



Reference Manual

Volume II Advanced Programming Guide

Version 6.23

January 31st 2005

CLIPS Advanced Programming Guide

Version 6.23 January 31st 2005

CONTENTS

License Information.....	i
Preface.....	iii
Acknowledgements.....	vii
Section 1 - Introduction.....	1
1.1 Warning About Interfacing With CLIPS	1
1.2 C++ Compatibility	2
1.3 Threads and Concurrency	2
1.4 Garbage Collection	3
Section 2 - Installing and Tailoring CLIPS.....	9
2.1 Installing CLIPS.....	9
2.1.1 Additional Considerations.....	12
2.2 Tailoring CLIPS.....	13
Section 3 - Integrating CLIPS with External Functions.....	21
3.1 Declaring User-Defined External Functions	21
3.2 Passing Arguments from CLIPS to External Functions.....	25
3.2.1 Determining the Number of Passed Arguments	25
3.2.2 Passing Symbols, Strings, Instance Names, Floats, and Integers	25
3.2.3 Passing Unknown Data Types	27
3.2.4 Passing Multifield Values	30
3.3 Returning Values To CLIPS From External Functions	32
3.3.1 Returning Symbols, Strings, and Instance Names	32
3.3.2 Returning Boolean Values	34
3.3.3 Returning External Addresses and Instance Addresses	35
3.3.4 Returning Unknown Data Types.....	36
3.3.5 Returning Multifield Values	38
3.4 User-Defined Function Example.....	42
Section 4 - Embedding CLIPS.....	45
4.1 Environment Functions.....	45
4.1.1 AddClearFunction.....	45
4.1.2 AddPeriodicFunction	46
4.1.3 AddResetFunction.....	46
4.1.4 BatchStar.....	47
4.1.5 Bload	47

4.1.6 Bsave.....	48
4.1.7 Build.....	48
4.1.8 Clear.....	48
4.1.9 Eval.....	49
4.1.10 FunctionCall.....	49
4.1.11 GetAutoFloatDividend.....	50
4.1.12 GetDynamicConstraintChecking.....	50
4.1.13 GetSequenceOperatorRecognition.....	50
4.1.14 GetStaticConstraintChecking.....	50
4.1.15 InitializeEnvironment.....	51
4.1.16 Load.....	51
4.1.17 RemoveClearFunction.....	52
4.1.18 RemovePeriodicFunction.....	52
4.1.19 RemoveResetFunction.....	52
4.1.20 Reset.....	53
4.1.21 Save.....	53
4.1.22 SetAutoFloatDividend.....	53
4.1.23 SetDynamicConstraintChecking.....	53
4.1.24 SetSequenceOperator Recognition.....	54
4.1.25 SetStaticConstraintChecking.....	54
4.2 Debugging Functions.....	55
4.2.1 DribbleActive.....	55
4.2.2 DribbleOff.....	55
4.2.3 DribbleOn.....	55
4.2.4 GetWatchItem.....	56
4.2.5 Unwatch.....	56
4.2.6 Watch.....	56
4.3 Deftemplate Functions.....	57
4.3.1 DeftemplateModule.....	57
4.3.2 FindDeftemplate.....	57
4.3.3 GetDeftemplateList.....	57
4.3.4 GetDeftemplateName.....	58
4.3.5 GetDeftemplatePPForm.....	58
4.3.6 GetDeftemplateWatch.....	58
4.3.7 GetNextDeftemplate.....	59
4.3.8 IsDeftemplateDeletable.....	59
4.3.9 ListDeftemplates.....	59
4.3.10 SetDeftemplateWatch.....	60
4.3.11 Undeftemplate.....	60
4.4 Fact Functions.....	60
4.4.1 Assert.....	60
4.4.2 AssertString.....	61
4.4.3 AssignFactSlotDefaults.....	62

4.4.4 CreateFact	62
4.4.5 DecrementFactCount	65
4.4.6 FactDeftemplate	65
4.4.7 FactExistp.....	65
4.4.8 FactIndex.....	66
4.4.9 Facts	66
4.4.10 FactSlotNames	67
4.4.11 GetFactDuplication	67
4.4.12 GetFactList.....	67
4.4.13 GetFactListChanged.....	68
4.4.14 GetFactPPForm.....	68
4.4.15 GetFactSlot.....	68
4.4.16 GetNextFact	69
4.4.17 GetNextFactInTemplate.....	69
4.4.18 IncrementFactCount.....	70
4.4.19 LoadFacts	71
4.4.20 LoadFactsFromString.....	71
4.4.21 PutFactSlot.....	71
4.4.22 Retract	72
4.4.23 SaveFacts	72
4.4.24 SetFactDuplication.....	73
4.4.25 SetFactListChanged	73
4.5 Deffacts Functions	74
4.5.1 DeffactsModule.....	74
4.5.2 FindDeffacts.....	74
4.5.3 GetDeffactsList	74
4.5.4 GetDeffactsName.....	75
4.5.5 GetDeffactsPPForm	75
4.5.6 GetNextDeffacts.....	75
4.5.7 IsDeffactsDeletable.....	76
4.5.8 ListDeffacts.....	76
4.5.9 Undeffacts	76
4.6 Defrule Functions.....	77
4.6.1 DefruleHasBreakpoint	77
4.6.2 DefruleModule	77
4.6.3 FindDefrule	77
4.6.4 GetDefruleList	77
4.6.5 GetDefruleName	78
4.6.6 GetDefrulePPForm.....	78
4.6.7 GetDefruleWatchActivations.....	78
4.6.8 GetDefruleWatchFirings.....	79
4.6.9 GetIncrementalReset.....	79
4.6.10 GetNextDefrule.....	79

4.6.11 IsDefruleDeletable	80
4.6.12 ListDefrules.....	80
4.6.13 Matches	80
4.6.14 Refresh	81
4.6.15 RemoveBreak.....	81
4.6.16 SetBreak.....	81
4.6.17 SetDefruleWatchActivations	81
4.6.18 SetDefruleWatchFirings.....	82
4.6.19 SetIncrementalReset.....	82
4.6.20 ShowBreaks	82
4.6.21 Undefrule	83
4.7 Agenda Functions	83
4.7.1 AddRunFunction.....	83
4.7.2 Agenda	84
4.7.3 ClearFocusStack.....	84
4.7.4 DeleteActivation	85
4.7.5 Focus	85
4.7.6 GetActivationName	85
4.7.7 GetActivationPPForm.....	86
4.7.8 GetActivationSalience	86
4.7.9 GetAgendaChanged	86
4.7.10 GetFocus	87
4.7.11 GetFocusStack	87
4.7.12 GetNextActivation	87
4.7.13 GetSalienceEvaluation.....	87
4.7.14 GetStrategy.....	88
4.7.15 ListFocusStack.....	88
4.7.16 PopFocus.....	88
4.7.17 RefreshAgenda.....	89
4.7.18 RemoveRunFunction	89
4.7.19 ReorderAgenda	89
4.7.20 Run.....	90
4.7.21 SetActivationSalience	90
4.7.22 SetAgendaChanged.....	90
4.7.23 SetSalienceEvaluation.....	91
4.7.24 SetStrategy	91
4.8 Defglobal Functions.....	91
4.8.1 DefglobalModule	92
4.8.2 FindDefglobal	92
4.8.3 GetDefglobalList.....	92
4.8.4 GetDefglobalName	93
4.8.5 GetDefglobalPPForm.....	93
4.8.6 GetDefglobalValue	93

4.8.7 GetDefglobalValueForm.....	93
4.8.8 GetDefglobalWatch	94
4.8.9 GetGlobalsChanged	94
4.8.10 GetNextDefglobal	94
4.8.11 GetResetGlobals.....	95
4.8.12 IsDefglobalDeletable	95
4.8.13 ListDefglobals.....	95
4.8.14 SetDefglobalValue.....	96
4.8.15 SetDefglobalWatch	96
4.8.16 SetGlobalsChanged.....	96
4.8.17 SetResetGlobals	97
4.8.18 ShowDefglobals.....	97
4.8.19 Undefglobal.....	97
4.9 Deffunction Functions.....	98
4.9.1 DeffunctionModule.....	98
4.9.2 FindDeffunction	98
4.9.3 GetDeffunctionList	98
4.9.4 GetDeffunctionName.....	99
4.9.5 GetDeffunctionPPForm	99
4.9.6 GetDeffunctionWatch	99
4.9.7 GetNextDeffunction.....	100
4.9.8 IsDeffunctionDeletable	100
4.9.9 ListDeffunctions.....	100
4.9.10 SetDeffunctionWatch.....	101
4.9.11 Undeffunction	101
4.10 Defgeneric Functions	101
4.10.1 DefgenericModule	101
4.10.2 FindDefgeneric.....	102
4.10.3 GetDefgenericList.....	102
4.10.4 GetDefgenericName.....	102
4.10.5 GetDefgenericPPForm	103
4.10.6 GetDefgenericWatch.....	103
4.10.7 GetNextDefgeneric	103
4.10.8 IsDefgenericDeletable.....	104
4.10.9 ListDefgenerics	104
4.10.10 SetDefgenericWatch	104
4.10.11 Undefgeneric.....	105
4.11 Defmethod Functions.....	105
4.11.1 GetDefmethodDescription	105
4.11.2 GetDefmethodList.....	106
4.11.3 GetDefmethodPPForm.....	106
4.11.4 GetDefmethodWatch	106
4.11.5 GetMethodRestrictions	107

4.11.6	GetNextDefmethod	107
4.11.7	IsDefmethodDeletable	108
4.11.8	ListDefmethods.....	108
4.11.9	SetDefmethodWatch	108
4.11.10	Undefmethod.....	109
4.12	Defclass Functions	109
4.12.1	BrowseClasses	109
4.12.2	ClassAbstractP	109
4.12.3	ClassReactiveP.....	110
4.12.4	ClassSlots.....	110
4.12.5	ClassSubclasses.....	110
4.12.6	ClassSuperclasses.....	111
4.12.7	DefclassModule	111
4.12.8	DescribeClass.....	112
4.12.9	FindDefclass.....	112
4.12.10	GetClassDefaultsMode	112
4.12.11	GetDefclassList.....	113
4.12.12	GetDefclassName.....	113
4.12.13	GetDefclassPPForm	113
4.12.14	GetDefclassWatchInstances.....	114
4.12.15	GetDefclassWatchSlots.....	114
4.12.16	GetNextDefclass	114
4.12.17	IsDefclassDeletable.....	114
4.12.18	ListDefclasses	115
4.12.19	SetClassDefaultsMode	115
4.12.20	SetDefclassWatchInstances	115
4.12.21	SetDefclassWatchSlots	116
4.12.22	SlotAllowedValues	116
4.12.23	SlotCardinality	116
4.12.24	SlotDirectAccessP.....	117
4.12.25	SlotExistP.....	117
4.12.26	SlotFacets.....	117
4.12.27	SlotInitableP.....	118
4.12.28	SlotPublicP.....	118
4.12.29	SlotRange.....	118
4.12.30	SlotSources.....	119
4.12.31	SlotTypes	119
4.12.32	SlotWritableP.....	120
4.12.33	SubclassP	120
4.12.34	SuperclassP	120
4.12.35	Undefclass.....	120
4.13	Instance Functions.....	121
4.13.1	BinaryLoadInstances.....	121

4.13.2 BinarySaveInstances	121
4.13.3 CreateRawInstance.....	122
4.13.4 DecrementInstanceCount	122
4.13.5 DeleteInstance.....	122
4.13.6 DirectGetSlot	123
4.13.7 DirectPutSlot.....	123
4.13.8 FindInstance	124
4.13.9 GetInstanceClass.....	124
4.13.10 GetInstanceName	124
4.13.11 GetInstancePPForm	125
4.13.12 GetInstancesChanged.....	125
4.13.13 GetNextInstance.....	125
4.13.14 GetNextInstanceInClass.....	126
4.13.15 GetNextInstanceInClassAndSubclasses.....	126
4.13.16 IncrementInstanceCount	127
4.13.17 Instances.....	129
4.13.18 LoadInstances.....	129
4.13.19 LoadInstancesFromString	129
4.13.20 MakeInstance	130
4.13.21 RestoreInstances.....	130
4.13.22 RestoreInstancesFromString	131
4.13.23 SaveInstances.....	131
4.13.24 Send.....	131
4.13.25 SetInstancesChanged	132
4.13.26 UnmakeInstance.....	132
4.13.27 ValidInstanceAddress	133
4.14 Defmessage-handler Functions	133
4.14.1 FindDefmessageHandler.....	133
4.14.2 GetDefmessageHandlerList	134
4.14.3 GetDefmessageHandlerName.....	134
4.14.4 GetDefmessageHandlerPPForm	134
4.14.5 GetDefmessageHandlerType	135
4.14.6 GetDefmessageHandlerWatch.....	135
4.14.7 GetNextDefmessageHandler.....	135
4.14.8 IsDefmessageHandlerDeletable	136
4.14.9 ListDefmessageHandlers	136
4.14.10 PreviewSend.....	137
4.14.11 SetDefmessageHandlerWatch.....	137
4.14.12 UndefmessageHandler	137
4.15 Definstances Functions	138
4.15.1 DefinstancesModule.....	138
4.15.2 FindDefinstances.....	138
4.15.3 GetDefinstancesList.....	138

4.15.4 GetDefinstancesName.....	139
4.15.5 GetDefinstancesPPForm.....	139
4.15.6 GetNextDefinstances.....	139
4.15.7 IsDefinstancesDeletable.....	140
4.15.8 ListDefinstances.....	140
4.15.9 Undefinstances.....	140
4.16 Defmodule Functions.....	141
4.16.1 FindDefmodule.....	141
4.16.2 GetCurrentModule.....	141
4.16.3 GetDefmoduleList.....	141
4.16.4 GetDefmoduleName.....	142
4.16.5 GetDefmodulePPForm.....	142
4.16.6 GetNextDefmodule.....	142
4.16.7 ListDefmodules.....	143
4.16.8 SetCurrentModule.....	143
4.17 Embedded Application Examples.....	143
4.17.1 User-Defined Functions.....	143
4.17.2 Manipulating Objects and Calling CLIPS Functions.....	145
Section 5 - Creating a CLIPS Run-time Program.....	149
5.1 Compiling the Constructs.....	149
Section 6 - Combining CLIPS with Languages Other Than C.....	153
6.1 Introduction.....	153
6.2 Ada and FORTRAN Interface Package Function List.....	153
6.3 Embedded CLIPS - Using an External Main Program.....	154
6.4 Asserting Facts into CLIPS.....	155
6.5 Calling a Subroutine from CLIPS.....	156
6.6 Passing Arguments from CLIPS to an External Function.....	157
6.7 String Conversion.....	160
6.8 Compiling and Linking.....	160
6.8.1 VMS Ada Version.....	160
6.8.2 VMS FORTRAN Version.....	161
6.8.3 CLIPS Library.....	162
6.9 Building an Interface Package.....	162
Section 7 - I/O Router System.....	163
7.1 Introduction.....	163
7.2 Logical Names.....	163
7.3 Routers.....	165
7.4 Router Priorities.....	166
7.5 Internal I/O Functions.....	167
7.5.1 ExitRouter.....	167
7.5.2 GetcRouter.....	167

7.5.3 PrintRouter	168
7.5.4 UngetcRouter	168
7.6 Router Handling Functions	169
7.6.1 ActivateRouter	169
7.6.2 AddRouter	170
7.6.3 DeactivateRouter	171
7.6.4 DeleteRouter	171
Section 8 - Memory Management	173
8.1 How CLIPS Uses Memory	173
8.2 Standard Memory Functions	174
8.2.1 GetConserveMemory	174
8.2.2 MemRequests	174
8.2.3 MemUsed	175
8.2.4 ReleaseMem	175
8.2.5 SetConserveMemory	176
8.2.6 SetOutOfMemoryFunction	176
Section 9 - Environments	179
9.1 Creating, selecting, and destroying environments	179
9.2 Environment Companion Functions	180
9.3 Standard Environment Functions	182
9.3.1 AddEnvironmentCleanupFunction	182
9.3.2 AllocateEnvironmentData	183
9.3.3 CreateEnvironment	184
9.3.4 DeallocateEnvironmentData	184
9.3.5 DestroyEnvironment	184
9.3.6 GetCurrentEnvironment	185
9.3.7 GetEnvironmentByIndex	185
9.3.8 GetEnvironmentData	185
9.3.9 GetEnvironmentIndex	186
9.3.10 SetCurrentEnvironment	186
9.3.11 SetCurrentEnvironmentByIndex	186
9.4 Environment Aware User-Defined Functions	187
9.5 Allocating Environment Data	188
9.6 Environment Globals	191
9.7 Other Considerations	191
Appendix A - Language Integration Listings	193
A.1 Ada Interface Package for CLIPS	193
A.2 FORTRAN Interface Package for VAX VMS	197
A.3 Function to Convert C Strings for VMS Ada or FORTRAN	200
Appendix B - I/O Router Examples	203

B.1 Dribble System	203
B.2 Better Dribble System	205
B.3 Batch System.....	206
B.4 Simple Window System	208
Appendix C - Update Release Notes	213
C.1 Version 6.23	213
C.2 Version 6.22	213
C.3 Version 6.21	214
C.4 Version 6.2	214
C.5 Version 6.1	215
C.6 Version 6.05	216
Index.....	217

License Information

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OF THIRD PARTY RIGHTS. IN NO EVENT SHALL THE AUTHORS BE LIABLE FOR ANY CLAIM, OR ANY SPECIAL INDIRECT OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

CLIPS is released as public domain software and as such you are under no obligation to pay for its use. However, if you derive commercial or monetary benefit from use of the software or just want to show support, please consider making a voluntary payment based on the worth of the software to you as compensation for the time and effort required to develop and maintain CLIPS. Payments can be made online at <http://order.kagi.com/?JKT>.

Preface

The History of CLIPS

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

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

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

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

Further enhancements transformed CLIPS from a training tool into a tool useful for the development and delivery of expert systems as well. Versions 4.0 and 4.1 of CLIPS, released

respectively in the summer and fall of 1987, featured greatly improved performance, external language integration, and delivery capabilities. Version 4.2 of CLIPS, released in the summer of 1988, was a complete rewrite of CLIPS for code modularity. Also included with this release were an architecture manual providing a detailed description of the CLIPS software architecture and a utility program for aiding in the verification and validation of rule-based programs. Version 4.3 of CLIPS, released in the summer of 1989, added still more functionality.

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

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

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

CLIPS Version 6.2

Version 6.2 of CLIPS contains two major enhancements. First, CLIPS now provides a mechanism which allows an embedded application to create multiple environments into which programs can be loaded. Second, an improved Windows 2000/XP CLIPS interface is now available and the Macintosh CLIPS interface has been enhanced to support MacOS X. For a detailed listing of differences between the 6.x releases of CLIPS, refer to appendix B of the *Basic Programming Guide* and appendix C of the *Advanced Programming Guide*.

CLIPS Documentation

Two documents are provided with CLIPS.

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

Acknowledgements

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

Many other individuals contributed to the design, development, review, and general support of CLIPS, including: Jack Aldridge, Carla 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, Bill Paseman, Cynthia Rathjen, Eric Raymond, Reza Razavipour, Marsha Renals, Monica Rua, Tim Saito, Michael Sullivan, Gregg Swietek, Eric Taylor, James Villarreal, Lui Wang, Bob Way, Jim Wescott, Charlie Wheeler, and Wes White.

Section 1 - Introduction

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

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

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

1.1 WARNING ABOUT INTERFACING WITH CLIPS

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

1.2 C++ COMPATIBILITY

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

1.3 THREADS AND CONCURRENCY

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

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

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

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

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

Warning

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

1.4 GARBAGE COLLECTION

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

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

The following example illustrates several important concepts:

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

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

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

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

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

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

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

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

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

The **gensym*** function entered at the command prompt returns the unique symbol *gen2*. This newly created symbol is assumed to be garbage until an outstanding reference to the symbol is

established. In this case, once the return value has been displayed and control returned to the command prompt, garbage collection is initiated as part of the command prompt loop and the data structures associated with the symbol can be deallocated,

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

```
main()
{
    DATA_OBJECT rtn;

    InitializeEnvironment();

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

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

If the values returned to an embedding program are never garbage collected, continuous execution would result in a program eventually running out of memory. CLIPS addresses this issue by automatically invoking garbage collection for the following embedded functions: **Assert**, **AssertString**, **Clear**, **DeleteInstance**, **DirectPutSlot**, **FunctionCall**, **MakeInstance**, **Reset**, **Send**, **SetDefglobalValue**, **Undefclass**, **Undeffacts**, **Undeffunction**, **Undefgeneric**, **Undefglobal**, **Undefinstances**, **Undefmethod**, **Undefrule**, **Undeftemplate**, and **UnmakeInstance**. Calling one of these functions will not garbage collect any data returned from that call, but it could garbage collect data returned from prior calls.

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

```
main()
{
    DATA_OBJECT rtn;
    char *str1, *str2;

    InitializeEnvironment();

    FunctionCall("gensym*", NULL, &rtn);
    str1 = DOTOString(rtn);

    /* Safe to refer to str1 here. */

    FunctionCall("gensym*", NULL, &rtn);
    str2 = DOTOString(rtn);

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

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

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

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

```
void      IncrementGCLocks();
void      DecrementGCLocks();
```

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

```
main()
{
    DATA_OBJECT rtn;
    char *str1, *str2;

    InitializeEnvironment();

    IncrementGCLocks();

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

    /* Safe to refer to str1 here. */

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

    /* Safe to refer to str1 here. */

    DecrementGCLocks();
}
```

In this case, the second call to **FunctionCall** can't garbage collect the string referenced by *str1*, so it is safe to refer to this string after the call. The same effect could also be achieved by moving the **IncrementGCLocks** call after the first call to **FunctionCall**.

The garbage collector should not be disabled indiscriminately as shown in the following example:

```
main()
{
    InitializeEnvironment();

    IncrementGCLocks();

    Load("mab.clp");
    Reset();
    Run(-1);

    DecrementGCLocks();
}
```

First, while calling **Reset** could trigger garbage collection on values returned to the embedding program, in this case there are no such values. Second, while **Load** and **Run** won't trigger garbage collection on values returned to the embedding program, they do trigger garbage collection to remove garbage generated during their execution. Garbage collection should only be disabled for brief periods of time. The primary execution of your program should occur with garbage collection enabled.

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

Section 2 - Installing and Tailoring CLIPS

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

2.1 INSTALLING CLIPS

CLIPS executables for DOS, Windows 95/98/NT/XP, and MacOS are available for download from the internet. See Appendix A in the *Basic Programming Guide* for details. To tailor CLIPS or to install it on another machine, the user must port the source code and create a new executable version.

Testing of CLIPS 6.23 included the following hardware/software environments:

- Dell Dimension 8250 running Windows XP Professional with DJGPP 3.21 (for creating 32-bit protected mode DOS applications), Microsoft Visual C++ 6.0, Borland C++ 5.0, and CodeWarrior 9.4.
- Apple iBook running MacOS X 10.3.7 using CodeWarrior 9.4 and Xcode 1.2.

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

1) Load the source code onto the user's system

The following C source files are necessary to set up the basic CLIPS system:

agenda.h	dffnxpsr.h	globlpsr.h	prccode.h
analysis.h	dfinsbin.h	immthpsr.h	prcdrfun.h
argaccs.h	dfinscmp.h	incrrset.h	prcdrpsr.h
bload.h	drive.h	inherpsr.h	prdcfun.h
bmathfun.h	ed.h	inscom.h	prntutil.h
bsave.h	emathfun.h	insfile.h	proflfun.h
classcom.h	engine.h	insfun.h	reorder.h
classexm.h	envrnmnt.h	insmngr.h	reteutil.h
classfun.h	evaluatn.h	insmoddp.h	retract.h
classinf.h	expressn.h	insmult.h	router.h
classini.h	exprnbin.h	inspsr.h	rulebin.h
classpsr.h	exprnops.h	insquery.h	rulebld.h

clips.h	exprnpsr.h	insqypsr.h	rulebsc.h
clsitpsr.h	extnfunc.h	iofun.h	rulecmp.h
cmptblty.h	factbin.h	lgcldpnd.h	rulecom.h
commlne.h	factbld.h	match.h	rulecstr.h
conscomp.h	factcmp.h	memalloc.h	ruledef.h
constant.h	factcom.h	miscfun.h	ruledlt.h
constrct.h	factfun.h	modulbin.h	rulelhs.h
constrnt.h	factgen.h	modulbsc.h	rulespr.h
crstrtgy.h	facthsh.h	modulcmp.h	scanner.h
cstrcbin.h	factlhs.h	moduldef.h	setup.h
cstrccmp.h	factmch.h	modulpsr.h	shrtlnkn.h
cstrccom.h	factmng.h	modulutl.h	sortfun.h
cstrepsr.h	factqpsr.h	msgcom.h	strngfun.h
cstrnbin.h	factqry.h	msgfun.h	strngrtr.h
cstrnchk.h	factprt.h	msgpass.h	symlbin.h
cstrncmp.h	factrete.h	msgpsr.h	symlcmp.h
cstrnops.h	factrhs.h	multifld.h	symbol.h
cstrnpsr.h	filecom.h	multifun.h	sysdep.h
cstrnutl.h	filertr.h	network.h	textpro.h
default.h	generate.h	objbin.h	tmpltbin.h
defs.h	genrcbin.h	objcmp.h	tmpltbsc.h
developr.h	genrccmp.h	object.h	tmpltcmp.h
dffctbin.h	genrccom.h	objrtbin.h	tmpltdef.h
dffctbsc.h	genrcexe.h	objrtbld.h	tmpltfun.h
dffctcmp.h	genrcfun.h	objrtcmp.h	tmpltlhs.h
dffctdef.h	genrcpsr.h	objrtfnx.h	tmpltpsr.h
dffctpsr.h	globlbin.h	objrtgen.h	tmpltrhs.h
dffnxbin.h	globlbsc.h	objrtmch.h	tmpltutl.h
dffnxcmp.h	globlcmp.h	parsefun.h	userdata.h
dffnxexe.h	globlcom.h	pattern.h	utility.h
dffnxfun.h	globldef.h	pprint.h	watch.h
agenda.c	drive.c	globlcom.c	pprint.c
analysis.c	edbasic.c	globldef.c	prccode.c
argaces.c	edmain.c	globlpsr.c	predrfun.c
bload.c	edmisc.c	immthpsr.c	predrpsr.c
bmthfun.c	edstruct.c	incrrset.c	prdcftun.c
bsave.c	edterm.c	inherpsr.c	prntutil.c
classcom.c	emathfun.c	inscom.c	proflfun.c
classexm.c	engine.c	insfile.c	reorder.c
classfun.c	envrnmnt.c	insfun.c	reteutil.c
classinf.c	evaluatn.c	insmng.h	retract.c
classini.c	expressn.c	insmoddp.c	router.c

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

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

2) **Modify all include statements (if necessary)**

All of the “.c” files and most of the “.h” files have #include statements. These #include statements may have to be changed to either match the way the compiler searches for include files or to include a different ".h" file.

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

Edit the `setup.h` file and set any special options. CLIPS uses 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.

If user-defined functions are needed, compile the source code for those functions as well and modify the `UserFunctions` or `EnvUserFunctions` 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*.

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.

DJGPP v3.21

The CLIPS 32 bit DOS executable for 386 or higher PCs was created using the free DJGPP C compiler. This executable can access memory beyond the DOS 640K limit, but your environment must have DMPI services available in order to run this executable. If you are

running Windows or OS/2, DPMI services are provided for you. Information on this compiler is available at the WWW URL <http://www.delorie.com/djgpp/>. A free DPMI server is also available at this location.

The built-in MicroEMACS editor will not work with this compiler, so the **EMACS_EDITOR** compiler directive in **setup.h** should be set to 0. With the compiler installed, the following command will compile CLIPS

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

UNIX

If the **EX_MATH** compiler directive is enabled, then the **-lm** option must be used when compiling CLIPS with the gcc command. Similarly, if the **EMACS_EDITOR** 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

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

2.2 TAILORING CLIPS

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

The first flag in **setup.h** indicates on what type of compiler/machine CLIPS is to run. The source code is sent out with the flag for GENERIC CLIPS turned on. When compiled in this mode, all system-dependent features of CLIPS are excluded and the program should run on any system. A number of other flags are available in this file, indicating the types of compilers/machines on which CLIPS has been compiled previously. If the user's implementation matches one of the available flags, set that flag to 1 and turn the **GENERIC** flag off (set it to 0). The code for most of the features controlled by the compiler/machine-type flag is in the **sysdep.c** file.

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

ALLOW_ENVIRONMENT_GLOBALS

If this flag is on, then global variables are used to track the current environment and environment indices. If this flag is off, then no global variables are used by CLIPS. If this is disabled, then **ENVIRONMENT_API_ONLY** will be automatically enabled and **EMACS_EDITOR** will be automatically disabled. This is on in the standard CLIPS executable.

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.

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.

BLOAD

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

BLOAD_AND_BSAVE

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

BLOAD_INSTANCES

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

BLOAD_ONLY

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

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.

BSAVE_INSTANCES

This flag controls the ability to save instances in binary format to a file via the **b~~save~~-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.

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.

CONSTRUCT_COMPILER

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

DEBUGGING_FUNCTIONS

This flag controls access to commands such as **agenda**, **facts**, **ppdefrule**, **ppdeffacts**, etc. This would be used to save some memory in **BLOAD_ONLY** or **RUN_TIME** systems. This flag is on in the standard CLIPS executable.

DEFFACTS_CONSTRUCT

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

DEFFUNCTION_CONSTRUCT

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

DEFGENERIC_CONSTRUCT

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

DEFGLOBAL_CONSTRUCT

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

DEFINSTANCES_CONSTRUCT

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

DEFMODULE_CONSTRUCT

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

DEFRULE_CONSTRUCT

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

DEFTEMPLATE_CONSTRUCT

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

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.

EMACS_EDITOR

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.

ENVIRONMENT_API_ONLY

If this flag is on, then the standard embedded functions require their first argument to be a generic pointer to an environment. This is off 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.

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.

FACT_SET_QUERIES

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

HELP_FUNCTIONS

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_DEFAULT** flag should be set to point to the full path name for the CLIPS help file. This is on in the standard CLIPS executable.

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.

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.

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.

INSTANCE_SET_QUERIES

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

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.

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.

MULTIFIELD_FUNCTIONS

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

OBJECT_SYSTEM

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

PROFILING_FUNCTIONS

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

RUN_TIME

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

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.

STRING_FUNCTIONS

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

TEXTPRO_FUNCTIONS

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

WINDOW_INTERFACE

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

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** or **EnvUserFunctions** (if the function is environment aware as described in section 9.4). 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:

```
void 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 = D0ToLong(temp); }
    else /* the type must be FLOAT */
        { firstNumber = (long) ceil(D0ToDouble(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 = D0ToLong(temp); }
    else if (GetType(temp) == FLOAT)
        { secondNumber = (long) ceil(D0ToDouble(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      GetDOBBegin(argument);
int      GetpDOBBegin(&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 **GetDOBBegin**, **GetpDOBBegin**, **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 ((GetMFTType(multifieldPtr,i) == STRING) ||
        (GetMFTType(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 = DOSToString(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 **TRUE** or **FALSE**. Alternatively, the function may be declare to return a SYMBOL type ('w') or UNKNOWN type ('u') and return the value of **FalseSymbol** or **TrueSymbol** macro.

```

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

#define FALSE 0
#define TRUE 1

void *FalseSymbol();
void *TrueSymbol();

```

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("positive1",'b',positive1,"positive1",
                    "11n");
*/

int positive1()
{
    DATA_OBJECT temp;

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

    if (ArgCountCheck("positive1",EXACTLY,1) == -1)
        { return(FALSE); }

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

    if (ArgTypeCheck("positive1",1,INTEGER_OR_FLOAT,&temp) == 0)
        { return(FALSE); }

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

```

```

    if (GetType(temp) == INTEGER)
        { if (D0ToLong(temp) <= 0L) return(FALSE); }
    else /* the type must be FLOAT */
        { if (D0ToDouble(temp) <= 0.0) return(FALSE); }

    return(TRUE);
}

```

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

```

#include "clips.h"

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

void *positivep2()
{
    DATA_OBJECT temp;

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

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

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

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

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

    if (GetType(temp) == INTEGER)
        { if (D0ToLong(temp) <= 0L) return(FalseSymbol()); }
    else /* the type must be FLOAT */
        { if (D0ToDouble(temp) <= 0.0) return(FalseSymbol()); }

    return(TrueSymbol());
}

```

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-XXXXXXX>

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(
    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,FalseSymbol());
    return;
}

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

if (! ArgTypeCheck("cube",1,INTEGER_OR_FLOAT,returnValuePtr))
{
    SetpType(returnValuePtr,SYMBOL);
    SetpValue(returnValuePtr,FalseSymbol());
    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;
Unsigned  size;
int  fieldPosition, type;
void *multifieldPtr;
void *value;

```

If a new multifield is to be created from an existing multifield, then the type and value of the existing multifield can be copied and the begin and end indices can be modified to obtain the appropriate subfields of the multifield value. If you wish to create a new multifield value that is not part of an existing multifield value, then use the function **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(
    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.          */
/*=====*/

SetpD0Begin(returnValuePtr,1);
SetpD0End(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(
    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 (GetpD0Begin(returnValuePtr) > GetpD0End(returnValuePtr))
    { return; }

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

    SetpD0Begin(returnValuePtr,GetpD0Begin(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(
    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.

```

void UserFunctions()
{
    extern void TripleNumber(DATA_OBJECT_PTR);

    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 **InitializeEnvironment** 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. This function must except an environment pointer if the environment companion function is used (see section 9.2).
- 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. This function must except an environment pointer if the environment companion function is used (see section 9.2).
- 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. This function must except an environment pointer if the environment companion function is used (see section 9.2).
- 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 BatchStar

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

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

Arguments: A string representing the name of the file.

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

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

4.1.5 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.

4.1.6 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.

4.1.7 Build

```
int    Build(constructString);
char  *constructString;
```

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

Arguments: 1) A string containing the construct to be added.

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

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

4.1.8 Clear

```
void Clear();
```

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

Arguments: None.

Returns: No meaningful return value.

Other: This function can trigger garbage collection.

4.1.9 Eval

```
int Eval(expressionString,&result);
char *expressionString;
DATA_OBJECT result;
```

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

Arguments:

- 1) A string containing the expression to be evaluated.
- 2) Caller's buffer for storing the result of the evaluation. See sections 3.2.3 and 3.2.4 for information on getting the value stored in a DATA_OBJECT.

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

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

4.1.10 FunctionCall

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

Other: This function can trigger garbage collection.

Example

```
DATA_OBJECT rtn;

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

4.1.11 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; FALSE (0) if the behavior is disabled and TRUE (1) if the behavior is enabled.

4.1.12 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; FALSE (0) if the behavior is disabled and TRUE (1) if the behavior is enabled.

4.1.13 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; FALSE (0) if the behavior is disabled and TRUE (1) if the behavior is enabled.

4.1.14 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; FALSE (0) if the behavior is disabled and TRUE (1) if the behavior is enabled.

4.1.15 InitializeEnvironment

```
void InitializeEnvironment();
```

- Purpose:** Initializes the CLIPS system. Must be called prior to any other CLIPS function call. NOTE: This function should be called only once.
- Arguments:** None.
- Returns:** No meaningful return value.

4.1.16 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 and no errors occurred while loading. If syntactic errors are in the constructs, **Load** still will attempt to read the entire file and error notices will be sent to **werror**.
- Other:** The **load** function is not available for use in run-time programs (since individual constructs can't be added or deleted). To execute different sets of constructs, the switching feature must be used in a run-time program (see section 5 for more details).

4.1.17 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.

4.1.18 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.

4.1.19 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.

4.1.20 Reset

```
void Reset();
```

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

Arguments: None.

Returns: No meaningful return value.

Other: This function can trigger garbage collection.

4.1.21 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.

4.1.22 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: TRUE (1) to enable the behavior and FALSE (0) to disable it.

Returns: Returns the old value for the behavior.

4.1.23 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: TRUE (1) to enable the behavior and FALSE (0) to disable it.

Returns: Returns the old value for the behavior.

4.1.24 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: TRUE (1) to enable the behavior and FALSE (0) to disable it.

Returns: Returns the old value for the behavior.

4.1.25 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: TRUE (1) to enable the behavior and FALSE (0) to disable it.

Returns: Returns the old value for the behavior.

4.2 DEBUGGING FUNCTIONS

The following function call controls the CLIPS debugging aids:

4.2.1 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 all 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

```
unsigned 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);
unsigned    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).
- Other:** This function can trigger garbage collection.

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).

Other: This function can trigger garbage collection.

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

4.4.2 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.

Other: This function can trigger garbage collection.

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

Examples

If the following deftemplate has been processed by CLIPS,

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

then the following fact

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

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

```
void AddExampleFact1()
{
    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(
    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 **Assert** 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;
```

```

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

IncrementGCLocks();

/*=====*/
/* 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"));
SetD0Begin(theValue,1);
SetD0End(theValue,2);

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

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

AssignFactSlotDefaults(newFact);

```

```

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

DecrementGCLocks();

/*=====*/
/* 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 FactDeftemplate

```

void *FactDeftemplate(factPtr);
void *factPtr;

```

Purpose: Returns the deftemplate associated with a fact.

Arguments: A generic pointer to a fact data structure.

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

4.4.7 FactExistp

```

long FactExistp(factPtr);
void *factPtr;

```

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

- Arguments:** 1) A generic pointer to a fact data structure.
- Returns:** An integer; zero (0) if the fact is not in the fact-list, otherwise a one (1).

4.4.8 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).

4.4.9 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.

4.4.10 FactSlotNames

```
void FactSlotNames(factPtr,&theValue);
DATA_OBJECT slotNames;
```

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

Arguments:

- 1) A generic pointer to a fact data structure.
- 2) A pointer to a DATA_OBJECT in which to place a multifield value containing the fact's slot name symbols. For ordered facts, a multifield value containing the single symbol *implied* is returned. See sections 3.2.3 and 3.2.4 for information on getting the value stored in a DATA_OBJECT.

Returns: No meaningful value.

4.4.11 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; FALSE (0) if the behavior is disabled and TRUE (1) if the behavior is enabled.

4.4.12 GetFactList

```
void GetFactList(&returnValue,theModule);
DATA_OBJECT returnValue;
void *theModule;
```

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

Arguments:

- 1) A 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 all modules.

Returns: No meaningful return value.

4.4.13 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.

4.4.14 GetFactPPForm

```
void GetFactPPForm(buffer,bufferLength,factPtr);
char *buffer;
unsigned 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.

4.4.15 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.

4.4.16 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.

4.4.17 GetNextFactInTemplate

```
void *GetNextFactInTemplate(templatePtr, factPtr);
void *templatePtr;
```

```
void *factPtr;
```

- Purpose:** Provides access to the list of facts for a particular deftemplate.
- Arguments:**
- 1) A generic pointer to a deftemplate.
 - 2) A generic pointer to a fact data structure (or NULL to get the first fact from the deftemplate's fact-list).
- Returns:** A generic pointer to the first fact of the specified deftemplate if *factPtr* is NULL, otherwise a generic pointer to the next fact of the specified deftemplate immediately following *factPtr*. If *factPtr* is the last fact belonging to the deftemplate, then NULL is returned.
- Other:** Once this generic pointer to the fact structure is obtained, the fields of the fact can be examined by using the macros **GetMFType** and **GetMFValue**. The values of a fact obtained using this function should never be changed. See **CreateFact** for details on accessing deftemplate facts.
- WARNING:** Do not call this function with a pointer to a fact that has been retracted. If the return value from **GetNextFactInTemplate** is stored as part of a persistent data structure or in a static data area, then the function **IncrementFactCount** should be called to insure that the fact cannot be disposed while external references to the fact still exist.

4.4.18 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.

4.4.19 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.

4.4.20 LoadFactsFromString

```
int    LoadFactsFromString(inputString,maximumPosition);
char  *inputString;
int    maximumPosition;
```

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

Arguments:

- 1) A string containing the fact definitions to be loaded.
- 2) The maximum number of characters to be read from the string. A value of -1 indicates the entire string.

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

4.4.21 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**.

4.4.22 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.

This function can trigger garbage collection.

4.4.23 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.

4.4.24 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: `TRUE` (1) to enable the behavior and `FALSE` (0) to disable it.

Returns: Returns the old value for the behavior.

4.4.25 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 all 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).
- Other:** This function can trigger garbage collection.

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 all 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

```
unsigned 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

```
unsigned 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; FALSE (0) if the behavior is disabled and 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);
unsigned    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);
unsigned 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: TRUE (1) to enable the behavior and 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).

Other: This function can trigger garbage collection.

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. This function must except an environment pointer if the environment companion function is used (see section 9.2).

- 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;
unsigned    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.

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 SetActivationSaliency

```
int SetActivationSaliency(activationPtr,newSaliency);
void *activationPtr;
int newSaliency;
```

- Purpose:** Sets the saliency value of an activation. The saliency value of the defrule which generated the activation is unchanged.
- Arguments:**
- 1) A generic pointer to an activation data structure.
 - 2) The new saliency value (which is not restricted to the -10000 to +10000 range).
- Returns:** The old saliency value of the activation.
- Other:** The function **ReorderAgenda** should be called after saliency 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 SetSaliencyEvaluation

```
int    SetSaliencyEvaluation(value);
int    value;
```

Purpose: Sets the saliency evaluation behavior (the C equivalent of the CLIPS **set-saliency-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 all 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;
unsigned    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

```
unsigned 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; FALSE (0) if globals are not reset and 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).

Other: This function can trigger garbage collection.

4.8.15 SetDefglobalWatch

```
void SetDefglobalWatch(newState,defglobalPtr);
unsigned 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: TRUE (1) to enable the behavior and 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).
- Other:** This function can trigger garbage collection.

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 all 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

```
unsigned 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);
unsigned 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).

Other: This function can trigger garbage collection.

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 all 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

```
unsigned 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);
unsigned 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).
- Other:** This function can trigger garbage collection.

4.11 DEFMETHOD FUNCTIONS

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

4.11.1 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

```
unsigned 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 to send 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);
unsigned 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).

Other: This function can trigger garbage collection.

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 GetClassDefaultsMode

```
unsigned short GetClassDefaultsMode();
```

Purpose: Returns the current class defaults mode (the C equivalent of the CLIPS **get-class-defaults-mode** command).

Arguments: None.

Returns: An integer (see SetClassDefaultsMode for the list of mode constants).

4.12.11 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 all modules.

Returns: No meaningful return value.

4.12.12 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.

4.12.13 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).

4.12.14 GetDefclassWatchInstances

```
unsigned 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).

4.12.15 GetDefclassWatchSlots

```
unsigned 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).

4.12.16 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.

4.12.17 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).

4.12.18 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.

4.12.19 SetClassDefaultsMode

```
unsigned short SetClassDefaultsMode(value);
unsigned short value;
```

- Purpose:** Sets the current class defaults mode (the C equivalent of the CLIPS **set-class-defaults-mode** command).
- Arguments:** The new value for the mode – one of the following defined integer constants:
- CONVENIENCE_MODE
CONSERVATION_MODE
- Returns:** Returns the old value for the mode.

4.12.20 SetDefclassWatchInstances

```
void SetDefclassWatchInstances(newState,defclassPtr);
```

```
unsigned 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.

4.12.21 SetDefclassWatchSlots

```
void SetDefclassWatchSlots(newState,defclassPtr);
unsigned 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.

4.12.22 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.

4.12.23 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.

4.12.24 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.

4.12.25 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.

4.12.26 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.

4.12.27 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.

4.12.28 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.

4.12.29 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.

4.12.30 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.

4.12.31 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.

4.12.32 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.

4.12.33 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.

4.12.34 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.

4.12.35 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).
- Other:** This function can trigger garbage collection.

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,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 `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.

Other: This function can trigger garbage collection.

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.

Other: This function can trigger garbage collection.

WARNING: This function bypasses message-passing.

4.13.8 FindInstance

```
void *FindInstance(theModule,instanceName,searchImports);
void *theModule;
char *instanceName;
unsigned 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;
unsigned  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 GetNextInstanceInClassAndSubclasses

```
void *GetNextInstanceInClassAndSubclasses(defclassPtr,instancePtr,
                                          iterationData);
void **defclassPtr,*instancePtr;
DATA_OBJECT *iterationData;
```

Purpose: Provides access to the list of instances for a particular class and its subclasses.

Arguments:

- 1) A generic pointer to a generic pointer to a class.
- 2) A generic pointer to an instance (or NULL to get the first instance in the specified class).
- 3) A pointer to a DATA_OBJECT in which instance iteration is stored. No initialization of this argument is required and the values stored in this argument are not intended for examination by the calling function.

Returns: A generic pointer to the first instance in the list of instances for the specified class and its subclasses if *instancePtr* is NULL, otherwise a pointer to the instance immediately following *instancePtr* in the

list or the next instance in a subclass of the class. If *instancePtr* is the last instance in the class and all its subclasses, then NULL is returned.

As the subclasses of the specified class are iterated through to find instances, the value stored in *defclassPtr* is updated to indicate the class of the instance returned by this function.

Example

```
DATA_OBJECT iterate;
void *theInstance;
void *theClass;

theClass = FindDefclass("USER");

for (theInstance = GetNextInstanceInClassAndSubclasses(&theClass,
                                                       NULL,&iterate);
     theInstance != NULL;
     theInstance = GetNextInstanceInClassAndSubclasses(&theClass,
                                                       theInstance,&iterate))
{
    PrintRouter(WDISPLAY,GetInstanceName(theInstance));
    PrintRouter(WDISPLAY,"\n");
}
```

4.13.16 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",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",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);
}

```

4.13.17 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.

4.13.18 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.

4.13.19 LoadInstancesFromString

```
long      LoadInstancesFromString(inputString,maximumPosition);
char  *inputString;
unsigned maximumPosition;
```

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

Arguments:

- 1) A string containing the instance definitions.
- 2) The maximum number of characters to be read from the string.
A value of -1 indicates the entire string.

Returns: Returns the number of instances loaded or -1 if there were problems using the string as an input source.

4.13.20 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.

Other: This function can trigger garbage collection.

Example

```
MakeInstance("(henry of boy (age 8))");
```

4.13.21 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.

4.13.22 RestoreInstancesFromString

```
long    RestoreInstancesFromString(inputString,maximumPosition);
char    *inputString;
unsigned maximumPosition;
```

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

Arguments:

- 1) A string containing the instance definitions.
- 2) The maximum number of characters to be read from the string.
A value of -1 indicates the entire string.

Returns: Returns the number of instances loaded or -1 if there were problems using the string as an input source.

4.13.23 SaveInstances

```
long    SaveInstances(fileName,saveCode,NULL,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 `TRUE`.

Returns: Returns the number of instances saved.

4.13.24 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.

Other: This function can trigger garbage collection.

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);
}
```

4.13.25 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.

4.13.26 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.
- Other:** This function can trigger garbage collection.

4.13.27 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 (TRUE to list them or FALSE to not list them).

Returns: No meaningful value. The second argument to this function is set to a multifield value containing the list of defmessage-handler constructs for the specified class. The multifield functions described in section 3.2.4 can be used to retrieve the defmessage-handler class, name, and type from the list. Note that the class, name, and type for each defmessage-handler are stored as triplets in the return multifield value.

4.14.3 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

```
unsigned 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 (TRUE to list them or 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).

Other: This function can trigger garbage collection.

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 all 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).
- Other:** This function can trigger garbage collection.

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()
{
    InitializeEnvironment();
    Load("constructs.clp");
    Reset();
    Run(-1L)
}

void TripleNumber(
    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;
}

void UserFunctions()
{
    extern void TripleNumber();

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

void EnvUserFunctions(
    void *theEnv))
{
}

```

- 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];

    InitializeEnvironment();

    /*=====*/
    /* 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. */
    /*=====*/

    FunctionCall("+","[c1] [c2]",&result);
    c3 = FindInstance(NULL,D0ToString(result),TRUE);

    /*=====*/
    /* Print out a summary of the complex addition using the */
    /* "print" and "magnitude" messages to get information */
    /* about the three complex numbers. */
    /*=====*/

    PrintRouter("stdout","The addition of\n\n");

    SetType(insdata,INSTANCE_ADDRESS);
    SetValue(insdata,c1);
    Send(&insdata,"print",NULL,&result);

```

```

PrintRouter("stdout", "\nand\n\n");

SetType(insdata, INSTANCE_ADDRESS);
SetValue(insdata, c2);
Send(&insdata, "print", NULL, &result);

PrintRouter("stdout", "\nis\n\n");

SetType(insdata, INSTANCE_ADDRESS);
SetValue(insdata, c3);
Send(&insdata, "print", NULL, &result);

PrintRouter("stdout", "\nand the resulting magnitude is\n\n");

SetType(insdata, INSTANCE_ADDRESS);
SetValue(insdata, c3);
Send(&insdata, "magnitude", NULL, &result);
sprintf(numbuf, "%lf\n", D0ToDouble(result));
PrintRouter("stdout", numbuf);
}

void UserFunctions()
{}

void EnvUserFunctions(
    void *theEnv)
{}

```

- 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 **load** 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 **load** 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 function **InitializeEnvironment** should not be called. Also do *not* call the **CommandLoop** or **RerouteStdin** functions which are normally called from the **main** function of a command line version of CLIPS. Do *not* define any functions in the **UserFunctions** or **EnvUserFunctions** functions. These functions are not called during initialization. All of the function definitions have already been compiled in the 'C' constructs code. In order for your run-time program to be loaded, a function must be called to initialize the constructs module. This function is defined in the 'C' constructs code, and its name is dependent upon the id used when translating the constructs to 'C' code. The name of the function is **InitCImage_<id>** where <id> is the integer used as the construct module <id>. In the example above, the function name would be **InitCImage_1**. The return value of this function is a pointer to an environment (see section 9) which was created and initialized to contain your run-time program. This initialization steps probably would be followed by any user initialization, then by a reset and run. Finally, when you are finished with a run-time module, you can call **DestroyEnvironment** to remove it. An example **main.c** file would be

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

main()
{
    void *theEnv;
    extern void *InitCImage_1();

    theEnv = InitCImage_1();
```

```

    •
    •
    •
    EnvReset(theEnv);
    EnvRun(theEnv,-1);
    •
    •
    •
    /* Any other code */
    DestroyEnvironment(theEnv);
}

void UserFunctions()
{
    /* UserFunctions is not called for a run-time version. */
}

void EnvUserFunctions(
    void *theEnv)
{
    /* EnvUserFunctions 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.

Important Note

In prior versions of CLIPS, it was possible to switch between different images by calling the **InitCImage** function of the desired run-time program while execution was halted. This mechanism is no longer available. Each call to separate **InitCImage** functions creates a unique environment into which the run-time program is loaded. You can thus switch between various runtime programs by using the environment API to specify which environment is the target of a command. Also note that only the first call to a given **InitCImage** function will create an environment containing the specified run-time program. Subsequent calls have no effect and a value of NULL is returned by the function. Once the **DestroyEnvironment** function has been called to remove an environment created by an **InitCImage** call, there is no way to reload the run-time program.

Section 6 - 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
InitializeEnvironment	xInitializeEnvironment
Reset	xReset
Load	xLoad
Run	xRun
Facts	xFacts
Watch	xWatch
Unwatch	xUnwatch
AssertString	xAssertString
Retract	xRetract
PrintRouter	xPrintRouter
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

    xInitializeEnvironment;

    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 xInitializeEnvironment
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
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).

```
$sacs create library [{user directory}].adalib]
$sacs set library [{user directory}].adalib]
$ada {Ada files, including the interface packages}
```

- 4) Export the Ada object code from the DEC ACS library.

```
$sacs 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

```
$sacs 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.
wprompt	The CLIPS prompt is sent to this logical name.
wdialog	All informational messages are sent to this logical name.
wdisplay	Requests to display CLIPS information, such as facts or rules, are sent to this logical name.
werror	All error messages are sent to this logical name.
wwarning	All warning messages are sent to this logical name.
wtrace	All watch information is sent to this logical name (with the exception of compilations which is sent to wdialog).

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 **PrintRouter** 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 ExitRouter

```
void ExitRouter(exitCode);
int exitCode;
```

Purpose: The function **ExitRouter** 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 **ExitRouter** calls the system function **exit** with the argument num after calling all exit functions associated with I/O routers.

7.5.2 GetcRouter

```
int GetcRouter(logicalName);
char *logicalName;
```

- Purpose:** The function **GetcRouter** 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 PrintRouter

```
int    PrintRouter(logicalName, str);
char *logicalName, *str;
```

- Purpose:** The function **PrintRouter** 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 UngetcRouter

```
int    UngetcRouter(ch, logicalName);
int    ch;
char *logicalName;
```

- Purpose:** The function **UngetcRouter** 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 **UngetcRouter** to ensure that character input from the function can be received from a custom interface. As with **GetcRouter**, user code should be prepared to handle special characters like the backspace on machines with unbuffered I/O.

7.6 ROUTER HANDLING FUNCTIONS

The following functions are used for creating, deleting, and handling I/O routers. They are intended for use within user-defined functions. Prototypes for these functions can be included by using the **clips.h** header file or the **router.h** header file.

7.6.1 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;
int  ch, 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 ungetc 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) if 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. Each of the router functions must except an environment pointer if the environment companion function is used (see section 9.2).

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 **memalloc.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, this option should not be enabled unless memory allocation routines are very slow since the CLIPS block memory routines tend to trade increased overhead for memory requests for faster speed.

Extensive effort has gone into making CLIPS garbage free. Theoretically, if an application can fit into the available memory on a machine, CLIPS should be able to run it forever. Of course, user-defined functions that use dynamic memory may affect this.

8.2 STANDARD MEMORY FUNCTIONS

CLIPS currently provides a few functions that can be used to monitor and control memory usage. Prototypes for these functions can be included by using the **clips.h** header file or the **memalloc.h** header file.

8.2.1 GetConserveMemory

```
int    GetConserveMemory();
```

Purpose: Returns the current value of the conserve memory behavior.

Arguments: None.

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

8.2.2 MemRequests

```
long int MemRequests();
```

Purpose: The function **MemRequests** will return the number of times CLIPS has requested memory from the operating system (the C equivalent of the CLIPS **mem-requests** command).

Arguments: None.

Returns: A long integer representing the number of requests CLIPS has made.

Other: When used in conjunction with **MemoryUsed**, the user can estimate the number of bytes CLIPS requests per call to **malloc**.

8.2.3 MemUsed

```
long int MemUsed();
```

Purpose: The function **MemUsed** will return the number of bytes CLIPS has currently in use or has held for later use (the C equivalent of the CLIPS **mem-used** command).

Arguments: None.

Returns: A long integer representing the number of bytes requested.

Other: The number of bytes used does not include any overhead for memory management or data creation. It does include all free memory being held by CLIPS for later use; therefore, it is not a completely accurate measure of the amount of memory actually used to store or process information. It is used primarily as a minimum indication.

8.2.4 ReleaseMem

```
long int ReleaseMem(howMuch, printMessage);
long int howMuch;
int printMessage;
```

Purpose: The function **ReleaseMem** will cause all free memory, or a specified amount, being held by CLIPS to be returned to the operating system (the C equivalent of the CLIPS **release-mem** command).

Arguments:

- 1) The number of bytes to be released. If this argument is -1, all memory will be released; otherwise, the specified number of bytes will be released.
- 2) A non-zero value causes a memory deallocation message to be printed when this function is called.

Returns: A long integer representing the actual amount of memory freed to the operating system.

Other: This function can be useful if a user-defined function requires memory but cannot get any from a **malloc** call. However, it should be used carefully. Excessive calls to **ReleaseMemory** will cause CLIPS to call **malloc** more often, which can reduce the performance of CLIPS.

8.2.5 SetConserveMemory

```
int      SetConserveMemory(value);
int      value;
```

Purpose: The function **SetConserveMemory** allows a user to turn on or off the saving of pretty print information. Normally, this information is saved. If constructs are never going to be pretty printed or saved, a significant amount of memory can be saved by not keeping the pretty print representation.

Arguments: A boolean value: FALSE (0) to keep pretty print information for newly loaded constructs and TRUE (1) to not keep this information for newly loaded constructs.

Returns: Returns the old value for the behavior.

Other: This function can save considerable memory space. It should be turned on before loading any constructs. It can be turned on or off as many times as desired. Constructs loaded while this is turned off can be displayed only by reloading the construct, even if the option is turned on subsequently.

8.2.6 SetOutOfMemoryFunction

```
int      (*SetOutOfMemoryFunction(outOfMemoryFunction))();
int      (*outOfMemoryFunction)();

int      outOfMemoryFunction(theEnv,size);
void      *theEnv;
unsigned long size;
```

Purpose: Allows the user to specify a function to be called when CLIPS cannot satisfy a memory request.

Arguments: A pointer to the function to be called when CLIPS cannot satisfy a memory request. This function is passed the size of the memory request which could not be satisfied and a pointer to the environment. It should return a non-zero value if CLIPS should not

attempt to allocate the memory again (and exit because of lack of available memory) or a zero value if CLIPS should attempt to allocate memory again.

Returns: Returns a pointer to the previously called out of memory function.

Other: Because the out of memory function can be called repeatedly for a single memory request, any user-defined out of memory function should return zero only if it has released memory.

Section 9 - Environments

CLIPS provides the ability to create multiple environments into which programs can be loaded. Each environment maintains its own set of data structures and can be run independently of the other environments.

9.1 CREATING, SELECTING, AND DESTROYING ENVIRONMENTS

If you have no need for multiple CLIPS programs loaded concurrently, there is no need to use any of the environment functions described in this section. The call to **InitializeEnvironment** automatically creates an environment for you and any subsequent calls to CLIPS functions will be applied to that environment. Environments can also be created using the **CreateEnvironment** function. The return value of the **CreateEnvironment** function is an anonymous (void *) pointer to an **environmentData** data structure. Environments created using the **CreateEnvironment** function are automatically initialized, so there is no need to call the **InitializeEnvironment** function.

Once multiple environments have been created, it is necessary to specify to which environment CLIPS function calls should be applied. This can be done in one of two ways. First, each of the CLIPS embedded function calls has a companion function call of the same name prefaced with “Env.” These companion function call accept an additional first argument: a generic pointer to an environment data structure. The CLIPS embedded function call is applied to this argument. CLIPS also supports the notion of a current environment. CLIPS embedded function calls that do not specify an environment are applied to the current environment. Environments newly created by the **CreateEnvironment** function call automatically become the current environment. The current environment can also be set by using the **SetCurrentEnvironment** function call. Environments also have an integer index associated with them that can be retrieved using the **GetEnvironmentIndex** function. The current environment can also be set by passing this index to the **SetCurrentEnvironmentByIndex** function.

Once you are done with an environment, it can be deleted with the **DestroyEnvironment** function call. This will deallocate all memory associated with that environment. Alternately if the **ALLOW_ENVIRONMENT_GLOBALS** compiler directive is enabled and you are ready to terminate execution of your program, the **DeallocateEnvironmentData** function call can be used to delete all existing environments and release the global memory used to keep track of environments.

If you have added your own user-defined functions or extensions to CLIPS and you want these to work properly with multiple environments, you need to make them environment aware. Principally this involves using the environment companion functions in place of the standard embedded function calls. In addition, any functions or extensions which use global data should allocate this data for each environment by using the **AllocateEnvironmentData** function.

Shown following are two example main programs which make use of environments. The first example uses the environment companion embedded function calls and the second example uses the standard embedded function calls.

Environments Using Environment Embedded Calls

```
void main()
{
    void *theEnv1, *theEnv2;

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

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

    EnvReset(theEnv1);
    EnvReset(theEnv2);

    EnvRun(theEnv1,-1);
    EnvRun(theEnv2,-1);

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

Environments Using Standard Embedded Calls

```
void main()
{
    void *theEnv1, *theEnv2;

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

    SetCurrentEnvironment(theEnv1);
    Load("program1.clp");
    Reset();
    Run(-1);

    SetCurrentEnvironment(theEnv2);
    Load("program2.clp");
    Reset();
    Run(-1);

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

9.2 ENVIRONMENT COMPANION FUNCTIONS

With a few exceptions, all of the CLIPS embedded function calls described in sections 3 through 8 have a companion function of the same name preceded with “Env”. The first argument to these

companion functions is a generic pointer to an environment data structure and the remaining arguments are the same as the standard embedded function. For example, the standard embedded function call for **Run** is defined as follows:

```
long int Run(runLimit);

long int runLimit;
```

The environment companion function for **Run** is defined as follows:

```
long int EnvRun(theEnv,runLimit);

void *theEnv;
long int runLimit;
```

The pointers to functions passed in to the companion functions for the **AddClearFunction**, **AddResetFunction**, **AddPeriodicFunction**, **AddRunFunction**, and **AddRouter** should have as an additional first argument a generic pointer to an environment. For example, the standard embedded function call for **AddClearFunction** is defined as follows:

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

void clearFunction();
```

The environment companion function for **AddClearFunction** is defined as follows:

```
int      EnvAddClearFunction(theEnv,clearItemName,envClearFunction,priority);

void *theEnv;
char *clearItemName;
void (*clearFunction)();
int      priority;

void envClearFunction(theEnv);

void *theEnv;
```

The **InitializeEnvironment** function does not have a companion function since this function is unnecessary when you explicitly create environments using the **CreateEnvironment** function. The following embedded functions all have environment companion functions, but it is not necessary to use the companion functions in order to be environment aware:

GetType	DOToString	GetDOLength	ValueToString
GetpType	DOPToString	GetpDOLength	ValueToDouble
SetType	DOToDouble	GetDOBgin	ValueToLong
SetpType	DOPToDouble	GetpDOBgin	ValueToInteger

GetValue	DOToFloat	GetDOEnd
GetpValue	DOPToFloat	GetpDOEnd
SetValue	DOToLong	SetDOBegin
SetpValue	DOPToLong	SetpDOBegin
GetMFTType	DOToInteger	SetDOEnd
GetMFValue	DOPToInteger	SetpDOEnd
SetMFTType	DOToPointer	
SetMFValue	DOPToPointer	

If the **ENVIRONMENT_API_ONLY** compiler directive is enabled, then the standard embedded functions require their first argument to be a generic pointer to an environment. For example, the function **Run** would be defined as follows:

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

void *theEnv;
long int runLimit;
```

This change only applies to the functions that are required in order to be environment aware. For example, the **GetType** function would not require an additional argument if this compiler directive were enabled. In addition, even with this compiler directive enabled, the “Env” companion functions are still available. Use of this compiler directive is a good way to verify that any code you have written is environment aware. It is also useful if, for whatever reason, you prefer that the CLIPS embedded function calls do not all begin with the “Env” prefix. By default, the **ENVIRONMENT_API_ONLY** compiler directive is disabled.

9.3 STANDARD ENVIRONMENT FUNCTIONS

The following functions are used to create and manipulate environments. Prototypes for these functions can be included by using the **clips.h** header file or the **envrnmnt.h** header file.

9.3.1 AddEnvironmentCleanupFunction

```
int AddEnvironmentCleanupFunction(theEnv,theName,theFunction,priority);
struct environmentData *theEnv;
char *theName;
void (*)(void *theFunction);
int priority;

void theFunction(void *);
```

Purpose: Adds a cleanup function that is called when an environment is destroyed.

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

- 2) The name associated with the environment cleanup function.
- 3) A pointer to the environment cleanup function which is to be called when the environment is deleted. When called, the function is passed a generic pointer to the environment being destroyed.
- 4) The priority of the environment cleanup function which determines the order in which cleanup functions are called (higher priority items are called first). The values -2000 to 2000 are reserved for CLIPS system defined run items and should not be used for user defined run items.

Returns: Boolean value. TRUE if the cleanup function was successfully added, otherwise FALSE.

Other: Environment cleanup functions created using this function are called after all the cleanup functions associated with environment data created using **AllocateEnvironmentData** have been called.

9.3.2 AllocateEnvironmentData

```
int AllocateEnvironmentData(theEnv,position,size,cleanupFunction);
void *theEnv;
unsigned int position;
unsigned long size;
void (*)(void *cleanupFunction);

void cleanupFunction(void *);
```

Purpose: Allocates environment specific data of the specified size.

Arguments:

- 1) A generic pointer to an environment data structure.
- 2) The integer position index used to reference the data.
- 3) The amount of environment data that needs to be allocated.
- 4) A pointer to a cleanup function that is called when the environment is destroyed. When called, the function is passed a generic pointer to the environment being destroyed. CLIPS automatically handles the allocation and deallocation of the base environment data created by this function (the amount of data specified by the *size* argument). You do not need to supply a cleanup function for this purpose and can supply NULL as this argument. If your base environment data contains pointers to memory that you allocate, then you need to either supply a cleanup function as this argument or add a cleanup function using **AddEnvironmentCleanupFunction**.

- Returns:** Boolean value. TRUE if the environment data was successfully allocated, otherwise FALSE.
- Other:** Environment cleanup functions specified using this function are called in ascending order of the position indices. If the deallocation of your environment data has order dependencies, you can either assign the position indices appropriately to achieve the proper order or you can use the **AddEnvironmentCleanupFunction** function to more explicitly specify the order in which your environment data must be deallocated.

9.3.3 CreateEnvironment

```
void *CreateEnvironment();
```

- Purpose:** Creates an environment and initializes it.
- Arguments:** None.
- Returns:** A generic pointer to an environment data structure. NULL is returned in the event of an error.

9.3.4 DeallocateEnvironmentData

```
int DeallocateEnvironmentData();
```

- Purpose:** Calls DestroyEnvironment to deallocate each existing environment and then deallocates the remaining memory used to keep track of environment allocations.
- Arguments:** None.
- Returns:** Boolean value. TRUE if the function was able to successfully deallocate the environment data, otherwise FALSE.
- Other:** You should not call this function if any environments are currently executing or you are not terminating your program. If the compiler directive **ALLOW_ENVIRONMENT_GLOBALS** is FALSE, this function does nothing.

9.3.5 DestroyEnvironment

```
int DestroyEnvironment(theEnv);
```

```
void *theEnv;
```

- Purpose:** Destroys the specified environment deallocating all memory associated with it.
- Arguments:** A generic pointer to an environment data structure.
- Returns:** Boolean value. TRUE if the environment was successfully destroyed, otherwise FALSE.
- Other:** You should not call this function to destroy an environment that is currently executing.

9.3.6 GetCurrentEnvironment

```
void *GetCurrentEnvironment();
```

- Purpose:** Returns a generic pointer to the current environment.
- Arguments:** None.
- Returns:** A generic pointer to the current environment. NULL is returned if there is no current environment.

9.3.7 GetEnvironmentByIndex

```
void *GetEnvironmentByIndex(envIndex);
unsigned long envIndex;
```

- Purpose:** Returns a generic pointer to the environment associated with the specified environment index.
- Arguments:** An unsigned long integer; the environment index of the environment to become the current environment.
- Returns:** A generic pointer to the environment associated with the specified environment index. NULL is returned if there is no such environment.

9.3.8 GetEnvironmentData

```
void *GetEnvironmentData(theEnv,position);
void *theEnv;
unsigned int position;
```

- Purpose:** Returns a generic pointer to the environment data associated with the specified position index.
- Arguments:**
- 1) A generic pointer to an environment data structure.
 - 2) An unsigned integer; the position index of the desired environment data.
- Returns:** A generic pointer; the environment data associated with the specified position index.

9.3.9 GetEnvironmentIndex

```
unsigned long GetEnvironmentIndex(theEnv);
void *theEnv;
```

- Purpose:** Returns the unique integer index associated with the specified environment.
- Arguments:** A generic pointer to an environment data structure.
- Returns:** An integer; the index associated with the specified environment.

9.3.10 SetCurrentEnvironment

```
void SetCurrentEnvironment(theEnv);
void *theEnv;
```

- Purpose:** Sets the current environment to the specified environment.
- Arguments:** A generic pointer to an environment data structure.
- Returns:** No meaningful return value.

9.3.11 SetCurrentEnvironmentByIndex

```
int SetCurrentEnvironmentByIndex(envIndex);
unsigned long envIndex;
```

- Purpose:** Sets the current environment to the environment associated with the specified environment index.
- Arguments:** An unsigned long integer; the environment index of the environment to become the current environment.

Returns: Boolean value. TRUE if the environment with the specified index existed and was set as the current environment, otherwise FALSE.

9.4 ENVIRONMENT AWARE USER-DEFINED FUNCTIONS

In order to support all environment features fully, any user-defined functions that you create must be environment aware. To be environment aware, user-defined function must satisfy the following conditions:

- 1) The user-defined function must be registered using either **EnvDefineFunction** or **EnvDefineFunction2**. Use of these functions inform CLIPS that your user-defined function is environment aware and accepts a generic pointer to an environment as its first argument.
- 2) You should register your functions from within **EnvUserFunctions** instead of **UserFunctions**. **EnvUserFunctions** is located in **main.c** and its single argument is a generic pointer to an environment. This pointer should be passed into your calls to either **EnvDefineFunction** or **EnvDefineFunction2**. The macro identifier PTIEF can be placed in front of a function name to cast it as a pointer to a function which accepts a generic pointer (the environment) as its single argument and returns an integer. This macro is analogous to the PTIF macro with the addition of the generic pointer to the environment.
- 3) Your user-defined function should accept an additional argument as its first argument: a generic pointer to an environment.
- 4) Your user-defined function should use the environment companion functions where required to be environment aware.
- 5) If your user-defined functions (or other extensions) make use of global data that could differ for each environment, you should allocate this data with the **AllocateEnvironmentData** function (see section 9.5).

Example

The following example shows the necessary modifications to the code from section 3.4 in order for the user-defined function to be environment aware.

```
void EnvUserFunctions(
    void *theEnv)
{
    EnvDefineFunction2(theEnv,"triple",'u',PTIEF TripleNumber, "TripleNumber",
                      "11n");
}

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

```

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

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

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

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

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

return;
}

```

9.5 ALLOCATING ENVIRONMENT DATA

If your user-defined functions (or other extensions) make use of global data that could differ for each environment, you should allocate this data with the **AllocateEnvironmentData** function. A call to this function has four arguments. The first is a generic pointer to the environment to which the data is being added.

The second argument is the integer position index. This is the value that you will pass in to the **GetEnvironmentData** function to retrieve the allocated environment data. This position index must be unique and if you attempt to use an index that has already been allocated, then the call to **AllocateEnvironmentData** will fail returning FALSE. To avoid collisions with environment positions predefined by CLIPS, use the macro constant **USER_ENVIRONMENT_DATA** as the base index for any position indices you define. This constant will always be greater than the

largest predefined position index used by CLIPS. The maximum number of environment position indices is specified by the macro constant `MAXIMUM_ENVIRONMENT_POSITIONS` found in the `envrnmnt.h` header file. A call to **AllocateEnvironmentData** will fail if the position index is greater than or equal this value. If this happens, you can simply increase the value of this macro constant to provide more environment positions.

The third argument is an integer indicating the size of the environment data that needs to be allocated. Typically you'll define a struct containing the various values you want stored in the environment data and use the `sizeof` operator to pass in the size of the struct to the function. When an environment is created directly using **CreateEnvironment** or indirectly using **InitializeEnvironment**, CLIPS automatically allocates memory of the size specified, initializes the memory to contain all zeroes, and stores the memory in the environment position associated with position index. When the environment is destroyed using **DestroyEnvironment**, CLIPS automatically deallocates the memory originally allocated for each environment data position. If the environment data contains pointers to memory that you allocate, it is your responsibility to deallocate this memory. You can do this by either specifying a cleanup function as the fourth argument in your **AllocateEnvironmentData** call or by adding a cleanup function using the **AddEnvironmentCleanupFunction** function.

The fourth argument is a pointer to a cleanup function. If this argument is not `NULL`, then the cleanup function associated with this environment position is called whenever an environment is deallocated using the **DestroyEnvironment** function. The cleanup functions are called in ascending order of the position indices.

As an example of allocating environment data, we'll look at a **get-index** function that returns an integer index starting with one and increasing by one each time it is called. For example:

```
CLIPS> (get-index)
1
CLIPS> (get-index)
2
CLIPS> (get-index)
3
CLIPS>
```

Each environment will need global data to store the current value of the index. The C source code that implements the environment data first needs to specify the position index and specify a data structure for storing the data:

```
#define INDEX_DATA USER_ENVIRONMENT_DATA + 0

struct indexData
{
    long index;
};

#define IndexData(theEnv) \
```

```
((struct indexData *) GetEnvironmentData(theEnv, INDEX_DATA))
```

First, the position index `GET_INDEX_DATA` is defined as `USER_ENVIRONMENT_DATA` with an offset of zero. If you were to define additional environment data, the offset would be increased each time by one to get to the next available position. Next, the *indexData* struct is defined. This struct contains a single member, *index*, which will use to store the next value returned by the **get-index** function. Finally, the `IndexData` macro is defined which merely provides a convenient mechanism for access to the environment data.

The next step in the C source code is to add the initialization code to the **EnvUserFunctions** function:

```
void EnvUserFunctions(
    void *theEnv)
{
    if (! AllocateEnvironmentData(theEnv, INDEX_DATA,
                                sizeof(struct indexData), NULL))
    {
        printf("Error allocating environment data for INDEX_DATA\n");
        exit(EXIT_FAILURE);
    }

    IndexData(theEnv)->index = 1;

    EnvDefineFunction2(theEnv, "get-index", 'l', PTIEF GetIndex, "GetIndex",
                      "00");
}
```

First, the call to **AllocateEnvironmentData** is made. If this fails, then an error message is printed and a call to **exit** is made to terminate the program. Otherwise, the *index* member of the environment data is initialized to one. If a starting value of zero was desired, it would not be necessary to perform any initialization since the value of *index* is automatically initialized to zero when the environment data is initialized. Finally, **EnvDefineFunction2** is called to register the **get-index** function.

The last piece of the C source code is the **GetIndex** C function which implements the **get-index** function:

```
long GetIndex(
    void *theEnv)
{
    if (EnvArgCountCheck(theEnv, "get-index", EXACTLY, 0) == -1)
    { return(0); }

    return(IndexData(theEnv)->index++);
}
```

This function is fairly straightforward. A generic pointer to the current environment is passed to the function since it was registered using **EnvDefineFunction2**. First a check for the correct

number of arguments is made and then a call to the **IndexData** macro is made to retrieve the *index* member of struct which is the return value. Use of the ++ operator increments the current value of the *index* member before the function returns.

9.6 ENVIRONMENT GLOBALS

The only global variables in the C source code for CLIPS are used to keep track of the current environment and the environment indices. If it is desired to remove these global variables, the **ALLOW_ENVIRONMENT_GLOBALS** compiler directive can be disabled. If disabled, you can no longer use the following functions: **GetCurrentEnvironment**, **GetEnvironmentByIndex**, **GetEnvironmentIndex**, **SetCurrentEnvironment**, and **SetCurrentEnvironmentByIndex**. In addition, if disabled the **ENVIRONMENT_API_ONLY** compiler directive is enabled and the **EMACS_EDITOR** compiler directive is disabled.

9.7 OTHER CONSIDERATIONS

The mechanism for loading run-time program has changed with the introduction of environments. See section 5 for more details.

Appendix A - Language Integration Listings

This appendix includes listings for various language interface packages described in section 6. The portability of these routines varies. Most of the code listed in the interface packages defined in sections A.1 and A.2 should be fairly portable. However, the string conversion routine in section A.3 is not as portable. For example, the Ada function **Convert_to_C_String** is probably portable to any Ada machine, yet the C function **MakeStringDsc** listed here is very specific to the DEC VMS. These functions should be typed in exactly as shown below.

A.1 ADA INTERFACE PACKAGE FOR CLIPS

The following listings are an Ada package specification and body for some of the CLIPS functions used in embedded CLIPS systems. The code is specific to the DEC Ada compiler because of the PRAGMA IMPORT_PROCEDURE. Other Ada compilers may provide a similar capability, and this package specification could be modified.

CLIPS Package Specification

```
package CLIPS is

-----
-- Initializes the CLIPS environment upon program startup.
    procedure xInitializeEnvironment;
-----
-- Resets the CLIPS environment.
    procedure xReset;
-----
-- Loads a set of constructs into the CLIPS database.  If there are syntactic
-- error in the constructs, xLoad will still attempt to read the
-- entire file, and error notices will be sent to werror.
-- Returns: an integer, zero if an error occurs.
    function xLoad (File_Name : in string) return integer;
-----
-- Allows Run_Limit rules to fire (execute).
-- -1 allows rules to fire until the agenda is empty.
-- Returns: Number of rules that were fired.
    function xRun (Run_Limit : in integer := -1) return integer;
-----
-- Lists the facts in the fact-list.
    procedure xFacts (Logical_Name : in string;
                     Module_Ptr : in integer;
                     First : in integer;
                     Last : in integer;
                     Max : in integer);
-----
-- Turns the watch facilities of CLIPS on.
```

```

function xWatch (Watch_Item : in string) return integer;
-----
-- Turns the watch facilities of CLIPS off.

function xUnwatch (Watch_Item : in string) return integer;
-----
-- Asserts a fact into the CLIPS fact-list. The function version
-- returns the Fact_Pointer required by xRetractFact.

function xAssertString (Pattern : in string) return integer;
-----
-- Causes a fact asserted by the ASSERT_FACT function to be retracted.
-- Returns: false if fact has already been retracted, else true.
-- Input of any value not returned by ASSERT_FACT will
-- cause CLIPS to abort.

function xRetract (Fact_Pointer : in integer) return integer;
-----
-- 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.

function xPrintRouter (Log_Name : in string ;
                      Str       : in string) return integer;
-----
-- Removes a rule from CLIPS.
-- Returns: false if rule not found, else true.

function xUndefrule (Rule_Name : in string) return integer;
-----

private

pragma INTERFACE (C, xInitializeEnvironment);
pragma IMPORT_PROCEDURE (INTERNAL => xInitializeEnvironment,
                        EXTERNAL => InitializeEnvironment);

pragma INTERFACE (C, xReset);
pragma IMPORT_PROCEDURE (INTERNAL => xReset,
                        EXTERNAL => Reset);

function cLoad (File_Name : in string) return integer;
pragma INTERFACE (C, cLoad);
pragma IMPORT_FUNCTION (INTERNAL  => cLoad,
                       EXTERNAL  => Load,
                       MECHANISM => REFERENCE);

pragma INTERFACE (C, xRun);
pragma IMPORT_FUNCTION (INTERNAL  => xRun,
                       EXTERNAL  => Run,
                       MECHANISM => VALUE);

procedure cFacts(Logical_Name : in string;
                 Module_Ptr : in integer;
                 First : in integer;
                 Last : in integer;

```

```

                                Max : in integer);
pragma INTERFACE (C, cFacts);
pragma IMPORT_PROCEDURE (INTERNAL => cFacts,
                        EXTERNAL => Facts,
                        MECHANISM => (REFERENCE, VALUE,
                                    VALUE, VALUE, VALUE));

function cWatch (Item : in string) return integer;
pragma INTERFACE (C, cWatch);
pragma IMPORT_FUNCTION (INTERNAL => cWatch,
                      EXTERNAL => Watch,
                      MECHANISM => REFERENCE);

function cUnwatch (Item : in string) return integer;
pragma INTERFACE (C, cUnwatch);
pragma IMPORT_FUNCTION (INTERNAL => cUnwatch,
                      EXTERNAL => Unwatch,
                      MECHANISM => REFERENCE);

function cAssertString (Pattern : in string) return integer;
pragma INTERFACE (C, cAssertString);
pragma IMPORT_FUNCTION (INTERNAL => cAssertString,
                      EXTERNAL => AssertString,
                      MECHANISM => REFERENCE);

function cRetract (Fact_Pointer : in integer) return integer;
pragma INTERFACE (C, cRetract);
pragma IMPORT_FUNCTION (INTERNAL => cRetract,
                      EXTERNAL => Retract,
                      MECHANISM => VALUE);

function cPrintRouter (Log_Name : in string ;
                      Str : in string) return integer;
pragma INTERFACE (C, cPrintRouter);
pragma IMPORT_FUNCTION (INTERNAL => cPrintRouter,
                      EXTERNAL => PrintRouter,
                      MECHANISM => REFERENCE);

function cUndefrule (Rule_Name : in string) return integer;
pragma INTERFACE (C, cUndefrule);
pragma IMPORT_FUNCTION (INTERNAL => cUndefrule,
                      EXTERNAL => Undefrule,
                      MECHANISM => REFERENCE);

end CLIPS;

```

CLIPS Package Body

```
package body CLIPS is
```

```

-----

function ADA_TO_C_STRING (Input_String : in string)
    return string is

    Out_String : string (1..Input_String'LAST+1);

begin

```

```

    for I in Input_String'RANGE loop
        if (Input_String (I) in ' ' .. '~' or
            Input_String (I) = ASCII.Cr      or
            Input_String (I) = ASCII.Lf     ) then
            Out_String (I) := Input_String (I);
        else
            Out_String (I) := ASCII.Nul;
        end if;
    end loop;
    Out_String (Out_String'LAST) := ASCII.Nul;
    return Out_String;
end ADA_TO_C_STRING;

```

```

function xLoad (File_Name : in string) return integer is
begin
    return cLoad (ADA_TO_C_STRING (File_Name));
end xLoad;

```

```

procedure xFacts (Logical_Name : in string;
                  Module_Ptr : in integer;
                  First : in integer;
                  Last : in integer;
                  Max : in integer) is
begin
    cFacts (ADA_TO_C_STRING (Logical_Name),Module_Ptr,First,Last,Max);
end xFacts;

```

```

function xWatch (Watch_Item : in string) return integer is
begin
    return cWatch (ADA_TO_C_STRING (Watch_Item));
end xWatch;

```

```

function xUnwatch (Watch_Item : in string) return integer is
begin
    return cUnwatch (ADA_TO_C_STRING (Watch_Item));
end xUnwatch;

```

```

function xAssertString (Pattern : in string) return integer is
begin
    return cAssertString (ADA_TO_C_STRING (Pattern));

```



```
end xAssertString;
```

```
-----
function xRetract (Fact_Pointer : in integer) return integer is
begin
  return cRetract (Fact_Pointer);
end xRetract;
```

```
-----
function xPrintRouter (Log_Name : in string ;
                      Str       : in string ) return integer is
begin
  return cPrintRouter (ADA_TO_C_STRING (Log_Name),
                      ADA_TO_C_STRING (Str));
end xPrintRouter;
```

```
-----
function xUndefrule (Rule_Name : in string) return integer is
begin
  return cUndefrule (ADA_TO_C_STRING (Rule_Name));
end xUndefrule;
```

```
end CLIPS;
```

A.2 FORTRAN INTERFACE PACKAGE FOR VAX VMS

The following pages are listings of the FORTRAN interface functions for the VAX VMS plus the internal functions used to convert FORTRAN character strings to C character strings and vice versa. Many of these functions may work with minor modifications on other machines; note, however, the use of the VMS argument passing modifier, %VAL, in some functions.

```
C
C-----
C
      SUBROUTINE xInitializeEnvironment
      CALL InitializeEnvironment
      RETURN
      END
C
C-----
C
      SUBROUTINE xReset
      CALL Reset
      RETURN
      END
C
C-----
```

```

C      INTEGER FUNCTION xLoad (FILE_NAME)

      CHARACTER * (*) FILE_NAME
      CHARACTER *80 C_FILE_NAME
      INTEGER C_FILE_NAME_POINTER, Load
      EQUIVALENCE (C_FILE_NAME, C_FILE_NAME_POINTER)

      CALL CONVERT_TO_C_STRING (FILE_NAME, C_FILE_NAME)
      xLoad = Load (C_FILE_NAME_POINTER)
      RETURN
      END

C
C-----
C
      INTEGER FUNCTION xRun (RUN_LIMIT)

      INTEGER RUN_LIMIT, Run

      xRun = Run (%VAL (RUN_LIMIT))
      RETURN
      END

C
C-----
C
      SUBROUTINE xFacts (LOGICAL_NAME, MODULE, BEGIN, END, MAX)

      CHARACTER * (*) LOGICAL_NAME
      INTEGER MODULE, BEGIN, END, MAX
      CHARACTER *80 C_LOGICAL_NAME
      INTEGER C_LOGICAL_NAME_POINTER
      EQUIVALENCE (C_LOGICAL_NAME, C_LOGICAL_NAME_POINTER)

      CALL CONVERT_TO_C_STRING (LOGICAL_NAME, C_LOGICAL_NAME)
      CALL Facts(C_LOGICAL_NAME_POINTER,%VAL (MODULE),
*           %VAL (BEGIN),%VAL (END),%VAL (MAX))
      RETURN
      END

C
C-----
C
      INTEGER FUNCTION xWatch (WATCH_ITEM)

      CHARACTER * (*) WATCH_ITEM
      CHARACTER *80 C_WATCH_ITEM
      INTEGER C_WATCH_ITEM_POINTER, Watch
      EQUIVALENCE (C_WATCH_ITEM, C_WATCH_ITEM_POINTER)

      CALL CONVERT_TO_C_STRING (WATCH_ITEM, C_WATCH_ITEM)
      xWatch = Watch (C_WATCH_ITEM_POINTER)
      RETURN
      END

C
C-----
C
      INTEGER FUNCTION xUnwatch (WATCH_ITEM)

      CHARACTER * (*) WATCH_ITEM

```

```

CHARACTER *80 C_WATCH_ITEM
INTEGER C_WATCH_ITEM_POINTER, Unwatch
EQUIVALENCE (C_WATCH_ITEM, C_WATCH_ITEM_POINTER)

CALL CONVERT_TO_C_STRING (WATCH_ITEM, C_WATCH_ITEM)
xUnwatch = Unwatch (C_WATCH_ITEM_POINTER)
RETURN
END

C
C-----
C
INTEGER FUNCTION xAssertString (PATTERN)
C
CHARACTER * (*) PATTERN
CHARACTER *80 C_PATTERN
INTEGER C_PATTERN_POINTER, AssertString
EQUIVALENCE (C_PATTERN, C_PATTERN_POINTER)
C
CALL CONVERT_TO_C_STRING (PATTERN, C_PATTERN)
xAssertString = AssertString (C_PATTERN_POINTER)
RETURN
END

C
C-----
C
INTEGER FUNCTION xRetract (FACT_ADDRESS)

INTEGER FACT_ADDRESS, Retract

xRetract = Retract (%VAL (FACT_ADDRESS))
RETURN
END

C
C-----
C
INTEGER FUNCTION xPrintRouter (LOG_NAME, PRINT_LINE)

CHARACTER * (*) LOG_NAME
CHARACTER * (*) PRINT_LINE
CHARACTER *80 C_LOG_NAME
CHARACTER *80 C_PRINT_LINE
INTEGER C_PRINT_LINE_POINTER, C_LOG_NAME_POINTER
EQUIVALENCE (C_PRINT_LINE, C_PRINT_LINE_POINTER),
*             (C_LOG_NAME, C_LOG_NAME_POINTER )

CALL CONVERT_TO_C_STRING (PRINT_LINE, C_PRINT_LINE)
CALL CONVERT_TO_C_STRING (LOG_NAME, C_LOG_NAME)
xPrintRouter = PrintRouter (C_LOG_NAME_POINTER,
*                           C_PRINT_LINE_POINTER)
RETURN
END

C
C-----
C
INTEGER FUNCTION xFindDefrule (RULE_NAME)

CHARACTER * (*) RULE_NAME
CHARACTER *80 C_RULE_NAME

```

```

        INTEGER C_RULE_NAME_POINTER, Undefrule
        EQUIVALENCE (C_RULE_NAME, C_RULE_NAME_POINTER)

        CALL CONVERT_TO_C_STRING (RULE_NAME, C_RULE_NAME)
        xFindDefrule = FindDefrule (C_RULE_NAME_POINTER)
    RETURN
    END
C
C-----
C
    INTEGER FUNCTION xUndefrule (RULE_NAME)

    CHARACTER * (*) RULE_NAME
    CHARACTER *80 C_RULE_NAME
    INTEGER C_RULE_NAME_POINTER, Undefrule
    EQUIVALENCE (C_RULE_NAME, C_RULE_NAME_POINTER)

    CALL CONVERT_TO_C_STRING (RULE_NAME, C_RULE_NAME)
    xUndefrule = Undefrule (C_RULE_NAME_POINTER)
    RETURN
    END
C
C-----
C
    SUBROUTINE CONVERT_TO_C_STRING (F_STRING, C_STRING)
    CHARACTER * (*) F_STRING, C_STRING

    K = LENGTH (F_STRING)
    DO 100 I = 1,K
        C_STRING (I:I) = F_STRING (I:I)
    100 CONTINUE
    K = K + 1
    C_STRING (K:K) = CHAR (0)
    RETURN
    END
C
C-----
C
    INTEGER FUNCTION LENGTH (STRING)

    CHARACTER * (*) STRING

    K = LEN (STRING)
    DO 100 I=K,1,-1
        IF(STRING(I:I) .NE. ' ') GO TO 150
    100 CONTINUE
    150 CONTINUE
    LENGTH = I
    RETURN
    END

```

A.3 FUNCTION TO CONVERT C STRINGS FOR VMS ADA OR FORTRAN

This function converts a C string to an Ada string. The **MakeStringDsc** function normally is stored in the same file together with the **UserFunctions** definition and any C interface

subroutines. The function is not portable and is specific to the VAX VMS environment. The definition of Ada string descriptors is implementation dependent, and access to those definitions from C also is implementation dependent. However, a very similar function could be written for any environment that supports Ada and C.

C Function: MakeStringDsc

(Note: This function definition is VAX VMS specific)

```
#include <ssdef.h>
#include <descrip.h>

struct dsc$descriptor_s *MakeStringDsc(c_str)
    char *c_str;
{
    struct dsc$descriptor_s *desc;

    desc = (struct dsc$descriptor_s *) malloc
        (sizeof (struct dsc$descriptor_s));

    /*   Define String Descriptor   */

    desc->dsc$w_length  = strlen(c_str);
    desc->dsc$a_pointer = c_str;
    desc->dsc$b_class   = DSC$K_CLASS_S;
    desc->dsc$b_dtype   = DSC$K_DTYPE_T;

    return(desc);
}
```


Appendix B - I/O Router Examples

The following examples demonstrate the use of the I/O router system. These examples show the necessary C code for implementing the basic capabilities described.

B.1 DRIBBLE SYSTEM

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

```

/*
First of all, we need a file pointer to the dribble file which will contain
the tracing information.
*/

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

static FILE *TraceFP = NULL;

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

int FindTrace(
    char *logicalName)
{
    if (strcmp(logicalName,"wtrace") == 0) return(TRUE);

    return(FALSE);
}

/*
We now need to define a function which will print the tracing information to
our trace file. The print function for our router is defined below.
*/

int PrintTrace(
    char *logicalName,
    char *str)
{
    fprintf(TraceFP,"%s",str);
}

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

int ExitTrace(
    int exitCode)
    {
        fclose(TraceFP);
    }
/* unused */

```

```
}
```

```
/*
```

There is no need to define a get character or ungetc character function since this router does not handle input.

A function to turn the trace mode on needs to be defined. This function will check if the trace file has already been opened. If the file is already open, then nothing will happen. Otherwise, the trace file will be opened and the trace router will be created. This new router will intercept tracing information intended for the user interface and send it to the trace file. The trace on function is defined below.

```
*/
```

```
int TraceOn()
```

```
{
    if (TraceFP == NULL)
    {
        TraceFP = fopen("trace.txt","w");
        if (TraceFP == NULL) return(FALSE);
    }
    else
    { return(FALSE); }
```

```
    AddRouter("trace",                                /* Router name */
              20,                                     /* Priority */
              FindTrace,                              /* Query function */
              PrintTrace,                             /* Print function */
              NULL,                                    /* Getc function */
              NULL,                                    /* Ungetc function */
              ExitTrace);                             /* Exit function */
```

```
    return(TRUE);
}
```

```
/*
```

A function to turn the trace mode off needs to be defined. This function will check if the trace file is already closed. If the file is already closed, then nothing will happen. Otherwise, the trace router will be deleted and the trace file will be closed. The trace off function is defined below.

```
*/
```

```
int TraceOff()
```

```
{
    if (TraceFP != NULL)
    {
        DeleteRouter("trace");

        if (fclose(TraceFP) == 0)
        {
            TraceFP = NULL;
            return(TRUE);
        }
    }
}
```

```
TraceFP = NULL;
return(FALSE);
```



```

    }

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

extern int TraceOn(), TraceOff();

DefineFunction("tron",'b',TraceOn, "TraceOn");
DefineFunction("troff",'b',TraceOff, "TraceOff");

/*
Compile and link the appropriate files. The trace functions should now be
accessible within CLIPS as external functions. For Example
    CLIPS>(tron)
    .
    .
    .
    CLIPS>(troff)
*/

```

B.2 BETTER DRIBBLE SYSTEM

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

```

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

int PrintTrace(
    char *logicalName,
    char *str)
{
    fprintf(TraceFP,"%s",str);
    DeactivateRouter("trace");
    PrintRouter(logicalName,str);
    ActivateRouter("trace");
}

/*
The TraceOn function must also be modified. The priority of the router
should be 40 instead of 20 since the router passes the output along to other
routers.
*/

int TraceOn()
{
    if (TraceFP == NULL)
    {
        TraceFP = fopen("trace.txt","w");
    }
}

```

```

        if (TraceFP == NULL) return(FALSE);
    }
    else
    { return(FALSE); }

    AddRouter("trace",
              40,
              FindTrace,
              PrintTrace,
              NULL,
              NULL,
              ExitTrace);
/* Router name      */
/* Priority          */
/* Query function    */
/* Print function    */
/* Getc function     */
/* Ungetc function   */
/* Exit function     */

    return(TRUE);
}

```

B.3 BATCH SYSTEM

Write the necessary functions that will allow batch input from the file "batch.txt" to the CLIPS top-level interface.

```

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

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

static FILE *BatchFP = NULL;

/*
We want to recognize any input requested from the logical name "stdin"
because all user input is received from this logical name. The recognizer
function for our router is defined below.
*/

int FindMybatch(
    char *logicalName)
{
    if (strcmp(logicalName,"stdin") == 0) return(TRUE);

    return(FALSE);
}

/*
We now need to define a function which will get and unget characters from our
batch file. The get and ungetc character functions for our router are
defined below.
*/

static char BatchBuffer[80];
static int BatchLocation = 0;

int GetcMybatch(
    char *logicalName)

```

```

{
    int rv;

    rv = getc(BatchFP);

    if (rv == EOF)
    {
        DeleteRouter("mybatch");
        fclose(BatchFP);
        return(GetcRouter(logicalName));
    }

    BatchBuffer[BatchLocation] = (char) rv;
    BatchLocation++;
    BatchBuffer[BatchLocation] = EOS;

    if ((rv == '\n') || (rv == '\r'))
    {
        PrintRouter("wprompt",BatchBuffer);
        BatchLocation = 0;
    }

    return(rv);
}

int UngetcMybatch(
    int ch,
    char *logicalName)                /* unused */
{
    if (BatchLocation > 0) BatchLocation--;
    BatchBuffer[BatchLocation] = EOS;
    return(ungetc(ch,BatchFP));
}

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

int ExitMybatch(
    int exitCode)                    /* unused */
{
    fclose(BatchFP);
}

/*
There is no need to define a print function since this router does not handle
output except for echoing the command line.
Now we define a function that turns the batch mode on.
*/

int MybatchOn()
{
    BatchFP = fopen("batch.txt","r");

    if (BatchFP == NULL) return(FALSE);

    AddRouter("mybatch",                /* Router name */

```

```

        20,                                /* Priority      */
        FindMybatch,                       /* Query function */
        NULL,                             /* Print function */
        GetcMybatch,                      /* Getc function  */
        UngetcMybatch,                   /* Ungetc function */
        ExitMybatch);                    /* Exit function  */

    return(TRUE);
}

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

extern int MybatchOn();

DefineFunction("mybatch",'b',MybatchOn, "MybatchOn");

/*
Compile and link the appropriate files. The batch function should now be
accessible within CLIPS as external function. For Example
CLIPS> (mybatch)
*/

```

B.4 SIMPLE WINDOW SYSTEM

Write the necessary functions using CURSES (a screen management function available in UNIX) that will allow a top/bottom split screen interface. Output sent to the logical name **top** will be printed in the upper window. All other screen I/O should go to the lower window. (NOTE: Use of CURSES may require linking with special libraries.)

```

/*
First of all, we need some pointers to the windows and a flag to indicate
that the windows have been initialized.
*/

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

WINDOW *LowerWindow, *UpperWindow;
int WindowInitialized = FALSE;

/*
We want to intercept any I/O requests that the standard interface would
handle. In addition, we also need to handle requests for the logical name
top. The recognizer function for our router is defined below.
*/

int FindScreen(
    char *logicalName)
{
    if ((strcmp(logicalName,"stdout") == 0) ||
        (strcmp(logicalName,"stdin") == 0) ||

```

```

        (strcmp(logicalName,"wprompt") == 0) ||
        (strcmp(logicalName,"wdisplay") == 0) ||
        (strcmp(logicalName,"wdialog") == 0) ||
        (strcmp(logicalName,"werror") == 0) ||
        (strcmp(logicalName,"wwarning") == 0) ||
        (strcmp(logicalName,"wtrace") == 0) ||
        (strcmp(logicalName,"top") == 0) )
    { return(TRUE); }

    return(FALSE);
}

/*
We now need to define a function which will print strings to the two windows.
The print function for our router is defined below.
*/

int PrintScreen(
    char *logicalName,
    char *str)
{
    if (strcmp(logicalName,"top") == 0)
    {
        wprintw(UpperWindow,"%s",str);
        wrefresh(UpperWindow);
    }
    else
    {
        wprintw(LowerWindow,"%s",str);
        wrefresh(LowerWindow);
    }
}

/*
We now need to define a function which will get and unget characters from the
lower window.  CURSES uses unbuffered input so we will simulate buffered
input for CLIPS.  The get and ungetc character functions for our router are
defined below.
*/

static int UseSave = FALSE;
static int SaveChar;
static int SendReturn = TRUE;

static char StrBuff[80] = {'\0'};
static int CharLocation = 0;

```

```

int GetcScreen(
    char *logicalName)
{
    int rv;

    if (UseSave == TRUE)
    {
        UseSave = FALSE;
        return(SaveChar);
    }

    if (StrBuff[CharLocation] == '\0')
    {
        if (SendReturn == FALSE)
        {
            SendReturn = TRUE;
            return('\n');
        }

        wgetstr(LowerWindow, StrBuff[80] );
        CharLocation = 0;
    }

    rv = StrBuff[CharLocation];
    if (rv == '\0') return('\n');
    CharLocation++;
    SendReturn = FALSE;
    return(rv);
}

int UngetcScreen(
    char ch,
    char *logicalName)
{
    UseSave = TRUE;
    SaveChar = ch;
    return(ch);
}

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

int ExitScreen(
    int num)                                /* unused */
{
    endwin();
}

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

int ScreenOn()
{
    int halfLines, i;

```

```

/* Has initialization already occurred? */

if (WindowInitialized == TRUE) return(FALSE);
else WindowInitialized = TRUE;

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

initscr();
echo();

AddRouter("screen",                                /* Router name */
          10,                                       /* Priority */
          FindScreen,                               /* Query function */
          PrintScreen,                             /* Print function */
          GetcScreen,                              /* Getc function */
          UngetcScreen,                           /* Ungetc function */
          ExitScreen);                             /* Exit function */

/* Create the two windows. */

halfLines = LINES / 2;
UpperWindow = newwin(halfLines, COLS, 0, 0);
LowerWindow = newwin(halfLines - 1, COLS, halfLines + 1, 0);

/* Both windows should be scrollable. */

scrollok(UpperWindow, TRUE);
scrollok(LowerWindow, TRUE);

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

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

wclear(UpperWindow);
wclear(LowerWindow);
wmove(LowerWindow, 0, 0);

return(TRUE);
}

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

int ScreenOff()
{
    /* Is CURSES already deactivated? */

    if (WindowInitialized == FALSE) return(FALSE);

    WindowInitialized = FALSE;

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

    DeleteRouter("screen");
    endwin();
}

```

```

    return(TRUE);
}

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

extern int ScreenOn(), ScreenOff();

DefineFunction("screen-on",'b',ScreenOn, "ScreenOn");
DefineFunction("screen-off",'b',ScreenOff, "ScreenOff");

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

```


Appendix C - Update Release Notes

The following sections denote the changes and bug fixes for CLIPS versions 6.05, 6.1, 6.2, 6.21, 6.22, and 6.23.

C.1 VERSION 6.23

- **FalseSymbol and TrueSymbol Changes** – The FalseSymbol and TrueSymbol constants were not defined as specified in the *Advanced Programming Guide*. These constants have now been defined as macros so that their corresponding environment companion functions (EnvFalseSymbol and EnvTrueSymbol) could be defined. See sections 3.3.2 and 9.2 for more details.
- **Run-time Program Bug Fix** – Files created by the constructs-to-c function for use in a run-time program generate compilation errors.
- **External Function Interface** - A new function has been added:

GetNextFactInTemplate (see section 4.4.17)

- **Compiler Directives** – The FACT_SET_QUERIES flag has been added. See section 2.2 for more details.
- **New Source Files** – New source files have been added (see section 2.1 for a complete list of source files):

factqpsr.c
factqpsr.h
factquery.c
factquery.h

C.2 VERSION 6.22

- **Function and Macro Corrections** – The following functions and macros were corrected to accept the correct number of arguments as specified in the *Advanced Programming Guide*:

Agenda
BatchStar
EnvGetActivationSalience
EnvBatchStar
EnvFactDeftemplate
EnvFactExistp

EnvFactList
EnvFactSlotNames
EnvGetInstanceInClassAndSubclasses
EnvLoadInstancesFromString
EnvRestoreInstancesFromString
EnvSetOutOfMemoryFunction
FactDeftemplate
FactExistp
FactList
FactSlotNames
GetInstanceInClassAndSubclasses
LoadInstancesFromString
RestoreInstancesFromString
SetOutOfMemoryFunction

C.3 VERSION 6.21

- **Introduction** – Added information on thread\concurrency and garbage collection issues (see sections 1.3 and 1.4).
- **External Function Interface** - Several new functions have been added including:

DeallocateEnvironmentData (see section 9.3.4)
DecrementGCLocks (see section 1.4)
FactDeftemplate (see section 4.4.6)
GetEnvironmentByIndex (see section 9.3.7)
IncrementGCLocks (see section 1.4)

C.4 VERSION 6.2

- **Environments** – It is now possible in an embedded application to create multiple environments into which programs can be loaded (see section 9).
- **External Function Interface** - Several new functions have been added including:

GetClassDefaultsMode (see section 4.12.10)
SetClassDefaultsMode (see section 4.12.19)

- **Run-time Programs** – Support for environments requires some changes in code for loading run-time programs (see section 5).

- **Compiler Directives** – Two new flags have been added: `ENVIRONMENT_API_ONLY` and `ALLOW_ENVIRONMENT_GLOBALS`.
- **New Source Files** – New source files have been added (see section 2.1 for a complete list of source files):

envrnmnt.c
envrnmnt.h

- **Deleted Source Files** – The following source files have been removed (see section 2.1 for a complete list of source files):

extobj.h

C.5 VERSION 6.1

- **C++ Compatible** – The CLIPS source code can now be compiled using either an ANSI C or C++ compiler. Minimally, non-ANSI C compilers must support full ANSI style function prototypes and the void data type in order to compile CLIPS.
- **Obsolete External Function Interface Changes** - The following functions should be replaced with the specified functions. If `clips.h` is included in your C files, macros will automatically map these functions to their new names.

CLIPSCallFunction (use **FunctionCall** instead)
ExitCLIPS (use **ExitRouter** instead)
GetcCLIPS (use **GetcRouter** instead)
InitializeCLIPS (use **InitializeEnvironment** instead)
PrintCLIPS (use **PrintRouter** instead)
UngetcCLIPS (use **UngetcRouter** instead)

- **Obsolete Constants and Variables Changes** - The following constants and variables should be replaced with the specified replacements. If `clips.h` is included in your C files, macros will automatically map the old names to their new names.

CLIPS_FALSE (use **FALSE** instead)
CLIPS_TRUE (use **TRUE** instead)
CLIPSFalseSymbol (use **FalseSymbol** instead)
CLIPSTrueSymbol (use **TrueSymbol** instead)
WCLIPS (use **WPROMPT** instead)

- **Source File Name Changes** - The names of the following source files have been changed:

clipsmem.h (use **memalloc.h** instead)

memory.c (use **memalloc.c** instead)

- **Compiler Directives** – Some of the compiler directive flags in the file setup.h have been changed. The CLP_TEXTPRO and CLP_HELP flags have been renamed to TEXTPRO_FUNCTIONS and HELP_FUNCTIONS. The ANSI_COMPILER flag has been removed.
- **External Function Interface** - Several new functions have been added including:

GetNextInstanceInClassAndSubclasses (see section 4.13.15)

- **New Source Files** – Several new source files have been added (see section 2.1 for a complete list of source files):

parsefun.c

parsefun.h

proffun.c

proffun.h

sortfun.c

sortfun.h

C.6 VERSION 6.05

- **Compiler Directives** - The CLP_EDIT flag in the file setup.h has been renamed to EMACS_EDITOR.
- **External Function Interface** - Several new functions have been added including:

BatchStar (see section 4.1.4)

Build (see section 4.1.7)

Eval (see section 4.1.9)

FactExistp (see section 4.4.7)

FactSlotNames (see section 4.4.10)

GetFactList (see section 4.4.12)

LoadFactsFromString (see section 4.4.20)

LoadInstancesFromString (see section 4.13.19)

RestoreInstancesFromString (see section 4.13.22)

Index

ActivateRouter	169	BLOAD_INSTANCES	14
Ada	iv, 153	BLOAD_ONLY	14, 15
AddClearFunction	45, 181	bload-instances	14, 121
AddDouble	36, 37	BLOCK_MEMORY	14, 174
AddEnvironmentCleanupFunction ..	182, 189	boolean	34
AddLong	36, 37	FALSE	34
AddPeriodicFunction	46, 181	TRUE	34
AddResetFunction	46, 181	BrowseClasses	109
AddRouter	170, 181	browse-classes	109
AddRunFunction	83, 181	bsave	14, 48
AddSymbol	32, 36, 37	BSAVE_INSTANCES	15
Advanced Programming Guide	iv, v	bsave-instances	15, 121
Agenda	213	build	19, 48, 151, 216
agenda	15, 84	C	iii, 153
AllocateEnvironmentData	179, 183, 188	call-next-handler	17
ALLOW_ENVIRONMENT_GLOBALS	13,	call-next-method	17
179, 184, 191, 215		call-specific-method	17
ANSI	19	ClassAbstractP	109
ANSI_COMPILER	216	class-abstractp	110
any-factp	17	ClassReactiveP	110
any-instancep	18	class-reactivep	110
ArgCountCheck	25	ClassSlots	110
ArgRangeCheck	25	class-slots	110
ArgTypeCheck	28, 30	ClassSubclasses	110
ART	iii	class-subclasses	111
Artificial Intelligence Section	iii	ClassSuperclasses	111
Assert	5, 60, 62, 72	class-superclasses	111
AssertString	5, 61, 72, 154, 155	Clear	5, 45, 48
assert-string	61	ClearFocusStack	84
AssignFactSlotDefaults	62	clear-focus-stack	84
AUXILIARY_MESSAGE_HANDLERS	14	CLIPS	iii
Basic Programming Guide	iv, v, 1, 23	CLIPS_FALSE	215
BASIC_IO	14	CLIPS_TRUE	215
batch*	47	CLIPSFalseSymbol	215
BatchStar	213	CLIPSTrueSymbol	215
BatchStar	47, 216	CLIPSFunctionCall	215
BinaryLoadInstances	121	clipsmem.h	216
BinarySaveInstances	121	CLIPSTrueSymbol	215
BLOAD	14, 47, 149	close	14, 164
BLOAD_AND_BSAVE	14	CLP_EDIT	216
		CLP_HELP	216

CLP_TEXTPRO	216	DEFRULE_CONSTRUCT	16
CommandLoop	150	DefruleHasBreakpoint	77
Common Lisp Object System	iv	DefruleModule	77
compiler directives	13	defrule-module	77
CONFLICT_RESOLUTION_STRATEGIE		DEFTEMPLATE_CONSTRUCT	16
S	15	DeftemplateModule	57
conserve-mem	174, 176	deftemplate-module	57
CONSTRUCT_COMPILER	15, 150	delayed-do-for-all-facts	17
constructs-to-c	19, 149, 213	delayed-do-for-all-instances	18
COOL	iv	delete\$	18
create\$	18	DeleteActivation	85
CreateEnvironment	179, 181, 184, 189	DeleteInstance	5, 122
CreateFact	60, 62, 69, 70	DeleteRouter	171
CreateMultifield	39	dependencies	18
CreateRawInstance	122	dependents	18
DeactivateRouter	171	DescribeClass	112
DeallocateEnvironmentData	179, 184, 214	describe-class	112
DEBUGGING_FUNCTIONS	15	DestroyEnvironment	179, 185, 189
DecrementFactCount	65, 72	DirectGetSlot	123
DecrementGCLocks	6, 214	DirectPutSlot	5, 123
DecrementInstanceCount	122	do-for-all-facts	17
DefclassModule	111	do-for-all-instances	18
defclass-module	111	do-for-fact	17
DEFFACTS_CONSTRUCT	15	do-for-instance	18
DeffactsModule	74	DOPTToDouble	28
deffacts-module	74	DOPTToFloat	28
DEFFUNCTION_CONSTRUCT	15	DOPTToInteger	28
DeffunctionModule	98	DOPTToLong	28
deffunction-module	98	DOPTToPointer	28
deffunctions		DOPTToString	28
calling from C	49	DOTToDouble	28
DEFGENERIC_CONSTRUCT	15	DOTToFloat	28
DefgenericModule	101	DOTToInteger	28
defgeneric-module	101	DOTToLong	28
DEFGLOBAL_CONSTRUCT	16	DOTToPointer	28
DefglobalModule	92	DOTToString	28
defglobal-module	92	DribbleActive	55
DefineFunction	21, 32, 34, 35, 36, 38	DribbleOff	55
DefineFunction2	23	dribble-off	55
DEFINSTANCES_CONSTRUCT	16	DribbleOn	55
DefinstancesModule	138	dribble-on	55
definstances-module	138	DYNAMIC_SALIENCE	16
DEFMODULE_CONSTRUCT	16	dynamic-get	123

dynamic-put.....	123	factquery.c.....	213
EMACS_EDITOR.....	13, 16 , 191, 216	factquery.h.....	213
embedded application.....	45	facts.....	15, 66 , 154
EnvBatchStar.....	213	FactSlot.....	67
EnvDefineFunction.....	187	FactSlotNames.....	214
EnvDefineFunction2.....	187 , 190	FactSlotNames.....	216
EnvFactDeftemplate.....	213	fact-slot-names.....	67
EnvFactExistp.....	213	FALSE.....	215
EnvFactList.....	214	FalseSymbol.....	34 , 213, 215
EnvFactSlotNames.....	214	fetch.....	19
EnvFalseSymbol.....	213	files	
EnvGetActivationSaliency.....	213	header	
EnvGetNextInstanceInClassAndSubclasses.....	214	clips.h.....	21, 28, 45, 167, 169, 174, 182
ENVIRONMENT_API_ONLY.....	17 , 182, 191, 215	envrnmnt.h.....	182
environmentData.....	179	memalloc.h.....	174
EnvLoadInstancesFromString.....	214	router.h.....	167, 169
EnvRestoreInstancesFromString.....	214	setup.h.....	12 , 13, 16, 150, 174
envrnmnt.c.....	215	source	
envrnmnt.h.....	189, 215	main.c.....	21, 43, 45, 145, 147, 150, 161, 162
EnvSetOutOfMemoryFunction.....	214	memalloc.c.....	173
EnvTrueSymbol.....	213	sysdep.c.....	13
EnvUserFunctions	21, 150	find-all-facts.....	17
eval.....	19, 49 , 151, 216	find-all-instances.....	18
EX_MATH.....	13, 17	FindDefclass.....	112
ExitCLIPS.....	215	FindDeffacts.....	74
ExitRouter.....	167 , 215	FindDeffunction.....	98
explode\$.....	18	FindDefgeneric.....	102
EXT_IO.....	17	FindDefglobal.....	92
external address.....	35	FindDefinstances.....	138
extobj.h.....	215	FindDefmessageHandler.....	133
FACT_SET_QUERIES.....	17 , 213	FindDefmodule.....	141
FactDeftemplate.....	214	FindDefrule.....	77 , 154
FactDeftemplate.....	65 , 214	FindDeftemplate.....	57
FactExistp.....	214	find-fact.....	17
FactExistp.....	65 , 216	FindInstance.....	124
fact-existp.....	65	find-instance.....	18
FactIndex.....	66	first\$.....	18
fact-index.....	66	float.....	26
FactList.....	214	Focus.....	85
factqpsr.c.....	213	format.....	17, 164, 165
factqpsr.h.....	213	FORTTRAN.....	153
		FunctionCall.....	5, 49 , 215

functions	
argument access	25, 157
calling from C	49
external.....	21 , 25, 27
library	
exit.....	167
free	173
getc.....	168
malloc.....	173, 175, 176
printf.....	168
sprintf	62
string conversion	
Convert_to_C_String	193
MakeStringDsc.....	157 , 193, 200
user-defined.....	21
garbage collection	3
genalloc	173
GENERIC	13
generic functions	
calling from C	49
genfree.....	173
GetActivationName	85
GetActivationPPForm.....	86
GetActivationSalience	86
GetAgendaChanged	86
GetAutoFloatDividend.....	50
get-auto-float-dividend.....	50
GetcCLIPS	215
GetClassDefaultsMode	112 , 214
get-class-defaults-mode	112
GetConserveMemory	174
GetcRouter	168 , 169, 215
GetCurrentEnvironment.....	185 , 191
GetCurrentModule	141
get-current-module.....	141
GetDefclassList.....	113
get-defclass-list	113
GetDefclassName.....	113
GetDefclassPPForm.....	113
GetDefclassWatchInstances.....	114
GetDefclassWatchSlots.....	114
GetDeffactsList	74
get-deffacts-list.....	74
GetDeffactsName.....	75
GetDeffactsPPForm	75
GetDeffunctionList	98
get-deffunction-list.....	99
GetDeffunctionName	99
GetDeffunctionPPForm	99
GetDeffunctionWatch	99
GetDefgenericList.....	102
get-defgeneric-list	102
GetDefgenericName.....	102
GetDefgenericPPForm.....	103
GetDefgenericWatch.....	103
GetDefglobalList.....	92
get-defglobal-list	92
GetDefglobalName	93
GetDefglobalPPForm.....	93
GetDefglobalValue	93
GetDefglobalValueForm.....	93
GetDefglobalWatch	94
GetDefinstancesList.....	138
get-definstances-list	139
GetDefinstancesName.....	139
GetDefinstancesPPForm	139
GetDefmessageHandlerList	134
get-defmessage-handler-list.....	134
GetDefmessageHandlerName.....	134
GetDefmessageHandlerPPForm	134
GetDefmessageHandlerType	135
GetDefmessageHandlerWatch.....	135
GetDefmethodDescription	105
GetDefmethodList.....	106
get-defmethod-list	106
GetDefmethodPPForm.....	106
GetDefmethodWatch	106
GetDefmoduleList.....	141
get-defmodule-list	142
GetDefmoduleName	142
GetDefmodulePPForm.....	142
GetDefruleList	77
get-defrule-list.....	78
GetDefruleName	78
GetDefrulePPForm.....	78
GetDefruleWatchActivations.....	78

GetDefruleWatchFirings.....	79	GetNextDefmessageHandler.....	135
GetDeftemplateList.....	57	GetNextDefmethod.....	107
get-deftemplate-list.....	57	GetNextDefmodule.....	142
GetDeftemplateName.....	58	GetNextDefrule.....	79
GetDeftemplatePPForm.....	58	GetNextDeftemplate.....	59
GetDeftemplateWatch.....	58	GetNextFact.....	69
GetDOBegin.....	30	GetNextFactInTemplate.....	69, 213
GetDOEnd.....	30	GetNextInstance.....	125
GetDOLength.....	30	GetNextInstanceInClass.....	126
GetDynamicConstraintChecking.....	50	GetNextInstanceInClassAndSubclasses...214	
get-dynamic-constraint-checking.....	50	GetNextInstanceInClassAndSubclasses 126,	216
GetEnvironmentByIndex.....	185, 191, 214	GetpDOBegin.....	30
GetEnvironmentData.....	186, 188	GetpDOEnd.....	30
GetEnvironmentIndex.....	179, 186, 191	GetpDOLength.....	30
GetFactDuplication.....	67	get-profile-percent-threshold.....	19
get-fact-duplication.....	67	GetpType.....	27
GetFactList.....	67, 216	GetpValue.....	30, 37
get-fact-list.....	67	GetResetGlobals.....	95
GetFactListChanged.....	68	get-reset-globals.....	95
GetFactPPForm.....	68	GetSalienceEvaluation.....	87
GetFactSlot.....	68	get-salience-evaluation.....	16, 88
GetFocus.....	87	GetSequenceOperatorRecognition.....	50
get-focus.....	87	get-sequence-operator-recognition.....	50
GetFocusStack.....	87	GetStaticConstraintChecking.....	50
get-focus-stack.....	87	get-static-constraint-checking.....	51
GetGlobalsChanged.....	94	GetStrategy.....	88
GetIncrementalReset.....	79	get-strategy.....	15, 88
get-incremental-reset.....	18, 79	GetType.....	27
GetInstanceClass.....	124	GetValue.....	30, 37
GetInstanceName.....	124	GetWatchItem.....	56
GetInstancePPForm.....	125	HELP_DEFAULT.....	17
GetInstancesChanged.....	125	HELP_FUNCTIONS.....	17, 216
GetMethodRestrictions.....	107	I/O router.....	163, 203
get-method-restrictions.....	107	priority.....	166
GetMFTType.....	30, 69, 70	IMPERATIVE_MESSAGE_HANDLERS.....	17
GetMFValue.....	30, 69, 70	IMPERATIVE_METHODS.....	17
GetNextActivation.....	87	implode\$.....	18
GetNextDefclass.....	114	INCREMENTAL_RESET.....	18
GetNextDeffacts.....	75	IncrementFactCount.....	61, 69, 70, 72
GetNextDeffunction.....	100	IncrementGCLocks.....	6, 214
GetNextDefgeneric.....	103	IncrementInstanceCount.....	127
GetNextDefglobal.....	94		
GetNextDefinstances.....	139		

Inference Corporation	iii	list-defmodules	143
InitCImage	150	ListDefrules	80
InitializeCLIPS	215	list-defrules	80
InitializeEnvironment	45, 51 , 150, 154, 179, 181, 189, 215	ListDeftemplates	59
insert\$	18	list-deftemplates	59
installation of CLIPS	9	ListFocusStack	88
instance address	35	list-focus-stack	88
Instance manipulation from C	121	load	14, 51 , 145, 147, 149, 151, 154
INSTANCE_PATTERN_MATCHING	18	LoadFacts	71 , 151
INSTANCE_SET_QUERIES	18	load-facts	71, 151
Instances	129	LoadFactsFromString	71 , 216
integer	26	LoadInstances	129 , 151
integration	1	load-instances	129, 151
Interfaces Guide	v	LoadInstancesFromString	214
IsDefclassDeletable	114	LoadInstancesFromString	129 , 216
IsDeffactsDeletable	76	logical names	163
IsDeffunctionDeletable	100	stdin	165
IsDefgenericDeletable	104	stdout	165
IsDefglobalDeletable	95	t	165
IsDefinstancesDeletable	140	wdialog	165 , 173
IsDefmessageHandlerDeletable	136	wdisplay	66, 137, 165
IsDefmethodDeletable	108	werror	51, 165
IsDefruleDeletable	80	wprompt	165
IsDeftemplateDeletable	59	wtrace	165
LISP	iii	wwarning	165
ListDefclasses	115	LOGICAL_DEPENDENCIES	18
list-defclasses	115	lowercase	19
ListDeffacts	76	main	144, 145, 150
list-deffacts	76	MakeInstance	5, 130
ListDeffunctions	100	make-instance	130
list-deffunctions	100	Matches	80
ListDefgenerics	104	MAXIMUM_ENVIRONMENT_POSITIO	
list-defgenerics	104	NS	189
ListDefglobals	95	memalloc.c	216
list-defglobals	95	memalloc.h	216
ListDefinstances	140	member\$	18
list-definstances	140	memory management	173
ListDefmessageHandlers	136	memory.c	216
list-defmessage-handlers	136	MemoryUsed	175
ListDefmethods	108	MemRequests	174
list-defmethods	108	mem-requests	174
ListDefmodules	143	MemUsed	175
		mem-used	175

Message-passing from C.....	131	release-mem	175
MULTIFIELD_FUNCTIONS	18, 19	RemoveBreak.....	81
NASA.....	iii	remove-break	81
next-handlerp.....	17	RemoveClearFunction	52
next-methodp.....	17	RemovePeriodicFunction.....	52
nth\$.....	18	RemoveResetFunction	52
OBJECT_SYSTEM	19	RemoveRunFunction	89
open.....	14, 164	ReorderAgenda	89
override-next-handler	17	replace\$	18
override-next-method	17	RerouteStdin.....	150
parameter.....	27	Reset.....	5, 47, 53 , 97, 154
parsefun.c	216	rest\$.....	18
parsefun.h.....	216	RestoreInstances.....	130
PopFocus.....	88	restore-instances.....	130, 131
pop-focus.....	88	RestoreInstancesFromString	214
portability	1	RestoreInstancesFromString	131 , 216
Portability Note		Retract	72 , 154, 155
format.....	3	RtnArgCount.....	25
ppdeffacts.....	15	RtnDouble	26
ppdefrule	15	RtnLexeme	26
PreviewSend.....	137	RtnLong	26
preview-send	137	RtnUnknown	27 , 30
PrintCLIPS.....	215	Run.....	90 , 154, 181
printout.....	14, 164, 165	RUN_TIME	15, 19 , 150
print-region.....	19	run-time module	149
PrintRouter.....	154, 166, 168 , 215	Save.....	53
profile.....	19	SaveFacts	72
profile-info	19	save-facts.....	73
profile-reset.....	19	SaveInstances.....	131
PROFILING_FUNCTIONS	19	save-instances.....	131
proflfun.c.....	216	Send.....	5, 131
proflfun.h.....	216	SetActivationSalience	90
progn\$	18	SetAgendaChanged.....	90
PTIEF.....	187	SetAutoFloatDividend	53
PTIF	187	set-auto-float-dividend.....	53
PutFactSlot.....	62, 71	SetBreak.....	81
read.....	14, 164, 165	set-break	81
readline.....	17, 165	SetClassDefaultsMode.....	115 , 214
Reference Manual	v, vii	set-class-defaults-mode.....	115
Refresh	81	SetConserveMemory.....	176
RefreshAgenda.....	89	SetCurrentEnvironment	179, 186 , 191
refresh-agenda.....	16, 89	SetCurrentEnvironmentByIndex.....	179, 186 , 191
ReleaseMem.....	175		

SetCurrentModule.....	143	SetType	37, 39
set-current-module	143	setup flags	13
SetDefclassWatch	115, 116	SetValue.....	37, 39
SetDeffunctionWatch.....	101	SHORT_LINK_NAMES.....	19
SetDefgenericWatch	104	ShowBreaks	82
SetDefglobalValue.....	5, 96	show-breaks	82
SetDefglobalWatch	96	ShowDefglobals.....	97
SetDefmessageHandlerWatch.....	137	show-defglobals.....	97
SetDefmethodWatch	108	SlotAllowedValues	116
SetDefruleWatchActivations	81	slot-allowed-values.....	116
SetDefruleWatchFirings.....	82	SlotCardinality	116
SetDeftemplateWatch	60	slot-cardinality	117
SetDOBegin	39	SlotDirectAccessP.....	117
SetDOEnd	39	SlotExistP.....	117
SetDynamicConstraintChecking.....	53	SlotFacets.....	117
set-dynamic-constraint-checking	54, 150	slot-facets	118
SetFactDuplication.....	73	SlotInitableP.....	118
set-fact-duplication	73	SlotPublicP.....	118
SetFactListChanged	73	SlotRange.....	118
SetGlobalsChanged.....	96	slot-range	119
SetIncrementalReset.....	82	SlotSources.....	119
set-incremental-reset	18, 82	slot-sources.....	119
SetInstancesChanged	132	SlotTypes	119
SetMFTType.....	39	slot-types	119
SetMFValue	39	SlotWritableP.....	120
SetMultifieldErrorValue	39	Smalltalk	iv
SetOutOfMemoryFunction	214	Software Technology Branch.....	iii
SetOutOfMemoryFunction	176	sortfun.c.....	216
SetpDOBegin	39	sortfun.h	216
SetpDOEnd	39	str-cat.....	19
set-profile-percent-threshold.....	19	str-compare.....	19
SetpType	37	str-index.....	19
SetpValue.....	37	STRING_FUNCTIONS.....	19
SetResetGlobals	97	str-length.....	19
set-reset-globals.....	97	SubclassP	120
SetSalienceEvaluation.....	91	subseq\$.....	18
set-salience-evaluation.....	91	subsetp.....	18
SetSequenceOperatorRecognition	54	sub-string.....	19
set-sequence-operator-recognition.....	54	SuperclassP	120
SetStaticConstraintChecking	54	symbol.....	26, 32
set-static-constraint-checking	54	sym-cat.....	19
SetStrategy	91	TEXTPRO_FUNCTIONS	19, 216
set-strategy	15, 91	toss	19

TRUE	215	Unwatch	56
TrueSymbol.....	34 , 213, 215	upcase.....	19
Undefclass.....	5, 120	USER_ENVIRONMENT_DATA.....	188
Undeffacts	5, 76	User's Guide	v, vii
Undeffunction	5, 101	UserFunctions 21 , 26, 45, 144, 150, 156,	
Undefgeneric.....	5, 105	157, 160, 161, 200	
Undefglobal.....	5, 97	ValidInstanceAddress	133
Undefinstances.....	5, 140 , 141	ValueToDouble.....	30, 37
UndefmessageHandler	137	ValueToInteger	30, 37
undefmessage-handler	137	ValueToLong	30, 37
Undefmethod.....	5, 109	ValueToString.....	30, 37
Undefrule	5, 83 , 154	Watch	56 , 154
Undeftemplate.....	5, 60	WCLIPS.....	215
UngetcCLIPS	215	WINDOW_INTERFACE.....	20
UngetcRouter	169 , 215	WPROMPT.....	215
UnmakeInstance.....	5, 132		