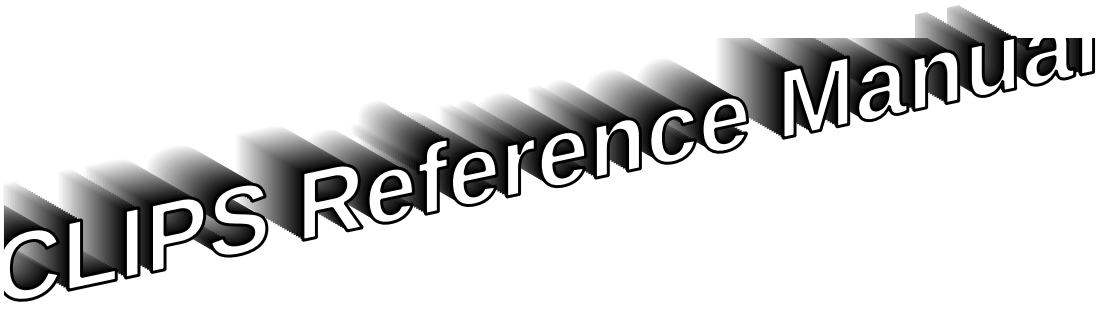
JSC-25012



Volume II
Advanced Programming Guide

CLIPS Version 6.03

June 15th 1995

Software Technology Branch

Lyndon B. Johnson Space Center

CLIPS Advanced Programming Guide

Version 6.03 June 15th 1995

CONTENTS

Preface

The History of CLIPS

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

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

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

The original intent of the prototype was to gain useful insight and knowledge about the construction of expert system tools and to lay the groundwork for the construction of a fully usable tool. The CLIPS prototype had numerous shortcomings, however, it demonstrated the feasibility of the project concept. After additional development, it became apparent that sufficient enhancements to the prototype would produce a low cost expert system tool that would be ideal for the purposes of training. Another year of development and internal use went into CLIPS improving its portability, performance, and functionality. A reference manual and user's guide were written during this time. The first release of CLIPS to groups outside of NASA, version 3.0, occurred in the summer of 1986.

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

Originally, the primary representation methodology in CLIPS was a forward chaining rule language based on the Rete algorithm (hence the Production System part of the CLIPS acronym). Version 5.0 of CLIPS, released in the spring of 1991, introduced two new programming paradigms: procedural programming (as found in languages such as C and Ada) and object-oriented programming (as found in languages such as the Common Lisp Object System and Smalltalk). The object-oriented programming language provided within CLIPS is called the CLIPS Object-Oriented Language (COOL). Version 5.1 of CLIPS, released in the fall of 1991, was primarily a software maintenance upgrade required to support the newly developed and/or enhanced X Window, MS-DOS, and Macintosh interfaces.

Because of its portability, extensibility, capabilities, and low-cost, CLIPS has received widespread acceptance throughout the government, industry, and academia. The development of CLIPS has helped to improve the ability to deliver expert system technology throughout the public and private sectors for a wide range of applications and diverse computing environments. CLIPS is being used by over 4,000 users throughout the public and private community including: all NASA sites and branches of the military, numerous federal bureaus, government contractors, universities, and many private companies. CLIPS is available at a nominal cost through COSMIC, the NASA software distribution center (for more on COSMIC, see appendix E of the *Basic Programming Guide*).

CLIPS Version 6.0

Version 6.0 of CLIPS contains five major enhancements. First, instances of user-defined classes in COOL can be pattern-matched on the left-hand side of rules. Second, CLIPS now contains considerable support for knowledge based systems software engineering. Support is now provided for building modular systems and many of the features previously available in CRSV are now directly supported in CLIPS (such as constraint consistency among uses of the same variable). Third, deftemplates can now have more than one multifield slot. Fourth, it is now possible to nest other conditional elements within a *not* conditional element and two new conditional elements, *exists* and *forall* , are supported. Fifth, a Windows 3.1 CLIPS interface is now available for PC compatible computers. In addition, an MS-DOS 386 version of CLIPS is available which can use extended memory. For a detailed listing of differences between versions 5.1 and 6.0 of CLIPS, refer to appendix D of the *Basic Programming Guide* and appendix C of the *Advanced Programming Guide*.

CLIPS Documentation

Three documents are provided with CLIPS.

- The *CLIPS Reference Manual* which is split into the following parts:
  - *Volume I - The Basic Programming Guide*, which provides the definitive description of CLIPS syntax and examples of usage.
  - *Volume II - The Advanced Programming Guide*, which provides detailed discussions of the more sophisticated features in CLIPS and is intended for people with extensive programming experience who are using CLIPS for advanced applications.
  - *Volume III - The Interfaces Guide*, which provides information on machine-specific interfaces.
- The *CLIPS User's Guide* which provides an introduction to CLIPS rule-based and object-oriented programming and is intended for people with little or no expert system experience.
- The *CLIPS Architecture Manual* which provides a detailed description of the CLIPS software architecture. This manual describes each module of CLIPS in terms of functionality and purpose. It is intended for people with extensive programming experience who are interested in modifying CLIPS or who want to gain a deeper understanding of how CLIPS works.

Acknowledgements

As with any large project, CLIPS is the result of the efforts of numerous people. The primary contributors have been: Robert Savely, previous branch chief of the STB and now chief scientist of advanced software technology at JSC, who conceived the project and provided overall direction and support; Chris Culbert, current branch chief of the STB, who managed the project, wrote the original *CLIPS Reference Manual*, and designed the original version of CRSV; Gary Riley, who designed and developed the rule-based portion of CLIPS, co-authored the *CLIPS Reference Manual* and *CLIPS Architecture Manual*, and developed the Macintosh interface for CLIPS; Brian Donnell, who designed and developed the CLIPS Object Oriented Language (COOL), co-authored the *CLIPS Reference Manual* and *CLIPS Architecture Manual*, and developed the previous MS-DOS interfaces for CLIPS; Bebe Ly, who was responsible for maintenance and enhancements to CRSV and is now responsible for developing the X Window interface for CLIPS; Chris Ortiz, who developed the Windows 3.1 interface for CLIPS; Dr. Joseph Giarratano of the University of Houston-Clear Lake, who wrote the *CLIPS User's Guide*; and Frank Lopez, who wrote the original prototype version of CLIPS.

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

Section 1 - Introduction

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

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

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

1.1 WARNING ABOUT INTERFACING WITH CLIPS

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

1.2 COMPATIBILITY WITH CLIPS VERSION 5.1

There are significant differences in external integration between CLIPS 5.1 and CLIPS 6.0. It is recommended that you completely scan the *Advanced Programming Guide* and carefully read Section 3 before converting user code from previous versions of CLIPS. Numerous changes have

been made to standardize the naming conventions of access functions, simplify certain operations, and provide additional functionality. Because many of the changes involved changing function names for consistency, the header file **cmptblty.h** has been provided (which should be included after the **clips.h** header file). This file provides macro definitions which map old functions and macros into CLIPS 6.0 functions and macros. Note that the **cmptblty.h** header file is provided only as a convenient mechanism for quickly upgrading old user code to run with CLIPS 6.0. It is recommended that old code be eventually converted to take advantage of the new access functions and macros.

1.3 USING ANSI PROTOTYPES

CLIPS 6.0 supports ANSI function prototypes. When including CLIPS header files, you can indicate whether prototypes should be used by defining the **ANSI_COMPILER** flag to be 1 within the **setup.h** header file. The **setup.h** header file is automatically included when the **clips.h** header file is used.

Section 2 - Installing and Tailoring CLIPS

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

2.1 INSTALLING CLIPS

A typical CLIPS package includes both documentation and a number of floppy disks in either the MS-DOS 2.1 format or the Macintosh format. The floppy disks contain a CLIPS executable, examples of CLIPS programs, and the CLIPS source code. Users *should* make copies of the distribution disks. If CLIPS is to be used on a Macintosh, the executable may be copied to a hard disk or to another floppy disk. If CLIPS is to be used on a MS-DOS machine, the executable may be copied using the installation program provided on the disk (see the readme.txt file on the disk for more details). No other installation is required to run the standard version of CLIPS. To tailor CLIPS or to install it on another machine, the user must port the source code and create a new executable version.

Internal testing of CLIPS covers many different hardware/software environments, including

- Sun Sparcstation running UNIX
- IBM PC 386 running DOS 5.0 with Zortech C++ v3.1 and Microsoft Windows 3.1 with Borland C++ v4.0
- Apple Macintosh IIfx and Power Macintosh running System 7.5 using Symantec C 7.0.3, Symantec C 8.0, and CodeWarrior 5.0

CLIPS was designed specifically for portability and has been installed on numerous other computers without making modifications to the source code. It *should* run on any system which supports a full Kernighan and Ritchie (K&R) C compiler, and it *will* run with any C compiler that is compatible with the ANSI C standard. CLIPS *cannot* be compiled using strict C++ compilers. C++ is not a proper superset of ANSI C, and therefore some ANSI compliant code cannot be compiled by strict C++ compilers. Specifically, old style K&R function declarations either do not compile or have different meanings in C++ than in ANSI C. CLIPS uses K&R style function definitions so that it will compile on both K&R and ANSI C compilers. Also, some compilers have extended syntax to support a particular platform which will add additional reserved words to the C language. In the event that this extended syntax conflicts with the CLIPS source, the user will have to edit the code. This usually only involves a global search-and-replace of the particular reserved word. The following steps describe how to create a new executable version of CLIPS:

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

   agenda.h analysis.h argacces.h bload.h

bmathfun.h	bsave.h	classcom.h	classexm.h
classfun.h	classinf.h	classini.h	classpsr.h
clips.h	clipsmem.h	clsltpsr.h	cmptblty.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	dfinscmp.h	drive.h
ed.h	engine.h	evaluatn.h	expressn.h
exprnbin.h	exprnops.h	exprnpsr.h	extnfunc.h
extobj.h	factbin.h	factbld.h	factcmp.h
factcom.h	factgen.h	facthsh.h	factlhs.h
factmch.h	factmngr.h	factprt.h	factrete.h
factrhs.h	filecom.h	filertr.h	generate.h
genrcbin.h	genrccmp.h	genrccom.h	genrcexe.h
genrcfun.h	genrcpsr.h	globlbin.h	globlbsc.h
globlcmp.h	globlcom.h	globldef.h	globlpsr.h
immthpsr.h	incrrset.h	inherpsr.h	inscom.h
insfile.h	insfun.h	insmngr.h	insmoddp.h
insmult.h	inspsr.h	insquery.h	insqypsr.h
lgcldpnd.h	match.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	pattern.h	pprint.h	prccode.h
prcdrfun.h	prcdrpsr.h	prntutil.h	reorder.h
reteutil.h	retract.h	router.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	shrtlnkn.h
strngfun.h	strngrtr.h	symblbin.h	symblcmp.h
symbol.h	sysdep.h	tmpltbin.h	tmpltbsc.h
tmpltcmp.h	tmpltcom.h	tmpltdef.h	tmpltfun.h
tmpltlhs.h	tmpltpsr.h	tmpltrhs.h	utility.h
watch.h			

agenda.c	analysis.c	argacces.c	bload.c
bmathfun.c	bsave.c	classcom.c	classexm.c

classfun.c	classinf.c	classini.c	classpsr.c
clsltpsr.c	commline.c	conscomp.c	constrct.c
constrnt.c	crstrtgy.c	cstrcbin.c	cstrccom.c
cstrcpsr.c	cstrnbin.c	cstrnchk.c	cstrncmp.c
cstrnops.c	cstrnpsr.c	cstrnutl.c	default.c
defins.c	developr.c	dffctbin.c	dffctbsc.c
dffctcmp.c	dffctdef.c	dffctpsr.c	dffnxbin.c
dffnxcmp.c	dffnxexe.c	dffnxfun.c	dffnxpsr.c
dfinsbin.c	dfinscmp.c	drive.c	edbasic.c
edmain.c	edmisc.c	edstruct.c	edterm.c
emathfun.c	engine.c	evaluatn.c	expressn.c
exprnbin.c	exprnops.c	exprnpsr.c	extnfunc.c
factbin.c	factbld.c	factcmp.c	factcom.c
factgen.c	facthsh.c	factlhs.c	factmch.c
factmngr.c	factprt.c	factrete.c	factrhs.c
filecom.c	filertr.c	generate.c	genrcbin.c
genrccmp.c	genrccom.c	genrcexe.c	genrcfun.c
genrcpsr.c	globlbin.c	globlbsc.c	globlcmp.c
globlcom.c	globldef.c	globlpsr.c	immthpsr.c
incrrset.c	inherpsr.c	inscom.c	insfile.c
insfun.c	insmngr.c	insmoddp.c	insmult.c
inspsr.c	insquery.c	insqypsr.c	iofun.c
lgcldpnd.c	main.c	memory.c	miscfun.c
modulbin.c	modulbsc.c	modulcmp.c	moduldef.c
modulpsr.c	modulutl.c	msgcom.c	msgfun.c
msgpass.c	msgpsr.c	multifld.c	multifun.c
objbin.c	objcmp.c	objrtbin.c	objrtbld.c
objrtcmp.c	objrtfnx.c	objrtgen.c	objrtmch.c
pattern.c	pprint.c	prccode.c	prcdrfun.c
prcdrpsr.c	prdctfun.c	prntutil.c	reorder.c
reteutil.c	retract.c	router.c	rulebin.c
rulebld.c	rulebsc.c	rulecmp.c	rulecom.c
rulecstr.c	ruledef.c	ruledlt.c	rulelhs.c
rulepsr.c	scanner.c	strngfun.c	strngrtr.c
symblbin.c	symblcmp.c	symbol.c	sysdep.c
textpro.c	tmpltbin.c	tmpltbsc.c	tmpltcmp.c
tmpltcom.c	tmpltdef.c	tmpltfun.c	tmpltlhs.c
tmpltpsr.c	tmpltrhs.c	utility.c	watch.c

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

2) **Modify all include statements (if necessary)**
   All of the “.c” files and most of the “.h” files have #include statements. These #include statements may have to be changed to either match the way the compiler searches for include files or to include a different ".h" file for a non-ANSI C compiler. Note: If an ANSI C compiler is being used and the compiler is set up properly, this step should be unnecessary.

3) **Tailor CLIPS environment and/or features**
   Edit the setup.h file and set any special options. CLIPS uses compiler directives to allow machine-dependent features. The first flag in the setup.h file tells CLIPS on what kind of machine the code is being compiled. The default setting for this flag is GENERIC, which will create a version of CLIPS that will run on any computer. The user may set this flag for the user's type of system. If the system type is unknown, the first flag should be set to GENERIC. If you change the system type to anything other than GENERIC, make sure that the version number of your compiler is greater than or equal to the version number listed in the setup.h file (as earlier versions of a compiler may not support some system dependent features). Other flags 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

4) **Compile all of the ".c" files to object code**
   Use the standard compiler syntax for the user's machine. The ".h" files are include files used by the other files and do not need to be compiled. Some options may have to be set, depending on the compiler. Many microcomputer compilers support either large or small memory compilation. CLIPS should *always* use the large memory option. Other compilers default to 8-character variable names but allow more with an option. CLIPS uses variables that require more than 8 characters to be distinctly identified; if necessary, this option should be set.

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

5) **Create the interactive CLIPS executable element**
   To create the interactive CLIPS executable, link together all of the object files. This executable will provide the interactive interface defined in section 2.1 of the *Basic Programming Guide*. On some machines, the default stack size is too small to run CLIPS properly. Usually, the default stack size can be changed during the link process. At least 4000 bytes of stack size are needed to run CLIPS reasonably.

2.1.1 Additional Considerations

Although compiling CLIPS should not be difficult even for inexperienced C programmers, some non-obvious problems can occur. One type of problem is linking with inappropriate system

libraries. Normally, default libraries are specified through the environment; i.e., not specified as a part of the compile/link process. On occasion, the default system libraries are inappropriate for use with CLIPS. For example, when using a compiler which supports different memory models, be sure to link with the system libraries that match the memory model under which the CLIPS code was compiled. The same can be said for floating-point models. Some computers provide multiple ways of storing floating-point numbers (typically differing in accuracy or speed of processing). Be sure to link with system libraries that use the same storage formats with which the CLIPS code was compiled. Some additional considerations for compiling CLIPS with specific compilers and/or operating systems are described following.

UNIX

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

```
cc -o clips *.c -lm -ltermcap
```

Macintosh (Symantec C V7.03 and V8.0)

Enable the **Separate STRS** and **Far DATA** options using the **Set Project Type...** menu item. Under this same menu item, set the **Partition (K)** size to at least 1000 and enable the **32-Bit Compatible** flag in the **SIZE Flags** pop-up menu. If the Macintosh interface is being compiled, enable the **MultiFinder Aware**, **Background NULL Events**, **Suspend & Resume Events**, and **HighLevelEvent-Aware** flags in the **SIZE flag** pop-up menu.

When using Symantec C 8.0, disable global optimizations. The CLIPS validation test suite fails when global optimizations are enabled.

Macintosh (Metrowerks CodeWarrior V5.0)

For unknown reasons, the 680x0 version of CodeWarrior was never able to create a 680x0 executable that could complete the CLIPS validation suite. If you need to create a 68K version of CLIPS, it is recommended that you use Symantec C 7.0 to do so.

Macintosh (MPW C V3.2)

When compiling the CLIPS source files, use the **-b3** option. When linking, use the **-srt** option.

IBM PC (Microsoft C V6.0A with MS-DOS)

When compiling the CLIPS source files, use the **/AL** option. When linking, use the **/SEGMENTS:256** and **/STACK:8000** options (the stack option provides a reasonable amout of space, however, this may need to be adjusted up or down depending upon the type of applications being run—applications using deffunctions, generic functions, and message-handlers will generally require more stack space). With the DOS 640K memory limit, it is not

possible to create an executable containing all of the CLIPS features. Some features must be disabled (see section 2.2).

IBM PC (Borland C++ V3.1 with MS-DOS)
When compiling and linking the CLIPS source files, use the **-ml** and **-d** options. With the DOS 640K memory limit, it is not possible to create an executable containing all of the CLIPS features. Some features must be disabled (see section 2.2).

IBM PC (Zortech C++ V3.1 with MS-DOS)
There is a compiler bug which manifests itself when dead code optimizations are performed. When compiling the CLIPS source files, specify the **-o-dc** option to remove dead code optimizations. In addition, do not use the **-s** option (stack overflow checking doesn't work properly for functions called from an interrupt handler). Use the **-mx** option when compiling and the **=16000** option when linking.

IBM PC (Borland C++ V4.0 with MS-WINDOWS)
When compiling the CLIPS source files, set the following options. For **Code Generation**: Allocate enums as ints and Duplicate strings merged. For **Memory Model**: Large, Never Assume SS Equals DS, Automatic Far Data, and Far Data Threshold set to 10 Bytes. For **Entry/Exit Code**: Windows smart callbacks, all functions exportable.

2.2 TAILORING CLIPS

CLIPS makes use of compiler directives 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.

ANSI_COMPILER This flag indicates whether the compiler being used follows the draft proposed ANSI C standards (including the ANSI C libraries). If this flag is on, the compiler is assumed to be a *fully* ANSI standard compiler, otherwise it is assumed to be a K & R standard compiler. This is on in the standard CLIPS executable.

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

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.

CONFLICT_RESOLUTION_STRATEGIES

This flag controls the availability of conflict resolution strategies (see sections 5.2 and 5.3 of the *Basic Programming Guide*) for use with the defrule construct. If it is off, then the depth conflict resolution strategy is the only strategy used and the functions set-strategy and get-strategy are not available. This is on in the standard CLIPS executable.

DYNAMIC_SALIENCE

This flag controls the availability of dynamic salience (see sections 5.2 and 5.4.9 of the *Basic Programming Guide*) for use with the defrule construct. If it is off, then dynamic salience can not be used and the functions refresh-agenda, get-salience-evaluation, and get-salience-evaluation are not available. This is on in the standard CLIPS executable.

INCREMENTAL_RESET

This flag controls the availability of incremental reset (see sections 5.1 and 12.1.7 of the *Basic Programming Guide*) for use with the defrule construct. If it is off, then newly defined rules are not aware of facts or instances that were created before the rule was defined. In addition, the functions set-incremental-reset and get-incremental-reset are not available if this flag is off. This is on in the standard CLIPS executable.

LOGICAL_DEPENDENCIES

This flag controls the availability of logical dependencies (see section 5.4.8 of the *Basic Programming Guide*) for use with the defrule construct. If it is off, then the logical CE cannot be used on the LHS of a rule and the functions dependencies and dependents are not available. This is on in the standard CLIPS executable.

DEFFACTS_CONSTRUCT

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

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.

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.

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.

IMPERATIVE_METHODS

This flag determines if the following functions are available for use in generic function methods: **next-methodp**, **call-next-method**, **override-next-method** and **call-specific-method**. These functions allow imperative control over the generic dispatch by calling shadowed methods (see section 8.5.3 of the *Basic Programming Guide*). This flag is on in the standard CLIPS executable. Turning this flag off can save some memory and marginally increase the speed of the generic dispatch.

OBJECT_SYSTEM

This flag controls the use of defclass, definstances, and defmessage-handler. If it is off, these constructs are not allowed which can save some memory. If this flag is on, the MULTIFIELD_FUNCTIONS flag should also be on if you want to be able to manipulate multifield slots. This is on in the standard CLIPS executable.

DEFINSTANCES_CONSTRUCT

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

IMPERATIVE_MESSAGE_HANDLERS

This flag determines if **around** message-handlers and the following functions are available for use in object message dispatch: **next-handlerp**, **call-next-handler** and **override-next-handler**. These functions allow imperative control over the message dispatch by calling shadowed message-handlers (see section 9.5.3 of the *Basic Programming Guide*). This flag is on in the standard CLIPS executable. Turning this flag off can save some memory and marginally increase the speed of the message dispatch.

AUXILIARY_MESSAGE_HANDLERS

This flag determines if **before** and **after** message-handlers are available for use in object message dispatch. These handler types enhance declarative control over the message dispatch (see section 9.4.3 of the *Basic Programming Guide*). This flag is on in the standard CLIPS executable. Turning this flag off can save some memory and marginally increase the speed of the message dispatch.

INSTANCE_SET_QUERIES

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

INSTANCE_PATTERN_MATCHING

This flag controls the ability to include object patterns on the LHS of rules (see section 5.4.1.8 of the *Basic Programming Guide*). This is on in the standard CLIPS executable. Turning this flag off can save some memory.

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.

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.

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.

EX_MATH 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.

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

CLP_HELP If this flag is on, the on-line help system will be available from the CLIPS top-level interface. When this is turned on, the **HELP_FILE** flag should be set to point to the full path name for the CLIPS help file. This is on in the standard CLIPS executable.

CLP_EDIT This flag controls the integrated MicroEMACS editor. If it is turned on, the editor will be available. If it is turned off, the editor will not be available but about 40K of memory will be saved. NOTE: The editor is machine dependent and will not run on all machines. See the **setup.h** file for a description of which machines can support the editor. This is on in the standard CLIPS executable.

CONSTRUCT_COMPILER

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

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.

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.

BASIC_IO This flag controls access to the basic I/O functions in CLIPS. These functions are printout, read, open, and close. 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.

EXT_IO This flag controls access to the extended I/O functions in CLIPS. These functions are format and readline. 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 or only the basic I/O routines. This is on in the standard CLIPS executable.

MULTIFIELD_FUNCTIONS

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

STRING_FUNCTIONS

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

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.

BLOCK_MEMORY This option controls memory allocation. If the flag is on, memory is allocated from the operating system in large blocks. This can improve performance if the system memory allocation routines are extremely inefficient or place arbitrary restrictions on the number of memory allocations that can be made. This flag is off in the standard CLIPS executable.

WINDOW_INTERFACE

This flag indicates that a windowed interface is being used. Currently, the help system uses this flag to determine whether it should handle more processing by itself or allow the interface to take care of more processing. This is off in the standard CLIPS executable.

SHORT_LINK_NAMES

ANSI C compilers must be able to distinguish between identifiers which use at least 31 significant characters. Some linkers, however, use considerably fewer characters when determining name conflicts (potentially as few as 6 characters). If this flag is on, then identifiers which cannot be uniquely distinguished within 6 characters are replaced with alternate names that are distinguishable with 6 characters. This is off in the standard CLIPS executable.

Section 3 - Integrating CLIPS with External Functions

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

3.1 DECLARING USER-DEFINED EXTERNAL FUNCTIONS

All external functions must be described to CLIPS so they can be properly accessed by CLIPS programs. User-defined functions are described to CLIPS by modifying the function **UserFunctions**. This function is initially in the CLIPS **main.c** file and may be modified there or moved to a user's file. Within **UserFunctions**, a call should be made to the **DefineFunction** routine for every function which is to be integrated with CLIPS. The user's source code then can be compiled and linked with CLIPS.

```
int      DefineFunction(functionName,functionType,
                        functionPointer,actualFunctionName);

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

An example **UserFunctions** declaration follows:

```
UserFunctions()
  {
   /*=======================================*/
   /* Declare your C functions if necessary. */
   /*=======================================*/

   extern double rta();
   extern VOID *dummy();

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

   DefineFunction("rta",'d',PTIF rta,"rta");
   DefineFunction("mul",'l',PTIF mul,"mul");
  }
```

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

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

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

Boolean functions should return a value of type int (0 for the symbol FALSE and any other value for the symbol TRUE). String, symbol, instance name, external address, and instance address functions should return a pointer of type VOID *. Character return values are converted by CLIPS to a symbol of length one. Integer return values are converted by CLIPS to long integers for internal storage. Single precision float values are converted by CLIPS to double precision float values for internal storage. If a user function is not going to return a value to CLIPS, the function should be defined as type VOID and this argument should be v for void. Return types *o* and *x* are only available if the object system has been enabled (see section 2.2).

Function types *j*, *k*, *m*, *n*, and *u* are all passed a data object as an argument in which the return value of function is stored. This allows a user defined function to return one of several possible return types. Function type *u* is the most general and can return any data type. By convention, function types *j*, *k*, *m*, and *n* return specific data types. CLIPS will signal an error if one of these functions return a disallowed type. See section 3.3.4 for more details on returning unknown data types.

The third argument is a pointer to the actual function, the compiled function name (an **extern** declaration of the function may be appropriate). The CLIPS name (first argument) need not be the same as the actual function name (third argument). The macro identifier PTIF can be placed

in front of a function name to cast it as a pointer to a function returning an integer (primarily to prevent warnings from compilers which allow function prototypes).

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

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

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

In place of **DefineFunction**, the **DefineFunction2** function can be used to provide additional information to CLIPS about the number and types of arguments expected by a CLIPS function or command.

```
int      DefineFunction2(functionName,functionType,
                         functionPointer,actualFunctionName,
                         functionRestrictions);

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

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

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

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

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

Examples

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

```
"3*"
```

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

```
"*5"
```

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

```
"35n"
```

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

```
"66fsui"
```

3.2 PASSING ARGUMENTS FROM CLIPS TO EXTERNAL FUNCTIONS

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

3.2.1 Determining the Number of Passed Arguments

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

```
int      RtnArgCount();
int      ArgCountCheck(functionName,restriction,count);
int      ArgRangeCheck(functionName,min,max);

int      restriction, count, min, max;
char    *functionName;
```

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

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

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

```
char    *RtnLexeme(argumentPosition);
double   RtnDouble(argumentPosition);
long     RtnLong(argumentPosition);

int      argumentPosition;
```

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

Example

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

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

/*
Use DefineFunction2("rta",'d',PTIF rta,"rta","22n");
*/

double rta()
  {
   double base, height;

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

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

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

   base = RtnDouble(1);
   height = RtnDouble(2);

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

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

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

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

3.2.3 Passing Unknown Data Types

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

```
#include "clips.h"           /* or "evaluatn.h" */

DATA_OBJECT *RtnUnknown(argumentPosition, &argument);

int    GetType(argument);
int    GetpType(&argument);
int    ArgTypeCheck(char *,argumentPosition,
                    expectedType,&argument);

char  *DOToString(argument);
char  *DOPToString(&argument);
double DOToDouble(argument);
double DOPToDouble(&argument);
float  DOToFloat(argument);
float  DOPToFloat(&argument);
long   DOToLong(argument);
long   DOPToLong(&argument);
int    DOToInteger(argument);
int    DOPToInteger(&argument);
VOID  *DOToPointer(argument);
VOID  *DOPToPointer(&argument);

int argumentPosition, expectedType;
DATA_OBJECT argument;
```

Function **RtnUnknown** should be called first. It copies the elements of the internal CLIPS structure that represent the unknown-type argument into the DATA_OBJECT structure pointed to by the second argument. It also returns a pointer to that same structure, passed as the second

argument. After obtaining a pointer to the DATA_OBJECT structure, a number of macros can be used to extract type information and the arguments value.

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

The function **ArgTypeCheck** can be used for error checking if a function expects a specific type of argument for a particular parameter. It returns a non-zero integer value if the parameter was of the specified type, otherwise it returns zero. The first argument is the name of the function to be printed within the error message if the type restriction is unsatisfied. The second argument is the index of the parameter to be tested. The third argument is the type restriction and must be one of the following CLIPS defined constants: STRING, SYMBOL, SYMBOL_OR_STRING, FLOAT, INTEGER, INTEGER_OR_FLOAT, MULTIFIELD, EXTERNAL_ADDRESS, INSTANCE_ADDRESS, INSTANCE_NAME, or INSTANCE_OR_INSTANCE_NAME. If the FLOAT type restriction is used, then integer values will be converted to floating-point numbers. If the INTEGER type restriction is used, then floating-point values will be converted to integers. The fourth argument is a pointer to a DATA_OBJECT structure in which the unknown parameter will be stored.

Example

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

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

/*
Use DefineFunction2("mul",'l',PTIF mul,"mul","22n");
*/
```

```
long mul()
  {
   DATA_OBJECT temp;
   long firstNumber, secondNumber;

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

   if (ArgCountCheck("mul",EXACTLY,2) == -1)
     { return(1L); }

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

   if (ArgTypeCheck("mul",1,INTEGER_OR_FLOAT,&temp) == 0)
     { return(1L); }

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

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

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

   RtnUnknown(2,&temp);

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

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

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

   return (firstNumber * secondNumber);
  }
```

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

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

3.2.4 Passing Multifield Values

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

```
#include "clips.h"          /* or "evaluatn.h" */

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

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

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

first element in the multifield value and the end index points to the last element in the multifield value. A multifield value of length one will have the same values for the begin and end indices. A multifield value of length zero will have an end index that is one less than the begin index.

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

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

Examples

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

```
#include "clips.h"

/*
Use DefineFunction2("mfl",'l',PTIF MFLength,"MFLength","11m");
*/

long int MFLength()
  {
   DATA_OBJECT argument;

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

   if (ArgCountCheck("mfl",EXACTLY,1) == -1) return(-1L);

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

   if (ArgTypeCheck("mfl",1,MULTIFIELD,&argument) == 0)
     { return(-1L); }

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

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

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

```
#include "clips.h"

/*
Use DefineFunction2("cmfc",'l',PTIF CntMFChars,"CntMFChars",
                    "11m");
*/

long int CntMFChars()
  {
   DATA_OBJECT argument;
   VOID *multifieldPtr;
   int end, i;
   long count = 0;
   char *tempPtr;

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

   if (ArgCountCheck("cmfc",EXACTLY,1) == -1) return(0L);

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

   if (ArgTypeCheck("cmfc",1,MULTIFIELD,&argument) == 0)
     { return(0L); }

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

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

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

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

   return(count);
  }
```

3.3 RETURNING VALUES TO CLIPS FROM EXTERNAL FUNCTIONS

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

3.3.1 Returning Symbols, Strings, and Instance Names

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

```
   #include "clips.h"           /* or "symbol.h" */

   VOID *AddSymbol(string);
   char *string;
```

Example

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

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

/*
Use DefineFunction2("reverse-str",'s',PTIF Reverse,"Reverse",
                    "11s");
*/

VOID *Reverse()
  {
   DATA_OBJECT temp;
   char *lexeme, *tempString;
   VOID *returnValue;
   int i, length;

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

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

   if (ArgCountCheck("reverse-str",EXACTLY,1) == -1)
     { return(AddSymbol("")); }

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

   if (ArgTypeCheck("reverse-str",1,STRING,&temp) == 0)
     { return(AddSymbol("")); }
   lexeme = DOToString(temp);

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

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

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

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

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

   returnValue = AddSymbol(tempString);
   free(tempString);
   return(returnValue);
  }
```

3.3.2 Returning Boolean Values

A user function may return a boolean value in one of two ways. The user may define an integer function and use **DefineFunction** to declare it as a BOOLEAN type ('b'). The function should then either return the value **CLIPS_TRUE** or **CLIPS_FALSE**. Alternatively, the function may be declare to return a SYMBOL type ('w') or UNKNOWN type ('u') and return the symbol **CLIPSFalseSymbol** or **CLIPSTrueSymbol**.

```
   #include "clips.h"           /* or "symbol.h" */

   #define CLIPS_FALSE 0
   #define CLIPS_TRUE  1

   VOID *CLIPSFalseSymbol
```

```
   VOID *CLIPSTrueSymbol
```

Examples

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

```
#include "clips.h"

/*
Use DefineFunction2("positivep1",'b',positivep1,"positivep1",
                    "11n");
*/

int positivep1()
  {
   DATA_OBJECT temp;

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

   if (ArgCountCheck("positivep1",EXACTLY,1) == -1)
     { return(CLIPS_FALSE); }

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

   if (ArgTypeCheck("positivep1",1,INTEGER_OR_FLOAT,&temp) == 0)
     { return(CLIPS_FALSE); }

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

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

   return(CLIPS_TRUE);
  }
```

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

```
#include "clips.h"

/*
Use DefineFunction("positivep2",'w',PTIF positivep2,"positivep2",
                   "11n");
*/

VOID *positivep2()
  {
   DATA_OBJECT temp;

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

   if (ArgCountCheck("positivep1",EXACTLY,1) == -1)
     { return(CLIPSFalseSymbol); }

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

   if (ArgTypeCheck("positivep1",1,INTEGER_OR_FLOAT,&temp) == 0)
     { return(CLIPSFalseSymbol); }

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

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

   return(CLIPSTrueSymbol);
  }
```

3.3.3 Returning External Addresses and Instance Addresses

A user function may return an external address or an instance address. The user should use **DefineFunction** to declare their function as returning an external address type ('a') or an instance address type ('x'). The function should then either return a pointer that has been typecast to (VOID *). Within CLIPS, the printed representation of an external address is

```
<Pointer-XXXXXXXX>
```

where XXXXXXXX is the external address. Note that it is up to the user to make sure that external addresses remain valid within CLIPS. The printed representation of an instance address is

```
<Instance-XXX>
```

where XXX is the name of the instance.

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

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

/*
Use DefineFunction2("malloc",'a',PTIF CLIPSmalloc,"CLIPSmalloc",
                    "00");
*/

VOID *CLIPSmalloc()
  { return((VOID *) malloc(100)); }
```

3.3.4 Returning Unknown Data Types

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

```
#include "clips.h"          /* or "evaluatn.h" */

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

VOID  *SetValue(argument,value)
VOID  *SetpValue(&argument,value)

VOID  *AddLong(longValue);
VOID  *AddDouble(doubleValue);

VOID  *GetValue(argument);
VOID  *GetpValue(&argument);
```

```
   char  *ValueToString(value);
   double ValueToDouble(value);
   long   ValueToLong(value);
   int    ValueToInteger(value);

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

Macros **SetType** and **SetpType** can be used to set the type of a DATA_OBJECT or DATA_OBJECT_PTR respectively. The type parameter should be one of the following CLIPS defined constants (note that these are not strings): SYMBOL, STRING, INTEGER, FLOAT, EXTERNAL_ADDRESS, INSTANCE_NAME, or INSTANCE_ADDRESS. Macros **SetValue** (for DATA_OBJECTs) and **SetpValue** (for DATA_OBJECT_PTRs) can be used to set the value of a DATA_OBJECT. The functions **AddSymbol** (for symbols, strings and instance names), **AddLong** (for integers) and **AddDouble** (for floats) can be used to produce values that can be used with these macros (external addresses and instance addresses can be used directly). Macros **GetValue** (for DATA_OBJECTs) and **GetpValue** (for DATA_OBJECT_PTRs) can be used to retrieve the value of a DATA_OBJECT. Note that the value for an external address or an instance address can be retrieved directly using one of these macros. For other data types, the macros **ValueToString** (for symbols, strings, and instance names), **ValueToLong** (for integers), **ValueToInteger** (for integers), and **ValueToDouble** (for floats) can be used to convert the retrieved value from a DATA_OBJECT to a C object of type char *, double, long, or integer respectively.

Example

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

```
#include "clips.h"

/*
Use DefineFunction2("cube",'u',PTIF cube,"cube","11n");
*/

VOID cube(returnValuePtr)
  DATA_OBJECT_PTR returnValuePtr;
  {
   VOID *value;
   long longValue;
   double doubleValue;

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

```
   if (ArgCountCheck("cube",EXACTLY,1) == -1)
     {
      SetpType(returnValuePtr,SYMBOL);
      SetpValue(returnValuePtr,CLIPSFalseSymbol);
      return;
     }

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

   if (! ArgTypeCheck("cube",1,INTEGER_OR_FLOAT,returnValuePtr))
     {
      SetpType(returnValuePtr,SYMBOL);
      SetpValue(returnValuePtr,CLIPSFalseSymbol);
      return;
     }

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

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

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

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

3.3.5 Returning Multifield Values

Multifield values can also be returned from an external function. When defining such an external function, the data type should be set to 'm' in the call to **DefineFunction**. Note that a multifield value can also be returned from a 'u' function, whereas only a multifield value should be returned

from an 'm' function. As with returning unknown data types, the user function will be passed a pointer of type DATA_OBJECT_PTR which can be modified to set up a multifield value. The following macros and functions are useful for this purpose:

```
VOID  *CreateMultifield(size);
int    SetMFType(multifieldPtr,fieldPosition,type);
VOID  *SetMFValue(multifieldPtr,fieldPosition,value);
int    SetDOBegin(returnValue,fieldPosition);
int    SetpDOBegin(&returnValue,fieldPosition);
int    SetDOEnd(returnValue,fieldPosition);
int    SetpDOEnd(&returnValue,fieldPosition);
VOID   SetMultifieldErrorValue(&returnValue);

DATA_OBJECT returnValue;
int size, fieldPosition;
VOID *multifieldPtr;
VOID *value;
```

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

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

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

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

Examples

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

```
#include "clips.h"

/*
Use DefineFunction2("sample4",'m',PTIF sample4,"sample4","00");
*/

VOID sample4(returnValuePtr)
  DATA_OBJECT_PTR returnValuePtr;
  {
   VOID *multifieldPtr;

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

   if (ArgCountCheck("sample4",EXACTLY,0) == -1)
     {
      SetMultifieldErrorValue(returnValuePtr);
      return;
     }

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

   multifieldPtr = CreateMultifield(2);

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

   SetMFType(multifieldPtr,1,SYMBOL);
   SetMFValue(multifieldPtr,1,AddSymbol("altitude"));

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

   SetMFType(multifieldPtr,2,FLOAT);
   SetMFValue(multifieldPtr,2,AddDouble(900.0));

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

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

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

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

   return;
  }
```

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

```
#include "clips.h"

/*
Use DefineFunction2("rest",'m',PTIF rest,"rest","11m");
*/

VOID rest(returnValuePtr)
  DATA_OBJECT_PTR returnValuePtr;
  {
   /*================================*/
   /* Check for exactly one argument. */
   /*================================*/

   if (ArgCountCheck("rest",EXACTLY,1) == -1)
     {
      SetMultifieldErrorValue(returnValuePtr);
      return;
     }

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

   if (ArgTypeCheck("rest",1,MULTIFIELD,returnValuePtr) == 0)
     {
      SetMultifieldErrorValue(returnValuePtr);
      return;
     }
```

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

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

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

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

3.4 USER-DEFINED FUNCTION EXAMPLE

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

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

2) Define the user function in a new file.

```
#include "clips.h"

double TripleNumber()
  {
   return(3.0 * RtnDouble(1));
  }
```

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

```
#include "clips.h"

VOID TripleNumber(returnValuePtr)
  DATA_OBJECT_PTR returnValuePtr;
  {
   VOID        *value;
   long         longValue;
   double       doubleValue;
```

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

   if (ArgCountCheck("triple",EXACTLY,1) == -1)
     {
      SetpType(returnValuePtr,INTEGER);
      SetpValue(returnValuePtr,AddLong(0L));
      return;
     }

   if (! ArgTypeCheck("triple",1,INTEGER_OR_FLOAT,returnValuePtr))
     {
      SetpType(returnValuePtr,INTEGER);
      SetpValue(returnValuePtr,AddLong(0L));
      return;
     }

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

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

   return;
  }
```

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

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

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

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

4) Modify the CLIPS **main.c** file to include the new UserFunctions definition.

```
UserFunctions()
  {
   extern VOID TripleNumber();

   DefineFunction2("triple",'u',PTIF TripleNumber, "TripleNumber",
                   "11n");
  }
```

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

6) Link all object code files.

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

Section 4 - Embedding CLIPS

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

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

(These statements may have to be tailored so the compiler on the user's system can find the CLIPS include file.) The user's main program must initialize CLIPS by calling the function **InitializeCLIPS** at some time prior to loading constructs. **UserFunctions** also must be defined, regardless of whether CLIPS calls any external functions. Compile and link all of the user's code with all CLIPS files *except* the object version of **main.c**. When running CLIPS as an embedded program, many of the capabilities available in the interactive interface (in addition to others) are available through function calls. The functions are documented in the following sections. Prototypes for these functions can be included by using the **clips.h** header file.

4.1 ENVIRONMENT FUNCTIONS

The following function calls control the CLIPS environment:

4.1.1 AddClearFunction

```
int   AddClearFunction(clearItemName,clearFunction,priority);
char *clearItemName;
VOID (*clearFunction)();
int   priority;

VOID clearFunction();
```

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

Arguments:

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

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

4.1.2 AddPeriodicFunction

```
int   AddPeriodicFunction(periodicItemName,periodicFunction,
                          priority);
char *periodicItemName;
VOID (*periodicFunction)();
int   priority;

VOID periodicFunction();
```

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

Arguments:

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

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

4.1.3 AddResetFunction

```
int   AddResetFunction(resetItemName,resetFunction,priority);
char *resetItemName;
VOID (*resetFunction)();
int   priority;

VOID resetFunction();
```

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

Arguments:

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

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

4.1.4 Bload

```
int   Bload(fileName);
char *fileName;
```

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

Arguments: A string representing the name of the file.

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

4.1.5 Bsave

```
int   Bsave(fileName);
char *fileName;
```

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

Arguments: A string representing the name of the file.

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

4.1.6 Clear

```
VOID Clear();
```

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

Arguments: None.

Returns: No meaningful return value.

4.1.7 CLIPSFunctionCall

```
VOID CLIPSFunctionCall(functionName,arguments,&result);
char *functionName,*arguments;
DATA_OBJECT result;
```

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

Arguments:
1) The name of the system function, deffunction or generic function to be called.
2) A string containing any *constant* arguments separated by blanks (this argument can be NULL).
3) Caller's buffer for storing the result of the function call. See sections 3.2.3 and 3.2.4 for information on getting the value stored in a DATA_OBJECT.

Returns: No meaningful return value.

Example

```
DATA_OBJECT rtn;

CLIPSFunctionCall("+","1 2",&rtn);
```

4.1.8 GetAutoFloatDividend

```
int   GetAutoFloatDividend();
```

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

Arguments: None.

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

4.1.9 GetDynamicConstraintChecking

```
int   GetDynamicConstraintChecking();
```

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

Arguments: None.

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

4.1.10 GetSequenceOperatorRecognition

```
int   GetSequenceOperatorRecognition();
```

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

Arguments: None.

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

4.1.11 GetStaticConstraintChecking

```
int   GetStaticConstraintChecking();
```

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

Arguments: None.

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

4.1.12 InitializeCLIPS

```
VOID InitializeCLIPS();
```

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

Arguments: None.

Returns: No meaningful return value.

4.1.13 Load

```
int   Load(fileName);
char *fileName;
```

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

Arguments: A string representing the name of the file.

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

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

4.1.14 RemoveClearFunction

```
int   RemoveClearFunction(clearItemName);
char *clearItemName;
```

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

Arguments: The name associated with the user-defined clear function. This is the same name that was used when the clear function was added with the function **AddClearFunction**.

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

4.1.15 RemovePeriodicFunction

```
int   RemovePeriodicFunction(periodicItemName);
char *periodicItemName;
```

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

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

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

4.1.16 RemoveResetFunction

```
int   RemoveResetFunction(resetItemName);
char *resetItemName;
```

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

Arguments: The name associated with the user-defined reset function. This is the same name that was used when the reset function was added with the function **AddResetFunction**.

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

4.1.17 Reset

```
VOID Reset();
```

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

Arguments: None.

Returns: No meaningful return value.

4.1.18 Save

```
int   Save(fileName);
char *fileName;
```

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

Arguments: A string representing the name of the file.

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

4.1.19 SetAutoFloatDividend

```
int   SetAutoFloatDividend(value);
int   value;
```

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

Arguments: The new value for the behavior: CLIPS_TRUE (1) to enable the behavior and CLIPS_FALSE (0) to disable it.

Returns: Returns the old value for the behavior.

4.1.20 SetDynamicConstraintChecking

```
int   SetDynamicConstraintChecking(value);
int   value;
```

Purpose: Sets the value of the dynamic constraint checking behavior (the C equivalent of the CLIPS command **set-dynamic-constraint-checking**). When this behavior is disabled (FALSE by default), newly created data objects (such as deftemplate facts and instances) do not have their slot values checked for constraint violations. When this behavior is enabled (TRUE), the slot values are checked for constraint violations. The return value for this function is the old value for the behavior.

Arguments: The new value for the behavior: CLIPS_TRUE (1) to enable the behavior and CLIPS_FALSE (0) to disable it.

Returns: Returns the old value for the behavior.

4.1.21 SetSequenceOperator Recognition

```
int   SetSequenceOperatorRecognition(value);
int   value;
```

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

Arguments: The new value for the behavior: CLIPS_TRUE (1) to enable the behavior and CLIPS_FALSE (0) to disable it.

Returns: Returns the old value for the behavior.

4.1.22 SetStaticConstraintChecking

```
int   SetStaticConstraintChecking(value);
int   value;
```

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

Arguments: The new value for the behavior: CLIPS_TRUE (1) to enable the behavior and CLIPS_FALSE (0) to disable it.

Returns: Returns the old value for the behavior.

4.2 DEBUGGING FUNCTIONS

The following function call controls the CLIPS debugging aids:

4.2.1 DribbleActive

```
int DribbleActive();
```

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

Arguments: None.

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

4.2.2 DribbleOff

```
int DribbleOff();
```

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

Arguments: None.

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

4.2.3 DribbleOn

```
int   DribbleOn(fileName);
char *fileName;
```

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

Arguments: The name of the file in which to store dribble information. Only one dribble file may be opened at a time.

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

4.2.4 GetWatchItem

```
int   GetWatchItem(item);
char *item;
```

Purpose: Returns the current value of a watch item.

Arguments: The item to be activated or deactivated which should be one of the following strings: *facts*, *rules*, *activations*, *focus*, *compilations*,

statistics, *globals*, *instances*, *slots*, *messages*, *message-handlers*, *generic-functions*, *method*, or *deffunctions.*

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

4.2.5 Unwatch

```
int   Unwatch(item);
char *item;
```

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

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

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

4.2.6 Watch

```
int   Watch(item);
char *item;
```

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

Arguments: The item to be activated which should be one of the following strings: facts, rules, activations, focus, compilations, statistics, globals, deffunctions, instances, slots, messages, message-handlers, generic-functions, methods, or all. If all is selected, all possible watch items will be traced.

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

4.3 DEFTEMPLATE FUNCTIONS

The following function calls are used for manipulating deftemplates.

4.3.1 DeftemplateModule

```
char *DeftemplateModule(deftemplatePtr);
VOID *deftemplatePtr;
```

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

Arguments: A generic pointer to a deftemplate.

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

4.3.2 FindDeftemplate

```
VOID *FindDeftemplate(deftemplateName);
char *deftemplateName;
```

Purpose: Returns a generic pointer to a named deftemplate.

Arguments: The name of the deftemplate to be found.

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

4.3.3 GetDeftemplateList

```
VOID GetDeftemplateList(&returnValue,theModule);
 DATA_OBJECT returnValue;
 VOID *theModule;
```

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

Arguments:

1) A pointer to the caller's DATA_OBJECT in which the return value will be stored. The multifield functions described in section 3.2.4 can be used to retrieve the deftemplate names from the list.
2) A generic pointer to the module from which the list will be extracted. A NULL pointer indicates that the list is to be extracted from al l modules.

Returns: No meaningful return value.

4.3.4 GetDeftemplateName

```
char *GetDeftemplateName(deftemplatePtr);
VOID *deftemplatePtr;
```

Purpose: Returns the name of a deftemplate.

Arguments: A generic pointer to a deftemplate data structure.

Returns: A string containing the name of the deftemplate.

4.3.5 GetDeftemplatePPForm

```
char *GetDeftemplatePPForm(deftemplatePtr);
VOID *deftemplatePtr;
```

Purpose: Returns the pretty print representation of a deftemplate.

Arguments: A generic pointer to a deftemplate data structure.

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

4.3.6 GetDeftemplateWatch

```
int GetDeftemplateWatch(deftemplatePtr);
VOID *deftemplatePtr;
```

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

Arguments: A generic pointer to a deftemplate data structure.

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

4.3.7 GetNextDeftemplate

```
VOID *GetNextDeftemplate(deftemplatePtr);
VOID *deftemplatePtr;
```

Purpose: Provides access to the list of deftemplates.

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

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

4.3.8 IsDeftemplateDeletable

```
int   IsDeftemplateDeletable(deftemplatePtr);
VOID *deftemplatePtr;
```

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

Arguments: A generic pointer to a deftemplate data structure.

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

4.3.9 ListDeftemplates

```
VOID ListDeftemplates(logicalName,theModule);
char *logicalName;
VOID *theModule;
```

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

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

Returns: No meaningful return value.

4.3.10 SetDeftemplateWatch

```
VOID SetDeftemplateWatch(newState,deftemplatePtr);
int    newState;
 VOID *deftemplatePtr;
```

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

Arguments: The new facts watch state and a generic pointer to a deftemplate data structure.

4.3.11 Undeftemplate

```
int   Undeftemplate(deftemplatePtr);
VOID *deftemplatePtr;
```

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

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

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

4.4 FACT FUNCTIONS

The following function calls manipulate and display information about facts.

4.4.1 Assert

```
VOID *Assert(factPtr);
VOID *factPtr;
```

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

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

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

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

4.4.2 AssertString

```
VOID *AssertString(string);
char *string;
```

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

Arguments: One argument; a pointer to a string containing a list of primitive data types (symbols, strings, integers, floats, and/or instance names).

Returns: A generic pointer to a fact structure.

Examples

If the following deftemplate has been processed by CLIPS,

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

then the following fact

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

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

```
VOID AddExampleFact1()
  {
   AssertString("(example (x 3) (y red) (z 1.5 b))");
  }
```

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

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

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

4.4.3 AssignFactSlotDefaults

```
int AssignFactSlotDefaults(theFact);
VOID *theFact;
```

Purpose: Assigns default values to a fact.

Arguments: A generic pointer to a fact data structure.

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

4.4.4 CreateFact

```
VOID *CreateFact(deftemplatePtr);
VOID *deftemplatePtr;
```

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

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

Returns: A generic pointer to a fact data structure.

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

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

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

these data types). For most situations in which a fact needs to be asserted, however, the **AssertString** function should be preferred (it is slighter slower than using the **CreateFact** and **Assert** functions, but it is much easier to use and less prone to being used incorrectly).

Examples

If the following deftemplate has been processed by CLIPS,

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

then the following fact

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

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

```
VOID AddExampleFact2()
  {
   VOID *newFact;
   VOID *templatePtr;
   VOID *theMultifield;
   DATA_OBJECT theValue;

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

   templatePtr = FindDeftemplate("example");
   newFact = CreateFact(templatePtr);
   if (newFact == NULL) return;

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

   theValue.type = INTEGER;
   theValue.value = AddLong(3);
   PutFactSlot(newFact,"x",&theValue);

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

   theValue.type = SYMBOL;
   theValue.value = AddSymbol("red");
   PutFactSlot(newFact,"y",&theValue);
```

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

   theMultifield = CreateMultifield(2);
   SetMFType(theMultifield,1,FLOAT);
   SetMFValue(theMultifield,1,AddDouble(1.5));
   SetMFType(theMultifield,2,SYMBOL);
   SetMFValue(theMultifield,2,AddSymbol("b"));
   SetDOBegin(theValue,1);
   SetDOEnd(theValue,2);

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

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

   AssignFactSlotDefaults(newFact);

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

   Assert(newFact);
  }
```

4.4.5 DecrementFactCount

```
VOID DecrementFactCount(factPtr);
VOID *factPtr;
```

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

Arguments: A generic pointer to a fact.

Returns: No meaningful return value.

4.4.6 FactIndex

```
long int FactIndex(factPtr);
VOID *factPtr;
```

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

Arguments: A generic pointer to a fact data structure.

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

4.4.7 Facts

```
VOID Facts(logicalName,theModule,start,end,max);
 char *logicalName;
 VOID *theModule;
 long start, end, max;
```

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

Arguments:

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

Returns: No meaningful return value.

4.4.8 GetFactDuplication

```
int   GetFactDuplication();
```

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

Arguments: None.

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

4.4.9 GetFactListChanged

```
int GetFactListChanged();
```

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

Arguments: None.

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

4.4.10 GetFactPPForm

```
VOID  GetFactPPForm(buffer,bufferLength,factPtr);
char *buffer;
int   bufferLength;
VOID *factPtr;
```

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

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

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

4.4.11 GetFactSlot

```
int GetFactSlot(factPtr,slotName,&theValue);
VOID *factPtr;
char *slotName;
DATA_OBJECT theValue;
```

Purpose: Retrieves a slot value from a fact.

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

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

4.4.12 GetNextFact

```
VOID *GetNextFact(factPtr);
VOID *factPtr;
```

Purpose: Provides access to the fact-list.

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

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

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

WARNING: Do not call this function with a pointer to a fact that has been retracted. If the return value from **GetNextFact** is stored as part of a persistent data structure or in a static data area, then the function

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

4.4.13 IncrementFactCount

```
VOID IncrementFactCount(factPtr);
VOID *factPtr;
```

Purpose: This function should be called for each external copy of pointer to a fact to let CLIPS know that such an outstanding external reference exists. As long as an fact's count is greater than zero, CLIPS will not release its memory because there may be outstanding pointers to the fact. However, the fact can still be *functionally* retracted, i.e. the fact will *appear* to no longer be in the fact-list. The fact address always can be safely *examined* using the fact access functions as long as the count for the fact is greater than zero. Retracting an already retracted fact will have no effect, however, the function **AddFact** should not be called twice for the same pointer created using **CreateFact**. Note that this function only needs to be called if you are storing pointers to facts that may later be referenced by external code after the fact has been retracted.

Arguments: A generic pointer to a fact.

Returns: No meaningful return value.

4.4.14 LoadFacts

```
int   LoadFacts(fileName);
char *fileName;
```

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

Arguments: A string representing the name of the file.

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

4.4.15 PutFactSlot

```
int PutFactSlot(factPtr,slotName,&theValue);
VOID *factPtr;
char *slotName;
DATA_OBJECT theValue;
```

Purpose: Sets the slot value of a fact.

Arguments:
1) A generic pointer to a fact data structure.
2) The name of the slot to be set (NULL should be used for the implied multifield slot of an implied deftemplate).
3) A pointer to a DATA_OBJECT that contains the slot's new value. A multifield or implied multifield slot should only be passed a multifield value. A single field slot should only be passed a single field value. See sections 3.3.3 and 3.3.4 for information on setting the value stored in a DATA_OBJECT.

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

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

4.4.16 Retract

```
int Retract(factPtr);
VOID *factPtr;
```

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

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

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

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

4.4.17 SaveFacts

```
int   SaveFacts(fileName,saveScope,NULL);
char *fileName;
 int   saveScope;
```

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

Arguments: A string representing the name of the file and an integer constant representing the scope for the facts being saved which should be either LOCAL_SAVE or VISIBLE_SAVE. The third argument is used internally by the CLIPS save-facts command and should be set to NULL when called from user code.

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

4.4.18 SetFactDuplication

```
int   SetFactDuplication(value);
int   value;
```

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

Arguments: The new value for the behavior: CLIPS_TRUE (1) to enable the behavior and CLIPS_FALSE (0) to disable it.

Returns: Returns the old value for the behavior.

4.4.19 SetFactListChanged

```
VOID SetFactListChanged(changedFlag);
int changedFlag;
```

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

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

Returns: Nothing useful.

4.5 DEFFACTS FUNCTIONS

The following function calls are used for manipulating deffacts.

4.5.1 DeffactsModule

```
char *DeffactsModule(theDeffacts);
VOID *theDeffacts;
```

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

Arguments: A generic pointer to a deffacts.

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

4.5.2 FindDeffacts

```
VOID *FindDeffacts(deffactsName);
char *deffactsName;
```

Purpose: Returns a generic pointer to a named deffacts.

Arguments: The name of the deffacts to be found.

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

4.5.3 GetDeffactsList

```
VOID GetDeffactsList(&returnValue,theModule);
 DATA_OBJECT returnValue;
 VOID *theModule;
```

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

Arguments:
1) A pointer to the caller's DATA_OBJECT in which the return value will be stored. The multifield functions described in section 3.2.4 can be used to retrieve the deffacts names from the list.
2) A generic pointer to the module from which the list will be extracted. A NULL pointer indicates that the list is to be extracted from al l modules.

Returns: No meaningful return value.

4.5.4 GetDeffactsName

```
char *GetDeffactsName(deffactsPtr);
VOID *deffactsPtr;
```

Purpose: Returns the name of a deffacts.

Arguments: A generic pointer to a deffacts data structure.

Returns: A string containing the name of the deffacts.

4.5.5 GetDeffactsPPForm

```
char *GetDeffactsPPForm(deffactsPtr);
VOID *deffactsPtr;
```

Purpose: Returns the pretty print representation of a deffacts.

Arguments: A generic pointer to a deffacts data structure.

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

4.5.6 GetNextDeffacts

```
VOID *GetNextDeffacts(deffactsPtr);
VOID *deffactsPtr;
```

Purpose: Provides access to the list of deffacts.

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

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

4.5.7 IsDeffactsDeletable

```
int   IsDeffactsDeletable(deffactsPtr);
VOID *deffactsPtr;
```

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

Arguments: A generic pointer to a deffacts data structure.

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

4.5.8 ListDeffacts

```
VOID ListDeffacts(logicalName,theModule);
char *logicalName;
VOID *theModule;
```

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

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

Returns: No meaningful return value.

4.5.9 Undeffacts

```
int   Undeffacts(deffactsPtr);
VOID *deffactsPtr;
```

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

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

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

4.6 DEFRULE FUNCTIONS

The following function calls are used for manipulating defrules.

4.6.1 DefruleHasBreakpoint

```
int  DefruleHasBreakpoint(defrulePtr);
VOID *defrulePtr;
```

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

Arguments: A generic pointer to a defrule data structure.

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

4.6.2 DefruleModule

```
char *DefruleModule(theDefrule);
VOID *theDefrule;
```

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

Arguments: A generic pointer to a defrule.

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

4.6.3 FindDefrule

```
VOID *FindDefrule(defruleName);
char *defruleName;
```

Purpose: Returns a generic pointer to a named defrule.

Arguments: The name of the defrule to be found.

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

4.6.4 GetDefruleList

```
VOID GetDefruleList(&returnValue,theModule);
DATA_OBJECT returnValue;
VOID *theModule;
```

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

Arguments:
1) A pointer to the caller's DATA_OBJECT in which the return value will be stored. The multifield functions described in section 3.2.4 can be used to retrieve the defrule names from the list.
2) A generic pointer to the module from which the list will be extracted. A NULL pointer indicates that the list is to be extracted from al l modules.

Returns: No meaningful return value.

4.6.5 GetDefruleName

```
char *GetDefruleName(defrulePtr);
VOID *defrulePtr;
```

Purpose: Returns the name of a defrule.

Arguments: A generic pointer to a defrule data structure.

Returns: A string containing the name of the defrule.

4.6.6 GetDefrulePPForm

```
char *GetDefrulePPForm(defrulePtr);
VOID *defrulePtr;
```

Purpose: Returns the pretty print representation of a defrule.

Arguments: A generic pointer to a defrule data structure.

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

4.6.7 GetDefruleWatchActivations

```
int GetDefruleWatchActivations(defrulePtr);
VOID *defrulePtr;
```

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

Arguments: A generic pointer to a defrule data structure.

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

4.6.8 GetDefruleWatchFirings

```
int GetDefruleWatchFirings(defrulePtr);
VOID *defrulePtr;
```

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

Arguments: A generic pointer to a defrule data structure.

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

4.6.9 GetIncrementalReset

```
int   GetIncrementalReset();
```

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

Arguments: None.

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

4.6.10 GetNextDefrule

```
VOID *GetNextDefrule(defrulePtr);
VOID *defrulePtr;
```

Purpose: Provides access to the list of defrules.

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

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

4.6.11 IsDefruleDeletable

```
int   IsDefruleDeletable(defrulePtr);
VOID *defrulePtr;
```

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

Arguments: A generic pointer to a defrule data structure.

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

4.6.12 ListDefrules

```
VOID ListDefrules(logicalName,theModule);
char *logicalName;
VOID *theModule;
```

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

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

Returns: No meaningful return value.

4.6.13 Matches

```
int Matches(defrulePtr);
VOID *defrulePtr;
```

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

Arguments: A generic pointer to a defrule data structure.

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

4.6.14 Refresh

```
int   Refresh(defrulePtr);
VOID *defrulePtr;
```

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

Arguments: A generic pointer to a defrule data structure.

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

4.6.15 RemoveBreak

```
int RemoveBreak(defrulePtr);
VOID *defrulePtr;
```

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

Arguments: A generic pointer to a defrule data structure.

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

4.6.16 SetBreak

```
VOID SetBreak(defrulePtr);
VOID *defrulePtr;
```

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

Arguments: A generic pointer to a defrule data structure.

Returns: No meaningful return value.

4.6.17 SetDefruleWatchActivations

```
VOID SetDefruleWatchActivations(newState,defrulePtr);
int    newState;
VOID *defrulePtr;
```

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

Arguments: The new activations watch state and a generic pointer to a defrule data structure.

4.6.18 SetDefruleWatchFirings

```
VOID SetDefruleWatchFirings(newState,defrulePtr);
int    newState;
VOID *defrulePtr;
```

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

Arguments: The new rule firing watch state and a generic pointer to a defrule data structure.

4.6.19 SetIncrementalReset

```
int   SetIncrementalReset(value);
int   value;
```

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

Arguments: The new value for the behavior: CLIPS_TRUE (1) to enable the behavior and CLIPS_FALSE (0) to disable it.

Returns: Returns the old value for the behavior.

4.6.20 ShowBreaks

```
VOID ShowBreaks(logicalName,theModule);
char *logicalName;
VOID *theModule;
```

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

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

Returns: No meaningful return value.

4.6.21 Undefrule

```
int   Undefrule(defrulePtr);
VOID *defrulePtr;
```

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

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

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

4.7 AGENDA FUNCTIONS

The following function calls are used for manipulating the agenda.

4.7.1 AddRunFunction

```
int    AddRunFunction(runItemName,runFunction,priority);
char  *runItemName;
VOID (*runFunction)();
int    priority;

VOID runFunction();
```

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

Arguments:

1) The name associated with the user-defined run function. This name is used by the function **RemoveRunFunction**.
2) A pointer to the user-defined function which is to be called after every rule firing.
3) The priority of the run item which determines the order in which run items are called (higher priority items are called first). The values -2000 to 2000 are reserved for CLIPS system defined run items and should not be used for user defined run items.

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

Example

This following function checks to see if a key on the keyboard has been hit. If a key has been hit, then the fact (stop-processing) is asserted into the fact-list.

```
VOID CheckKB()
  {
   if (CheckKeyboardStatus() == KB_HIT)
     { AssertString("stop-processing"); }
  }
```

This function can now be added to the list of functions called after every rule firing by making the following function call.

```
AddRunFunction("check-kb",checkKB,3000);
```

4.7.2 Agenda

```
VOID Agenda(logicalName,theModule)
 char *logicalName;
 VOID *theModule;
```

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

Arguments:

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

Returns: No meaningful return value.

4.7.3 ClearFocusStack

```
VOID ClearFocusStack();
```

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

Arguments: None.

Returns: No meaningful return value.

4.7.4 DeleteActivation

```
int   DeleteActivation(activationPtr);
VOID *activationPtr;
```

Purpose: Removes an activation from the agenda.

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

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

4.7.5 Focus

```
VOID Focus(defmodulePtr);
VOID *defmodulePtr;
```

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

Arguments: A generic pointer to a defmodule data structure.

Returns: No meaningful value.

4.7.6 GetActivationName

```
char *GetActivationName(activationPtr);
VOID *activationPtr;
```

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

Arguments: A generic pointer to an activation data structure.

Returns: A string containing a defrule name.

4.7.7 GetActivationPPForm

```
VOID  GetActivationPPForm(buffer,bufferLength,activationPtr);
char *buffer;
int   bufferLength;
VOID *activationPtr;
```

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

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

4.7.8 GetActivationSalience

```
int   GetActivationSalience(activationPtr);
VOID *activationPtr;
```

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

Arguments: A generic pointer to an activation data structure.

Returns: The integer salience value of an activation.

4.7.9 GetAgendaChanged

```
int GetAgendaChanged();
```

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

Arguments: None.

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

4.7.10 GetFocus

```
VOID *GetFocus();
```

Purpose: Returns the module associated with the current focus (the C equivalent of the CLIPS **get-focus** function).

Arguments: None.

Returns: A generic pointer to a defmodule data structure (or NULL if the focus stack is empty).

4.7.11 GetFocusStack

```
VOID GetFocusStack(&returnValue);
DATA_OBJECT returnValue;
```

Purpose: Returns the module names in the focus stack as a multifield value in the returnValue DATA_OBJECT (the C equivalent of the CLIPS **get-focus-stack** function).

Arguments: A pointer to the caller's DATA_OBJECT in which the return value will be stored. The multifield functions described in section 3.2.4 can be used to retrieve the defrule names from the list.

4.7.12 GetNextActivation

```
VOID *GetNextActivation(activationPtr);
VOID *activationPtr;
```

Purpose: Provides access to the list of activations on the agenda.

Arguments: A generic pointer to an activation data structure (or NULL to get the first activation on the agenda).

Returns: A generic pointer to the first activation on the agenda if *activationPtr* is NULL, otherwise a generic pointer to the activation immediately following *activationPtr* on the agenda. If *activationPtr* is the last activation on the agenda, then NULL is returned.

4.7.13 GetSalienceEvaluation

```
int   GetSalienceEvaluation();
```

Purpose: Returns the current salience evaluation behavior (the C equivalent of the CLIPS **get-salience-evaluation** command).

Arguments: None.

Returns: An integer (see SetSalienceEvaluation for the list of defined constants).

4.7.14 GetStrategy

```
int   GetStrategy();
```

Purpose: Returns the current conflict resolution strategy (the C equivalent of the CLIPS **get-strategy** command).

Arguments: None.

Returns: An integer (see SetStrategy for the list of defined strategy constants).

4.7.15 ListFocusStack

```
VOID ListFocusStack(logicalName);
char *logicalName;
```

Purpose: Prints the current focus stack (the C equivalent of the CLIPS **list-focus-stack** command).

Arguments: The logical name to which the listing output is sent.

Returns: No meaningful return value.

4.7.16 PopFocus

```
VOID *PopFocus();
```

Purpose: Removes the current focus from the focus stack and returns the module associated with that focus (the C equivalent of the CLIPS **pop-focus** function).

Arguments: None.

Returns: A generic pointer to a defmodule data structure.

4.7.17 RefreshAgenda

```
VOID RefreshAgenda(theModule);
VOID *theModule;
```

Purpose: Recomputes the salience values for all activations on the agenda and then reorders the agenda (the C equivalent of the CLIPS **refresh-agenda** command).

Arguments: A generic pointer to the module containing the agenda to be refreshed. A NULL pointer indicates that the agendas of all modules should be refreshed.

Returns: No meaningful return value.run

4.7.18 RemoveRunFunction

```
int   RemoveRunFunction(runItemName);
char *runItemName;
```

Purpose: Removes a named function from the list of functions to be called after every rule firing.

Arguments: The name associated with the user-defined run function. This is the same name that was used when the run function was added with the function **AddRunFunction**.

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

4.7.19 ReorderAgenda

```
VOID ReorderAgenda(theModule);
VOID *theModule;
```

Purpose: Reorders the agenda based on the current conflict resolution strategy and current activation saliences.

Arguments: A generic pointer to the module containing the agenda to be reordered. A NULL pointer indicates that the agendas of all modules should be reordered.

Returns: No meaningful return value.

4.7.20 Run

```
long int Run(runLimit);
long int runLimit;
```

Purpose: Allows rules to execute (the C equivalent of the CLIPS **run** command).

Arguments: An integer which defines how many rules should fire before returning. If runLimit is a negative integer, rules will fire until the agenda is empty.

Returns: Returns an integer value; the number of rules that were fired.

4.7.21 SetActivationSalience

```
int   SetActivationSalience(activationPtr,newSalience);
VOID *activationPtr;
int   newSalience;
```

Purpose: Sets the salience value of an activation. The salience value of the defrule which generated the activation is unchanged.

Arguments:
1) A generic pointer to an activation data structure.
2) The new salience value (which is not restricted to the -10000 to +10000 range).

Returns: The old salience value of the activation.

Other: The function **ReorderAgenda** should be called after salience values have been changed to update the agenda.

4.7.22 SetAgendaChanged

```
VOID SetAgendaChanged(changedFlag);
int changedFlag;
```

Purpose: Sets the internal boolean flag which indicates when changes to the agenda of rule activations have occurred. This function is normally used to reset the flag to zero after GetAgendaChanged() returns non-zero.

Arguments: An integer indicating whether changes in the agenda have occurred (non-zero) or not (0).

Returns: Nothing useful.

4.7.23 SetSalienceEvaluation

```
int   SetSalienceEvaluation(value);
int   value;
```

Purpose: Sets the salience evaluation behavior (the C equivalent of the CLIPS **set-salience-evaluation** command).

Arguments: The new value for the behavior – one of the following defined integer constants:

WHEN_DEFINED
WHEN_ACTIVATED
EVERY_CYCLE

Returns: Returns the old value for the behavior.

4.7.24 SetStrategy

```
int   SetStrategy(value);
int   value;
```

Purpose: Sets the conflict resolution strategy (the C equivalent of the CLIPS **set-strategy** command).

Arguments: The new value for the behavior – one of the following defined integer constants:

DEPTH_STRATEGY
BREADTH_STRATEGY
LEX_STRATEGY
MEA_STRATEGY
COMPLEXITY_STRATEGY

SIMPLICITY_STRATEGY
RANDOM_STRATEGY

Returns: Returns the old value for the strategy.

4.8 DEFGLOBAL FUNCTIONS

The following function calls are used for manipulating defglobals.

4.8.1 DefglobalModule

```
char *DefglobalModule(theDefglobal);
VOID *theDefglobal;
```

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

Arguments: A generic pointer to a defglobal.

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

4.8.2 FindDefglobal

```
VOID *FindDefglobal(globalName);
char *globalName;
```

Purpose: Returns a generic pointer to a named defglobal.

Arguments: The name of the defglobal to be found (e.g. *x* for ?*x*).

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

4.8.3 GetDefglobalList

```
VOID GetDefglobalList(&returnValue,theModule);
 DATA_OBJECT returnValue;
 VOID *theModule;
```

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

Arguments:

1) A pointer to the caller's DATA_OBJECT in which the return value will be stored. The multifield functions described in section 3.2.4 can be used to retrieve the defglobal names from the list.
2) A generic pointer to the module from which the list will be extracted. A NULL pointer indicates that the list is to be extracted from al l modules.

Returns: No meaningful return value.

4.8.4 GetDefglobalName

```
char *GetDefglobalName(defglobalPtr);
VOID *defglobalPtr;
```

Purpose: Returns the name of a defglobal.

Arguments: A generic pointer to a defglobal data structure.

Returns: A string containing the name of the defglobal (e.g. *x* for ?*x*).

4.8.5 GetDefglobalPPForm

```
char *GetDefglobalPPForm(defglobalPtr);
VOID *defglobalPtr;
```

Purpose: Returns the pretty print representation of a defglobal.

Arguments: A generic pointer to a defglobal data structure.

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

4.8.6 GetDefglobalValue

```
int   GetDefglobalValue(globalName,&vPtr);
char *globalName;
DATA_OBJECT vPtr;
```

Purpose: Returns the value of a defglobal.

Arguments:

1) The name of the global variable to be retrieved (e.g. *y* for ?*y*).

2) A pointer to a DATA_OBJECT in which the value is stored (see sections 3.2.3 and 3.3.4 for details on this data structure).

Returns: An integer; zero (0) if the defglobal was not found, otherwise a one (1). The DATA_OBJECT vPtr is assigned the current value of the defglobal.

4.8.7 GetDefglobalValueForm

```
VOID  GetDefglobalValueForm(buffer,bufferLength,defglobalPtr);
char *buffer;
int   bufferLength;
VOID *defglobalPtr;
```

Purpose: Returns a printed representation of a defglobal and its current value in the caller's buffer. For example,

```
?*x* = 5
```

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

4.8.8 GetDefglobalWatch

```
int GetDefglobalWatch(defglobalPtr);
VOID *defglobalPtr;
```

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

Arguments: A generic pointer to a defglobal data structure.

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

4.8.9 GetGlobalsChanged

```
int GetGlobalsChanged();
```

Purpose: Determines if any changes to global variables have occurred. If this function returns a non-zero integer, it is the user's responsibility to call SetGlobalsChanged(0) to reset the internal flag. Otherwise, this

function will continue to return non-zero even when no changes have occurred. This function is primarily used to determine when to update a display tracking global variables.

Arguments: None.

Returns: 0 if no changes to global variables have occurred, non-zero otherwise.

4.8.10 GetNextDefglobal

```
VOID *GetNextDefglobal(defglobalPtr);
VOID *defglobalPtr;
```

Purpose: Provides access to the list of defglobals.

Arguments: A generic pointer to a defglobal data structure (or NULL to get the first defglobal).

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

4.8.11 GetResetGlobals

```
int   GetResetGlobals();
```

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

Arguments: None.

Returns: An integer; CLIPS_FALSE (0) if globals are not reset and CLIPS_TRUE (1) if globals are reset.

4.8.12 IsDefglobalDeletable

```
int   IsDefglobalDeletable(defglobalPtr);
VOID *defglobalPtr;
```

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

Arguments: A generic pointer to a defglobal data structure.

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

4.8.13 ListDefglobals

```
VOID ListDefglobals(logicalName,theModule);
char *logicalName;
VOID *theModule;
```

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

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

Returns: No meaningful return value.

4.8.14 SetDefglobalValue

```
int   SetDefglobalValue(globalName,&vPtr);
char *globalName;
DATA_OBJECT vPtr;
```

Purpose: Sets the value of a defglobal.

Arguments:
1) The name of the global variable to be set (e.g. *y* for ?*y*).
2) A pointer to a DATA_OBJECT in which the new value is contained (see sections 3.2.3 and 3.3.4 for details on this data structure).

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

4.8.15 SetDefglobalWatch

```
VOID SetDefglobalWatch(newState,defglobalPtr);
int    newState;
 VOID *defglobalPtr;
```

Purpose: Sets the globals watch item for a specific defglobal.

Arguments: The new globals watch state and a generic pointer to a defglobal data structure.

4.8.16 SetGlobalsChanged

```
VOID SetGlobalsChanged(changedFlag);
int changedFlag;
```

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

Arguments: An integer indicating whether changes in global variables have occurred (non-zero) or not (0).

Returns: Nothing useful.

4.8.17 SetResetGlobals

```
int   SetResetGlobals(value);
int   value;
```

Purpose: Sets the reset-globals behavior (the C equivalent of the CLIPS **set-reset-globals** command). When this behavior is enabled (by default) global variables are reset to their original values when the **reset** command is performed.

Arguments: The new value for the behavior: CLIPS_TRUE (1) to enable the behavior and CLIPS_FALSE (0) to disable it.

Returns: Returns the old value for the behavior.

4.8.18 ShowDefglobals

```
VOID ShowDefglobals(logicalName,theModule);
char *logicalName;
VOID *theModule;
```

Purpose: Prints the list of defglobals and their current values (the C equivalent of the CLIPS **show-defglobals** command).

Arguments: 1) The logical name to which the listing output is sent.

2) A generic pointer to the module containing the defglobals to be displayed. A NULL pointer indicates that defglobals in all modules should be displayed.

Returns: No meaningful return value.

4.8.19 Undefglobal

```
int   Undefglobal(defglobalPtr);
VOID *defglobalPtr;
```

Purpose: Removes a defglobal from CLIPS (the C equivalent of the CLIPS **undefglobal** command).

Arguments: A generic pointer to a defglobal data structure. If the NULL pointer is used, then all defglobals will be deleted.

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

4.9 DEFFUNCTION FUNCTIONS

The following function calls are used for manipulating deffunctions.

4.9.1 DeffunctionModule

```
char *DeffunctionModule(theDeffunction);
VOID *theDeffunction;
```

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

Arguments: A generic pointer to a deffunction.

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

4.9.2 FindDeffunction

```
VOID *FindDeffunction(deffunctionName);
char *deffunctionName;
```

Purpose: Returns a generic pointer to a named deffunction.

Arguments: The name of the deffunction to be found.

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

4.9.3 GetDeffunctionList

```
VOID GetDeffunctionList(&returnValue,theModule);
DATA_OBJECT returnValue;
VOID *theModule;
```

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

Arguments:
1) A pointer to the caller's DATA_OBJECT in which the return value will be stored. The multifield functions described in section 3.2.4 can be used to retrieve the deffunction names from the list.
2) A generic pointer to the module from which the list will be extracted. A NULL pointer indicates that the list is to be extracted from al l modules.

Returns: No meaningful return value.

4.9.4 GetDeffunctionName

```
char *GetDeffunctionName(deffunctionPtr);
VOID *deffunctionPtr;
```

Purpose: Returns the name of a deffunction.

Arguments: A generic pointer to a deffunction data structure.

Returns: A string containing the name of the deffunction.

4.9.5 GetDeffunctionPPForm

```
char *GetDeffunctionPPForm(deffunctionPtr);
VOID *deffunctionPtr;
```

Purpose: Returns the pretty print representation of a deffunction.

Arguments: A generic pointer to a deffunction data structure.

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

4.9.6 GetDeffunctionWatch

```
int GetDeffunctionWatch(deffunctionPtr);
VOID *deffunctionPtr;
```

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

Arguments: A generic pointer to a deffunction data structure.

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

4.9.7 GetNextDeffunction

```
VOID *GetNextDeffunction(deffunctionPtr);
VOID *deffunctionPtr;
```

Purpose: Provides access to the list of deffunctions.

Arguments: A generic pointer to a deffunction data structure (or NULL to get the first deffunction).

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

4.9.8 IsDeffunctionDeletable

```
int   IsDeffunctionDeletable(deffunctionPtr);
VOID *deffunctionPtr;
```

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

Arguments: A generic pointer to a deffunction data structure.

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

4.9.9 ListDeffunctions

```
VOID ListDeffunctions(logicalName,theModule);
char *logicalName;
VOID *theModule;
```

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

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

Returns: No meaningful return value.

4.9.10 SetDeffunctionWatch

```
VOID SetDeffunctionWatch(newState,deffunctionPtr);
int    newState;
 VOID *deffunctionPtr;
```

Purpose: Sets the deffunctions watch item for a specific deffunction.

Arguments: The new deffunctions watch state and a generic pointer to a deffunction data structure.

4.9.11 Undeffunction

```
int   Undeffunction(deffunctionPtr);
VOID *deffunctionPtr;
```

Purpose: Removes a deffunction from CLIPS (the C equivalent of the CLIPS **undeffunction** command).

Arguments: A generic pointer to the deffunction (NULL means to delete all deffunctions).

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

4.10 DEFGENERIC FUNCTIONS

The following function calls are used for manipulating generic functions.

4.10.1 DefgenericModule

```
char *DefgenericModule(theDefgeneric);
VOID *theDefgeneric;
```

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

Arguments: A generic pointer to a defgeneric.

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

4.10.2 FindDefgeneric

```
VOID *FindDefgeneric(defgenericName);
char *defgenericName;
```

Purpose: Returns a generic pointer to a named generic function.

Arguments: The name of the generic to be found.

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

4.10.3 GetDefgenericList

```
VOID GetDefgenericList(&returnValue,theModule);
DATA_OBJECT returnValue;
VOID *theModule;
```

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

Arguments:

1) A pointer to the caller's DATA_OBJECT in which the return value will be stored. The multifield functions described in section 3.2.4 can be used to retrieve the defgeneric names from the list.

2) A generic pointer to the module from which the list will be extracted. A NULL pointer indicates that the list is to be extracted from al l modules.

Returns: No meaningful return value.

4.10.4 GetDefgenericName

```
char *GetDefgenericName(defgenericPtr);
VOID *defgenericPtr;
```

Purpose: Returns the name of a generic function.

Arguments: A generic pointer to a defgeneric data structure.

Returns: A string containing the name of the generic function.

4.10.5 GetDefgenericPPForm

```
char *GetDefgenericPPForm(defgenericPtr);
VOID *defgenericPtr;
```

Purpose: Returns the pretty print representation of a generic function.

Arguments: A generic pointer to a defgeneric data structure.

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

4.10.6 GetDefgenericWatch

```
int GetDefgenericWatch(defgenericPtr);
VOID *defgenericPtr;
```

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

Arguments: A generic pointer to a defgeneric data structure.

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

4.10.7 GetNextDefgeneric

```
VOID *GetNextDefgeneric(defgenericPtr);
VOID *defgenericPtr;
```

Purpose: Provides access to the list of generic functions.

Arguments: A generic pointer to a defgeneric data structure (or NULL to get the first generic function).

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

4.10.8 IsDefgenericDeletable

```
int   IsDefgenericDeletable(defgenericPtr);
VOID *defgenericPtr;
```

Purpose: Indicates whether or not a particular generic function and all its methods can be deleted.

Arguments: A generic pointer to a defgeneric data structure.

Returns: An integer: zero (0) if the generic function and all its methods cannot be deleted, otherwise a one (1).

4.10.9 ListDefgenerics

```
VOID ListDefgenerics(logicalName,theModule);
char *logicalName;
VOID *theModule;
```

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

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

Returns: No meaningful return value.

4.10.10 SetDefgenericWatch

```
VOID SetDefgenericWatch(newState,defgenericPtr);
int    newState;
VOID *defgenericPtr;
```

Purpose: Sets the defgenerics watch item for a specific defgeneric.

Arguments: The new generic-functions watch state and a generic pointer to a defgeneric data structure.

4.10.11 Undefgeneric

```
int   Undefgeneric(defgenericPtr);
VOID *defgenericPtr;
```

Purpose: Removes a generic function and all its methods from CLIPS (the C equivalent of the CLIPS **undefgeneric** command).

Arguments: A generic pointer to the generic function (NULL means to delete all generic functions).

Returns: An integer: zero (0) if the generic function and all its methods could not be deleted, otherwise a one (1).

4.11 DEFMETHOD FUNCTIONS

The following function calls are used for manipulating generic function methods.

4.11.1 GetDefmethodDescription

```
VOID GetDefmethodDescription(buffer,bufferLength,
                             defgenericPtr,methodIndex);
char *buf;
int bufLength;
VOID *defgenericPtr;
unsigned methodIndex;
```

Purpose: Stores a synopsis of the method parameter restrictions in the caller's buffer.

Arguments:
1) A pointer to the caller's buffer.
2) The maximum number of characters which could be stored in the caller's buffer (not including space for the terminating null character).

3) A generic pointer to a defgeneric data structure.
4) The index of the generic function method.

Returns: No meaningful return value.

4.11.2 GetDefmethodList

```
VOID GetDefmethodList(defgenericPtr,&returnValue);
VOID *defgenericPtr;
DATA_OBJECT returnValue;
```

Purpose: Returns the list of currently defined defmethods for the specified defgeneric. This function is the C equivalent of the CLIPS **get-defmethod-list** command).

Arguments:
1) A generic pointer to the defgeneric (NULL for all defgenerics).
2) A pointer to the DATA_OBJECT in which the list of defmethod constructs is to be stored.

Returns: A multifield value containing the list of defmethods constructs for the specified defgeneric. The multifield functions described in section 3.2.4 can be used to retrieve the defmethod names and indices from the list. Note that the name and index for each defmethod are stored as pairs in the return multifield value.

4.11.3 GetDefmethodPPForm

```
char *GetDefmethodPPForm(defgenericPtr,methodIndex);
VOID *defgenericPtr;
unsigned methodIndex;
```

Purpose: Returns the pretty print representation of a generic function method.

Arguments:
1) A generic pointer to a defgeneric data structure.
2) The index of the generic function method.

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

4.11.4 GetDefmethodWatch

```
int GetDefmethodWatch(defgenericPtr,methodIndex);
VOID *defgenericPtr;
 unsigned methodIndex
```

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

Arguments: A generic pointer to a defgeneric data structure and the index of the generic function method.

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

4.11.5 GetMethodRestrictions

```
VOID GetMethodRestrictions(defgenericPtr,methodIndex,
                           &returnValue);
VOID *defgenericPtr;
unsigned methodIndex;
DATA_OBJECT returnValue;
```

Purpose: Returns the restrictions for the specified method. This function is the C equivalent of the CLIPS **get-method-restrictions** function.

Arguments:
1) A generic pointer to the defgeneric (NULL for all defgenerics).
2) The index of the generic function method.
3) A pointer to the DATA_OBJECT in which the method restrictions are stored.

Returns: A multifield value containing the restrictions for the specified method (the description of the **get-method-restrictions** function in the Basic Programming Guide explains the meaning of the fields in the multifield value). The multifield functions described in section 3.2.4 can be used to retrieve the method restrictions from the list.

4.11.6 GetNextDefmethod

```
unsigned GetNextDefmethod(defgenericPtr,methodIndex);
VOID *defgenericPtr;
unsigned methodIndex;
```

Purpose: Provides access to the list of methods for a particular generic function.

Arguments:

1) A generic pointer to a defgeneric data structure.
2) The index of a generic function method (0 to get the first method of the generic function).

Returns:

The index of the first method in the list of methods for the generic function if *methodIndex* is 0, otherwise the index of the method immediately following *methodIndex* in the list of methods for the generic function. If *methodIndex* is the last method in the list of methods for the generic function, then 0 is returned.

4.11.7 IsDefmethodDeletable

```
int   IsDefmethodDeletable(defgenericPtr,methodIndex);
VOID *defgenericPtr;
unsigned methodIndex;
```

Purpose:

Indicates whether or not a particular generic function method can be deleted.

Arguments:

1) A generic pointer to a defgeneric data structure.
2) The index of the generic function method.

Returns:

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

4.11.8 ListDefmethods

```
VOID ListDefmethods(logicalName,defgenericPtr);
char *logicalName;
VOID *defgenericPtr;
```

Purpose:

Prints the list of methods for a particular generic function (the C equivalent of the CLIPS **list-defmethods** command).

Arguments:

1) The logical name of the output destination to which tosend the method listing
2) A generic pointer to the generic function (NULL to list methods for all generic functions).

Returns:

No meaningful return value.

4.11.9 SetDefmethodWatch

```
VOID SetDefmethodWatch(newState,defgenericPtr,methodIndex);
int    newState;
VOID *defgenericPtr;
unsigned methodIndex
```

Purpose: Sets the methods watch item for a specific defmethod.

Arguments: The new methods watch state, a generic pointer to a defgeneric data structure, and the index of the generic function method.

4.11.10 Undefmethod

```
int   Undefmethod(defgenericPtr,methodIndex);
VOID *defgenericPtr;
unsigned methodIndex;
```

Purpose: Removes a generic function method from CLIPS (the C equivalent of the CLIPS **undefmethod** command).

Arguments:
1) A generic pointer to a defgeneric data structure (NULL to delete all methods for all generic functions).
2) The index of the generic function method (0 to delete all methods of the generic function - must be 0 if *defgenericPtr* is NULL).

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

4.12 DEFCLASS FUNCTIONS

The following function calls are used for manipulating defclasses.

4.12.1 BrowseClasses

```
VOID BrowseClasses(logicalName,defclassPtr);
char *logicalName;
VOID *defclassPtr;
```

Purpose: Prints a “graph” of all classes which inherit from the specified class. This function is the C equivalent of the CLIPS **browse-classes** command.

Arguments: 1) The logical name of the output destination to which to send the browse display.
2) A generic pointer to the class which is to be browsed.

Returns: No meaningful return value.

4.12.2 ClassAbstractP

```
int ClassAbstractP(defclassPtr);
VOID *defclassPtr;
```

Purpose: Determines if a class is concrete or abstract, i.e. if a class can have direct instances or not. This function is the C equivalent of the CLIPS **class-abstractp** command.

Arguments: A generic pointer to the class.

Returns: The integer 1 if the class is abstract, or 0 if the class is concrete.

4.12.3 ClassReactiveP

```
int ClassReactiveP(defclassPtr);
VOID *defclassPtr;
```

Purpose: Determines if a class is reactive or non-reactive, i.e. if objects of the class can match object patterns. This function is the C equivalent of the CLIPS **class-reactivep** command.

Arguments: A generic pointer to the class.

Returns: The integer 1 if the class is reactive, or 0 if the class is non-reactive.

4.12.4 ClassSlots

```
VOID ClassSlots(defclassPtr,&result,inheritFlag);
VOID *defclassPtr;
DATA_OBJECT result;
int inheritFlag;
```

Purpose: Groups the names of slots of a class into a multifield data object. This function is the C equivalent of the CLIPS **class-slots** command.

Arguments: 1) A generic pointer to the class.

2) Pointer to the data object in which to store the multifield. See sections 3.3.3 and 3.3.4 for information on getting the value stored in a DATA_OBJECT.
3) The integer 1 to include inherited slots or 0 to only include explicitly defined slots.

Returns: No meaningful return value.

4.12.5 ClassSubclasses

```
VOID ClassSubclasses(defclassPtr,&result,inheritFlag);
VOID *defclassPtr;
DATA_OBJECT result;
int inheritFlag;
```

Purpose: Groups the names of subclasses of a class into a multifield data object. This function is the C equivalent of the CLIPS **class-subclasses** command.

Arguments: 1) A generic pointer to the class.
2) Pointer to the data object in which to store the multifield. See sections 3.3.3 and 3.3.4 for information on setting the value stored in a DATA_OBJECT.
3) The integer 1 to include inherited subclasses or 0 to only include direct subclasses.

Returns: No meaningful return value.

4.12.6 ClassSuperclasses

```
VOID ClassSuperclasses(defclassPtr,&result,inheritFlag);
VOID *defclassPtr;
DATA_OBJECT result;
int inheritFlag;
```

Purpose: Groups the names of superclasses of a class into a multifield data object. This function is the C equivalent of the CLIPS **class-superclasses** command.

Arguments: 1) A generic pointer to the class.
2) Pointer to the data object in which to store the multifield.
3) The integer 1 to include inherited superclasses or 0 to only include direct superclasses.

Returns: No meaningful return value.

4.12.7 DefclassModule

```
char *DefclassModule(theDefclass);
VOID *theDefclass;
```

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

Arguments: A generic pointer to a defclass.

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

4.12.8 DescribeClass

```
VOID DescribeClass(logicalName,defclassPtr);
char *logicalName;
VOID *defclassPtr;
```

Purpose: Prints a summary of the specified class including: abstract/concrete behavior, slots and facets (direct and inherited) and recognized message-handlers (direct and inherited). This function is the C equivalent of the CLIPS **describe-class** command.

Arguments: 1) The logical name of the output destination to which to send the description.
2) A generic pointer to the class which is to be described.

Returns: No meaningful return value.

4.12.9 FindDefclass

```
VOID *FindDefclass(defclassName);
char *defclassName;
```

Purpose: Returns a generic pointer to a named class.

Arguments: The name of the class to be found.

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

4.12.10 GetDefclassList

```
VOID GetDefclassList(&returnValue,theModule);
DATA_OBJECT returnValue;
VOID *theModule;
```

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

Arguments:
1) A pointer to the caller's DATA_OBJECT in which the return value will be stored. The multifield functions described in section 3.2.4 can be used to retrieve the defclass names from the list.
2) A generic pointer to the module from which the list will be extracted. A NULL pointer indicates that the list is to be extracted from al l modules.

Returns: No meaningful return value.

4.12.11 GetDefclassName

```
char *GetDefclassName(defclassPtr);
VOID *defclassPtr;
```

Purpose: Returns the name of a class.

Arguments: A generic pointer to a defclass data structure.

Returns: A string containing the name of the class.

4.12.12 GetDefclassPPForm

```
char *GetDefclassPPForm(defclassPtr);
VOID *defclassPtr;
```

Purpose: Returns the pretty print representation of a class.

Arguments: A generic pointer to a defclass data structure.

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

4.12.13 GetDefclassWatchInstances

```
int GetDefclassWatchInstances(defclassPtr);
VOID *defclassPtr;
```

Purpose: Indicates whether or not a particular defclass is being watched for instance creation and deletions.

Arguments: A generic pointer to a defclass data structure.

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

4.12.14 GetDefclassWatchSlots

```
int GetDefclassWatchSlots(defclassPtr);
VOID *defclassPtr;
```

Purpose: Indicates whether or not a particular defclass is being watched for slot changes.

Arguments: A generic pointer to a defclass data structure.

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

4.12.15 GetNextDefclass

```
VOID *GetNextDefclass(defclassPtr);
VOID *defclassPtr;
```

Purpose: Provides access to the list of classes.

Arguments: A generic pointer to a defclass data structure (or NULL to get the first class).

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

4.12.16 IsDefclassDeletable

```
int   IsDefclassDeletable(defclassPtr);
VOID *defclassPtr;
```

Purpose: Indicates whether or not a particular class and all its subclasses can be deleted.

Arguments: A generic pointer to a defclass data structure.

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

4.12.17 ListDefclasses

```
VOID ListDefclasses(logicalName,theModule);
char *logicalName;
VOID *theModule;
```

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

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

Returns: No meaningful return value.

4.12.18 SetDefclassWatchInstances

```
VOID SetDefclassWatchInstances(newState,defclassPtr);
int    newState;
 VOID *defclassPtr;
```

Purpose: Sets the instances watch item for a specific defclass.

Arguments: The new instances watch state and a generic pointer to a defclass data structure.

4.12.19 SetDefclassWatchSlots

```
VOID SetDefclassWatchSlots(newState,defclassPtr);
int    newState;
 VOID *defclassPtr;
```

Purpose: Sets the slots watch item for a specific defclass.

Arguments: The new slots watch state and a generic pointer to a defclass data structure.

4.12.20 SlotAllowedValues

```
VOID SlotAllowedValues(defclassPtr,slotName,&result);
VOID *defclassPtr;
char *slotName;
DATA_OBJECT result;
```

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

Arguments: 1) A generic pointer to the class.
2) Name of the slot.
3) Pointer to the data object in which to store the multifield. The multifield functions described in section 3.2.4 can be used to retrieve the allowed values from the list.

Returns: No meaningful return value.

4.12.21 SlotCardinality

```
VOID SlotCardinality(defclassPtr,slotName,result);
VOID *defclassPtr;
char *slotName;
DATA_OBJECT *result;
```

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

Arguments: 1) A generic pointer to the class.
2) Name of the slot.
3) Pointer to the data object in which to store the multifield.

Returns: No meaningful return value.

4.12.22 SlotDirectAccessP

```
int SlotDirectAccessP(defclassPtr,slotName);
VOID *defclassPtr,
char *slotName;
```

Purpose: Determines if the specified slot is directly accessible.

Arguments:
1) A generic pointer to a defclass data structure.
2) The name of the slot.

Returns: An integer: 1 if the slot is directly accessible, otherwise 0.

4.12.23 SlotExistP

```
int SlotExistP(defclassPtr,slotName,inheritFlag);
VOID *defclassPtr,
char *slotName;
 int inheritFlag;
```

Purpose: Determines if the specified slot exists.

Arguments:
1) A generic pointer to a defclass data structure.
2) The name of the slot.

Returns: An integer: If inheritFlag is 0 and the slot is directly defined in the specified class, then 1 is returned, otherwise 0 is returned. If inheritFlag is 1 and the slot is defined either in the specified class or an inherited class, then 1 is returned, otherwise 0 is returned.

4.12.24 SlotFacets

```
VOID SlotFacets(defclassPtr,slotName,result);
VOID *defclassPtr;
char *slotName;
DATA_OBJECT *result;
```

Purpose: Groups the facet values of a class slot into a multifield data object. This function is the C equivalent of the CLIPS **slot-facets** command. See section 10.8.1.11 in the Basic Programming Guide for more detail.

Arguments:
1) A generic pointer to the class.
2) Name of the slot.
3) Pointer to the data object in which to store the multifield.

Returns: No meaningful return value.

4.12.25 SlotInitableP

```
int SlotInitableP(defclassPtr,slotName);
VOID *defclassPtr,
char *slotName;
```

Purpose: Determines if the specified slot is initable.

Arguments:
1) A generic pointer to a defclass data structure.
2) The name of the slot.

Returns: An integer: 1 if the slot is initable, otherwise 0.

4.12.26 SlotPublicP

```
int SlotPublicP(defclassPtr,slotName);
VOID *defclassPtr,
char *slotName;
```

Purpose: Determines if the specified slot is public.

Arguments:
1) A generic pointer to a defclass data structure.
2) The name of the slot.

Returns: An integer: 1 if the slot is public, otherwise 0.

4.12.27 SlotRange

```
VOID SlotRange(defclassPtr,slotName,result);
VOID *defclassPtr;
char *slotName;
DATA_OBJECT *result;
```

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

Arguments:
1) A generic pointer to the class.
2) Name of the slot.
3) Pointer to the data object in which to store the multifield.

Returns: No meaningful return value.

4.12.28 SlotSources

```
VOID SlotSources(defclassPtr,slotName,result);
VOID *defclassPtr;
char *slotName;
DATA_OBJECT *result;
```

Purpose: Groups the names of the class sources of a slot into a multifield data object. This function is the C equivalent of the CLIPS **slot-sources** command. See section 10.8.1.12 in the Basic Programming Guide for more detail.

Arguments: 1) A generic pointer to the class.
2) Name of the slot.
3) Pointer to the data object in which to store the multifield.

Returns: No meaningful return value.

4.12.29 SlotTypes

```
VOID SlotTypes(defclassPtr,slotName,result);
VOID *defclassPtr;
char *slotName;
DATA_OBJECT *result;
```

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

Arguments: 1) A generic pointer to the class.
2) Name of the slot.
3) Pointer to the data object in which to store the multifield.

Returns: No meaningful return value.

4.12.30 SlotWritableP

```
int SlotWritableP(defclassPtr,slotName);
VOID *defclassPtr,
char *slotName;
```

Purpose: Determines if the specified slot is writable.

Arguments: 1) A generic pointer to a defclass data structure.
2) The name of the slot.

Returns: An integer: 1 if the slot is writable, otherwise 0.

4.12.31 SubclassP

```
int SubclassP(defclassPtr1,defclassPtr2);
VOID *defclassPtr1, *defclassPtr2;
```

Purpose: Determines if a class is a subclass of another class.

Arguments: 1) A generic pointer to a defclass data structure.
2) A generic pointer to a defclass data structure.

Returns: An integer: 1 if the first class is a subclass of the second class.

4.12.32 SuperclassP

```
int SuperclassP(defclassPtr1,defclassPtr2);
VOID *defclassPtr1, *defclassPtr2;
```

Purpose: Determines if a class is a superclass of another class.

Arguments: 1) A generic pointer to a defclass data structure.
2) A generic pointer to a defclass data structure.

Returns: An integer: 1 if the first class is a superclass of the second class.

4.12.33 Undefclass

```
int   Undefclass(defclassPtr);
VOID *defclassPtr;
```

Purpose: Removes a class and all its subclasses from CLIPS (the C equivalent of the CLIPS **undefclass** command).

Arguments: A generic pointer to a defclass data structure.

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

4.13 INSTANCE FUNCTIONS

The following function calls are used for manipulating instances.

4.13.1 BinaryLoadInstances

```
long BinaryLoadInstances(fileName);
char *fileName;
```

Purpose: Loads a set of instances from a binary file into the CLIPS data base (the C equivalent of the CLIPS **bload-instances** command).

Arguments: A string representing the name of the binary file.

Returns: Returns the number of instances restored or -1 if the file could not be accessed.

4.13.2 BinarySaveInstances

```
long BinarySaveInstances(fileName,saveCode,NULL,CLIPS_TRUE);
char *fileName;
int saveCode;
```

Purpose: Saves the instances in the system to the specified binary file (the C equivalent of the CLIPS **bsave-instances** command).

Arguments:
1) A string representing the name of the binary file.
2) An integer flag indicating whether to save local (current module only) or visible instances. Use either the constant LOCAL_SAVE or VISIBLE_SAVE.
3) Should always be NULL.
4) Should always be CLIPS_TRUE.

Returns: Returns the number of instances saved.

4.13.3 CreateRawInstance

```
VOID *CreateRawInstance(defclassPtr,instanceName);
VOID *defclassPtr;
char *instanceName;
```

Purpose: Creates an empty instance with the specified name of the specified class. No slot overrides or class default initializations are performed for the instance.

Arguments: 1) A generic pointer to the class of the new instance.
2) The name of the new instance.

Returns: A generic pointer to the new instance, NULL on errors.

WARNING: This function bypasses message-passing.

4.13.4 DecrementInstanceCount

```
VOID DecrementInstanceCount(instancePtr);
VOID *instancePtr;
```

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

Arguments: A generic pointer to the instance.

Returns: No meaningful return value.

4.13.5 DeleteInstance

```
int DeleteInstance(instancePtr);
VOID *instancePtr;
```

Purpose: Deletes the specified instance(s).

Arguments: A generic pointer to the instance to be deleted. If the pointer is NULL, all instances in the system are deleted.

Returns: Non-zero if successful, 0 otherwise.

WARNING: This function bypasses message-passing.

4.13.6 DirectGetSlot

```
VOID DirectGetSlot(instancePtr,slotName,result);
VOID *instancePtr;
char *slotName;
DATA_OBJECT *result;
```

Purpose: Stores the value of the specified slot of the specified instance in the caller's buffer (the C equivalent of the CLIPS **dynamic-get** function).

Arguments:
1) A generic pointer to the instance.
2) The name of the slot.
3) The caller's buffer for the slot value. See sections 3.2.3 and 3.2.4 for information on getting the value stored in a DATA_OBJECT.

Returns: No meaningful return value.

WARNING: This function bypasses message-passing.

4.13.7 DirectPutSlot

```
int DirectPutSlot(instancePtr,slotName,newValue);
VOID *instancePtr;
char *slotName;
DATA_OBJECT *newValue;
```

Purpose: Stores a value in the specified slot of the specified instance (the C equivalent of the CLIPS **dynamic-put** function).

Arguments:
1) A generic pointer to the instance.
2) The name of the slot.
3) The caller's buffer containing the new value (an error is generated if this value is NULL). See sections 3.3.3 and 3.3.4 for information on setting the value stored in a DATA_OBJECT.

Returns: Returns an integer; if zero, an error occurred while setting the slot. If non-zero, no errors occurred.

WARNING: This function bypasses message-passing.

4.13.8 FindInstance

```
VOID *FindInstance(theModule,instanceName,searchImports);
VOID *theModule;
char *instanceName;
int searchImports;
```

Purpose: Returns the address of the specified instance.

Arguments:
1) A generic pointer to the module to be searched (NULL to search the current module).
2) The name of the instance (should not include a module specifier).
3) A boolean flag indicating whether imported modules should also be searched: TRUE to search imported modules, otherwise FALSE.

Returns: A generic pointer to the instance, NULL if the instance does not exist.

4.13.9 GetInstanceClass

```
VOID *GetInstanceClass(instancePtr);
VOID *instancePtr;
```

Purpose: Determines the class of an instance.

Arguments: A generic pointer to an instance.

Returns: A generic pointer to the class of the instance.

4.13.10 GetInstanceName

```
char *GetInstanceName(instancePtr);
VOID *instancePtr;
```

Purpose: Determines the name of an instance.

Arguments: A generic pointer to an instance.

Returns: The name of the instance.

4.13.11 GetInstancePPForm

```
VOID GetInstancePPForm(buffer,bufferLength,instancePtr);
char *buffer;
int   bufferLength;
VOID *instancePtr;
```

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

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

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

4.13.12 GetInstancesChanged

```
int GetInstancesChanged();
```

Purpose: Determines if any changes to instances of user-defined instances have occurred, e.g. instance creations/deletions or slot value changes. If this function returns a non-zero integer, it is the user's responsibility to call SetInstancesChanged(0) to reset the internal flag. Otherwise, this function will continue to return non-zero even when no changes have occurred. This function is primarily used to determine when to update a display tracking instances.

Arguments: None.

Returns: 0 if no changes to instances of user-defined classes have occurred, non-zero otherwise.

4.13.13 GetNextInstance

```
VOID *GetNextInstance(instancePtr);
VOID *instancePtr;
```

Purpose: Provides access to the list of instances.

Arguments: A generic pointer to an instance (or NULL to get the first instance in the list).

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

4.13.14 GetNextInstanceInClass

```
VOID *GetNextInstanceInClass(defclassPtr,instancePtr);
VOID *defclassPtr,*instancePtr;
```

Purpose: Provides access to the list of instances for a particular class.

Arguments:
1) A generic pointer to a class.
2) A generic pointer to an instance (or NULL to get the first instance in the specified class).

Returns: A generic pointer to the first instance in the list of instances for the specified class if *instancePtr* is NULL, otherwise a pointer to the instance immediately following *instancePtr* in the list. If *instancePtr* is the last instance in the class, then NULL is returned.

4.13.15 IncrementInstanceCount

```
VOID IncrementInstanceCount(instancePtr);
VOID *instancePtr;
```

Purpose: This function should be called for each external copy of an instance address to let CLIPS know that such an outstanding external reference exists. As long as an instance's count is greater than zero, CLIPS will not release its memory because there may be outstanding pointers to the instance. However, the instance can still be *functionally* deleted, i.e. the instance will *appear* to no longer be in the system. The instance address always can be safely passed to instance access functions as long as the count for the instance is greater than zero. These functions will recognize when an instance has been functionally deleted.

Arguments: A generic pointer to the instance.

Returns: No meaningful return value.

Example

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

VOID InstanceReferenceExample
  {
   VOID *myInstancePtr;

   myInstancePtr = FindInstance(NULL,"my-instance",CLIPS_TRUE);

   /*=========================================*/
   /* Instance my-instance could be potentially */
   /* deleted during the run.                 */
   /*=========================================*/

   Run(-1L);

   /*=========================================*/
   /* This next function call could dereference */
   /* a dangling pointer and cause a crash.    */
   /*=========================================*/

   DeleteInstance(myInstancePtr);
  }

/*=========*/
/* Correct */
/*=========*/

VOID InstanceReferenceExample
  {
   VOID *myInstancePtr;

   myInstancePtr = FindInstance(NULL,"my-instance",CLIPS_TRUE);

   /*=====================================================*/
   /* The instance is correctly marked so that a dangling */
   /* pointer cannot be created during the run.           */
   /*=====================================================*/

   IncrementInstanceCount(myInstancePtr);
   Run(-1L);
   DecrementInstanceCount(myInstancePtr);

   /*=============================================================*/
   /* The instance can now be safely deleted using the pointer. */
   /*=============================================================*/

   DeleteInstance(myInstancePtr);
  }
```

4.13.16 Instances

```
VOID Instances(logicalName,modulePtr,className,subclassFlag);
char *logicalName;
VOID *defmodulePtr;
char *className;
int subclassFlag;
```

Purpose: Prints the list of all direct instances of a specified class currently in the system (the C equivalent of the CLIPS **instances** command).

Arguments:
1) The logical name to which output is sent.
2) A generic pointer to a defmodule data structure (NULL indicates to list all instances of all classes in all modules—the third and fourth arguments are ignored).
3) The name of the class for which to list instances (NULL indicates to list all instances of all classes in the specified module—the fourth argument is ignored).
4) A flag indicating whether or not to list recursively direct instances of subclasses of the named class in the specified module. 0 indicates no, and any other value indicates yes.

Returns: No meaningful return value.

4.13.17 LoadInstances

```
long LoadInstances(fileName);
char *fileName;
```

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

Arguments: A string representing the name of the file.

Returns: Returns the number of instances loaded or -1 if the file could not be accessed.

4.13.18 MakeInstance

```
VOID *MakeInstance(makeCommand);
char *makeCommand;
```

Purpose: Creates and initializes an instance of a user-defined class (the C equivalent of the CLIPS **make-instance** function).

Arguments: A string containing a **make-instance** command in the format below:

(<instance-name> of <class-name> <slot-override>*)
<slot-override> :== (<slot-name> <constant>*)

Returns: A generic pointer to the new instance, NULL on errors.

Example

```
MakeInstance("(henry of boy (age 8))");
```

4.13.19 RestoreInstances

```
long RestoreInstances(fileName);
char *fileName;
```

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

Arguments: A string representing the name of the file.

Returns: Returns the number of instances restored or -1 if the file could not be accessed.

4.13.20 SaveInstances

```
long SaveInstances(fileName,saveCode,NULL,CLIPS_TRUE);
char *fileName;
int saveCode;
```

Purpose: Saves the instances in the system to the specified file (the C equivalent of the CLIPS **save-instances** command).

Arguments:
1) A string representing the name of the file.
2) An integer flag indicating whether to save local (current module only) or visible instances. Use either the constant LOCAL_SAVE or VISIBLE_SAVE.
3) Should always be NULL.
4) Should always be CLIPS_TRUE.

Returns: Returns the number of instances saved.

4.13.21 Send

```
VOID Send(instanceBuffer,msg,msgArgs,result);
DATA_OBJECT *instanceBuffer, *result;
char *msg,*msgArgs;
```

Purpose: Message-passing from C Sends a message with the specified arguments to the specified object and stores the result in the caller's buffer (the C equivalent of the CLIPS **send** function).

Arguments:

1) A data value holding the object (instance, symbol, float, etc.) which will receive the message.
2) The message.
3) A string containing any *constant* arguments separated by blanks (this argument can be NULL).
4) Caller's buffer for storing the result of the message. See sections 3.2.3 and 3.2.4 for information on getting the value stored in a DATA_OBJECT.

Returns: No meaningful return value.

Example

```
VOID SendMessageExample()
  {
   DATA_OBJECT insdata, rtn;
   VOID *myInstancePtr;

   myInstancePtr = MakeInstance("(my-instance of MY-CLASS");
   SetType(insdata,INSTANCE_ADDRESS);
   SetValue(insdata,myInstancePtr);
   Send(&insdata,"my-msg","1 abc 3",&rtn);
  }
```

4.13.22 SetInstancesChanged

```
VOID SetInstancesChanged(changedFlag);
int changedFlag;
```

Purpose: Sets the internal boolean flag which indicates when changes to instances of user-defined classes have occurred. This function is normally used to reset the flag to zero after GetInstancesChanged() returns non-zero.

Arguments: An integer indicating whether changes in instances of user-defined classes have occurred (non-zero) or not (0).

Returns: Nothing useful.

4.13.23 UnmakeInstance

```
int UnmakeInstance(instancePtr);
VOID *instancePtr;
```

Purpose: This function is equivalent to DeleteInstance except that it uses message-passing instead of directly deleting the instance(s).

Arguments: A generic pointer to the instance to be deleted. If the pointer is NULL, all instances in the system are deleted.

Returns: Non-zero if successful, 0 otherwise.

4.13.24 ValidInstanceAddress

```
int ValidInstanceAddress(instancePtr);
VOID *instancePtr;
```

Purpose: Determines if an instance referenced by an address still exists. See the description of IncrementInstanceCount.

Arguments: The address of the instance.

Returns: The integer 1 if the instance still exists, 0 otherwise.

4.14 DEFMESSAGE-HANDLER FUNCTIONS

The following function calls are used for manipulating defmessage-handlers.

4.14.1 FindDefmessageHandler

```
unsigned FindDefmessageHandler(defclassPtr,
                               handlerName,handlerType);
VOID *defclassPtr,
char *handlerName,*handlerType;
```

Purpose: Returns an index to the specified message-handler within the list of handlers for a particular class.

Arguments:
1) A generic pointer to the class to which the handler is attached.
2) The name of the handler.

3) The type of the handler: around, before, primary or after.

Returns: An index to the specified handler if it exists, otherwise 0.

4.14.2 GetDefmessageHandlerList

```
VOID GetDefmessageHandlerList(defclassPtr,&returnValue,
                              includeInheritedp);
VOID *defclassPtr;
DATA_OBJECT returnValue;
int includeInheritedp
```

Purpose: Returns the list of currently defined defmessage-handlers for the specified class. This function is the C equivalent of the CLIPS **get-defmessage-handler-list** command).

Arguments:
1) A generic pointer to the class (NULL for all classes).
2) A pointer to the DATA_OBJECT in which the list of defmessage-handler constructs is to be stored.
3) An integer flag indicating whether to list inherited handlers (CLIPS_TRUE to list them or CLIPS_FALSE to not list them).

Returns: No meaningful value. The second argument to this function is set to a multifield value containing the list of defmessage-handler constructs for the specified class. The multifield functions described in section 3.2.4 can be used to retrieve the defmessage-handler class, name, and type from the list. Note that the class, name, and type for each defmessage-handler are stored as triplets in the return multifield value.

4.14.3 GetDefmessageHandlerName

```
char *GetDefmessageHandlerName(defclassPtr,handlerIndex);
VOID *defclassPtr;
unsigned handlerIndex;
```

Purpose: Returns the name of a message-handler.

Arguments:
1) A generic pointer to a defclass data structure.
2) The index of a message-handler.

Returns: A string containing the name of the message-handler.

4.14.4 GetDefmessageHandlerPPForm

```
char *GetDefmessageHandlerPPForm(defclassPtr,handlerIndex);
VOID *defclassPtr;
unsigned handlerIndex;
```

Purpose: Returns the pretty print representation of a message-handler.

Arguments:
1) A generic pointer to a defclass data structure.
2) The index of a message-handler.

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

4.14.5 GetDefmessageHandlerType

```
char *GetDefmessageHandlerType(defclassPtr,handlerIndex);
VOID *defclassPtr;
unsigned handlerIndex;
```

Purpose: Returns the type (around, before, primary or after) of a message-handler.

Arguments:
1) A generic pointer to a defclass data structure.
2) The index of a message-handler.

Returns: A string containing the type of the message-handler.

4.14.6 GetDefmessageHandlerWatch

```
int GetDefmessageHandlerWatch(defclassPtr,handlerIndex);
VOID *defclassPtr;
 unsigned handlerIndex
```

Purpose: Indicates whether or not a particular defmessage-handler is being watched.

Arguments: A generic pointer to a defclass data structure and the index of the message-handler.

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

4.14.7 GetNextDefmessageHandler

```
unsigned GetNextDefmessageHandler(defclassPtr,handlerIndex);
VOID *defclassPtr;
unsigned handlerIndex;
```

Purpose: Provides access to the list of message-handlers.

Arguments:
1) A generic pointer to a defclass data structure.
2) An index to a particular message-handler for the class (or 0 to get the first message-handler).

Returns: An index to the first handler in the list of handlers if *handlerIndex* is 0, otherwise an index to the handler immediately following *handlerIndex* in the list of handlers for the class. If *handlerIndex* is the last handler in the list of handlers for the class, then 0 is returned.

4.14.8 IsDefmessageHandlerDeletable

```
int   IsDefmessageHandlerDeletable(defclassPtr,handlerIndex);
VOID *defclassPtr;
unsigned handlerIndex;
```

Purpose: Indicates whether or not a particular message-handler can be deleted.

Arguments:
1) A generic pointer to a defclass data structure.
2) The index of a message-handler.

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

4.14.9 ListDefmessageHandlers

```
VOID ListDefmessageHandlers(logicalName,defclassPtr,
                            includeInheritedp);
char *logicalName;
VOID *defclassPtr;
int includeInheritedp
```

Purpose: Prints the list of message-handlers for the specified class. This function is the C equivalent of the CLIPS **list-defmessage-handlers** command).

Arguments:
1) The logical name to which the listing output is sent.
2) A generic pointer to the class (NULL for all classes).
3) An integer flag indicating whether to list inherited handlers (CLIPS_TRUE to list them or CLIPS_FALSE to not list them).

Returns: No meaningful return value.

4.14.10 PreviewSend

```
VOID PreviewSend(logicalName,defclassPtr,messageName);
char *logicalName;
VOID *defclassPtr;
char *messageName;
```

Purpose: Prints a list of all applicable message-handlers for a message sent to an instance of a particular class (the C equivalent of the CLIPS **preview-send** command). Output is sent to the logical name **wdisplay**.

Arguments:
1) The logical name to which output is sent.
2) A generic pointer to the class.
3) The message name.

Returns: No meaningful return value.

4.14.11 SetDefmessageHandlerWatch

```
VOID SetDefmessageHandlerWatch(newState,defclassPtr,
                               handlerIndex);
int    newState;
 VOID *defclassPtr;
 unsigned handlerIndex
```

Purpose: Sets the message-handlers watch item for a specific defmessage-handler.

Arguments: The new message-handlers watch state, a generic pointer to a defclass data structure, and the index of the message-handler.

4.14.12 UndefmessageHandler

```
int   UndefmessageHandler(defclassPtr,handlerIndex);
VOID *defclassPtr;
unsigned handlerIndex;
```

Purpose: Removes a message-handler from CLIPS (similar but *not* equivalent to the CLIPS **undefmessage-handler** command - see WildDeleteHandler).

Arguments:
1) A generic pointer to a defclass data structure (NULL to delete all message-handlers in all classes).
2) The index of the message-handler (0 to delete all message-handlers in the class - must be 0 if *defclassPtr* is NULL).

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

4.15 DEFINSTANCES FUNCTIONS

The following function calls are used for manipulating definstances.

4.15.1 DefinstancesModule

```
char *DefinstancesModule(theDefinstances);
VOID *theDefinstances;
```

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

Arguments: A generic pointer to a definstances.

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

4.15.2 FindDefinstances

```
VOID *FindDefinstances(definstancesName);
char *definstancesName;
```

Purpose: Returns a generic pointer to a named definstances.

Arguments: The name of the definstances to be found.

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

4.15.3 GetDefinstancesList

```
VOID GetDefinstancesList(&returnValue,theModule);
DATA_OBJECT returnValue;
VOID *theModule;
```

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

Arguments:
1) A pointer to the caller's DATA_OBJECT in which the return value will be stored. The multifield functions described in section 3.2.4 can be used to retrieve the definstances names from the list.
2) A generic pointer to the module from which the list will be extracted. A NULL pointer indicates that the list is to be extracted from al l modules.

Returns: No meaningful return value.

4.15.4 GetDefinstancesName

```
char *GetDefinstancesName(definstancesPtr);
VOID *definstancesPtr;
```

Purpose: Returns the name of a definstances.

Arguments: A generic pointer to a definstances data structure.

Returns: A string containing the name of the definstances.

4.15.5 GetDefinstancesPPForm

```
char *GetDefinstancesPPForm(definstancesPtr);
VOID *definstancesPtr;
```

Purpose: Returns the pretty print representation of a definstances.

Arguments: A generic pointer to a definstances data structure.

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

4.15.6 GetNextDefinstances

```
VOID *GetNextDefinstances(definstancesPtr);
VOID *definstancesPtr;
```

Purpose: Provides access to the list of definstances.

Arguments: A generic pointer to a definstances data structure (or NULL to get the first definstances).

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

4.15.7 IsDefinstancesDeletable

```
int   IsDefinstancesDeletable(definstancesPtr);
VOID *definstancesPtr;
```

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

Arguments: A generic pointer to a definstances data structure.

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

4.15.8 ListDefinstances

```
VOID ListDefinstances(logicalName,theModule);
char *logicalName;
VOID *theModule;
```

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

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

Returns: No meaningful return value.

4.15.9 Undefinstances

```
int   Undefinstances(definstancesPtr);
VOID *definstancesPtr;
```

Purpose: Removes a definstances from CLIPS (the C equivalent of the CLIPS **undefinstances** command).

Arguments: A generic pointer to a definstances data structure.

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

4.16 DEFMODULE FUNCTIONS

The following function calls are used for manipulating defmodules.

4.16.1 FindDefmodule

```
VOID *FindDefmodule(defmoduleName);
char *defmoduleName;
```

Purpose: Returns a generic pointer to a named defmodule.

Arguments: The name of the defmodule to be found.

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

4.16.2 GetCurrentModule

```
VOID *GetCurrentModule();
```

Purpose: Returns the current module (the C equivalent of the CLIPS **get-current-module** function).

Arguments: None.

Returns: A generic pointer to the generic defmodule data structure that is the current module.

4.16.3 GetDefmoduleList

```
VOID GetDefmoduleList(&returnValue);
DATA_OBJECT returnValue;
```

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

Arguments: A pointer to the caller's DATA_OBJECT in which the return value will be stored. The multifield functions described in section 3.2.4 can be used to retrieve the defmodule names from the list.

Returns: No meaningful return value.

4.16.4 GetDefmoduleName

```
char *GetDefmoduleName(defmodulePtr);
VOID *defmodulePtr;
```

Purpose: Returns the name of a defmodule.

Arguments: A generic pointer to a defmodule data structure.

Returns: A string containing the name of the defmodule.

4.16.5 GetDefmodulePPForm

```
char *GetDefmodulePPForm(defmodulePtr);
VOID *defmodulePtr;
```

Purpose: Returns the pretty print representation of a defmodule.

Arguments: A generic pointer to a defmodule data structure.

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

4.16.6 GetNextDefmodule

```
VOID *GetNextDefmodule(defmodulePtr);
VOID *defmodulePtr;
```

Purpose: Provides access to the list of defmodules.

Arguments: A generic pointer to a defmodule data structure (or NULL to get the first defmodule).

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

4.16.7 ListDefmodules

```
VOID ListDefmodules(logicalName);
char *logicalName;
```

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

Arguments: 1) The logical name to which the listing output is sent.

Returns: No meaningful return value.

4.16.8 SetCurrentModule

```
VOID *SetCurrentModule(defmodulePtr);
VOID *defmodulePtr;
```

Purpose: Sets the current module to the specified module (the C equivalent of the CLIPS **set-current-module** function).

Arguments: A generic pointer to a defmodule data structure.

Returns: A generic pointer to the previous current defmodule data structure.

4.17 EMBEDDED APPLICATION EXAMPLES

4.17.1 User-Defined Functions

This section lists the steps needed to define and use an embedded CLIPS application. The example given is the same system used in section 3.4, now set up to run as an embedded application.

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

2) Define the user function (TripleNumber), a new main routine, and UserFunctions in a new file. These could go in separate files if desired. For this example, they will all be included in a single file.

```
#include "clips.h"

main()
  {
   InitializeCLIPS();
   Load("constructs.clp");
   Reset();
   Run(-1L)
  }


VOID TripleNumber(returnValuePtr)
  DATA_OBJECT_PTR returnValuePtr;
  {
   VOID        *value;
   long         longValue;
   double       doubleValue;

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

   if (ArgCountCheck("triple",EXACTLY,1) == -1)
     {
      SetpType(returnValuePtr,INTEGER);
      SetpValue(returnValuePtr,AddLong(0L));
      return;
     }

   if (! ArgTypeCheck("triple",1,INTEGER_OR_FLOAT,returnValuePtr))
     {
      SetpType(returnValuePtr,INTEGER);
      SetpValue(returnValuePtr,AddLong(0L));
      return;
     }
```

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

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

   return;
  }

UserFunctions()
  {
   extern VOID TripleNumber();

   DefineFunction2("triple",'u',PTIF TripleNumber, "TripleNumber",
                   "11n");
  }
```

3) Define constructs which use the new function in a file called **constructs.clp** (or any file; just be sure the call to **Load** loads all necessary constructs prior to execution).

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

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

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

4) Compile all CLIPS files, *except* **main.c**, along with all user files.

5) Link all object code files.

6) Execute new CLIPS executable.

4.17.2 Manipulating Objects and Calling CLIPS Functions

This section lists the steps needed to define and use an embedded CLIPS application. The example illustrates how to call deffunctions and generic functions as well as manipulate objects from C.

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

2) Define a new main routine in a new file.

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

main()
  {
   VOID *c1,*c2,*c3;
   DATA_OBJECT insdata,result;
   char numbuf[20];

   InitializeCLIPS();

   /*=======================================================*/
   /* Load the classes, message-handlers, generic functions */
   /* and generic functions necessary for handling complex  */
   /* numbers.                                              */
   /*=======================================================*/

   Load("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("(c1 of COMPLEX (real 1) (imag 10))");
   c2 = CreateRawInstance(FindDefclass("COMPLEX"),"c2");

   result.type = INTEGER;
   result.value = AddLong(3L);
   DirectPutSlot(c2,"real",&result);

   result.type = INTEGER;
   result.value = AddLong(-7L);
   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.            */
   /*=========================================================*/

   CLIPSFunctionCall("+","[c1] [c2]",&result);
   c3 = FindInstance(NULL,DOToString(result),CLIPS_TRUE);

   /*=========================================================*/
   /* Print out a summary of the complex addition using the */
   /* "print" and "magnitude" messages to get information   */
   /* about the three complex numbers.                      */
   /*=========================================================*/

   PrintCLIPS("stdout","The addition of\n\n");

   SetType(insdata,INSTANCE_ADDRESS);
   SetValue(insdata,c1);
   Send(&insdata,"print",NULL,&result);

   PrintCLIPS("stdout","\nand\n\n");

   SetType(insdata,INSTANCE_ADDRESS);
   SetValue(insdata,c2);
   Send(&insdata,"print",NULL,&result);

   PrintCLIPS("stdout","\nis\n\n");

   SetType(insdata,INSTANCE_ADDRESS);
   SetValue(insdata,c3);
   Send(&insdata,"print",NULL,&result);

   PrintCLIPS("stdout","\nand the resulting magnitude is\n\n");

   SetType(insdata,INSTANCE_ADDRESS);
   SetValue(insdata,c3);
   Send(&insdata,"magnitude",NULL,&result);
   sprintf(numbuf,"%lf\n",DOToDouble(result));
   PrintCLIPS("stdout",numbuf);
  }

UserFunctions()
  {}
```

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)
   (role concrete)
   (slot real (create-accessor read-write))
   (slot imag (create-accessor read-write)))
```

```
(defmethod + ((?a COMPLEX) (?b COMPLEX))
   (make-instance of COMPLEX
      (real (+ (send ?a get-real) (send ?b get-real)))
      (imag (+ (send ?a get-imag) (send ?b get-imag)))))

(defmessage-handler COMPLEX magnitude ()
   (sqrt (+ (** ?self:real 2) (** ?self:imag 2))))
```

4) Compile all CLIPS files, *except* **main.c**, along with all user files.

5) Link all object code files.

6) Execute new CLIPS executable.

Section 5 - Creating a CLIPS Run-time Program

5.1 COMPILING THE CONSTRUCTS

This section describes the procedure for creating a CLIPS run-time module. A run-time program compiles all of the constructs (defrule, deffacts, deftemplate, etc.) into a single executable and reduces the size of the executable image. A run-time program will not run any faster than a program loaded using the **load** or **bload** commands. The **constructs-to-c** command used to generate a run-time program creates files containing the C data structures that would dynamically be allocated if the **load** or **bload** command was used. With the exception of some initialization routines, the **constructs-to-c** command does not generate any executable code. The primary benefits of creating a run-time program are: applications can be delivered as a single executable file; loading constructs as part of an executable is faster than loading them from an text or binary file; the CLIPS portion of the run-time program is smaller because the code needed to parse constructs can be discarded; and less memory is required to represent your program's constructs since memory for them is statically rather than dynamically allocated.

Creating a run-time module can be achieved with the following steps:

1) Start CLIPS and load in all of the constructs that will constitute a run-time module. Call the **constructs-to-c** command using the following syntax:

   (constructs-to-c <file-name> <id> [<max-elements>])

   where <file-name> is a string or a symbol, <id> is an integer, and the optional argument <max-elements> is also 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). It is possible to limit the size of the generated files by using the <max-elements> argument. This argument indicates the maximum number of structures which may be placed in a single array stored in a file. Where possible, if this number is exceeded new files will be created to store additional information. This feature is useful for compilers that may place a limitation on the size of a file that may be compiled.

Note that the .c extension is added by CLIPS. When giving the file name prefix, users should consider the maximum number of characters their system allows in a file name. For example, under MS-DOS, only eight characters are allowed in the file name. For very large systems, it is possible for CLIPS to add up to 5 characters to the file name prefix. Therefore, for system which allow only 8 character file names, the prefix should be no more than 3 characters.

Constraint information associated with constructs is not saved to the C files generated by the **constructs-to-c** command unless dynamic constraint checking is enabled (using the **set-dynamic-constraint-checking** command).

The **constructs-to-c** command is not available in the standard CLIPS distribution executable. Users wishing to create a run-time program must recompile CLIPS to include this capability (see section 2.2 for information on tailoring CLIPS and the CONSTRUCT_COMPILER setup flag).

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. The user still must call the function **InitializeCLIPS** in the main module. It will have been modified to make appropriate initializations for the run-time version. 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 the **UserFunctions** function. The function **UserFunctions** is not called during initialization. All of the function definitions have already been compiled in the 'C' constructs code. In addition to calling **InitializeCLIPS**, 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**. These two initialization steps probably would be followed by any user initialization, then by a reset and run. An example **main.c** file would be

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

main()
  {
   InitializeCLIPS();
   InitCImage_1();
        •
        •                          /* Any user Initialization */
        •
   Reset();
   Run(-1L);
        •
        •                          /* Any other code */
        •
  }

UserFunctions()
  {
   /* UserFunctions is not called for a run-time version. */
  }
```

4) Recompile all of the CLIPS source code (the RUN_TIME flag should still be 1). This causes several modifications in the CLIPS code. The run-time CLIPS module does not have the capability to load new constructs. Do NOT change any other compiler flags! Because of the time involved in recompiling CLIPS, it may be appropriate to recompile the run-time version of CLIPS into a separate library from the full version of CLIPS.

5) Link all regular CLIPS modules together with any user-defined function modules and the 'C' construct modules. Make sure that any user-defined functions have global scope. *Do not* place the construct modules within a library for the purposes of linking (the regular CLIPS modules, however, can be placed in a library). Some linkers (most notably the VAX VMS linker) will not correctly resolve references to global data that is stored in a module consisting only of global data.

6) The run-time module which includes user constructs is now ready to run.

Note that individual constructs may not be added or removed in a run-time environment. Because of this, the **load** function is not available for use in run-time programs. The clear command will also not remove any constructs (although it will clear facts and instances). Use calls to the InitCImage_... functions to clear the environment and replace it with a new set of constructs. In addition, the **eval** and **build** functions do not work in a run-time environment.

Since new constructs can't be added, a run-time program can't dynamically load a deffacts or definstances construct. To dynamically load facts and/or instances in a run-time program, the CLIPS **load-facts** and **load-instances** functions or the C **LoadFacts** and **LoadInstances** functions should be used in place of deffacts and definstances constructs.

Switching between different images created using the constructs-to-c function is now supported. Switching, however, may not occur while a CLIPS program is executing (e.g. you cannot call a function from the RHS of a rule which switches a different image into memory). Note that switching between construct images will clear all facts and instances from the CLIPS environment. It is possible to switch to the same image more than once. An example main program which switches between two different construct images is shown following.

```
main()
  {
   extern int failure;

   InitializeCLIPS();

   /*==================================*/
   /* Set up and run the first portion */
   /* of the expert system.            */
   /*==================================*/

   InitCImage_1();

   Reset();
   Run(-1L);

   /*============================================*/
   /* Check a global variable that was defined to */
   /* indicate which portion of the expert system */
   /* to run next.                               */
   /*============================================*/

   if (failure) InitCImage_2();
   else InitCImage_3();

   /*================================================*/
   /* Run the remaining portion of the expert system. */
   /*================================================*/

   Reset();
   Run(-1L);
  }
```

5.1.1 Additional Considerations

The construct compiler is a feature that does not work as well as might be desired on some machines. It has been tested on a VAX using VMS, a SUN workstation using UNIX, a Macintosh with Think C and MPW C, and an IBM PC AT using several different compilers. All machines are able to produce run-time modules for relatively small programs (several dozen constructs). However, the Macintosh and the IBM PC AT compilers have limitations and/or

additional compilation and link options which must be taken into consideration when large amounts of static data are defined in a program. These considerations are described below.

Macintosh (THINK C V5.04)
Enable the **Far DATA** option using the **Set Project Type...** menu item *before* compiling the CLIPS source files and constructs-to-c generated files. Note that individual source files are restricted to less than 32K of static data, so it may be necessary to limit the size of the files generated by the **constructs-to-c** command (see step 1 above).

Macintosh (MPW C V3.2)
When compiling (C command) and linking (Link command) the CLIPS source files and constructs-to-c generated files, use the **-b3** and **-model far** options. In addition, use the **-srt** option for linking.

IBM PC AT (Microsoft C V6.0A using MS-DOS)
It is recommended that you use a <max-elements> size of about 300 when using the constructs-to-c command to limit the size of the generated files. All files (both CLIPS source files and constructs-to-c generated files) should be compiled using the **/AH** and **/Gt1024** options. The resulting object files should be linked using the **/SEG:256**, **/ST:8192**, and **/NOI** options. You may wish to use other options as well or use different parameters for some of the options above.

IBM PC AT (Borland C++ V3.1 using Windows 3.1)
Borland C does not allow the huge memory module to be used for a Windows application, so the limit of 64K of static data prevents using the construct compiler for anything but very small programs.

IBM PC AT (Borland C++ V3.1 using MS-DOS)
It is recommended that you use a <max-elements> size of about 300 when using the constructs-to-c command to limit the size of the generated files. All files (both CLIPS source files and constructs-to-c generated files) should be compiled using the **-mh** and **-d** options.

IBM PC AT (Zortech C++ V3.1 using MS-DOS)
There is a compiler bug which manifests itself when dead code optimizations are performed. When compiling the CLIPS source files, specify the **-o-dc** option to remove dead code optimizations. In addition, use the **-mx** option when compiling and the **=16000** option when linking.

5.1.2 PORTING COMPILED CONSTRUCTS

Unlike previous version of files generated by the rules-to-c command, the files generated for version 6.0 by the constructs-to-c function should be completely portable to other machines.

Section 6 - Combining CLIPS with Languages Other Than C

CLIPS is developed in C and is most easily combined with user functions written in C. However, other languages can be used for user-defined functions, and CLIPS even may be embedded within a program written in another language. Users wishing to embed CLIPS with Ada should consider using CLIPS/Ada (see the *CLIPS/Ada Advanced Programming Guide*).

6.1 INTRODUCTION

This section will describe how to combine CLIPS with Ada or FORTRAN routines. Specific code examples will be used to illustrate the concepts. The code used in these examples is valid for VAX VMS systems which have the DEC C compiler, the DEC FORTRAN compiler, and the DEC Ada compiler.

Three basic capabilities are needed for complete language mixing.

- A program in another language may be used as the main program.
- The C access functions to CLIPS can be called from the other language and have parameters passed to them.
- Functions written in the other language can be called by CLIPS and have parameters passed to them.

The integration of CLIPS (and C) with other languages requires an understanding of how each language passes parameters between routines. In general, interface functions will be needed to pass parameters from C to another language and from another language to C. The basic concepts of mixed language parameter passing are the same regardless of the language or machine. However, since every machine and operating system passes parameters differently, specific details (and code) may differ from machine to machine. To improve usability and to minimize the amount of recoding needed for each machine, interface packages can be developed which allow user routines to call the standard CLIPS embedded command functions. The details of passing information *from* external routines to CLIPS generally are handled inside of the interface package. To pass parameters from CLIPS *to* an external routine, users will have to write interface functions. Example interface packages for VMS FORTRAN and VMS Ada to selected CLIPS functions are listed in appendix A. Section 6.9 will discuss how to construct an interface package for other machines/compilers.

6.2 ADA AND FORTRAN INTERFACE PACKAGE FUNCTION LIST

The Ada and FORTRAN interface packages in appendix A provide many of the embedded CLIPS commands discussed in section 4 of this manual. Each function in the interface package

prepends an *x* to the beginning of the corresponding C function name. A list of the C functions and their FORTRAN or Ada corollaries which are provided in the interface packages listed in the appendices appears below.

C Function	Ada/FORTRAN Function
InitializeCLIPS	xInitializeCLIPS
Reset	xReset
Load	xLoad
Run	xRun
Facts	xFacts
Watch	xWatch
Unwatch	xUnwatch
AssertString	xAssertString
Retract	xRetract
PrintCLIPS	xPrintCLIPS
FindDefrule	xFindDefrule
Undefrule	xUndefrule

The arguments to these functions are the same as described in section 4, however, the corresponding data type in either Ada or FORTRAN should be passed as a parameter. For example, when using Ada, the function xLoad should be passed an Ada string, not a C string (the function xLoad will perform the conversion). FORTRAN function names defined above do *not* follow ANSI 77 name standards. The VMS FORTRAN implementation described in this section allows long function names.

6.3 EMBEDDED CLIPS - USING AN EXTERNAL MAIN PROGRAM

Any program may be used as the main program for embedded CLIPS applications. The main program works essentially the same as in C.

Example Ada Main Program

```
with CLIPS;    use CLIPS;

with TEXT_IO;  use TEXT_IO;

procedure MAIN is

  File_Name        : string (1..50);
  File_Open_Status : integer;
  Rules_Fired      : integer;

begin

  xInitializeCLIPS;

  File_Name (1..7) := "mab.clp";
```

```
    -- Load rules
  File_Open_Status := xLoad (File_Name);

  if File_Open_Status = 1 then
    xReset;
    Rules_Fired := xRun (-1);
    PUT (integer'IMAGE (Rules_Fired));
    PUT_LINE ("  Rules Fired");
  else
    PUT_LINE ("Unable to open rules file");
  end if;

end MAIN;
```

Example FORTRAN Main Program

```
      PROGRAM MAIN
C
      INTEGER xLoad, FILE_OPEN_STATUS
      CHARACTER *8 FILE_NAME
      INTEGER xRun, RULES_FIRED
C
      CALL xInitializeCLIPS
C
      FILE_NAME = 'mab.clp'
      FILE_OPEN_STATUS = xLoad (FILE_NAME)

      IF (FILE_OPEN_STATUS .EQ. 1) THEN
        CALL xReset
        RULES_FIRED = xRun (-1)
        WRITE (6,100) RULES_FIRED
      ELSE
        WRITE (6,101)
      END IF

 100  FORMAT (I8,' RULES FIRED')
 101  FORMAT (' UNABLE TO OPEN RULES FILE')
      STOP
      END

      SUBROUTINE UserFunctions
      RETURN
      END
```

6.4 ASSERTING FACTS INTO CLIPS

An external function may assert a fact into CLIPS by calling **xAssertString**. External functions also may retract a fact previously asserted from outside of CLIPS. Note that the parameter passed to xRetract must have been received from a call to xAssertString. Any other value will cause unpredictable results.

Ada Example

```
Fact_Pointer             : integer;
Not_Previously_Retracted : boolean;

Fact_Pointer := xAssertString ("dummy hello");
Not_Previously_Retracted := xRetract (Fact_Pointer);
```

FORTRAN Example

```
CHARACTER *20 FACT_STRING
INTEGER xAssertString, FACT_POINTER
INTEGER xRetract, NOT_PREVIOUSLY_RETRACTED

FACT_STRING = 'dummy hello'
FACT_POINTER = xAssertString (FACT_STRING)
NOT_PREVIOUSLY_RETRACTED = xRetract (FACT_POINTER)
```

6.5 CALLING A SUBROUTINE FROM CLIPS

Like any other user-defined functions, subroutines written in other languages may be called from CLIPS. Depending on the language, the return value from the function call may or may not be useful. For example, most FORTRAN implementations allow a return value from a function but not from a subroutine. In these instances, the subroutine may be called for side effect only. As with defined functions written in C, the user must create an entry in **UserFunctions** for the subroutine (see section 3.1). An **extern** definition also must appear in the same file as the **UserFunctions** function, defining the type of data that the function will return. If the function does not return a value (Ada procedures or FORTRAN subroutines), it should be defined as returning a void value. See section 3.1 for the allowed return values for user-defined functions.

Ada Example

```
1:procedure DISPLAY is
2:--  Standard Ada definitions and declarations
3:begin
4:--
5:--  Any kind of normal Ada code may be used
6:--
          •
          •
          •
7:--
8:end DISPLAY;
```

FORTRAN Example

```
       subroutine display
□
C   Any kind of normal FORTRAN code may be used
C
        •
        •
        •
C
       return
       end
```

UserFunctions entry for either example

```
extern VOID display();

UserFunctions()
  {
   DefineFunction("display",'v',PTIF display,"display");
        •
        •
        •
   /* Any other user-defined functions. */
        •
        •
        •
  }
```

6.6 PASSING ARGUMENTS FROM CLIPS TO AN EXTERNAL FUNCTION

Arguments may be passed from CLIPS to an external function. CLIPS does not actually pass arguments to the function; instead arguments must be pulled from internal CLIPS buffers by using the functions described in section 3. Although the argument access functions could be called directly from Ada or FORTRAN, it probably is easier to write an interface function in C. CLIPS will call the C routine, which gathers the arguments and passes them in the proper manner to the external subprogram.

In this situation, the user must ensure argument compatibility. In particular, string variables must be converted from C arrays to FORTRAN or Ada string descriptors. The actual code used in the interface routine for argument conversion will depend on the language. Examples are given below for Ada and FORTRAN. Each example assumes the subroutine is called as follows:

```
(dummy 3.7 "An example string")
```

VMS Ada Example

Note the procedure definition in line 2 of the Ada routine. The numerical value is defined as an IN OUT type and the string as an IN. Also note the compiler PRAGMA on line 4-5. PRAGMA is DEC-Ada-specific, and a similar statement will be needed for other compilers. Following the Ada routine is an example of a C interface function that calls the Ada subroutine. The C routine

must convert a C string into an Ada string descriptor using the **MakeStringDsc** (see section 6.7 for more on string conversion) function as shown in line 16 of the C routine. Note that the C function passes the *address* of the numerical parameters to the Ada subprogram (line 16) and a pointer to a descriptor for the string parameter. Note also that the **UserFunctions** definition (lines 21-24) calls the dummy C routine, not the Ada program.

```
package DUMMY_PKG is
  procedure DUMMY (Value : in out float  ;
                   Name  : in     string);

     (The following two lines are DEC Ada specific)

  pragma EXPORT_PROCEDURE (DUMMY
                           PARAMETER_TYPES => (float,string));

end DUMMY_PKG;

-- Ada interface to CLIPS internal functions, see Appendix A
with CLIPS_INTERNALS; use CLIPS_INTERNALS;

PACKAGE Dummy_PKG IS

package body DUMMY_PKG is

 procedure DUMMY (Value : in out float ;
                  Name  : in     string) is

  begin
--  Value and Name may be used as normal Ada variables.
--  Name should not be modified by this procedure since
--  it has a direct pointer to a CLIPS C string.

  end DUMMY;

end DUMMY_PKG;
```

C interface routine

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


   (The following two lines are VAX VMS specific)

#include <descrip.h>
struct dsc$descriptor_s *MakeStringDsc();

c_dummy()
  {
   double value;
   char *name;
   extern int dummy();

   value = RtnDouble(1);
   name  = RtnLexeme(2);

   dummy(&value, MakeStringDsc(name));

   return(0);
  }

UserFunctions()
  {
   DefineFunction("dummy", 'i', c_dummy, "c_dummy");
  }
```

VMS FORTRAN Example

The VMS FORTRAN routine looks very similar to the Ada routine and, in fact, uses the same C interface function listed for VMS Ada.

```
      subroutine dummy(value, name)
C
      REAL value
      CHARACTER *80 name
C
C   value and name may now be used as normal FORTRAN variables
C
       •
       •
       •
C
      return
      end
```

Note that the previous two examples performed the string conversion in C, not in the language (Ada or FORTRAN) to which the string was being passed. On some machines, it may be easier to convert the string in the language (Ada or FORTRAN) to which the string is being passed rather than in the language (C) from which the string is being passed.

6.7 STRING CONVERSION

Much of the information that needs to be passed between CLIPS and another language typically is stored as strings. The storage of string variables can differ radically between languages. Both Ada and FORTRAN use a special (machine-dependent) string descriptor for string data types, whereas C uses simple arrays. Because of this difference, special functions must be defined to convert FORTRAN or Ada strings to C strings and back. The implementation of these functions will be different for every language and computer. Typically, two functions are needed: one to convert an Ada or a FORTRAN string to a C string, and one to convert a C string to an Ada or a FORTRAN string descriptor. When converting C strings that have been provided by CLIPS to strings suitable for other languages, *do not* modify the original C string. The following table shows the string conversion routines provided in the interface packages in appendix A.

Environment	Function to Convert TO a C string	Function to Convert FROM a C string
VMS Ada	ADA_TO_C_STRING	MakeStringDsc
VMS FORTRAN	CONVERT_TO_C_STRING	MakeStringDsc

The interface package does all of the converting from Ada or FORTRAN strings to C strings. Users will have to convert from C when defining functions that are passed parameters from CLIPS. Appendix A.3 has a listing for a function that will convert C strings to Ada or FORTRAN character strings under VAX VMS.

6.8 COMPILING AND LINKING

After all routines are defined, they must be compiled and linked to execute. The manner of compilation will depend on the machine on which the user is working. Two examples are given below: one for VMS Ada and one for VMS FORTRAN.

6.8.1 VMS Ada Version

1) Copy all of the CLIPS include files and Ada interface package to the user directory.

$copy [{CLIPS master directory}**]*.h [**{user directory}**]**
$copy [{CLIPS master directory}**]*.ada [**{user directory}**]**

2) Create an object file from the file holding the **UserFunctions** definition.

$cc usrfuncs.c

3) Set up the Ada library and compile the Ada routine(s).

$acs create library [{user directory}**.adalib]**
$acs set library [{user directory}**.adalib]**
$ada {Ada files, including the interface packages}

4) Export the Ada object code from the DEC ACS library.

$acs export/main {Ada files, including the interface package}

5) Define the link libraries and link all of the files together. Note that, prior to linking, each user must define the standard link libraries with the **define lnk$library** command. This usually is done once in the **login.com** file during login. This definition may be different for each VMS system.

$link/executable={exec name} {Ada files}, **usrfuncs, [**{CLIPS master directory}**] clipslib/library**

This will create an embedded version of CLIPS using an Ada routine as the main program. To create a program that uses the CLIPS interface but calls Ada subprograms, modify step 4 to read

$acs export {user's Ada packages}

5) Copy the CLIPS **main.c** file from the CLIPS master directory and remove the **UserFunctions** definition from the CLIPS **main.c** routine. Then recompile

$cc main

6) Link with the following command:

$link/executable={exec name} {Ada files}**, main, usrfuncs , [**{CLIPS master directory}**] clipslib/library**

6.8.2 VMS FORTRAN Version

1) Copy all of the CLIPS include files to the user directory.

$copy [{CLIPS master directory}**]*.h [**{user directory}**]**

2) Create an object file from the file holding the **UserFunctions** definition.

$cc usrfuncs.c

3) Compile the FORTRAN routine(s).

$fortran {FORTRAN files}

4) Link all of the files together.

$link/executable={exec name} {FORTRAN files}, **usrfuncs, [**{CLIPS master directory}**] clipslib/library, clipsforlib/library**

Note that one of the FORTRAN programs *must* be a main program.

6.8.3 CLIPS Library

All of the previous examples assume a CLIPS library has been created on the user's machine. A CLIPS library can be made with any standard object code library program and should include all of the CLIPS object code files *except* the **main.c** file. A library also may be made for the interface packages.

6.9 BUILDING AN INTERFACE PACKAGE

To develop an interface package for CLIPS and FORTRAN, Ada, or any other language, the primary need is the string conversion routines. Once these have been developed, the rest of the interface package should look very similar to the examples shown in appendices A.1 to A.3. The majority of the conversion work should be done in the interface package. Note that if a CLIPS function takes no arguments then it is not necessary to write an interface function for it. For example, the function ListFacts takes no arguments and has no return value and can therefore be called directly (however, some languages, such as Ada, will require the function to be declared). The Ada listing in appendix A.1 use pragmas to map the C ListFacts function to the Ada xListFacts function (for consistency with the other functions which are proceeded by an x). The FORTRAN listings in appendix A include interface routines to function which do not require them as well. The functions listed in appendix A also directly mimic the equivalent C functions. That is, functions which return the integer 0 or 1 in C have the exact same value returned by their Ada and FORTRAN counterparts (rather than a boolean or logical value). It would normally be more useful to directly map these integers values into their boolean counterparts (TRUE or FALSE) in the other language.

Section 7 - I/O Router System

The **I/O router** system provided in CLIPS is quite flexible and will allow a wide variety of interfaces to be developed and easily attached to CLIPS. The system is relatively easy to use and is explained fully in sections 7.1 through 7.4. The CLIPS I/O functions for using the router system are described in sections 7.5 and 7.6, and finally, in appendix B, some examples are included which show how I/O routing could be used for simple interfaces.

7.1 INTRODUCTION

The problem that originally inspired the idea of I/O routing will be considered as an introduction to I/O routing. Because CLIPS was designed with portability as a major goal, it was not possible to build a sophisticated user interface that would support many of the features found in the interfaces of commercial expert system building tools. A prototype was built of a semi-portable interface for CLIPS using the CURSES screen management package. Many problems were encountered during this effort involving both portability concerns and CLIPS internal features. For example, every statement in the source code which used the C print function, **printf**, for printing to the terminal had to be replaced by the CURSES function, **wprintw**, which would print to a window on the terminal. In addition to changing function call names, different types of I/O had to be directed to different windows. The tracing information was to be sent to one window, the command prompt was to appear in another window, and output from printout statements was to be sent to yet another window.

This prototype effort pointed out two major needs: First, the need for generic I/O functions that would remain the same regardless of whether I/O was directed to a standard terminal interface or to a more complex interface (such as windows); and second, the need to be able to specify different sources and destinations for I/O. I/O routing was designed in CLIPS to handle these needs. The concept of I/O routing will be further explained in the following sections.

7.2 LOGICAL NAMES

One of the key concepts of I/O routing is the use of **logical names**. An analogy will be useful in explaining this concept. Consider the Acme company which has two computers: computers X and Y. The Acme company stores three data sets on these two computers: a personnel data set, an accounting data set, and a documentation data set. One of the employees, Joe, wishes to update the payroll information in the accounting data set. If the payroll information was located in directory A on computer Y, Joe's command would be

```
update Y:[A]payroll
```

If the data were moved to directory B on computer X, Joe’s command would have to be changed to

```
update X:[B]payroll
```

To update the payroll file, Joe must know its location. If the file is moved, Joe must be informed of its new location to be able to update it. From Joe's point of view, he does not care where the file is located physically. He simply wants to be able to specify that he wants the information from the accounting data set. He would rather use a command like

```
update accounting:payroll
```

By using logical names, the information about where the accounting files are located physically can be hidden from Joe while still allowing him to access them. The locations of the files are equated with logical names as shown here.

```
accounting     =  X:[A]
documentation  =  X:[C]
personnel      =  Y:[B]
```

Now, if the files are moved, Joe does not have to be informed of their relocation so long as the logical names are updated. This is the power of using logical names. Joe does not have to be aware of the physical location of the files to access them; he only needs to be aware that accounting is the logical name for the location of the accounting data files. Logical names allow reference to an object without having to understand the details of the implementation of the reference.

In CLIPS, logical names are used to send I/O requests without having to know which device and/or function is handling the request. Consider the message that is printed in CLIPS when rule tracing is turned on and a rule has just fired. A typical message would be

```
FIRE    1 example-rule:  f-0
```

The routine that requests this message be printed should not have to know where the message is being sent. Different routines are required to print this message to a standard terminal, a window interface, or a printer. The tracing routine should be able to send this message to a logical name (for example, **trace-out**) and should not have to know if the device to which the message is being sent is a terminal or a printer. The logical name **trace-out** allows tracing information to be sent simply to "the place where tracing information is displayed." In short, logical names allow I/O requests to be sent to specific locations without having to specify the details of how the I/O request is to be handled.

Many functions in CLIPS make use of logical names. Both the **printout** and **format** functions require a logical name as their first argument. The **read** function can take a logical name as an optional argument. The **open** function causes the association of a logical name with a file, and the **close** function removes this association.

Several logical names are predefined by CLIPS and are used extensively throughout the system code. These are

Name	Description
stdin	The default for all user inputs. The **read** and **readline** functions read from **stdin** if **t** is specified as the logical name.
stdout	The default for all user outputs. The **format** and **printout** functions send output to **stdout** if **t** is specified as the logical name.
wclips	The CLIPS prompt is sent to this logical name.
wdialog	All informational messages are sent to this logical name.
wdisplay	Requests to display CLIPS information, such as facts or rules, are sent to this logical name.
werror	All error messages are sent to this logical name.
wwarning	All warning messages are sent to this logical name.
wtrace	All watch information is sent to this logical name (with the exception of compilations which is sent to wdialog).

7.3 ROUTERS

The use of logical names has solved 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 question each router to discover whether it can handle an I/O request. Routers with high priorities are questioned before routers with low priorities. Priorities are very important when dealing with one or more routers that can each process the same I/O request. This is particularly true when a router is going to redefine the standard user interface. The router associated with the standard interface will handle the same I/O requests as the new router; but, if the new router is given a higher priority, the standard router will never receive any I/O requests. The new router will "intercept" all of the I/O requests. Priorities will be discussed in more detail in the next section.

The third component of a router consists of the functions which actually handle an I/O request. These include functions for printing strings, getting a character from an input buffer, returning a character to an input buffer, and a function to clean up (e.g., close files, remove windows) when CLIPS is exited.

7.4 ROUTER PRIORITIES

Each I/O router has a priority. Priority determines which routers are queried first when determining the router that will handle an I/O request. Routers with high priorities are queried before routers with low priorities. Priorities are assigned as integer values (the higher the integer, the higher the priority). Priorities are important because more than one router can handle an I/O request for a single logical name, and they enable the user to define a custom interface for CLIPS. For example, the user could build a custom router which handles all logical names normally handled by the default router associated with the standard interface. The user adds the custom router with a priority higher than the priority of the router for the standard interface. The custom router will then intercept all I/O requests intended for the standard interface and specially process those requests to the custom interface.

Once the router system sends an I/O request out to a router, it considers the request satisfied. If a router is going to share an I/O request (i.e., process it) then allow other routers to process the request also, that router must deactivate itself and call **PrintCLIPS** again. These types of routers should use a priority of either 30 or 40. An example is given in appendix B.2.

Priority	**Router Description**
50	Any router that uses "unique" logical names and does not want to share I/O with catch-all routers.
40	Any router that wants to grab standard I/O and is willing to share it with other routers. A dribble file is a good example of this type of router. The dribble file router needs to grab all output that normally would go to the terminal so it can be placed in the dribble file, but this same output also needs to be sent to the router which displays output on the terminal.

30	Any router that uses "unique" logical names and is willing to share I/O with catch-all routers.
20	Any router that wants to grab standard logical names and is not willing to share them with other routers.
10	This priority is used by a router which redefines the default user interface I/O router. Only one router should use this priority.
0	This priority is used by the default router for handling standard and file logical names. Other routers should not use this priority.

7.5 INTERNAL I/O FUNCTIONS

The following functions are called internally by CLIPS. These functions search the list of active routers and determine which router should handle an I/O request. Some routers may wish to deactivate themselves and call one of these functions to allow the next router to process an I/O request. Prototypes for these functions can be included by using the **clips.h** header file or the **router.h** header file.

7.5.1 ExitCLIPS

```
VOID ExitCLIPS(exitCode);
int exitCode;
```

Purpose:	The function **ExitCLIPS** calls the exit function associated with each active router before exiting CLIPS.
Arguments:	The *exitCode* argument corresponds to the value that normally would be sent to the system **exit** function. Consult a C system manual for more details on the meaning of this argument.
Returns:	No meaningful return value.
Info:	The function **ExitCLIPS** calls the system function **exit** with the argument num after calling all exit functions associated with I/O routers.

7.5.2 GetcCLIPS

```
int   GetcCLIPS(logicalName);
char *logicalName;
```

Purpose: The function **GetcCLIPS** queries all active routers until it finds a router that recognizes the logical name associated with this I/O request to get a character. It then calls the get character function associated with that router.

Arguments: The logical name associated with the get character I/O request.

Returns: An integer; the ASCII code of the character.

Info: This function should be used by any user-defined function in place of **getc** to ensure that character input from the function can be received from a custom interface. On machines which default to unbuffered I/O, user code should be prepared to handle special characters like the backspace.

7.5.3 PrintCLIPS

```
int   PrintCLIPS(logicalName,str);
char *logicalName, *str;
```

Purpose: The function **PrintCLIPS** queries all active routers until it finds a router that recognizes the logical name associated with this I/O request to print a string. It then calls the print function associated with that router.

Arguments:
1) The logical name associated with the location at which the string is to be printed.
2) The string that is to be printed.

Returns: Returns a non-zero value if the logical name is recognized, otherwise it returns zero.

Info: This function should be used by any user-defined function in place of **printf** to ensure that output from the function can be sent to a custom interface.

7.5.4 UngetcCLIPS

```
int   UngetcCLIPS(ch,logicalName);
int   ch;
char *logicalName;
```

Purpose: The function **UngetcCLIPS** queries all active routers until it finds a router that recognizes the logical name associated with this I/O request. It then calls the ungetc function associated with that router.

Arguments:
1) The ASCII code of the character to be returned.
2) The logical name associated with the ungetc character I/O request.

Returns: Returns *ch* if successful, otherwise -1.

Info: This function should be used by any user-defined function in place of **UngetcCLIPS** to ensure that character input from the function can be received from a custom interface. As with **GetcCLIPS**, user code should be prepared to handle special characters like the backspace on machines with unbuffered I/O.

7.6 ROUTER HANDLING FUNCTIONS

The following functions are used for creating, deleting, and handling I/O routers. They are intended for use within user-defined functions. Prototypes for these functions can be included by using the **clips.h** header file or the **router.h** header file.

7.6.1 ActivateRouter

```
int   ActivateRouter(routerName);
char *routerName;
```

Purpose: The function **ActivateRouter** activates an existing I/O router. This router will be queried to see if it can handle an I/O request. Newly created routers do not have to be activated.

Arguments: The name of the I/O router to be activated.

Returns: Returns a non-zero value if the logical name is recognized, otherwise it returns zero.

7.6.2 AddRouter

```
int AddRouter(routerName,priority,queryFunction,printFunction,
              getcFunction,ungetcFunction,exitFunction);

char *routerName;
int priority;
int (*queryFunction)(), (*printFunction)();
int (*getcFunction)(), (*ungetcFunction)(), (*exitFunction)();

int queryFunction(logicalName);
int printFunction(logicalName,str);
int getcFunction(logicalName);
int ungetcFunction(ch,logicalName);
int exitFunction(exitCode);

char *logicalName, *str, ch;
int   exitCode;
```

Purpose: The function **AddRouter** adds a new I/O router to the list of I/O routers.

Arguments:

1) The name of the I/O router. This name is used to reference the router by the other I/O router handling functions.
2) The priority of the I/O router. I/O routers are queried in descending order of priorities.
3) A pointer to the query function associated with this router. This query function should accept a single argument, a logical name, and return either TRUE (1) or FALSE (0) depending upon whether the router recognizes the logical name.
4) A pointer to the print function associated with this router. This print function should accept two arguments: a logical name and a character string. The return value of the print function is not meaningful.
5) A pointer to the get character function associated with this router. The get character function should accept a single argument, a logical name. The return value of the get character function should be an integer which represents the character or end of file (EOF) read from the source represented by logical name.
6) A pointer to the ungetc character function associated with this router. The ungetc character function accepts two arguments: a logical name and a character. The return value of the unget character function should be an integer which represents the character which was passed to it as an argument if the ungetc is successful or end of file (EOF) is the ungetc is not successful.

7) A pointer to the exit function associated with this router. The exit function should accept a single argument: the exit code represented by num.

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

Info: I/O routers are active upon being created. See the examples in appendix B for further information on how to use this function.

7.6.3 DeactivateRouter

```
int   DeactivateRouter(routerName);
char *routerName;
```

Purpose: The function **DeactivateRouter** deactivates an existing I/O router. This router will not be queried to see if it can handle an I/O request. The syntax of the **DeactivateRouter** function is as follows.

Arguments: The name of the I/O router to be deactivated.

Returns: Returns a non-zero value if the logical name is recognized, otherwise it returns zero.

7.6.4 DeleteRouter

```
int   DeleteRouter(routerName);
char *routerName;
```

Purpose: The function **DeleteRouter** removes an existing I/O router from the list of I/O routers.

Arguments: The name of the I/O router to be deleted.

Returns: Returns a non-zero value if the logical name is recognized, otherwise it returns zero.

Section 8 - Memory Management

Efficient use of memory is a very important aspect of an expert system tool. Expert systems are highly memory intensive and require comparatively large amounts of memory. To optimize both storage and processing speed, CLIPS does much of its own memory management. Section 8.1 describes the basic memory management scheme used in CLIPS. Section 8.2 describes some functions that may be used to monitor/ control memory usage.

8.1 HOW CLIPS USES MEMORY

The CLIPS internal data structures used to represent constructs and other data entities require the allocation of dynamic memory to create and execute. Memory can also be released as these data structures are no longer needed and are removed. All requests, either to allocate memory or to free memory, are routed through the CLIPS memory management functions. These functions request memory from the operating system and store previously used memory for reuse. By providing its own memory management, CLIPS is able to reduce the number of **malloc** calls to the operating system. This is very important since **malloc** calls are handled differently on each machine, and some implementations of **malloc** are very inefficient.

When new memory is needed by any CLIPS function, CLIPS first checks its own data buffers for a pointer to a free structure of the type requested. If one is found, the stored pointer is returned. Otherwise, a call is made to **malloc** for the proper amount of data and a new pointer is returned.

When a data structure is no longer needed, CLIPS saves the pointer to that memory against the next request for a structure of that type. Memory actually is released to the operating system only under limited circumstances. If a **malloc** call in a CLIPS function returns NULL,*all* free memory internally stored by CLIPS is released to the operating system and the **malloc** call is tried again. This usually happens during rule execution, and the message

```
*** DEALLOCATING MEMORY ***

*** MEMORY  DEALLOCATED ***
```

will be printed out to the **wdialog** stream. Users also may force memory to be released to the operating system (see section 8.2).

CLIPS uses the generic C function **malloc** to request memory. Some machines provide lower-level memory allocation/deallocation functions that are considerably faster than **malloc**. Generic CLIPS memory allocation and deallocation functions are stored in the **memory.c** file and are called **genalloc** and **genfree**. The call to **malloc** and **free** in these functions could be replaced to improve performance on a specific machine.

Some machines have very inefficient memory management services. When running on the such machines, CLIPS can be made to request very large chunks of memory and internally allocate smaller chunks of memory from the larger chunks. This technique bypasses numerous calls to **malloc** thus improving performance. This behavior can be enabled by setting the BLOCK_MEMORY compiler option in the **setup.h** header file to 1 (see section 2.2). In general,