function **ShowBreaks** is the C equivalent of the CLIPS **show-breaks** command. The **env** parameter is a pointer to a previously created environment; the **logicalName** parameter is the router output destination; and the **d** parameter is a pointer to a defmodule. If the parameter **d** is a null pointer, then breakpoints for defrules in all modules will be listed; otherwise, only breakpoints for defrules in the specified module will be listed.

12.6.6 Matches

```
void Matches(
   Defrule *d,
   Verbosity v,
   CLIPSValue *out);

typedef enum
  {
   VERBOSE,
   SUCCINCT,
   TERSE
  } Verbosity;
```

The function **Matches** is the C equivalent of the CLIPS **matches** command. The **d** parameter is a pointer to a **Defrule**; the **v** parameter specifies the level of information displayed in the printed output; and the **out** parameter is a pointer to a CLIPSValue allocated by the caller. The output of the function call—a multifield containing three integer fields indicating the number of pattern matches, partial matches, and activations—is stored in the **out** parameter value.

12.6.7 Refresh

```
void Refresh(
   Defrule *d);
```

The function **Refresh** is the C equivalent of the CLIPS **refresh** command.

12.7 Agenda Functions

The following function calls are used for manipulating the agenda.

12.7.1 Iteration and Listing

```
Activation *GetNextActivation(
   Environment *env,
   Activation *a);

FocalModule *GetNextFocus(
   Environment *env,
   FocalModule *fm);

void GetFocusStack(
   Environment *env,
   CLIPSValue *out);

void Agenda(
   Environment *env,
   const char *logicalName,
   Defmodule *d);

void ListFocusStack(
   Environment *env,
   const char *logicalName);
```

The function **GetNextActivation** provides iteration support for the list of activations on the agenda of the current module in an environment. If parameter **a** is a null pointer, then a pointer to the first **Activation** on the agenda of the current module in the environment specified by parameter **env** is returned by this function; otherwise, a pointer to the next **Activation** following the **Activation** specified by parameter **a** is returned. If parameter **a** is the last **Activation** on the agenda of the current module in the specified environment, a null pointer is returned.

The function **GetNextFocus** provides iteration support for the list of modules on the focus stack of an environment. If parameter **fm** is a null pointer, then a pointer to the first **FocusModule** on the focus stack in the environment specified by parameter **env** is returned by this function; otherwise, a pointer to the next **FocusModule** following the **FocusModule** specified by parameter **fm** is returned. If parameter **fm** is the last **FocusModule** on the agenda of the current module in the specified environment, a null pointer is returned.

The function **GetFocusStack** is the C equivalent of the CLIPS **get-focus-stack** function. Parameter **env** is a pointer to a previously created environment; and parameter **out** is a pointer to a **CLIPSValue** allocated by the caller. The output of the function call—a multifield containing a list of defmodule names—is stored in the **out** parameter value.

The function **Agenda** is the C equivalent of the CLIPS **agenda** command. Parameter **env** is a pointer to a previously created environment; parameter **logicalName** is the router output destination; and parameter **d** is a pointer to a **Defmodule**. If parameter **d** is a null pointer, then the agenda of every module is listed; otherwise, only the agenda of the specified module is listed.

The function **ListFocusStack** is the C equivalent of the CLIPS **list-focus-stack** command. Parameter **env** is a pointer to a previously created environment; and parameter **logicalName** is the router output destination.

12.7.2 Activation Attributes

```
const char *ActivationRuleName(
   Activation *a);

void ActivationPPForm(
   Activation *a,
   StringBuilder *sb);

int ActivationGetSalience(
   Activation *a);

int ActivationSetSalience(
   Activation *a,
   int s);
```

The function **ActivationRuleName** returns the name of the defrule that generated the activation specified by parameter **a**.

The function **ActivationPPForm** stores the text representation of the **Activation** specified by parameter **a** in the **StringBuilder** specified by parameter **sb**.

The function **ActivationGetSalience** returns the salience of the activation specified by parameter **a**. This salience value may be different from the the salience value of the defrule which generated the activation (due to dynamic salience).

The function **ActivationSetSalience** sets the salience of the activation specified by parameter **a** to the integer specified by parameter **s**. Salience values greater than 10,000 will assign the value 10,000 instead and salience values less than -10,000 will assign the value -10 instead. The function **ReorderAgenda** should be called after salience values have been changed to update the agenda.

12.7.3 FocalModule Attributes

```
const char *FocalModuleName(
  FocalModule *fm);

Defmodule *FocalModuleModule(
  FocalModule *fm);
```

The function **FocalModuleName** returns the name of the defmodule associated with the **FocalModule** specified by parameter **fm**.

The function **FocalModuleModule** returns a pointer to the **Defmodule** associated with the **FocalModule** specified by parameter **fm**.

12.7.4 Rule Fired Callback Functions

```
bool AddBeforeRuleFiresFunction(
   Environment *env,
   const char *name,
   RuleFiredFunction *f,
   int p,
   void *context);

bool AddAfterRuleFiresFunction(
   Environment *env,
   const char *name,
   RuleFiredFunction *f,
   int p,
   void *context);

bool RemoveBeforeRuleFiresFunction(
   Environment *env,
   const char *name);
```

```
bool RemoveAfterRuleFiresFunction(
   Environment *env,
   const char *name);

typedef void RuleFiredFunction(
   Environment *env,
   Activation *a,
   void *context);
```

The function **AddBeforeRuleFiresFunction** adds a callback function to the list of functions invoked after a rule executes. Parameter **env** is a pointer to a previously created environment; parameter **name** is a string that uniquely identifies the callback for removal using **RemoveBeforeRuleFiresFunction**; parameter **f** is a pointer to the callback function of type **RuleFiredFunction**; parameter **p** is the priority of the callback function; and parameter **context** is a user supplied pointer to data that is passed to the callback function when it is invoked (a null pointer should be used if there is no data that needs to be passed to the callback function). The **priority** parameter determines the order in which the callback functions are invoked (higher priority items are called first); the values -2000 to 2000 are reserved for internal use by CLIPS. This function returns true if the callback function was successfully added; otherwise, it returns false.

The function **AddAfterRuleFiresFunction** adds a callback function to the list of functions invoked after a rule executes. Parameter **env** is a pointer to a previously created environment; parameter **name** is a string that uniquely identifies the callback for removal using **RemoveAfterRuleFiresFunction**; parameter **f** is a pointer to the callback function of type **RuleFiredFunction**; parameter **p** is the priority of the callback function; and parameter **context** is a user supplied pointer to data that is passed to the callback function when it is invoked (a null pointer should be used if there is no data that needs to be passed to the callback function). The **priority** parameter determines the order in which the callback functions are invoked (higher priority items are called first); the values -2000 to 2000 are reserved for internal use by CLIPS. This function returns true if the callback function was successfully added; otherwise, it returns false.

When invoked, the **RuleFiredFunction** is passed parameter **a** that is a pointer to the **Activation** being executed. In the event that no rules are executed, the callbacks added by **AddAfterRuleFiresFunction** are invoked once with the parameter value **a** set to a null pointer.

The function **RemoveBeforeRuleFiresFunction** removes a callback function from the list of functions invoked before a rule executes. Parameter **env** is a pointer to a previously created environment; and parameter **name** is the string used to identify the callback when it was added using **AddBeforeRuleFiresFunction**. The function returns true if the callback was successfully removed; otherwise, it returns false.

The function **RemoveAfterRuleFiresFunction** removes a callback function from the list of functions invoked after a rule executes. Parameter **env** is a pointer to a previously created environment; and parameter **name** is the string used to identify the callback when it was added using **AddAfterRuleFiresFunction**. The function returns true if the callback was successfully removed; otherwise, it returns false.

12.7.6 Manipulating the Focus Stack

```
void ClearFocusStack(
   Environment *env);

void Focus(
   Defmodule *d);

Defmodule *PopFocus(
   Environment *env);

Defmodule *GetFocus(
   Environment *env);
```

The function **ClearFocusStack** is the C equivalent of the CLIPS **clear-focus-stack** command.

The function **Focus** is the C equivalent of the CLIPS **focus** command.

The function **PopFocus** is the C equivalent of the CLIPS **pop-focus** function. It removes the current focus from the focus stack and returns the **Defmodule** associated with that focus.

The function **GetFocus** is the C equivalent of the CLIPS **get-focus** function. It returns a pointer to the **Defmodule** that is the current focus on the focus stack. A null pointer is returned if the focus stack is empty.

12.7.7 Manipulating the Agenda

```
void RefreshAgenda(
   Defmodule *d);

void RefreshAllAgendas(
   Environment *env);

void ReorderAgenda(
   Defmodule *d);
```

```
void ReorderAllAgendas(
   Environment *env);

void DeleteActivation(
   Activation *a);

void DeleteAllActivations(
   Defmodule *d);
```

The function **RefreshAgenda** is the C equivalent of the CLIPS **refresh-agenda** command. For the agenda of the module specified by parameter **d**, it recomputes the salience values for all activations and then reorders the agenda.

The function **RefreshAllAgendas** invokes the **RefreshAgenda** function for every module in the evironment specified by parameter **env**.

The function **ReorderAgenda** reorders the agenda of the module specified by parameter **d** using the current conflict resolution strategy and current activation saliences.

The function **ReorderAllAgendas** invokes the **ReorderAgenda** function for every module in the evironment specified by parameter **env**.

The function **DeleteActivation** removes an activation from its agenda.

The function **DeleteAllActivations** removes all activations from the agenda of the module specified by parameter **d**;

12.7.8 Detecting Changes to the Agenda

```
bool GetAgendaChanged(
   Environment *env);

void SetAgendaChanged(
   Environment *env,
   bool b);
```

The function **GetAgendaChanged** returns true if changes to the agenda for the environment specified by parameter **env** have occurred (either activations, firings, or deactivations); otherwise, it returns false. To track future changes, **SetAgendaChanged** should reset the change tracking value to false.

The function **SetAgendaChanged** sets the agenda change tracking value for the environment specified by the parameter **env** to the value specified by the parameter **b**.

12.7.9 Settings

```
SalienceEvaluationType GetSalienceEvaluation(
   Environment *env);

SalienceEvaluationType SetSalienceEvaluation(
   Environment *env,
   SalienceEvaluationType set);

StrategyType GetStrategy(
   Environment *env);

StrategyType SetStrategy(
   Environment *env,
   StrategyType st);

typedef enum
  {
   WHEN_DEFINED,
   WHEN_ACTIVATED,
   EVERY_CYCLE
  } SalienceEvaluationType;

typedef enum
  {
   DEPTH_STRATEGY,
   BREADTH_STRATEGY,
   LEX_STRATEGY,
   MEA_STRATEGY,
   COMPLEXITY_STRATEGY,
   SIMPLICITY_STRATEGY,
   RANDOM_STRATEGY
  } StrategyType;
```

The function **GetSalienceEvaluation** is the C equivalent of the CLIPS **get-salience-evaluation** command. The **env** parameter is a pointer to a previously created environment. This function returns the enumeration value corresponding to the current setting.

The function **SetSalienceEvaluation** is the C equivalent of the CLIPS **set-salience-evaluation** command). The **env** parameter is a pointer to a previously created environment; and the parameter **set** is the new setting for the behavior. This function returns the old setting for the behavior.

The function **GetStrategy** is the C equivalent of the CLIPS **get-strategy** command. The **env** parameter is a pointer to a previously created environment. This function returns the enumeration value corresponding to the current setting.

The function **SetStrategy** is the C equivalent of the CLIPS **set-strategy** command. The **env** parameter is a pointer to a previously created environment; and the parameter **st** is the new setting for the behavior. This function returns the old setting for the behavior.

12.7.10 Examples

12.7.10.1 Calling a Function After Each Rule Firing

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

```
#include "clips.h"

void PrintPeriod(Environment *,Activation *,void *);

int main()
  {
   Environment *env;

   env = CreateEnvironment();

   Build(env,"(defrule loop"
             "   ?f <- (loop)"
             "   =>"
             "   (retract ?f)"
             "   (assert (loop)))");

   AssertString(env,"(loop)");

   AddAfterRuleFiresFunction(env,"print-dot",PrintPeriod,0,NULL);

   Run(env,20);

   Write(env,"\n");
  }

void PrintPeriod(
  Environment *env,
  Activation *a,
  void *context)
  {
   Write(env,".");
  }
```

When run, the program produces the following output:

```
.....................
```

12.8 Defglobal Functions

The following function calls are used for manipulating defglobals.

12.8.1 Search, Iteration, and Listing

```
Defglobal *FindDefglobal(
   Environment *env,
   const char *name);

Defglobal *GetNextDefglobal(
   Environment *env,
   Defglobal *d);

void GetDefglobalList(
   Environment *env,
   CLIPSValue *out,
   Defmodule *d);

void ListDefglobals(
   Environment *env,
   const char *logicalName,
   Defmodule *d);

void ShowDefglobals(
   Environment *env,
   const char *logicalName,
   Defmodule *d);
```

The function **FindDefglobal** searches for the defrule specified by the **name** parameter in the environment specified by the **env** parameter. For example, to retrieve the value of the global variable ?*x*, use the value "x" for the **name** parameter. This function returns a pointer to the named defglobal if it exists; otherwise, it returns a null pointer.

The function **GetNextDefglobal** provides iteration support for the list of defglobals in the current module. If parameter **d** is a null pointer, then a pointer to the first **Defglobal** in the current module is returned by this function; otherwise, a pointer to the next **Defglobal**

following the **Defglobal** specified by parameter **d** is returned. If parameter **d** is the last **Defglobal** in the current module, a null pointer is returned.

The function **GetDefglobalList** is the C equivalent of the CLIPS **get-defglobal-list** function). The **env** parameter is a pointer to a previously created environment; the **out** parameter is a pointer to a **CLIPSValue** allocated by the caller; and the **d** parameter is a pointer to a **Defmodule**. The output of the function call—a multifield containing a list of defglobal names—is stored in the **out** parameter value. If the parameter **d** is a null pointer, then defglobals in all modules will be included in the out parameter value; otherwise, only defglobals in the specified module will be included.

The function **ListDefglobals** is the C equivalent of the CLIPS **list-defglobals** command. The parameter **env** is a pointer to a previously created environment; the parameter **logicalName** is the router output destination; and the parameter **d** is a pointer to a defmodule. If the parameter **d** is a null pointer, then defglobals in all modules will be listed; otherwise, only defglobals in the specified module will be listed.

The function **ShowDefglobals** is the C equivalent of the CLIPS **show-defglobals** command. The parameter **env** is a pointer to a previously created environment; the parameter **logicalName** is the router output destination; and the parameter **d** is a pointer to a defmodule. If the parameter **d** is a null pointer, then defglobals in all modules will be listed with their current value; otherwise, only defglobals in the specified module will be listed with their current value.

12.8.2 Attributes

```
const char *DefglobalModule(
   Defglobal *d);

const char *DefglobalName(
   Defglobal *d);

const char *DefglobalPPForm(
   Defglobal *d);

void DefglobalValueForm(
   Defglobal *d,
   StringBuilder *sb);

void DefglobalGetValue(
   Defglobal *d,
   CLIPSValue *out);
```

```
void DefglobalSetValue(
   Defglobal *d,
   CLIPSValue *value);

void DefglobalSetInteger (
   Defglobal *d,
   long long value);

void DefglobalSetFloat (
   Defglobal *d,
   double value);

void DefglobalSetSymbol (
   Defglobal *d,
   const char *value);

void DefglobalSetString (
   Defglobal *d,
   const char *value);

void DefglobalSetInstanceName (
   Defglobal *d,
   const char *value);

void DefglobalSetCLIPSInteger (
   Defglobal *d,
   CLIPSInteger *value);

void DefglobalSetCLIPSFloat (
   Defglobal *d,
   CLIPSFloat *value);

void DefglobalSetCLIPSLexeme (
   Defglobal *d,
   CLIPSLexeme *value);

void DefglobalSetFact (
   Defglobal *d,
   Fact *value);

void DefglobalSetInstance (
   Defglobal *d,
   Instance *value);
```

```
void DefglobalSetMultifield (
   Defglobal *d,
   Multifield *value);

void DefglobalSetCLIPSExternalAddress (
   Defglobal *d,
   CLIPSExternalAddress *value);
```

The function **DefglobalModule** is the C equivalent of the CLIPS **defglobal-module** command. The return value of this function is the name of the module in which the **Defglobal** specified by the **d** parameter is defined.

The function **DefglobalName** returns the name of the defglobal specified by parameter **d**.

The function **DefglobalPPForm** returns the text representation of the defglobal specified by parameter **d**. The null pointer is returned if the text representation is not available.

The function **DefglobalValueForm** returns a string representation of a defglobal and its current value. Parameter **d** is a pointer to a **Defglobal**; and parameter **sb** is a pointer to a **StringBuilder** allocated by the caller in which the representation is stored.

The function **DefglobalGetValue** returns the value of the defglobal specified by parameter **d** in parameter **out**, a **CLIPSValue** allocated by the caller.

The function **DefglobalSet...** functions set the value of the defglobal specified by parameter **d** to the value specified by parameter **value**, a **CLIPSValue** allocated by the caller. This function can trigger garbage collection.

12.8.3 Deletion

```
bool DefglobalIsDeletable(
   Defglobal *d);

bool Undefglobal(
   Defglobal *d,
   Environment *env);
```

The function **DefglobalIsDeletable** returns true if the defglobal specified by parameter **d** can be deleted; otherwise it returns false.

The function **Undefglobal** is the C equivalent of the CLIPS **undefglobal** command. It deletes the defglobal specified by parameter **d**; or if parameter **d** is a null pointer, it deletes all defglobals in the environment specified by parameter **env**. The function returns true if the deletion was successful; otherwise, it returns false.

12.8.4 Watching and Detecting Changes to Defglobals

```
bool DefglobalGetWatch(
   Defglobal *d);

void DeftemplateSetWatch(
   Deftemplate *d,
   bool b);

bool GetGlobalsChanged(
   Environment *env);

void SetGlobalsChanged(
   Environment *env,
   bool b);
```

The function **DefglobalGetWatch** returns true if the defglobal specified by parameter **d** is being watched; otherwise, it returns false.

The function **DefglobalSetWatch** sets the watch state for the defglobal specified by parameter **d**. If parameter **b** is true, the watch state is enabled; otherwise, it is disabled.

The function **GetGlobalsChanged** returns true if changes to global variables for the environment specified by parameter **env** have occurred (either additions, deletions, or value modifications); otherwise, it returns false. To track future changes, **SetGlobalsChanged** should reset the change tracking value to false.

The function **SetGlobalsChanged** sets the global change tracking value for the environment specified by the parameter **env** to the value specified by the parameter **b**.

12.8.5 Reset Globals Behavior

```
bool GetResetGlobals(
   Environment *env);

bool SetResetGlobals(
   Environment *env,
   bool b);
```

The function **GetResetGlobals** is the C equivalent of the CLIPS **get-reset-globals** command. Parameter **env** is a pointer to a previously created environment. This function returns true if the behavior is enabled; otherwise, it returns false.

The function **SetResetGlobals** is the C equivalent of the CLIPS **set-reset-globals** command). Parameter **env** is a pointer to a previously created environment; and parameter **b** is

the new setting for the behavior (either true to enable it or false to disable it). This function returns the old setting for the behavior.

12.8.6 Examples

12.8.6.1 Listing, Watching, and Setting the Value of Defglobals

```
int main()
  {
   Environment *env;
   CLIPSValue value;

   env = CreateEnvironment();

   Build(env,"(defglobal ?*x* = 3)");
   Build(env,"(defglobal ?*y* = (create$ a b c))");

   Write(env,"Listing Globals:\n\n");

   ListDefglobals(env,STDOUT,NULL);

   Write(env,"\nShowing Values:\n\n");

   ShowDefglobals(env,STDOUT,NULL);

   Write(env,"\nSetting Values:\n\n");

   Watch(env,GLOBALS);

   Eval(env,"(* 3 4)",&value);
   DefglobalSetValue(FindDefglobal(env,"x"),&value);

   value.lexemeValue = CreateString(env,"123 Main St.");
   DefglobalSetValue(FindDefglobal(env,"y"),&value);

   DestroyEnvironment(env);
  }
```

When run, the program produces the following output:

```
Listing Globals:

MAIN:
   x
   y
For a total of 2 defglobals.

Showing Values:
```

```
MAIN:
   ?*x* = 3
   ?*y* = (a b c)

Setting Values:

:== ?*x* ==> 12 <== 3
:== ?*y* ==> "123 Main St." <== (a b c)
```

12.9 Deffunction Functions

The following function calls are used for manipulating deffunctions.

12.9.1 Search, Iteration, and Listing

```
Deffunction *FindDeffunction(
   Environment *env,
   const char *name);

Deffunction *GetNextDeffunction(
   Environment *env,
   Deffunction *d);

void GetDeffunctionList(
   Environment *env,
   CLIPSValue *out,
   Defmodule *d);

void ListDeffunctions(
   Environment *env,
   const char *logicalName,
   Defmodule *d);
```

The function **FindDeffunction** searches for the deffunction specified by parameter **name** in the environment specified by parameter **env**. This function returns a pointer to the named deffunction if it exists; otherwise, it returns a null pointer.

The function **GetNextDeffunction** provides iteration support for the list of deffunctions in the current module. If parameter **d** is a null pointer, then a pointer to the first **Deffunction** in the current module is returned by this function; otherwise, a pointer to the next **Deffunction** following the **Deffunction** specified by parameter **d** is returned. If parameter **d** is the last **Deffunction** in the current module, a null pointer is returned.

The function **GetDeffunctionList** is the C equivalent of the CLIPS **get-deffunction-list** function. Parameter **env** is a pointer to a previously created environment; parameter **out** is a

pointer to a **CLIPSValue** allocated by the caller; and parameter **d** is a pointer to a **Defmodule**. The output of the function call—a multifield containing a list of deffunction names—is stored in the **out** parameter value. If parameter **d** is a null pointer, then deffunctions in all modules will be included in parameter **out**; otherwise, only deffunctions in the specified module will be included.

The function **ListDeffunctions** is the C equivalent of the CLIPS **list-deffunctions** command. Parameter **env** is a pointer to a previously created environment; parameter **logicalName** is the router output destination; and parameter **d** is a pointer to a defmodule. If parameter **d** is a null pointer, then deffunctions in all modules will be listed; otherwise, only deffunctions in the specified module will be listed.

12.9.2 Attributes

```
const char *DeffunctionModule(
   Deffunction *d);

const char *DeffunctionName(
   Deffunction *d);

const char *DeffunctionPPForm(
   Deffunction *d);
```

The function **DeffunctionModule** is the C equivalent of the CLIPS **deffunction-module** command. The return value of this function is the name of the module in which the deffunction specified by parameter **d** is defined.

The function **DeffunctionName** returns the name of the deffunction specified by the **d** parameter.

The function **DeffunctionPPForm** returns the text representation of the **Deffunction** specified by the **d** parameter. The null pointer is returned if the text representation is not available.

12.9.3 Deletion

```
bool DeffunctionIdDeletable(
   Deffunction *d);

bool Undeffunction(
   Deffunction *d,
   Environment *env);
```

The function **DeffactsIsDeletable** returns true if the deffacts specified by parameter **d** can be deleted; otherwise it returns false.

The function **Undeffunction** is the C equivalent of the CLIPS **undeffunction** command). It deletes the deffacts specified by parameter **d**; or if parameter **d** is a null pointer, it deletes all deffacts in the environment specified by parameter **env**. This function returns true if the deletion is successful; otherwise, it returns false.

12.9.4 Watching Deffunctions

```
bool DeffunctionGetWatch(
   Deffunction *d);

void DeffunctionSetWatch(
   Deffunction *d,
   bool b);
```

The function **DeffunctionGetWatch** returns true if the watch state is enabled for the deffunction specified by the **d** parameter value; otherwise, it returns false.

The function **DeffunctionSetWatch** sets the watch state for the deffunction specified by the **d** parameter value to the value specified by the parameter **b**.

12.10 Defgeneric Functions

The following function calls are used for manipulating generic functions.

12.10.1 Search, Iteration, and Listing

```
Defgeneric *FindDefgeneric(
   Environment *env,
   const char *name);

Defgeneric *GetNextDefgeneric(
   Environment *env,
   Defgeneric *d);

void GetDefgenericList(
   Environment *env,
   CLIPSValue *out,
   Defmodule *d);
```

```
void ListDefgenerics(
   Environment *env,
   const char *logicalName,
   Defmodule *d);
```

The function **FindDefgeneric** searches for the defgeneric specified by parameter **name** in the environment specified by parameter **env**. This function returns a pointer to the named defgeneric if it exists; otherwise, it returns a null pointer.

The function **GetNextDefgeneric** provides iteration support for the list of defgenerics in the current module. If parameter **d** is a null pointer, then a pointer to the first **Defgeneric** in the current module is returned by this function; otherwise, a pointer to the next **Defgeneric** following the **Defgeneric** specified by parameter **d** is returned. If parameter **d** is the last **Defgeneric** in the current module, a null pointer is returned.

The function **GetDefgenericList** is the C equivalent of the CLIPS **get-defgeneric-list** function. Parameter **env** is a pointer to a previously created environment; parameter **out** is a pointer to a **CLIPSValue** allocated by the caller; and parameter **d** is a pointer to a **Defmodule**. The output of the function call—a multifield containing a list of defgeneric names—is stored in the **out** parameter value. If parameter **d** is a null pointer, then defgenerics in all modules will be included in parameter **out**; otherwise, only defgenerics in the specified module will be included.

The function **ListDefgenerics** is the C equivalent of the CLIPS **list-defgenerics** command). Parameter **env** is a pointer to a previously created environment; parameter **logicalName** is the router output destination; and parameter **d** is a pointer to a defmodule. If parameter **d** is a null pointer, then defgenerics in all modules will be listed; otherwise, only defgenerics in the specified module will be listed.

12.10.2 Attributes

```
const char *DefgenericModule(
   Defgeneric *d);

const char *DefgenericName(
   Defgeneric *d);

const char *DefgenericPPForm(
   Defgeneric *d);
```

The function **DefgenericModule** is the C equivalent of the CLIPS **defgeneric-module** command. The return value of this function is the name of the module in which the defgeneric specified by parameter **d** is defined.

The function **DefgenericName** returns the name of the defgeneric specified by the **d** parameter.

The function **DefgenericPPForm** returns the text representation of the **Defgeneric** specified by the **d** parameter. The null pointer is returned if the text representation is not available.

12.10.3 Deletion

```
bool DefgenericIsDeletable(
   Defgeneric *d);

bool Undefgeneric(
   Defgeneric *d,
   Environment *env);
```

The function **DefgenericIsDeletable** returns true if the defgeneric specified by parameter **d** can be deleted; otherwise it returns false.

The function **Undefgeneric** is the C equivalent of the CLIPS **undefgeneric** command. It deletes the defgeneric specified by parameter **d**; or if parameter **d** is a null pointer, it deletes all defgenerics in the environment specified by parameter **env**. This function returns true if the deletion is successful; otherwise, it returns false.

12.10.4 Watching Defgenerics

```
bool DefgenericGetWatch(
   Defgeneric *d);

void DefgenericSetWatch(
   Defgeneric *d,
   bool b);
```

The function **DefgenericGetWatch** returns true if execution is being watched for the defgeneric specified by the **d** parameter value; otherwise, it returns false.

The function **DefgenericSetWatch** sets the watch state for the defgeneric specified by the **d** parameter value to the value specified by the parameter **b**.

12.11 Defmethod Functions

The following function calls are used for manipulating generic function methods.

12.11.1 Iteration and Listing

```
unsigned GetNextDefmethod(
   Defgeneric *d,
   unsigned index);

void GetDefmethodList(
   Environment *environment,
   CLIPSValue *out,
   Defgeneric *d);

void ListDefmethods(
   Environment *env,
   const char *logicalName,
   Defgeneric *d);
```

The function **GetNextDefmethod** provides iteration support for the list of defmethods for a defgeneric. If parameter **index** is a 0, then a pointer to the first **Defmethod** for the defgeneric specified by parameter **d** is returned by this function; otherwise, a pointer to the next **Defmethod** following the **Defmethod** specified by parameter **index** is returned. If parameter **index** is the last **Defmethod** for the specified defgeneric, 0 is returned.

The function **GetDefmethodList** is the C equivalent of the CLIPS **get-defmethod-list** command. Parameter **env** is a pointer to a previously created environment; parameter **out** is a pointer to a **CLIPSValue** allocated by the caller; and parameter **d** is a pointer to a **Defgeneric**. The output of the function call—a multifield containing a list of defmethod name and index pairs—is stored in the **out** parameter value. If parameter **d** is a null pointer, then defmethods for all defgenerics will be included in parameter **out**; otherwise, only defmethods for the specified defgeneric will be included.

The function **ListDefmethods** is the C equivalent of the CLIPS **list-defmethods** command. Parameter **env** is a pointer to a previously created environment; parameter **logicalName** is the router output destination; and parameter **d** is a pointer to a defgeneric. If parameter **d** is a null pointer, then defmethods for all defgenerics will be listed; otherwise, only defmethods for the specified defgeneric will be listed.

12.11.2 Attributes

```
void DefmethodDescription(
   Defgeneric *d,
   unsigned index,
   StringBuilder *sb);
```

```
const char *DefmethodPPForm(
   Defgeneric *d,
   unsigned index);

void GetMethodRestrictions(
   Defgeneric *d,
   unsigned index,
   CLIPSValue *out);
```

The function **DefmethodDescription** provides a synopsis of a method's parameter restrictions. Parameter **d** is a pointer to a **Defgeneric**; parameter **index** is the method index; and parameter **sb** is a pointer to a **StringBuilder** allocated by the caller in which the method description is stored.

The function **DefmethodPPForm** returns the text representation of the **Defmethod** specified by parameter **d,** the generic function, and parameter **index**, the method index. The null pointer is returned if the text representation is not available.

The function **GetMethodRestrictions** is the C equivalent of the CLIPS **get-method-restrictions** function. Parameter **d** is a pointer to a generic function; parameter **index** is a method index; and parameter **out** is a pointer to a CLIPSValue allocated by the caller. The output of the function call—a multifield containing the method restrictions—is stored in the **out** parameter value.

12.11.3 Deletion

```
bool DefmethodIsDeletable(
   Defgeneric *d,
   unsigned index);

bool Undefmethod(
   Defgeneric *d,
   unsigned index,
   Environment *env);
```

The function **DefmethodIsDeletable** returns true if the defmethod specified by parameter **d**, the generic function, and **index**, the method index, can be deleted; otherwise it returns false.

The function **Undefmethod** is the C equivalent of the CLIPS **undefmethod** command. It deletes the defmethod specified by parameter **d**, parameter **index**, and parameter **env**. If parameter **d** is a null pointer and parameter **index** is 0, it deletes all methods for all generic functions in the environment specified by parameter **env**; if parameter **d** is not a null pointer and parameter **index** is 0, it deletes all methods of the generic function; otherwise, the defmethod

specified by the generic function and method index is deleted. This function returns true if the deletion is successful; otherwise, it returns false.

12.11.4 Watching Methods

```
bool DefmethodGetWatch(
   Defgeneric *d,
   unsigned index);

void DefmethodSetWatch(
   Defgeneric *d,
   unsigned index,
   bool b);
```

The function **DefmethodGetWatch** returns true if the method specified by parameter **d**, the generic function, and parameter **index**, the method index, is being watched; otherwise, it returns false.

The function **DefmethodSetWatch** sets the method watch state for the defmethod specified by the **d** parameter, the generic function, and parameter **index**, the method index, to the value specified by the parameter **b**.

12.12 Defclass Functions

The following function calls are used for manipulating defclasses.

12.12.1 Search, Iteration, and Listing

```
Defclass *FindDefclass(
   Environment *env,
   const char *name);

Defclass *GetNextDefclass(
   Environment *env,
   Defclass *d);

void GetDefclassList(
   Environment *env,
   CLIPSValue *out,
   Defmodule *d);

void ListDefclasses(
   Environment *env,
```

```
   const char *logicalName,
   Defmodule *d);

void BrowseClasses(,
   Defclass *d,
   const char *logicalName);

void DescribeClass(
   Defclass *d,
   const char *logicalName);
```

The function **FindDefclass** searches for the defclass specified by parameter **name** in the environment specified by parameter **env**. This function returns a pointer to the named defclass if it exists; otherwise, it returns a null pointer.

The function **GetNextDefclass** provides iteration support for the list of defclasses in the current module. If parameter **d** is a null pointer, then a pointer to the first **Defclass** in the current module is returned by this function; otherwise, a pointer to the next **Defclass** following the **Defclass** specified by parameter **d** is returned. If parameter **d** is the last **Defclass** in the current module, a null pointer is returned.

The function **GetDefclassList** is the C equivalent of the CLIPS **get-defclass-list** function). Parameter **env** is a pointer to a previously created environment; parameter **out** is a pointer to a **CLIPSValue** allocated by the caller; and parameter **d** is a pointer to a **Defmodule**. The output of the function call—a multifield containing a list of defclass names—is stored in the **out** parameter value. If parameter **d** is a null pointer, then defclasses in all modules will be included in parameter **out**; otherwise, only defclasses in the specified module will be included.

The function **ListDefclass** is the C equivalent of the CLIPS **list-defclass** command. Parameter **env** is a pointer to a previously created environment; parameter **logicalName** is the router output destination; and parameter **d** is a pointer to a defmodule. If parameter **d** is a null pointer, then defclasses in all modules will be listed; otherwise, only defclasses in the specified module will be listed.

The function **BrowseClasses** is the C equivalent of the CLIPS **browse-classes** command. It prints a "graph" of all classes which inherit from the class specified by parameter **d** to the router output destination specified by the **logicalName** parameter.

The function **DescribeClass** is the C equivalent of the CLIPS **describe-class** command. It prints a summary of the class specified by parameter **d** to the router output destination specified by the **logicalName** parameter. This summary includes abstract/concrete behavior, slots and facets (direct and inherited), and recognized message-handlers (direct and inherited).

12.12.2 Class Attributes

```
const char *DefclassModule(
   Defclass *d);

const char *DefclassName(
   Defclass *d);

const char *DefclassPPForm(
   Defclass *d);

void ClassSlots(
   Defclass *d,
   CLIPSValue *out,
   bool inherit);

void ClassSubclasses(
   Defclass *d,
   CLIPSValue *out,
   bool inherit);

void ClassSuperclasses(
   Defclass *d,
   CLIPSValue *out,
   bool inherit);
```

The function **DefclassModule** is the C equivalent of the CLIPS **defclass-module** function. The return value of this function is the name of the module in which the defclass specified by parameter **d** is defined.

The function **DefclassName** returns the name of the defclass specified by the **d** parameter.

The function **DefclassPPForm** returns the text representation of the **Defclass** specified by the **d** parameter. The null pointer is returned if the text representation is not available.

The function **ClassSlots** is the C equivalent of the CLIPS **class-slots** command. Parameter **d** is a pointer to a **Defclass**; parameter **out** is a pointer to a **CLIPSValue** allocated by the caller; and parameter **inherit** is a boolean flag. The output of the function call—a multifield containing the defclass's slot names—is stored in the **out** parameter value. If the **inherit** parameter is true, then inherited slots are included; otherwise, only slot explicitly defined by the class are included.

The function **ClassSubclasses** is the C equivalent of the CLIPS **class-subclasses** command. Parameter **d** is a pointer to a **Defclass**; parameter **out** is a pointer to a **CLIPSValue** allocated by the caller; and parameter **inherit** is a boolean flag. The output of the function call—a multifield containing the defclass's subclass names—is stored in the **out** parameter value. If the

inherit parameter is true, then inherited subclasses are included; otherwise, only direct subclasses explicitly defined by the class are included.

The function **ClassSuperclasses** is the C equivalent of the CLIPS **class-superclasses** command. Parameter **d** is a pointer to a **Defclass**; parameter **out** is a pointer to a **CLIPSValue** allocated by the caller; and parameter **inherit** is a boolean flag. The output of the function call—a multifield containing the defclass's superclass names—is stored in the **out** parameter value. If the **inherit** parameter is true, then inherited superclasses are included; otherwise, only direct superclasses explicitly defined by the class are included.

12.12.3 Deletion

```
bool DefclassIsDeletable(
   Defclass *d);

bool Undefclass(
   Defclass *d,
   Environment *env);
```

The function **DefclassIsDeletable** returns true if the defclass specified by parameter **d** can be deleted; otherwise it returns false.

The **Undefclass** function is the C equivalent of the CLIPS **undefclass** command. It deletes the defclass specified by parameter **d**; or if parameter **d** is a null pointer, it deletes all defclasses in the environment specified by parameter **env**. This function returns true if the deletion is successful; otherwise, it returns false.

12.12.4 Watching Instances and Slots

```
bool DefclassGetWatchInstances(
   Defclass *d);

bool DefclassGetWatchSlots(
   Defclass *d);

void DefclassSetWatchInstances(
   Defclass *d,
   bool b);

void DefclassSetWatchSlots(
   Defclass *d,
   bool b);
```

The function **DefclassGetWatchInstances** returns true if instances (creations and deletions) are being watched for the defclass specified by the **d** parameter value; otherwise, it returns false.

The function **DefclassGetWatchSlots** returns true if slot changes are being watched for the defclass specified by the **d** parameter value; otherwise, it returns false.

The function **DeftemplateSetWatchInstances** sets the instances creation and deletion watch state for the defclass specified by the **d** parameter value to the value specified by the parameter **b**.

The function **DeftemplateSetWatchSlots** sets the slot changes watch state for the defclass specified by the **d** parameter value to the value specified by the parameter **b**.

12.12.5 Class Predicates

```
bool ClassAbstractP(
   Defclass *d);

bool ClassReactiveP(
   Defclass *d);

bool SubclassP(
   Defclass *d1,
   Defclass *d2);

bool SuperclassP(
   Defclass *d1,
   Defclass *d2);
```

The function **ClassAbstractP** is the C equivalent of the CLIPS **class-abstractp** command. It returns true if the defclass specified by parameter **d** is abstract; otherwise, it returns false.

The function **ClassReactiveP** is the C equivalent of the CLIPS **class-reactivep** command. It returns true if the defclass specified by parameter **d** is reactive; otherwise, it returns false.

The function **SubclassP** returns true if the class specified by parameter **d1** is a subclass of the class specified by parameter **d2**; otherwise, it returns false.

The function **SuperclassP** returns true if the class specified by parameter **d1** is a superclass of the class specified by parameter **d2**; otherwise, it returns false.

12.12.6 Slot Attributes

```
bool SlotAllowedClasses(
   Defclass *d,
   const char *name,
   CLIPSValue *out);

bool SlotAllowedValues(
   Defclass *d,
   const char *name,
   CLIPSValue *out);

bool SlotCardinality(
   Defclass *d,
   const char *name,
   CLIPSValue *out);

bool SlotDefaultValue(
   Defclass *d,
   const char *name,
   CLIPSValue *out);

bool SlotFacets(
   Defclass *d,
   const char *name,
   CLIPSValue *out);

bool SlotRange(
   Defclass *d,
   const char *name,
   CLIPSValue *out);

bool SlotSources(
   Defclass *d,
   const char *name
   CLIPSValue *out);

bool SlotTypes(
   Defclass *d,
   const char *name,
   CLIPSValue *out);
```

The function **SlotAllowedClasses** is the C equivalent of the CLIPS **slot-allowed-classes** function. The function **SlotAllowedValues** is the C equivalent of the CLIPS **slot-allowed-values** function. The function **SlotCardinality** is the C equivalent of the CLIPS **slot-**

cardinality function. The function **SlotDefaultValue** is the C equivalent of the CLIPS **slot-default-value** function. The function **SlotFacets** is the C equivalent of the CLIPS **slot-facets** command. The function **SlotRange** is the C equivalent of the CLIPS **slot-range** function. The function **SlotSources** is the C equivalent of the CLIPS **slot-sources** command. The function **SlotTypes** is the C equivalent of the CLIPS **slot-types** function.

Parameter **d** is a pointer to a **Defclass**; parameter **name** specifies a valid slot name for the specified defclass; and parameter **out** is a pointer to a **CLIPSValue** allocated by the caller. The output of the function call—a multifield containing the attribute values—is stored in the **out** parameter value. These function return true if a valid slot name was specified and the output value is successfully set; otherwise, false is returned.

12.12.7 Slot Predicates

```
bool SlotDirectAccessP(
   Defclass *d,
   const char *name);

bool SlotExistP(
   Defclass *d,
   const char *name,
   bool inherit);

bool SlotInitableP(
   Defclass *d,
   const char *name);

bool SlotPublicP(
   Defclass *d,
   const char *name);

bool SlotWritableP(
   Defclass *d,
   const char *name);
```

The function **SlotDirectAccessP** is the C equivalent of the CLIPS **slot-direct-accessp** function. Parameter **d** is a pointer to a **Defclass**; and parameter **name** specifies a slot name. This function returns true if the slot is directly accessible; otherwise, it returns false.

The function **SlotExistP** is the C equivalent of the CLIPS **slot-existp** function. Parameter **d** is a pointer to a **Defclass**; parameter **name** specifies a slot name; and parameter **inherit** is a boolean flag. This function returns true if the specified slot exists; otherwise, it returns false. If the **inherit** parameter value is true, then inherited classes will be searched for the slot; otherwise, only the specified class will be searched.

The function **SlotInitableP** is the C equivalent of the CLIPS **slot-initablep** function. Parameter **d** is a pointer to a **Defclass**; and parameter **name** specifies a slot name. This function returns true if the slot is initable; otherwise, it returns false.

The function **SlotPublicP** is the C equivalent of the CLIPS **slot-publicp** function. Parameter **d** is a pointer to a **Defclass**; and parameter **name** specifies a slot name. This function returns true if the slot is public; otherwise, it returns false.

The function **SlotWritableP** is the C equivalent of the CLIPS **slot-writablep** function. Parameter **d** is a pointer to a **Defclass**; and parameter **name** specifies a slot name. This function returns true if the slot is writable; otherwise, it returns false.

12.12.8 Settings

```
ClassDefaultsMode GetClassDefaultsMode(
   Environment *env);

ClassDefaultsMode SetClassDefaultsMode(
   Environment *env,
   ClassDefaultsMode mode);

typedef enum
  {
   CONVENIENCE_MODE,
   CONSERVATION_MODE
  } ClassDefaultsMode;
```

The function **GetClassDefaultsMode** is the C equivalent of the CLIPS **get-class-defaults-mode** command. The **env** parameter is a pointer to a previously created environment. This function returns the ClassDefaultsMode enumeration corresponding to the current setting.

The function **SetClassDefaultsMode** the C equivalent of the CLIPS command **set-class-defaults-mode**). The **env** parameter is a pointer to a previously created environment; the **mode** parameter is the new setting for the behavior. This function returns the old setting for the behavior.

12.13 Instance Functions

The following function calls are used for manipulating instances.

12.13.1 Search, Iteration, and Listing

```
Instance *FindInstance(
   Environment *env,
   Defmodule *d,
   const char *name,
   bool searchImports);

Instance *GetNextInstance(
   Environment *env,
   Instance *i);

Instance *GetNextInstanceInClass(
   Defclass *d,
   Instance *i);

Instance *GetNextInstanceInClassAndSubclasses(
   Defclass **d,
   Instance *i,
   UDFValue *iterator);

void Instances(
   Environment *env,
   const char *logicalName,
   Defmodule *d,
   const char *className,
   bool listSubclasses);
```

The function **FindInstance** searches for the named instance specified by parameter **name** in the module specified by parameter **d** in the environment specified by parameter **env**. If parameter **d** is a null pointer, then the current module will be searched. If parameter **searchImports** is true, then imported modules will also be searched for the instance. If the named instance is found, a pointer to it is returned; otherwise, the null pointer is returned.

The function **GetNextInstance** provides iteration support for the list of instance in an environment. If parameter **i** is a null pointer, then a pointer to the first **Instance** in the environment specified by parameter **env** is returned by this function; otherwise, a pointer to the next **Instance** following the **Instance** specified by parameter **i** is returned. If parameter **i** is the last **Instance** in the specified environment, a null pointer is returned.

The function **GetNextInstanceInClass** provides iteration support for the list of instances belonging to a defclass. If parameter **i** is a null pointer, then a pointer to the first **Instance** for the defclass specified by parameter **d** is returned by this function; otherwise, a pointer to the next

Instance of the specified defclass following the **Instance** specified by parameter **i** is returned. If parameter **i** is the last **Instance** for the specified defclass, a null pointer is returned.

The function **GetNextInstanceInClassAndSubclasses** provides iteration support for the list of instances belonging to a defclass and its subclasses. If parameter **i** is a null pointer, then a pointer to the first **Instance** for the defclass or subclasses specified by parameter **d** is returned by this function; otherwise, a pointer to the next **Instance** of the specified defclass or its subclasses following the **Instance** specified by parameter **i** is returned. If parameter **i** is the last **Instance** for the specified defclass or its subclasses, a null pointer is returned. Parameter **d** is a pointer to a pointer to a **Defclass** declared by the caller. As the subclasses of the specified class are iterated through to find instances, the value referenced by the **d** parameter value is updated to indicate the class of the instance returned by this function. Parameter **iterator** is a pointer to a **UDFValue** declared by the caller that is used to store instance iteration information; no initialization of this argument is required and the values stored in this argument are not intended for examination by the calling function.

The function **Instances** is the C equivalent of the CLIPS **instances** command. Parameter **env** is a pointer to a previously created environment; parameter **logicalName** is the router output destination; parameter **d** is a pointer to a **Defmodule**; parameter **name** is the name of a defclass; and parameter **listSubclasses** is a boolean value indicating whether instances of subclasses should be listed. If parameter **d** is a null pointer, then all instances of all classes in all modules are listed (and the parameters values for **name** and **listSubclasses** are ignored). If parameter **d** is not a null pointer and parameter **name** is a null pointer, all instance of all classes in the specified module are listed (and parameter **listSubclasses** is ignored).

12.13.2 Attributes

```
Defclass *InstanceClass(
   Instance *i);

const char *InstanceName(
   Instance *i);

void InstancePPForm(
   Instance *i,
   StringBuilder *sb);

GetSlotError DirectGetSlot(
   Instance *i,
   const char *name,
   CLIPSValue *out);

PutSlotError DirectPutSlot(
  Instance *i,
```

```
  const char *name,
  CLIPSValue *value);

PutSlotError DirectPutSlotInteger(
   Instance *i,
   const char *name,
   long long value);

PutSlotError DirectPutSlotFloat(
   Instance *i,
   const char *name,
   double value);

PutSlotError DirectPutSlotSymbol(
   Instance *i,
   const char *name,
   const char *value);

PutSlotError DirectPutSlotString(
   Instance *i,
   const char *name,
   const char *value);

PutSlotError DirectPutSlotInstanceName(
   Instance *i,
   const char *name,
   const char *value);

PutSlotError DirectPutSlotCLIPSInteger(
   Instance *i,
   const char *name,
   CLIPSInteger *value);

PutSlotError DirectPutSlotCLIPSFloat(
   Instance *i,
   const char *name,
   CLIPSFloat *value);

PutSlotError DirectPutSlotCLIPSLexeme(
   Instance *i,
   const char *name,
   CLIPSLexeme *value);

PutSlotError DirectPutSlotFact(
   Instance *i,
```

```
   const char *name,
   Fact *value);

PutSlotError DirectPutSlotInstance(
   Instance *i,
   const char *name,
   Instance *value);

PutSlotError DirectPutSlotMultifield(
   Instance *i,
   const char *name,
   Multifield *value);

PutSlotError DirectPutSlotCLIPSExternalAddress(
   Instance *i,
   const char *name,
   CLIPSExternalAddress *value);

typedef enum
  {
   GSE_NO_ERROR,
   GSE_NULL_POINTER_ERROR,
   GSE_INVALID_TARGET_ERROR,
   GSE_SLOT_NOT_FOUND_ERROR,
  } GetSlotError;

typedef enum
  {
   PSE_NO_ERROR,
   PSE_NULL_POINTER_ERROR,
   PSE_INVALID_TARGET_ERROR,
   PSE_SLOT_NOT_FOUND_ERROR,
   PSE_TYPE_ERROR,
   PSE_RANGE_ERROR,
   PSE_ALLOWED_VALUES_ERROR,
   PSE_CARDINALITY_ERROR,
   PSE_ALLOWED_CLASSES_ERROR,
   PSE_RULE_NETWORK_ERROR
  } PutSlotError;
```

The function **InstanceClass** returns a pointer to the **Defclass** associated with the **Instance** specified by parameter **i**.

The function **InstanceName** returns the instance name of the **Instance** specified by parameter **i**.

The function **InstancePPForm** stores the text representation of the **Instance** specified by the parameter **i** in the **StringBuilder** specified by parameter **sb**.

The function **DirectGetSlot** is the C equivalent of the CLIPS **dynamic-get** function. It retrieves the slot value specified by parameter **name** from the instance specified by parameter **i** and stores it in parameter **out**, a **CLIPSValue** allocated by the caller. This function bypasses message-passing.

The **DirectGetSlot** function returns GSE_NO_ERROR if the slot value is successfully retrieved; otherwise it returns GSE_NULL_POINTER_ERROR if any of the function arguments are NULL pointers, GSE_INVALID_TARGET_ERROR if the instance specified by parameter **i** has been deleted, and GSE_SLOT_NOT_FOUND_ERROR if the instance does not have the specified slot.

The function **DirectPutSlot** function is the C equivalent of the CLIPS **dynamic-put** function. It sets the slot value specified by parameter **name** of the instance specified by parameter **i** to the value stored in parameter **value**, a **CLIPSValue** allocated and set by the caller. This function bypasses message-passing. The additional **DirectPutSlot...** functions are wrappers for the **DirectPutSlot** function which allow you to assign other CLIPS and C data types to a slot without the need to allocate a **CLIPSValue** structure.

The **DirectPutSlot** function returns PSE_NO_ERROR if the slot value is successfully set; otherwise it returns PSE_NULL_POINTER_ERROR if any of the function arguments are NULL pointers, PSE_INVALID_TARGET_ERROR if the instance specified by parameter **i** has been deleted, PSE_SLOT_NOT_FOUND_ERROR if the instance does not have the specified slot, and PSE_RULE_NETWORK_ERROR if an error occurred while the slot assignment was being processed in the rule network. When dynamic constraint checking is enabled, the remaining enumeration values indicate that the specified slot value violates one of the constraints for the allowed types or values for the slot.

12.13.3 Deletion

```
UnmakeInstanceError UnmakeAllInstances(
   Environment *env);

UnmakeInstanceError DeleteInstance(
   Instance *i);

UnmakeInstanceError DeleteAllInstances(
   Environment *env);
```

```
bool ValidInstanceAddress(
   Instance *i);
```

The function **UnmakeAllInstances** deletes all instances in the environment specified by parameter **env** using message-passing. It returns UE_NO_ERROR if all instances were successfully deleted. See the **Unmake** command in section 3.3.4 for the list of error codes in the **UnmakeInstanceError** enumeration.

The function **DeleteInstance** directly deletes the instance specified by parameter **i** bypassing message-passing. The function **DeleteAllInstances** directly deletes all instances in the environment specified by parameter **env** bypassing message-passing. Both functions returns UE_NO_ERROR if deletion is successful. See the **Unmake** command in section 3.3.4 for the list of error codes in the **UnmakeInstanceError** enumeration.

The function **ValidInstanceAddress** determines if an instance referenced by an address still exists. It returns true if the instance still exists; otherwise, it returns false. The parameter **i** must be a **Instance** that has not been deleted or has been retained (see section 5.2).

12.13.4 Loading and Saving Instances

```
long BinaryLoadInstances(
   Environment *env,
   const char *fileName);

long LoadInstances(
   Environment *env,
   const char *fileName);

long LoadInstancesFromString(
   Environment *env,
   const char *str,
   size_t length);

long RestoreInstances(
   Environment *env,
   const char *fileName);

long RestoreInstancesFromString(
   Environment *env,
   const char *str,
   size_t length);

long BinarySaveInstances(
   Environment *env,
```

```
   const char *fileName,
   SaveScope saveCode);

long SaveInstances(
   Environment *env,
   const char *fileName,
   SaveScope saveCode);

typedef enum
  {
   LOCAL_SAVE,
   VISIBLE_SAVE
  } SaveScope;
```

The function **BinaryLoadInstances** is the C equivalent of the CLIPS **bload-instances** command. Parameter **env** is a pointer to a previously created environment; and parameter **fileName** is a full or partial path string to binary instances save file created using **BinarySaveInstances**. This function returns the number of instances loaded, or -1 if an error occurs.

The function **LoadInstances** is the C equivalent of the CLIPS **load-instances** command. Parameter **env** is a pointer to a previously created environment; and parameter **fileName** is a full or partial path string to an ASCII or UTF-8 file containing instances. This function returns the number of instances loaded, or -1 if an error occurs.

The function **LoadInstancesFromString** loads a set of instances from a string input source (much like the **LoadInstances** function only using a string for input rather than a file). Parameter **env** is a pointer to a previously created environment; parameter **str** is a string containing instances; and parameter **length** is the maximum number of characters to be read from the input string. If the **length** parameter value is SIZE_MAX, then the **str** parameter value must be terminated by a null character; otherwise, the **length** parameter value indicates the maximum number characters that will be read from the **str** parameter value. This function returns the number of instances restored, or -1 if an error occurs.

The function **RestoreInstances** is the C equivalent of the CLIPS **restore-instances** command. Parameter **env** is a pointer to a previously created environment; and parameter **fileName** is a full or partial path string to an ASCII or UTF-8 file containing instances. This function returns the number of instances restored, or -1 if an error occurs.

The function **RestoreInstancesFromString** loads a set of instances from a string input source (much like the **RestoreInstances** function only using a string for input rather than a file). Parameter **env** is a pointer to a previously created environment; parameter **str** is a string containing instances; and parameter **length** is the maximum number of characters to be read from the input string. If the **length** parameter value is SIZE_MAX, then the **str** parameter value

must be terminated by a null character; otherwise, the **length** parameter value indicates the maximum number characters that will be read from the **str** parameter value. This function returns the number of instances restored, or -1 if an error occurs.

The function **BinarySaveInstances** is the C equivalent of the CLIPS **bsave-instances** command. Parameter **env** is a pointer to a previously created environment; parameter **fileName** is a full or partial path string to the binary instances save file that will be created; and parameter **scope** indicates whether all instances visible to the current module should be saved (VISIBLE_SAVE) or just those associated with defclasses defined in the current module (LOCAL_SAVE). This function returns the number of instances saved.

The function **SaveInstances** is the C equivalent of the CLIPS **save-instances** command. Parameter **env** is a pointer to a previously created environment; parameter **fileName** is a full or partial path string to the instance save file that will be created; and parameter **scope** indicates whether all instances visible to the current module should be saved (VISIBLE_SAVE) or just those associated with defclasses defined in the current module (LOCAL_SAVE).

12.13.5 Detecting Changes to Instances

```
bool GetInstancesChanged(
   Environment *env);

void SetInstancesChanged(
   Environment *env,
   bool b);
```

The function **GetInstancesChanged** returns true if changes to instances for the environment specified by parameter **env** have occurred (either creations, deletions, or slot value changes); otherwise, it returns false. To track future changes, **SetInstanceChanged** should reset the change tracking value to false.

The function **SetInstancesChanged** sets the instances change tracking value for the environment specified by the parameter **env** to the value specified by the parameter **b**.

12.13.6 Send

```
void Send(
   Environment *env,
   CLIPSValue *in,
   const char *msg,
   const char *msgArgs
   CLIPSValue *out);
```

The function **Send** is the C equivalent of the CLIPS **send** function. Parameter **env** is a pointer to a previously created environment; parameter **in** is a CLIPSValue allocated and assigned the value of the object (instance, instance name, symbol, etc.) to receive the message; parameter **msg** is the message to be received; parameter **msgArgs** is a string containing the constants arguments to the message separated by spaces (a null pointer indicates no arguments); and parameter **out** is a CLIPSValue allocated by the caller that is assigned the return value of the message call. If the calling function does not need to examine the return value of the message, a null pointer can be specified for the **out** parameter value. This function can trigger garbage collection.

12.13.7 Examples

12.13.7.1 Instance Iteration

```
#include "clips.h"

int main()
  {
   Environment *theEnv;
   UDFValue iterate;
   Instance *theInstance;
   Defclass *theClass;

   theEnv = CreateEnvironment();

   Build(theEnv,"(defclass A (is-a USER))");
   Build(theEnv,"(defclass B (is-a USER))");

   MakeInstance(theEnv,"(a1 of A)");
   MakeInstance(theEnv,"(a2 of A)");
   MakeInstance(theEnv,"(b1 of B)");
   MakeInstance(theEnv,"(b2 of B)");

   theClass = FindDefclass(theEnv,"USER");

   for (theInstance = GetNextInstanceInClassAndSubclasses(&theClass,NULL,&iterate);
        theInstance != NULL;
        theInstance = GetNextInstanceInClassAndSubclasses(&theClass,
                                                          theInstance,&iterate))
     { Writeln(theEnv,InstanceName(theInstance)); }

   DestroyEnvironment(theEnv);
  }
```

The output when running this example is:

```
a1
a2
```

```
b1
b2
```

12.13.7.2 Send

```
#include "clips.h"

int main()
  {
   Environment *theEnv;
   char *cs;
   CLIPSValue insdata;

   theEnv = CreateEnvironment();

   Build(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))";
   Build(theEnv,cs);

   insdata.instanceValue = MakeInstance(theEnv,"(my-instance of MY-CLASS)");
   Send(theEnv,&insdata,"my-msg","1 abc 3",NULL);

   DestroyEnvironment(theEnv);
  }
```

The output when running this example is:

```
1 abc 3
```

12.14 Defmessage-handler Functions

The following function calls are used for manipulating defmessage-handlers.

12.14.1 Search, Iteration, and Listing

```
unsigned FindDefmessageHandler(
   Defclass *d,
   const char *name,
   const char *type);
```

```
unsigned GetNextDefmessageHandler(
   Defclass *d,
   unsigned id);

void GetDefmessageHandlerList(
   Environment *env,
   Defclass *d,
   CLIPSValue *out,
   bool inherited);

void ListDefmessageHandlers(
   Environment *env,
   Defclass *d,
   const char *logicalName,
   bool inherited);
```

The function **FindDefmessageHandler** searches for the defmessage-handler specified by parameters **name** and **type** in the defclass specified by parameter **d**. Parameter **type** should be one of the following values: "around", "before", "primary", or "around". This function returns an integer id for the defmessage-handler if it exists; otherwise, it returns 0.

The function **GetNextDefmessageHandler** provides iteration support for the list of defmessage-handlers for a defclass. If parameter **id** is a 0, then the integer id to the first defmessage-handler for the defclass specified by parameter **d** is returned by this function; otherwise, the integer id to the next defmessage-handler following the defmessage-handler specified by parameter **id** is returned. If parameter **id** is the last defmessage-handler for the specified defclass, 0 is returned.

The function **GetDefmessageHandlerList** is the C equivalent of the CLIPS **get-defmessage-handler-list** command. Parameter **env** is a pointer to a previously created environment; parameter **d** is a pointer to a **Defclass**; parameter **out** is a pointer to a **CLIPSValue** allocated by the caller; and parameter **inherited** is a boolean value that indicates whether inherited message-handlers are included. The output of the function call—a multifield containing a list of class name/handler name/handler type triplets—is stored in the **out** parameter value. If parameter **d** is a null pointer, then message-handlers for all defclasses will be included in parameter **out**; otherwise, only message-handlers for the specified defclass will be included.

The function **ListDefmessageHandlers** is the C equivalent of the CLIPS **list-defmessage-handlers** command. Parameter **env** is a pointer to a previously created environment; parameter **d** is a pointer to a defclass; parameter **logicalName** is the router output destination; and parameter **inherited** is a boolean value that indicates whether inherited message-handlers are listed. If parameter **d** is a null pointer, then defmessage-handlers for all defclasses will be listed; otherwise, only defmessage-handlers for the specified defclass will be listed.

12.14.2 Attributes

```
const char *DefmessageHandlerName(
   Defclass *defclassPtr,
   unsigned id);

const char *DefmessageHandlerPPForm(
   Defclass *d,
   unsigned id);

const char *GetDefmessageHandlerType(
   Defclass *d,
   unsigned id);
```

The function **DefmessageHandlerName** returns the name of the defmessage-handler specified by parameters **d**, a **Defclass**, and **id**, the message-handler id.

The function **DefmessageHandlerPPForm** returns the text representation of the defmessage-handler specified by parameter **d,** a defclass, and parameter **id**, a message-handler id. The null pointer is returned if the text representation is not available.

The function **DefmessageHandlerType** returns the type of the defmessage-handler specified by parameter **d,** a defclass, and parameter **id**, a message-handler id. The return value of this function is one of the following values: "around", "before", "primary", or "around".

12.14.3 Deletion

```
bool DefmessageHandlerIsDeletable(
   Defclass *d,
   unsigned id);

bool UndefmessageHandler(
   Defclass *d,
   unsigned id,
   Environment *env);
```

The function **DefmessageHandleIsDeletable** returns true if the defmessage-handler specified by parameter **d**, a defclass, and **id**, a message-handler id, can be deleted; otherwise it returns false.

The function **UndefmessageHandler** is the C equivalent of the CLIPS **undefmessage-handler** command. It deletes the defmessage-handler specified by parameter **d**, parameter **d**, and parameter **env**. If parameter **d** is a null pointer and parameter **id** is 0, it deletes all message-handlers for all classes in the environment specified by parameter **env**; if parameter **d** is not a

null pointer and parameter **id** is 0, it deletes all message-handlers of the defclass; otherwise, the defmessage-handler specified by the defclass and message-handler id is deleted. This function returns true if the deletion is successful; otherwise, it returns false.

12.14.4 Watching Message-Handlers

```
bool DefmessageHandlerGetWatch(
   Defclass *d,
   unsigned id);

void DefmessageHandlerSetWatch(
   Defclass *d,
   unsigned id,
   bool b);
```

The function **DefmessageHandlerGetWatch** returns true if the message-handler specified by parameter **d**, the defclass, and parameter **id**, the message-handler id, is being watched; otherwise, it returns false.

The function **DefmessageHandlerSetWatch** sets the message-handler watch state for the defmessage-handler specified by parameter **d**, the defclass, and parameter **id**, the message-handler id, to the value specified by the parameter **b**.

12.14.5 PreviewSend

```
void PreviewSend(
   Defclass *d,
   const char *logicalName,
   const char *message);
```

The function **PreviewSend** is the C equivalent of the CLIPS **preview-send** command. Parameter **d** is a pointer to a defclass; parameter **logicalName** is the router output destination; and parameter **message** is the name of the message-handler to be previewed.

12.14.6 Example

This example demonstrates how to preview a send message and watch specific message-handlers when a message is executed. The following output shows how you would typically perform this task from the CLIPS command prompt:

```
CLIPS> (defclass A (is-a USER))
CLIPS> (defmessage-handler A foo (?x))
CLIPS> (defmessage-handler A foo before (?x))
```

```
CLIPS> (defmessage-handler A foo after (?x))
CLIPS> (defclass B (is-a A))
CLIPS> (defmessage-handler B foo (?x) (call-next-handler))
CLIPS> (defmessage-handler B foo around (?x) (call-next-handler))
CLIPS> (preview-send B foo)
>> foo around in class B
| >> foo before in class A
| << foo before in class A
| >> foo primary in class B
| | >> foo primary in class A
| | << foo primary in class A
| << foo primary in class B
| >> foo after in class A
| << foo after in class A
<< foo around in class B
CLIPS> (watch message-handlers A foo primary)
CLIPS> (watch message-handlers B foo around)
CLIPS> (make-instance [b1] of B)
[b1]
CLIPS> (send [b1] foo 3)
HND >> foo around in class B
       ED:1 (<Instance-b1> 3)
HND >> foo primary in class A
       ED:1 (<Instance-b1> 3)
HND << foo primary in class A
       ED:1 (<Instance-b1> 3)
HND << foo around in class B
       ED:1 (<Instance-b1> 3)
FALSE
CLIPS>
```

To achieve the same result from a C program, change the contents of the main.c source file to the following:

```
#include "clips.h"

int main()
  {
   Environment *env;
   Defclass *classA, *classB;

   env = CreateEnvironment();

   Build(env,"(defclass A (is-a USER))");
   Build(env,"(defmessage-handler A foo (?x))");
   Build(env,"(defmessage-handler A foo before (?x))");
   Build(env,"(defmessage-handler A foo after (?x))");
   Build(env,"(defclass B (is-a A))");
   Build(env,"(defmessage-handler B foo (?x) (call-next-handler))");
   Build(env,"(defmessage-handler B foo around (?x) (call-next-handler))");

   classA = FindDefclass(env,"A");
```

```
    classB = FindDefclass(env,"B");

    Write(env,"Preview Send:\n\n");

    PreviewSend(classB,STDOUT,"foo");

    Write(env,"\nWatch Handlers and Send Message:\n\n");

    DefmessageHandlerSetWatch(classA,
                              FindDefmessageHandler(classA,"foo","primary"),
                              true);

    DefmessageHandlerSetWatch(classB,
                              FindDefmessageHandler(classB,"foo","around"),
                              true);

    MakeInstance(env,"([b1] of B)");

    Eval(env,"(send [b1] foo 3)",NULL);

    DestroyEnvironment(env);
   }
```

The following output will be produced when the program is run:

```
Preview Send:

>> foo around in class B
| >> foo before in class A
| << foo before in class A
| >> foo primary in class B
| | >> foo primary in class A
| | << foo primary in class A
| << foo primary in class B
| >> foo after in class A
| << foo after in class A
<< foo around in class B

Watch Handlers and Send Message:

HND >> foo around in class B
       ED:1 (<Instance-b1> 3)
HND >> foo primary in class A
       ED:1 (<Instance-b1> 3)
HND << foo primary in class A
       ED:1 (<Instance-b1> 3)
HND << foo around in class B
       ED:1 (<Instance-b1> 3)
```

12.15 Definstances Functions

The following function calls are used for manipulating definstances.

12.15.1 Search, Iteration, and Listing

```
Definstances *FindDefinstances(
   Environment *env,
   const char *name);

Definstances *GetNextDefinstances(
   Environment *env,
   Definstances *d);

void GetDefinstancesList(
   Environment *env,
   CLIPSValue *out,
   Defmodule *d);

void ListDefinstances(
   Environment *env,
   char *logicalName,
   Defmodule *d);
```

The function **FindDefinstances** searches for the definstances specified by parameter **name** in the environment specified by parameter **env**. This function returns a pointer to the named definstances if it exists; otherwise, it returns a null pointer.

The function **GetNextDefinstances** provides iteration support for the list of definstances in the current module. If parameter **d** is a null pointer, then a pointer to the first **Definstances** in the current module is returned by this function; otherwise, a pointer to the next **Definstances** following the **Definstancess** specified by parameter **d** is returned. If parameter **d** is the last **Deffacts** in the current module, a null pointer is returned.

The function **GetDefinstancesList** the C equivalent of the CLIPS **get-definstances-list** function. Parameter **env** is a pointer to a previously created environment; parameter **out** is a pointer to a **CLIPSValue** allocated by the caller; and parameter **d** is a pointer to a **Defmodule**. The output of the function call—a multifield containing a list of definstances names—is stored in the **out** parameter value. If parameter **d** is a null pointer, then definstancess in all modules will be included in parameter **out**; otherwise, only definstancess in the specified module will be included.

The function **ListDefinstances** is the C equivalent of the CLIPS **list-definstances** command. Parameter **env** is a pointer to a previously created environment; parameter

logicalName is the router output destination; and parameter **d** is a pointer to a defmodule. If parameter **d** is a null pointer, then definstances in all modules will be listed; otherwise, only definstances in the specified module will be listed.

12.15.2 Attributes

```
const char *DefinstancesModule(
   Definstances *d);

const char *DefinstancesName(
   Definstances *d);

const char *DefinstancesPPForm(
   Definstances *d);
```

The function **DefinstancesModule** is the C equivalent of the CLIPS **definstances-module** command.

The function **DefinstancesName** returns the name of the deffacts specified by the **d** parameter.

The function **DefinstancePPForm** returns the text representation of the **Definstancess** specified by the **d** parameter. The null pointer is returned if the text representation is not available.

12.15.3 Deletion

```
bool DefinstancesIsDeletable(
   Definstances *d);

bool Undefinstances(
   Definstances *d,
   Environment *env);
```

The function **DefinstancesIsDeletable** returns true if the definstances specified by parameter **d** can be deleted; otherwise it returns false.

The function **Undefinstances** is the C equivalent of the CLIPS **undefinstances** command. It deletes the definstances specified by parameter **d**; or if parameter **d** is a null pointer, it deletes all definstances in the environment specified by parameter **env**. This function returns true if the deletion is successful; otherwise, it returns false.

12.16 Defmodule Functions

The following function calls are used for manipulating defmodules.

12.16.1 Search, Iteration, and Listing

```
Defmodule *FindDefmodule(
   Environment *env,
   const char *name);

Defmodule *GetNextDefmodule(
   Environment *env,
   Defmodule *d);

void GetDefmoduleList(
   Environment *env,
   CLIPSValue *out);

void ListDefmodules(
   Environment *env,
   const char *logicalName);
```

The function **FindDefmodule** searches for the defmodule specified by parameter **name** in the environment specified by parameter **env**. This function returns a pointer to the named defmodule if it exists; otherwise, it returns a null pointer.

The function **GetNextDefmodule** provides iteration support for the list of defmodules. If parameter **d** is a null pointer, then a pointer to the first **Defmodule** in the environment is returned by this function; otherwise, a pointer to the next **Defmodule** following the **Defmodule** specified by parameter **d** is returned. If parameter **d** is the last **Defmodule** in the environment, a null pointer is returned.

The function **GetDefmoduleList** is the C equivalent of the CLIPS **get-defmodule-list** function. Parameter **env** is a pointer to a previously created environment; and parameter **out** is a pointer to a **CLIPSValue** allocated by the caller. The output of the function call—a multifield containing a list of defmodule names—is stored in the **out** parameter value.

The function **ListDefmodules** is the C equivalent of the CLIPS **list-defmodules** command. Parameter **env** is a pointer to a previously created environment; and parameter **logicalName** is the router output destination.

12.16.2 Attributes

```
const char *DefmoduleName(
   Defmodule *d);

const char *DefmodulePPForm(
   Defmodule *d);
```

The function **DefmoduleName** returns the name of the defmodule specified by the **d** parameter.

The function **DefmodulePPForm** returns the text representation of the **Defmodule** specified by the **d** parameter. The null pointer is returned if the text representation is not available.

12.16.3 Current Module

```
Defmodule *GetCurrentModule(
   Environment *env);

Defmodule *SetCurrentModule(
   Environment *env,
   Defmodule *d);
```

The function **GetCurrentModule** is the C equivalent of the CLIPS **get-current-module** function. Parameter **env** is a pointer to a previously created environment. This function returns a pointer to the current environment.

The function **SetCurrentModule** is the C equivalent of the CLIPS **set-current-module** function. Parameter **env** is a pointer to a previously created environment; and parameter **d** is a pointer to a **Defmodule** that will become the current module. The return value of this function is the prior current module.

12.17 Standard Memory Functions

CLIPS provides functions that can be used to monitor and control memory usage.

12.17.1 Memory Allocation and Deallocation

```
void *genalloc(
   Environment *env,
   size_t size);
```

```
void genfree(
   Environment *env,
   void *ptr,
   size_t size);
```

The function **genalloc** allocates a block of memory using the CLIPS memory management routines. CLIPS caches memory to improve to performance. Calls to genalloc will first attempt to allocate memory from the cache before making a call to the C library **malloc** function. Parameter **env** is a pointer to a previously allocated environment; and parameter **size** is the number of bytes to be allocated. This function returns a pointer to a memory block of the specified size if successful; otherwise, a null pointer is returned.

The function **genfree** returns a block of memory to the CLIPS memory management routines. Calls to **genfree** adds the freed memory to the CLIPS cache for later reuse. The CLIPS **ReleaseMem** function can be used to clear the cache and release memory back to the operating system. Parameter **env** is a pointer to a previously allocated environment; parameter **ptr** is a pointer to memory previously allocated by **genalloc**; and parameter **size** is the size in bytes of the block of memory.

12.17.2 Settings

```
bool GetConserveMemory(
   Environment *env);

bool SetConserveMemory(
   Environment *env,
   bool b;)
```

The function **GetConserveMemory** returns the current value of the conserve memory behavior. If enabled (true), newly loaded constructs do not have their text (printy print) representation stored with the construct. If there is no need to save or pretty print constructs, this will reduce the amount of memory needed to load constructs.

The function **SetConserveMemory** sets the conserve memory behavior for the environment specified by the parameter **env** to the value specified by the parameter **b**. Changing the value for this behavior does not affect existing constructs.

12.17.3 Memory Tracking

```
long long MemRequests(
   Environment *env);
```

```
long long MemUsed(
   Environment *env);

long long ReleaseMem(
   Environment *env,
   long long limit);
```

The function **MemRequests** is the C equivalent of the CLIPS **mem-requests** command. The return value of this function is the number of memory requests currently held by CLIPS. When used in conjunction with **MemUsed**, the user can estimate the number of bytes CLIPS requests per call to **malloc**.

The function **MemUsed** is the C equivalent of the CLIPS **mem-used** command. The return value of this function is the total number of bytes requested and currently held by CLIPS. 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.

The function **ReleaseMem** is the C equivalent of the CLIPS **release-mem** command. It allows free memory being cached by CLIPS to be returned to the operating system. Parameter **env** is a previously allocated environment; and parameter **limit** is the amount of memory to be released. If the **limit** parameter value is 0 or less, then all cached memory will be released; otherwise, cached memory will be released until the amount of released memory exceeds the **limit** parameter value. The return value of this function is the amount of cached memory released.

12.18 Embedded Application Examples

12.18.1 User-Defined Functions

12.18.2 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 "clips.h"

int main()
  {
   Environment *env;
   Instance *c1, *c2, *c3;
   CLIPSValue insdata, result;

   env = CreateEnvironment();

   /*=========================================*/
   /* Load the code for handling complex numbers. */
   /*=========================================*/

   Load(env,"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 = MakeInstance(env,"(c1 of COMPLEX (real 1) (imag 10))");
   c2 = CreateRawInstance(env,FindDefclass(env,"COMPLEX"),"c2");

   result.integerValue = CreateInteger(env,3);
   DirectPutSlot(c2,"real",&result);

   result.integerValue = CreateInteger(env,-7);
   DirectPutSlot(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.            */
   /*=========================================================*/

   Eval(env,"(+ [c1] [c2])",&result);
   c3 = FindInstance(env,NULL,result.lexemeValue->contents,true);

    /*=======================================================*/
   /* Print out a summary of the complex addition using the */
   /* "print" and "magnitude" messages to get information   */
   /* about the three complex numbers.                      */
   /*=======================================================*/

   Write(env,"The addition of\n\n");

   insdata.instanceValue = c1;
   Send(env,&insdata,"print",NULL,NULL);

   Write(env,"\nand\n\n");
```

```
   insdata.instanceValue = c2;
   Send(env,&insdata,"print",NULL,NULL);

   Write(env,"\nis\n\n");

   insdata.instanceValue = c3;
   Send(env,&insdata,"print",NULL,NULL);

   Write(env,"\nand the resulting magnitude is ");

   Send(env,&insdata,"magnitude",NULL,&result);
   WriteCLIPSValue(env,STDOUT,&result);
   Write(env,"\n");

   return 0;
  }
```

3) Define constructs which use the new function in a file called **complex.clp** (or any file; just be sure the call to **Load** loads all necessary constructs prior to execution).

```
(defclass COMPLEX
   (is-a USER)
   (slot real)
   (slot imag))

(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. The output is:

```
The addition of

[c1] of COMPLEX
(real 1)
(imag 10)

and

[c2] of COMPLEX
(real 3)
(imag -7)
```

```
is

[gen1] of COMPLEX
(real 4)
(imag 3)

and the resulting magnitude is 5.000000
```

Appendix A:
Support Information

A.1 Questions and Information

The URL for the CLIPS Web page is http://www.clipsrules.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 support@clipsrules.net.

A.2 Documentation

The CLIPS Reference Manuals and other documentation is available at http://www.clipsrules.net/?q=Documentation.

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 changes were introduced in version 6.4 of CLIPS.

- **Environment API** – The environment API is the only supported API. The Env prefix has been removed from all API calls.

- **Void Pointers** – The use of generic (or universal) pointers in API calls when an appropriate typed pointer exists has been discontinued.

- **Bool Support** – The APIs now utilize the bool type for representing boolean values.

- **Primitive Value Redesign** – The implementation of primitive values has been redesigned. The function **AddSymbol** has been replaced with the functions **CreateSymbol**, **CreateString**, **CreateInstanceName**, and **CreateBoolean**. The function **AddLong** has been replaced with the function **CreateInteger**. The function **AddDouble** has been replaced with the function **CreateFloat**. The function **CreateMultifield** has been replaced with the **MultifieldBuilder** API and the functions **EmptyMultifield** and **StringToMultifield**. See section 4.1 and section 6 for more information.

- **User Defined Function API Redesign** – The User Defined Function API has been redesigned. The DefineFunction function has been renamed to **AddUDF** and its parameters have changed. The function **RtnArgCount** has been renamed to **UDFArgumentCount** and its parameters have changed. The functions **ArgCountCheck** and **ArgRangeCheck** are no longer supported since the UDF argument count is automatically checked before a UDF is invoked. The function **ArgTypeCheck** has been replaced with the functions **UDFFirstArgument**, **UDFNextArgument**, **UDFNthArgument**, and **UDFHasNextArgument**. See section 8 for more information.

- **I/O Router API Redesign** – The I/O router API has been redesigned. The parameters for the **AddRouter** function have changed. The **GetcRouter**, **UngetcRouter**, and **PrintRouter** functions have been renamed to **ReadRouter**, **UnreadRouter**, and **WriteString**. The **WPROMPT**, **WDISPLAY**, **WTRACE**, and **WDIALOG** logical names are no longer supported. The **WWARNING** and **WERROR** logical names have been renamed to **STDWRN** and **STDERR**. See section 9 for more information.

- **Garbage Collection API Redesign** – The Garbage Collection API has been redesigned. The functions **IncrementGCLocks**, **DecrementGCLocks**,

IncrementFactCount, **DecrementFactCount**, **IncrementInstanceCount**, **and DecrementInstanceCount** are no longer supported. The **Retain** and **Release** functions should be used to prevent primitive values from being garbage collected. See section 5 for more information.

- **FactBuilder and FactModifier APIs** – The FactBuilder and FactModifier APIs provides functions for creating a modifying facts. See section 7 for more information.

- **InstanceBuilder and InstanceModifier APIs** – The InstanceBuilder and InstanceModifier APIs provides functions for creating a modifying instances. See section 7 for more information.

- **StringBuilder API** – The StringBuilder API provides functions for dynamically allocating and appending strings. See section 4.3 for more information.

- **Eval Function** – The Eval function is now available for use in run-time programs. See sections 4.2 and 11 for more information.

- **Compiler Directives** – The **ALLOW_ENVIRONMENT_GLOBALS** flag has been removed. The **VAX_VMS** flag has been removed.

Index