Technical Report

MOTOROLA INC.

Cambridge Research Center

Monsoon Software Interface Specifications

Kenneth R. Traub[a], ed.

Michael J. Beckerle[a]

James E. Hicks[b]

Gregory M. Papadopoulos[b]

Andrew Shaw[b]

Jonathan Young[b]

[a]Motorola Cambridge Research Center
[b]Massachusetts Institute of Technology

Motorola Technical Report MCRC-TR-1
January 1990

Also published as MIT Laboratory for Computer Science Computation Structures Group Memo 296.

This report describes research done at the Motorola, Inc., Cambridge Research Center and the Laboratory for Computer Science of the Massachusetts Institute of Technology. Funding for the Laboratory for Computer Science is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-84-K-0099.

LIMITED RIGHTS LEGEND:
Contract No.: ONR Grant No. N00014-89-J-1988
Contractor: Massachusetts Institute of Technology
Subcontractor: Motorola, Inc.
Limited rights in this Technical Data is not subject to an expiration date.

One Kendall Square, Building 200
Cambridge MA 02139

LIMITED RIGHTS LEGEND
Contract No.: ONR Grant No. N00014-89-J-1988
Contractor: Massachusetts Institute of Technology
Subcontractor: Motorola, Inc.
Limited rights in this Technical Data are not subject to an expiration date.

Contents

Chapter 1

Introduction

This document is intended for those who are developing and/or maintaining the software support for Monsoon in all of its incarnations, software and hardware. It defines all of the major interfaces between software components.

This is a work in progress. Nevertheless, the current version should be relatively stable; for the next few months, updates to this document will be made in the form of addenda, rather than by producing newer versions of the entire document. The material in Chapters 1, 2, 3, 4, 6, and 7 are not likely to change significantly in the future. Chapter 5 on IOCF is less stable, and will probably change as more experience is gained with compiling for Monsoon, with resource management, and with debugging of Id programs. Some of the machine control aspects of the MMI (Section 2.5) may also change in the future.

The current version is dated January, 1990, and every page should bear the legend "Version 003, January, 1990." If it is much past January, 1990, you should check to see if any addenda are available, or if the entire document has been revised.[1]

Throughout, square brackets are used to indicate things that definately need revision before this document is final. The lack of square brackets is not to be construed as an indication that something is not subject to change.

Editorial control of this document rests with the Motorola Cambridge Research Center. Contact the editor if you want to make changes.

1.1 Software Overview: User's Perspective

The purpose of the Monsoon software system is to provide a development environment for experimenting with the Monsoon architecture and the programming languages in which Monsoon programs are written. The software system must be flexible enough to accomodate a variety of language/execution vehicles for performing such experiments, and provide a level of functionality at least as good as the present TTDA Id World.

For the time being, the only language for programming Monsoon is Id, so the different ways an experimenter can use the development system can be classified as different ways of executing an Id program. (Monsoon Assembly Language is discussed in Section 1.5.) The ways of executing Id include:

1. Compiling Monsoon object code from Id and executing on a single-processor Monsoon hardware configuration.

[1] If your copy says "master draft" at the bottom of any page, then you have an internal copy which may be out of date or simply erroneous in places. You should try to obtain a released copy, with all current addenda.

2. Compiling Monsoon object code from Id and executing on a multiple-processor Monsoon hardware configuration.

3. Compiling Monsoon object code from Id and executing on a software emulation of the Monsoon architecture.

4. Compiling Id into abstract Explicit Token Store (ETS) code and executing it on a special ETS interpreter.

5. Compiling Id directly into machine code for an existing von Neumann uniprocessor.

These possibilities have different advantages for different users. For users only interested in experimenting with the Id programming language, (1) and (2) offer the fastest execution vehicle, while (5) is a close substitute in case the user does not have access to Monsoon hardware. For users who want to study the parallel behavior of Id and dataflow architectures, (1) and (2) again are the fastest vehicles, but (3) and (4) have the advantages of offering a wider variety of statistics collection modes (as well as not requiring access to Monsoon hardware). For users who want to study the detailed behavior of Monsoon or who are debugging problems with the hardware, (3) is the logical choice.

This document will primarily focus on execution methods (1), (2), and (3). These three methods require essentially the same software support; indeed, (1) is just a special case of (2), and (3) is the same as (1) but with the processor hardware replaced by a software emulator program. At the time of this writing it is not clear whether the ETS interpreter in (4) will be a separate program from the emulator in (3). Method (5), compiling Id directly for von Neumann uniprocessors, is a totally separate project with its own set of software issues.

Hereafter, the term "Monsoon" will refer to both the hardware and the software emulator, unless otherwise specified.

Given that we are speaking of execution methods (1), (2), and (3), a user of the software system at any given time is performing one of the following activities, each of which involves one or more component programs of the software system (described in detail in the sections which follow):

- Preparing or revising an Id program. Programs: GNU Emacs running Id Mode.
- Compiling an Id program into Monsoon object code. Programs: Id Compiler.
- Loading a compiled program into Monsoon's memory. Programs: Monsoon Loader.
- Executing the program and possibly gathering statistics about its run-time behavior. Programs: Execution Manager, Id Run Time System, and either the Monsoon Interface Software (if running on Monsoon hardware) or MINT (the emulator).
- Debugging an Id program; *i.e.*, examining the state of Monsoon's memory after execution halts due to an error or otherwise. Programs: Id Debugger.
- Reviewing and analyzing the run-time statistics collected during execution. Programs: Statistics Viewer.

The user may switch from one activity to another by explicity invoking the appropriate program. A more productive environment, however, is provided by the "Id World" program, which integrates the above components into a unified X-window framework. This gives the

illusion of a Lisp Machine like development system while avoiding the software complexity that would result if it really were a single large program.

Rationale: Organizing the software system as a collection of small programs is a departure from the TTDA Id World system, in which all activities save editing and compilation were handled by the GITA program. The main advantage of the new structure is obviously one of simplicity: each of the pieces has a small and well-defined task, and more importantly, they are constrained to interact only through well-documented interfaces. It also means that the user can still use the entire system, albeit not as conveniently, if he does not have the window support required by Id World, or if he wants to perform a large number of experiments in "batch" mode.

1.2 Software Architecture

The previous section described the user's activities in interacting with the Monsoon software system and mentioned the components of the software system associated with each activity. Figure 1.1 shows the overall architecture of the software system. A brief description of the function of each component follows.

Id Compiler The Id Compiler compiles an Id program into executable code for Monsoon. As the compiler supports separate compilation of the Id procedures comprising a complete program, it in general requires as input information about previously compiled procedures, and produces as output information for the benefit of subsequent compilations. This is indicated as "separate compilation information" in Figure 1.1.

Loader The Loader loads compiler output into Monsoon's memory, performing relocation and dynamic linking as necessary.

Execution Manager The Execution Manager supervises the execution of an Id program on Monsoon. It initiates the execution of a top-level Id procedure with user-supplied arguments, and reports the result to the user when it finishes. It also handles I/O requests from the running Id program, and optionally gathers statistics during execution. Note that the Execution Manager plays only a supervisory role during execution; it does not participate in the computation being performed.

Id Debugger The Id Debugger allows the user to examine and modify the state of Monsoon after an Id program terminates, whether because of normal termination or because an error occured. In the latter case, the user may make modifications and then have the Execution Manager resume execution. (See also the description of the MONASM Debugger, in Section 1.5, which may also be useful to experts in debugging Id programs.)

Statistics Viewer The Statistics Viewer processes raw data collected by the Execution Manager during execution of an Id program. It allows the user to peruse the data on a graphics terminal, and also produces Postscript files for later hardcopy.

Monsoon Interface Software This software provides the link between the Loader, Execution Manager, and Id Debugger and the actual Monsoon hardware. It converts requests to read and write Monsoon memory into actual VMEbus transactions.

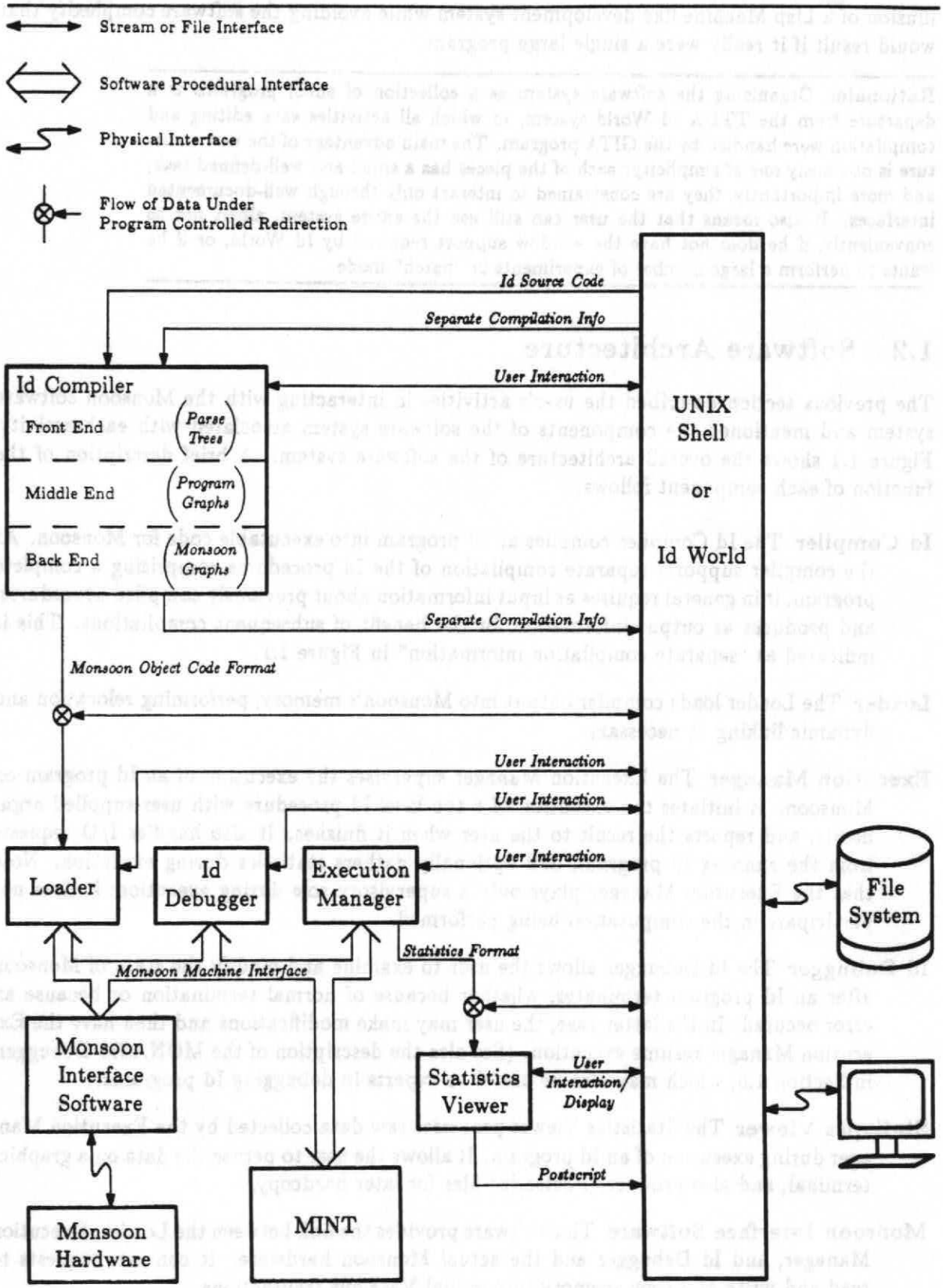

Figure 1.1: Monsoon Software Architecture

MINT The Monsoon INTerpreter provides a "plug-compatible" software emulation of the Monsoon hardware. Not only is it useful for users who do not actually have access to Monsoon hardware, but also it allows a richer set of statistics gathering modes.

Id Run Time System (Not shown in Figure 1.1.) The Id Run Time System, which runs on Monsoon itself, provides memory management and I/O support to an executing Id program. In particular, the Run Time System is the collection of primitives for managing activation frames, heap storage, and I/O, used by the Id compiler in compiling constructs of the Id programming language. In this sense, the Run Time System may be considered part of the user's Id program, a sort of system library.

Id Mode (Not shown in Figure 1.1.) Id Mode is a set of enhancements to GNU Emacs to assist in the preparation of Id programs. Among other features, it includes an indenter for Id.

Id World While it is possible to use the system by invoking the programs above through the Unix shell, Id World provides a more user-friendly interface. It uses the X-window system to present a unified Id development environment, giving the illusion to the user that the components of the Monsoon software system are a single program.

These components interact through the following six interfaces:

Monsoon Machine Interface (MMI) The MMI provides the path by which programs running on the front-end examine and manipulate Monsoon's machine state at the macro-architecture level. It is a set of procedural abstractions, providing calls for reading and writing Monsoon's data, instruction, and token queue memories, as well as for manipulating statistics registers and controlling execution. This same interface is used to access the state of the software emulation of Monsoon, MINT. This permits the Loader, Id Debugger, and Execution Manager to be shared between Monsoon and MINT.

Currently this interface is just defined as a set of procedural abstractions. There are plans to define a network protocol as well, so that programs on either side of the interface would not have to be compiled together. This would have the added benefits that programs on either side of the interface could be written in different programming languages, and it would simplify the organization of systems with more than one front-end processor (any system with more than four PE's).

The MMI is defined in Chapter 2.

Monsoon Object Code Format (MOC) This is the file format for input to the Loader. It is a general purpose format for loading any kind of data into Monsoon, and includes features which support dynamic relocation and linking. MOC itself has no knowledge of any particular programming language, and so can serve as the target for any compiler for Monsoon. MOC format is expressed in terms of CIOBL, and is defined in Chapter 4.

Id Object Code Format Programs compiled by the Id Compiler have a particular structure of which the Id Debugger and to some extent the Execution Manager must be aware. This includes the conventions for representing Id objects and for encoding source locators and identifier names into object modules. Id Object Code Format is expressed in terms of MOC, and is defined in Chapter 5.

Statistics Format This is the file format for recording raw data collected by the Execution Manager during execution of a program on Monsoon. Statistics Format is expressed in terms of CIOBL, and is defined in Chapter 6.

CIOBL The Common Input/Output Base Language is a method for encoding objects such as numbers, character strings, symbols, *etc.*, into files. It provides different encoding methods suitable for different purposes; *e.g.*, a character-based encoding for transmission over electronic mail *vs.* a binary encoding for maximum space efficiency. The MOC and Statistics formats are defined in terms of CIOBL, which is defined in Chapter 7. That chapter also defines a standard library of procedures for the convenience of programs which read or write CIOBL files.

Proto-Memory Manager (PMM) The PMM is a data structure which provides the most primitive level of memory mangagement for the Monsoon system. Its primary role is to allow the resources provided by a particular configuration of Monsoon to be divided among the programs which compete for them, which include the Loader and Id Run Time System. The PMM is the primary way that such programs are insulated from changes in machine configuration. The PMM is described in Chapter 3.

The current design of the Monsoon software system uses the Monsoon hardware as an attached, special-purpose processor to a front-end processor, which is a standard von Neumann CPU running Unix. In this scenario, the Monsoon processor is only active when actually running an Id program. At other times, the Monsoon processor is halted, and other programs running on the front-end are editing user code, compiling it, loading it into Monsoon through the hardware scan path, or examining Monsoon's machine state for debugging. For the purposes of this document and for the first implementation of the software, the reader should assume that the only software which runs on the Monsoon hardware is a user's Id program and the Id Run Time System which supports it.

Rationale: This is not unlike the relationship between the Lisp Machine and the GITA program in the old TTDA Id World system. Ultimately, one would hope for a completely self-sufficient Monsoon system, where software supporting program development including the compiler would run on Monsoon itself. A front-end processor, if it existed at all, would only serve to processes I/O transactions. It is felt, however, that it is premature to move to such a configuration—more experience is needed just running programs on Monsoon, and language features to support systems programming are still in their infancy. There are middle grounds between the two extremes; for example, the Monsoon processor could be running a rudimentary operating system rather than being halted when not executing a user's Id program. From this configuration, the Loader, Execution Manager, and Id Debugger, could be migrated onto Monsoon. At that point, Monsoon can stand alone for the purpose of running programs, and the Compiler and Statistics Viewer can ultimately be written in Id and run on Monsoon. The software architecture was intentionally designed to facilitate exactly this evolutionary path.

1.3 Software Components

This section describes each of the components of the software system to support Id. Some additional components for supporting Monsoon Assembly Language (MONASM) are described in Section 1.5. Also, the Monsoon Interface Software (MIS) component is described in more detail in Section 1.6.

1.3.1 Id Compiler

Function Compiles an Id program into object code for Monsoon.

Inputs (1) One or more top-level procedures or declarations written in the Id programming language. (2) Separate compilation information about other procedures not currently being compiled (see "Discussion," below).

Outputs (1) Monsoon object code in MOC format, conforming to conventions established by Id Object Code Format. (2) Separate compilation and assumption information for use by subsequent invocations of the compiler (see "Discussion," below). (3) Error messages in a human readable form but parseable by a program invoking the compiler (*e.g.*, so that a text editor can locate where errors in a previous compilation occured).

Interaction None, although error messages may be directed to the terminal.

Relevant Specifications Id Language, Monsoon instruction set, Id Run Time System program interface, MOC, Id Object Code Format. [To be defined: separate compilation information specification, error message format.]

Other Features Has options for controlling whether optimization takes place, and also for selecting between the instruction set executable by the hardware and an extended instruction set for gathering idealized statistics from MINT (see "Discussion," below).

Discussion The compiler supports separate compilation; *i.e.*, not all procedures comprising a program need be compiled at once. When compiling a procedure, however, the compiler may need information about other procedures to which it refers, such as the number of arguments, type information, *etc.* This is called "separate compilation information." Separate compilation information produced by the compiler includes not only information for use by subsequent compilations, but also a record of what assumptions were made about prior compilations, so that consistency can be verified before running the compiled program.

Normally, requests to allocate heap memory or frame storage are compiled as calls to the Id Run Time System. For the purposes of gathering idealized statistics from MINT, it is useful to compile these requests as fictitious opcodes, so that statistics can be gathered which are not influenced by Run Time System implementation.

1.3.2 Loader

Function The Loader takes an object code file and loads it into Monsoon's memory.

Inputs Object code file in MOC format.

Outputs Manipulates Monsoon machine state through MMI.

Interaction None.

Relevant Specifications MOC, PMM, MMI.

Other Features Has an option which checks that there are no unresolved dynamic links for a given set of modules, informing the invoking program if there are.

Discussion MOC modules are for the most part relocatable and dynamically linked, so it is up to the loader to manage areas of memory into which it loads object modules. The loader will have a private data structure to keep track of these areas, and also to record unresolved dynamic links. This data structure may be kept in an area of Monsoon memory or perhaps off-line, but in either case should be a PMM area accessed via the MMI so that it may be migrated onto Monsoon at a later time.

1.3.3 Execution Manager

Function Initiates and supervises the actual execution of a program on Monsoon.

Inputs Command-line parameters indicating which procedure to invoke and what arguments to pass.

Outputs Result from executed program, run-time statistics.

Interaction Services I/O requests from the executing Monsoon program.

Relevant Interfaces MMI, Id Object Code Format, Statistics Format.

Other Features Offers a selection of statistics gathering modes.

Discussion During execution, the Execution Manager is in the following loop:

1. Run Monsoon until activity ceases.
2. Read statistics registers, transfer to profiles being collected.
3. Advance the abscissa (timestep) of profiles being collected.
4. Zero statistics registers.
5. Prepare Monsoon for next timestep.
6. Go to Step 1.

The meaning of "until activity ceases" depends on the configuration of Monsoon's queueing system. If Monsoon is configured to enqueue and dequeue from the same queue, activity will cease only when the program is finished, so that a "timestep" corresponds to a complete execution of the program. On the other hand, if configured to enqueue into one queue but dequeue from another, activity will cease after only a portion of the program has executed. "Prepare Monsoon for next timestep" in this case means exchange the queues, and the profiles will be an approximation to the idealized parallelism profiles collected by TTDA GITA. MINT will provide more complex queueing systems offering a variety of statistics modes. See the MINT White Paper[2] for more details.

I/O requests take place during Step (1) above.

1.3.4 Id Debugger

Function Allows the user to interactively examine and modify the state of Monsoon after a Monsoon program has terminated, either due to normal termination or due to an error. It presents the state in a manner as closely tied to the Id source code of the Monsoon program as possible.

[2]K. R. Traub, "MINT White Paper," Motorola Cambridge Research Center Technical Report MCRC-TR-2, October, 1989.

Inputs None.

Outputs Manipulates Monsoon machine state through MMI.

Interaction User dialog for examining and modifying machine state.

Relevant Interfaces MMI, Id Object Code Format, Id Run Time System program interface.

Discussion The debugger requires the compiler to include in object code information such as variable names, *etc.*

1.3.5 Statistics Viewer

Function Analyzes, displays, and facilitates manipulation of run-time statistics gathered by the Execution Manager.

Inputs Run-time statistics in Statistics Format.

Outputs (1) Postscript for hardcopying statistics. (2) Edited statistics files.

Interaction Interactive display and analysis of statistics through X-window graphics protocol.

Relevant Interfaces Statistics Format, Postscript, X-window System.

Discussion Should allow scrolling through large statistics, comparison and scaling of different statistics from same run or from different runs, editing of descriptive information accompanying statistics, perhaps some specific kinds of high-level analysis (*e.g.*, speedup curves).

1.3.6 Monsoon Interface Software

Function Implements the MMI interface to the Monsoon hardware.

Inputs MMI function calls.

Outputs VMEbus transactions to control Monsoon hardware components.

Interaction None.

Relevant Interfaces MMI, Monsoon hardware interface.

Discussion The Monsoon Interface Software actually consists of several subcomponents; see Section 1.6 for more details.

1.3.7 MMI Network Client/Server

Function Extends the connection between the user and Monsoon sides of the MMI interface across the network, by converting MMI calls into remote procedure calls.

Discussion In its simplest form, the MMI is simply a set of library procedures linked individually with the Loader, Execution Manager, and Id Debugger. Each procedure controls some aspect of the Monsoon hardware, typically reading or writing some portion of machine state. In this view, there are really two version of the MMI: one which is part of the Monsoon Interface Software, and another which is part of the MINT program.

A more flexible version of the MMI uses the network as an "extension cord" between the caller of MMI procedures and the callee. In this case, the Loader, Execution Manager, and Id Debugger are each linked with an MMI Client program, and the Monsoon Interface Software and MINT are each linked with an MMI Server program. The MMI Client provides implementations of the MMI functions which make remote procedure calls, while the MMI server receives remote procedure calls from the network and makes the corresponding MMI calls.

See Section 1.4 for a discussion of the various configurations.

1.3.8 MINT

Function Provides a "plug-compatible" software emulation of the Monsoon hardware.

Inputs MMI function calls.

Outputs None.

Interaction Optional indication of activity (*i.e.*, print a dot every 1000 tokens).

Relevant Interfaces MMI, Monsoon instruction set.

Other Features Provides not only the hardware's instruction set, but also additional, fictitious opcodes for gathering idealized statistics. Potentially supports a larger address space and greater number of statistics registers than hardware. Supports a variety of queueing systems, some which model the hardware, others suitable for gathering idealized statistics.

Discussion Most of the code emulating the ALU is machine generated from microcode and/or ETS specifications; see Section 1.6.

1.3.9 Id Run Time System

Function Runs on Monsoon to provide heap and other resource management support to an executing Id program.

Inputs None.

Outputs None.

Interaction None.

Relevant Interfaces Id Object Code Format, Id Run Time System program interface.

Discussion The first version of the Run Time System will provide "system calls" for allocating and deallocating activation frames, for allocating and deallocating heap storage, and to perform some form of I/O. Its duties will eventually grow to encompass automatic reclamation of at least a part of the heap (*i.e.*, garbage collection) as well as more sophisticated I/O, as research in these areas progresses. As it is responsible for managing activation frames, it may also eventually include parallelism controls such as load balancing or throttling.

1.3.10 Id Mode

Function A Major Mode for the GNU Emacs editor for editing Id source code.

Inputs None.

Outputs None.

Interaction Through GNU Emacs.

Relevant Interfaces Id language, GNU internal interfaces.

1.3.11 Id World

Function Integrates all other software system components into a coherent, integrated Id development environment.

Inputs None.

Outputs None.

Interaction Provides a window-based framework including panes for editing, running, viewing statistics, *etc.*

Relevant Interfaces X-window system, command-level interfaces of Id Compiler, Loader, Execution Manager, Id Debugger, Statistics Viewer, and GNU Emacs.

Discussion In addition to simply invoking the other software components, Id World must do some bookkeeping to give the illusion of an integrated environment. For example, if the user directs Emacs to compile a single Id procedure, Id World should supply the Id Compiler not only with the procedure to be compiled, but also the appropriate separate compilation information based on the history of what procedures have already been loaded into Monsoon.

One possible design option is to use GNU Emacs as a substrate for building Id World. It would be good to provide a version which works in the absence of X-window support, but which is not terribly inconvenient to use because of it.

1.4 Software Configurations

The previous sections on software architecture described the pieces of the Monsoon software system, the interfaces through which they interact, and the overall flow of data between them as depicted in Figure 1.1. This section examines how the pieces are assembled to form a complete working environment for particular configurations of the Monsoon development environment. That is, this section focuses on how different software components are linked together to form executable programs, and how those programs are run in one or more processes on one or more hosts to form a complete working environment. The various components are building blocks, and the way in which they are assembled will differ depending on whether one is building an environment for using the emulator interactively, for using the emulator in batch mode, for using a single-processor Monsoon, for using a multi-processor Monsoon, for using a Monsoon located at a remote location, *etc.* The keys to this flexibility are the interfaces through which

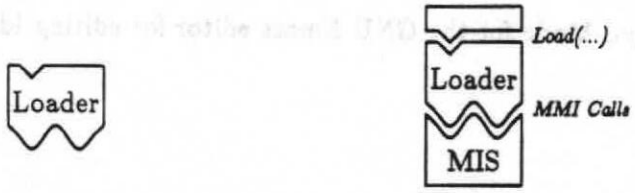

Figure 1.2: Depiction of (a) A Software Component; (b) A Complete Program

the components interact: procedural interfaces between components combined at link time, and file, network, and command-line interfaces between components executing as separate processes.

This section will not present an exhaustive listing of all possible combinations of components, but rather will illustrate the principles involved with a few examples. The point here is not to explain the function of each component as in Sections 1.1 and 1.3, or to illustrate the flow of data between them as in Section 1.2. Rather, the goal is to show which components are combined to form a complete executable program, and how these programs are configured at an operating system level to comprise a complete system. Most of the variation in configurations is centered around the programs found in the lower right quadrant of Figure 1.1—the programs which actually manipulate the state of Monsoon. The illustrations in this section will for the most part omit the more "off line" programs such as the Compiler and the Statistics Viewer, and may not include all paths of data flow, particularly user interaction.

A graphic convention illustrated in Figure 1.2 shows how software components are combined into executable programs. Part (a) of the figure is the graphic for a particular software component, the loader. Each curved surface of the graphic is meant to indicate a particular procedural interface, that is, a path to another software component through subroutine calls. In the case of the loader, the wavy surface at the bottom indicates calls made by the loader to read and write Monsoon's memory, those calls being a part of the Monsoon Machine Interface (MMI), documented in Chapter 2. The notched surface at the top indicates calls to the top-level procedure of the loader, the only call in this interface being the call `load(...)` which directs the loader to load data from some file or stream. The component in part (a) of the figure is not a program which can be run on its own; for example, if it makes a call to an MMI procedure there is no definition of that procedure to respond to the call.

Figure 1.2b shows how the loader component is used as the core of a complete loader program, by linking it with other components that mesh with the loader's interfaces. In this case, the loader is shown linked with two other components. The Monsoon Interface Software (MIS) provides an implementation of MMI calls which manipulates Monsoon hardware through the VMEbus. The small component on top of the loader is a simple command-line interpreter, which takes a command line provided by the Unix Shell (via **argc** and **argv**) and converts it into a call to the top-level loader procedure `load`. The conglomeration of these components, then, is a complete Unix program which loads a program indicated on the command line onto the Monsoon hardware attached to the local host.

Complete programs will appear as rectangles, as shown in Figure 1.2b. There are several things that can be inferred about any group of components forming a rectangle:

- It is complete: there are no subroutines without callers or subroutine calls to undefined procedures, with the exception of a single "main" procedure and O/S calls.

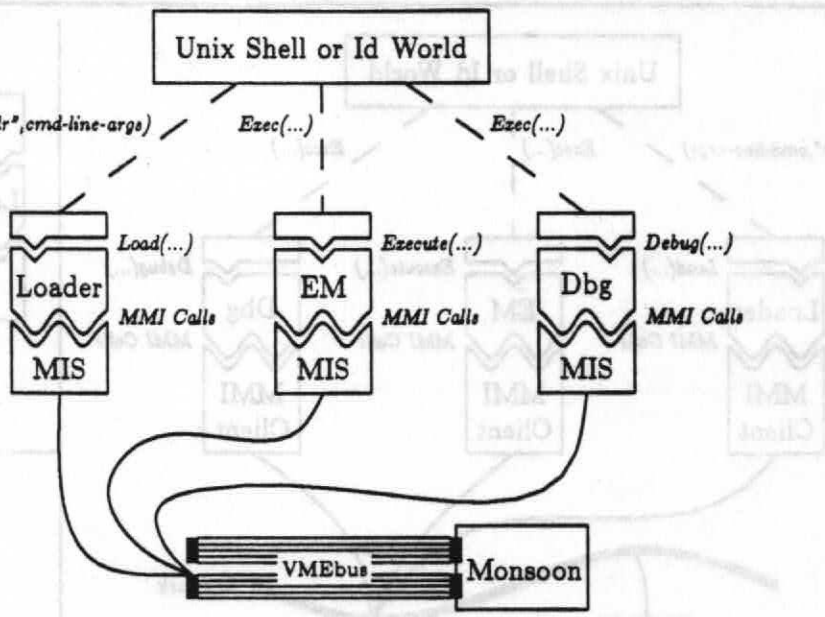

Figure 1.3: Configuration for a Single-PE Monsoon

- Any interaction with other rectangles is through the operating system, typically in the form of I/O.
- The components within the rectangle interact through well-defined procedural interfaces, each depicted by some interior boundary. The components are combined through the implementation language's linking mechanism.
- Because of the previous point, it is highly desirable for all components within a rectangle to be written in the same programming language. On the other hand, little difficulty results if *different* rectangles are written in different languages.
- The lifetimes of all components within a rectangle are tied together, but different rectangles may have different lifetimes. This is important when the programs represented by the rectangles have state.

The main purpose of this section, then, is to illustrate the relationships between the software components as they relate to the above four points.

1.4.1 Configuration for a Single-PE Monsoon

The most straightforward configuration of the Monsoon software system is that supporting a single Monsoon processor on the local host, illustrated in Figure 1.3. Whenever Monsoon is being manipulated, there are three separate entities involved:

1. The Monsoon processor itself, which is accessible via the VMEbus.
2. A shell or other top-level program for invoking programs which manipulate Monsoon.
3. Three programs invoked by the shell at different times, each of which manipulates the machine state of Monsoon. These programs embody the Loader, Execution Manager, and Debugger.

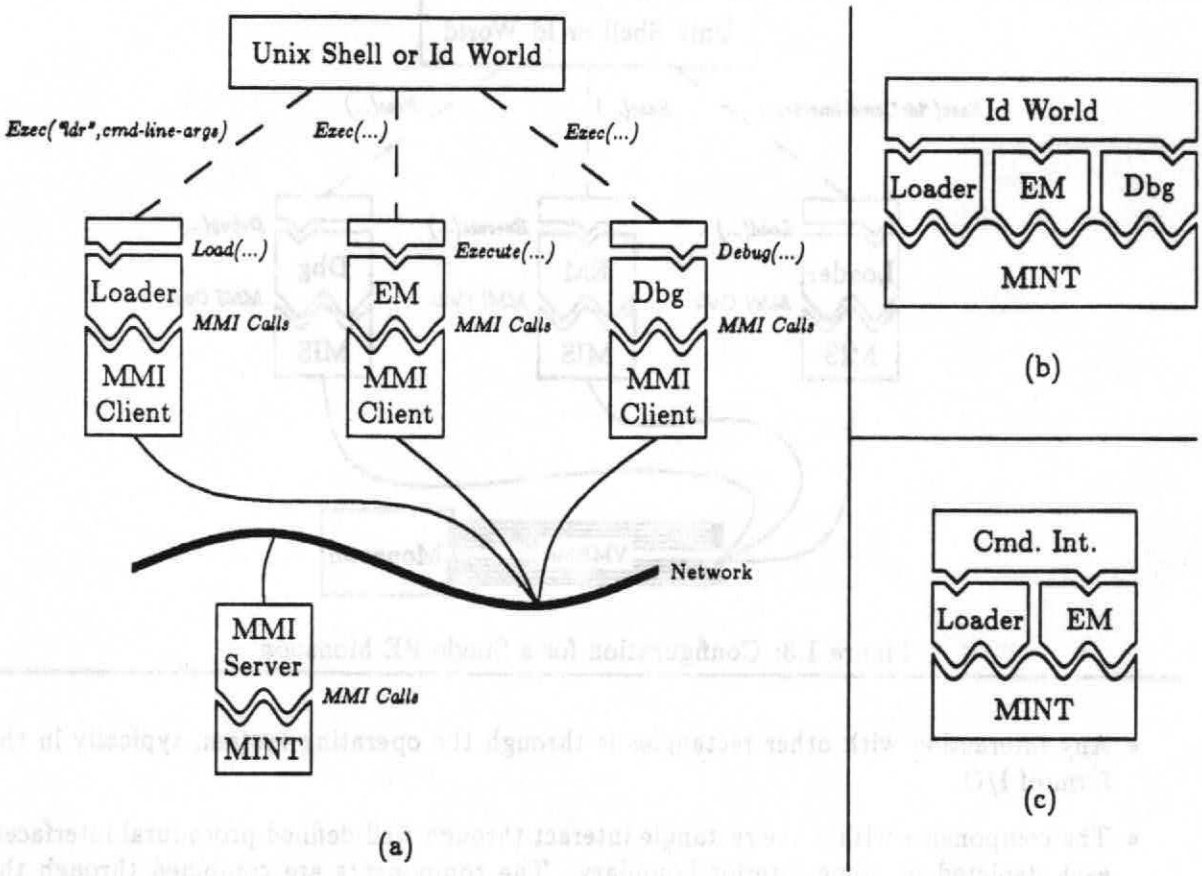

Figure 1.4: Configurations for MINT

The lines drawn between various rectangles loosely indicate O/S-level interaction between them. The dashed lines between the shell and the three other programs are meant to suggest that the latter programs are invoked, at different times, as subprocesses of the shell. The curved lines joining the programs to the VMEbus indicate operaing system calls for VMEbus I/O.

It should be noted that the shell or Id World program also invokes the Id Compiler, Statistics Viewer, and other O/S utilities at various times, but these are omitted from the diagram since they don't involve Monsoon.

One key feature of this configuration is that the processor state of Monsoon persists even as the Loader, Execution Manager, and Debugger come and go. Thus, after the loader is invoked and terminates, the Id Code it loaded into Monsoon is still around when the Execution Manager is invoked to run it.

1.4.2 Configuration for MINT

The software configuration for running MINT is very similar to that for a single-PE Monsoon, as shown in Figure 1.4a. The noteworthy aspect is that MMI calls are extended through the network by the MMI Client and Server components. This allows MINT to run in a separate process from the Loader, Execution Manager, and Debugger, so that its state is maintained across invocations of the latter three programs. Normally, the MINT process would be on the same host as the other programs, so the "network" link does not really go across any physical

network hardware. Rather, it is a software commmunication path provided by the operating system (*e.g.*, a pair of Unix "sockets").

Notice that the use of the MMI Client/Server is not limited to MINT. Indeed, the component labeled "MINT" in Figure 1.4a could be replaced by the Monsoon Interface Software, which would access the Monsoon Hardware via the VMEbus. This configuration is actually preferable to that described in the previous section, for two reasons. First, it shares the same Loader, Execution Manger, and Debugger programs with MINT (because in both cases the inner components are linked with the MMI Client, rather than with the MIS for the hardware and the Client for the emulator). Second, it allows the Monsoon hardware to be located on a different host. While the latter feature was not so important for MINT, it is for the hardware as every host may not have a Monsoon board. See also the next section for why the network link is useful.

Of course, it is not necessary to use the MMI Client/Server with MINT; MINT can be linked directly with the other components, as shown in Figure 1.4b. The disadvantage of this configuration is that it is one big program, which for all practical purposes requires that all the components be written in the same programming language (inter-language working is possible, of course, but is extremely awkward given that the two languages we are interested in are Lisp and C). There is, however, a situation in which a variant of this configuration is useful. If an experimenter wants to run a big simulation in "batch" mode on a supercomputer such as a Cray, a configuration consisting of MINT, the Loader, the Execution Manager, and a trivial command interpreter can be prepared as shown in Figure 1.4c. The command interpreter reads a series of "load" commands followed by one or more "execute" commands, the net effect being to read some compiled code into MINT and produce simulation results (statistics).

1.4.3 Configurations for a Large Monsoon

Finally, Figure 1.5 shows a configuration for a large Monsoon, one with a total of more than eight PE's and I-structure boards. This many boards requires more than one VMEbus cabinet, and therefore more than one host processor. Here the network link is essential, as the front-end software (Loader, *etc.*) must communicate to several other hosts. The MMI Clients are initialized to know that the first eight boards are contacted at one network address, the next eight at another, and so on. Beyond that, all software components are the same.

1.5 Monsoon Assembly Language

Monsoon is a general purpose multiprocessor, and may be programmed in a variety of languages, not just Id. Of course, for each language to be used on Monsoon there must be a compiler, and Id is the only high-level language for which we expect to have a compiler in the near future. On the other hand, we will be defining an assembly language for Monsoon, and an assembler for that language. Monsoon assembly language is called MONASM.

The software architecture from the MONASM programmer's point of view is pretty much the same as in Figure 1.1. The only difference is that the Id Compiler is replaced by the Monsoon Assembler, MONASM, and the Id Debugger is replaced by the MONASM Debugger. All other components of the software system are identical. Figure 1.6 shows the software architecture with the MONASM programs added.

The two new components are described below.

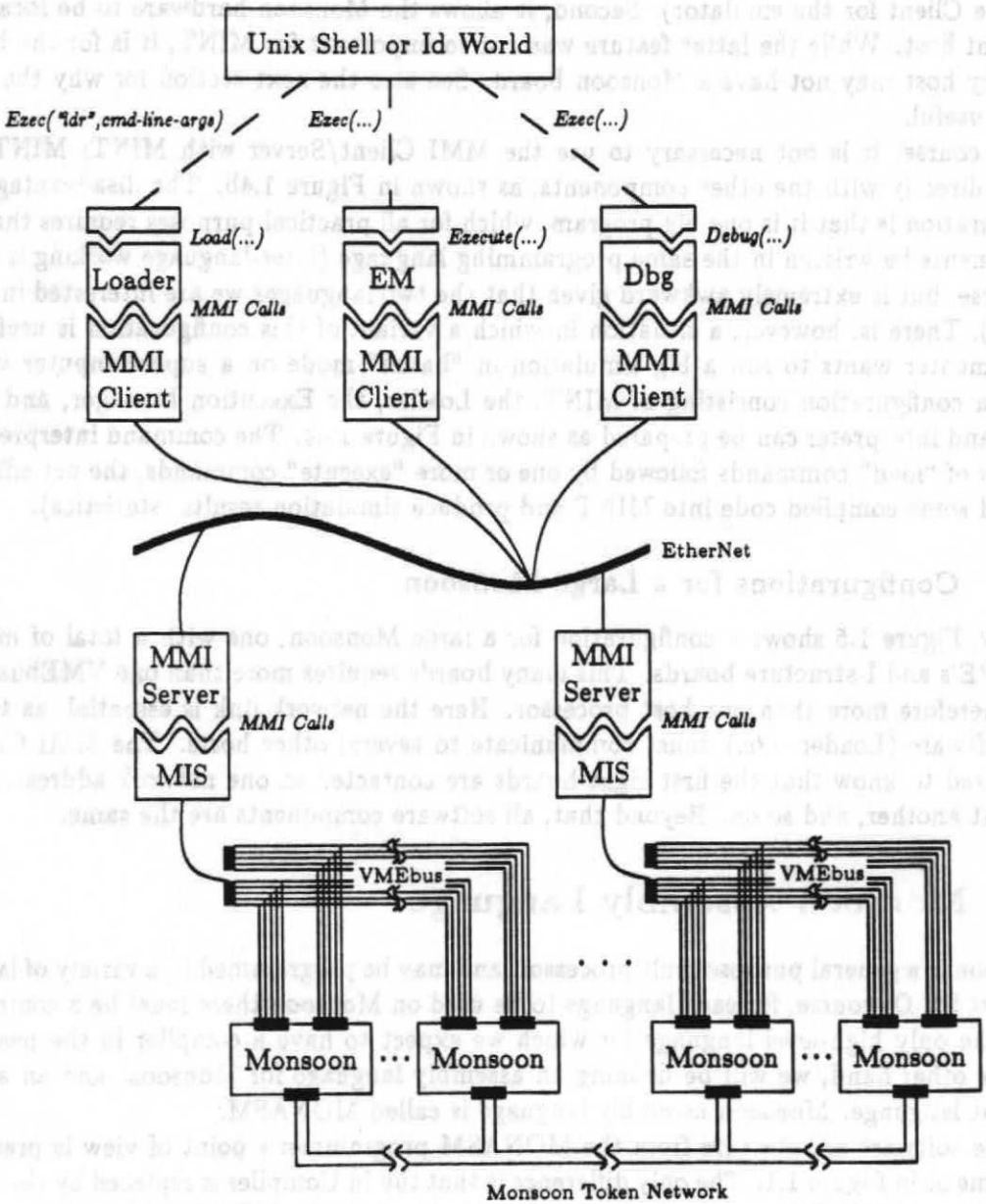

Figure 1.5: Configuration for a Large Monsoon

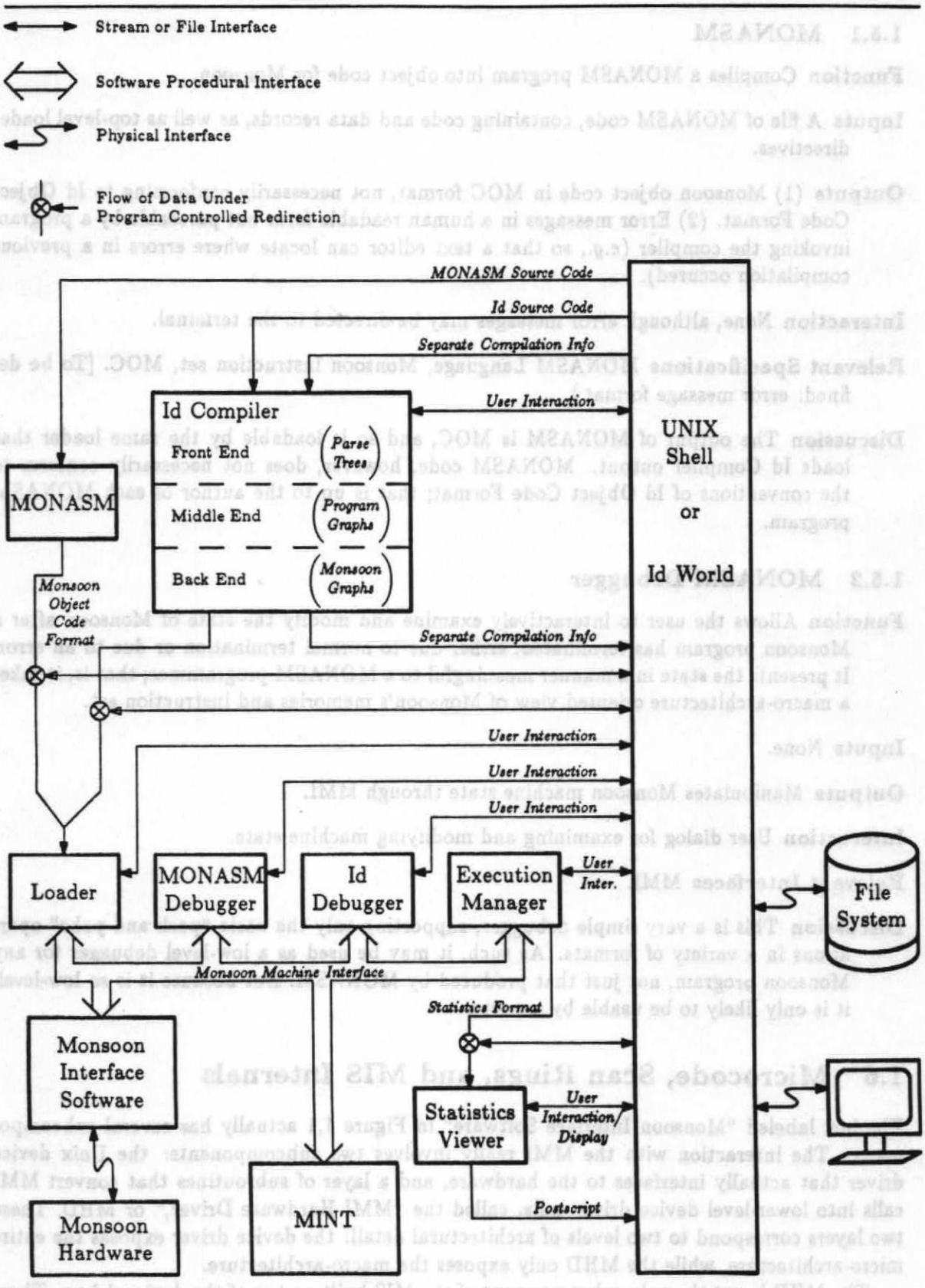

Figure 1.6: Monsoon Software Architecture with MONASM

1.5.1 MONASM

Function Compiles a MONASM program into object code for Monsoon.

Inputs A file of MONASM code, containing code and data records, as well as top-level loader directives.

Outputs (1) Monsoon object code in MOC format, not necessarily conforming to Id Object Code Format. (2) Error messages in a human readable form but parseable by a program invoking the compiler (*e.g.*, so that a text editor can locate where errors in a previous compilation occured).

Interaction None, although error messages may be directed to the terminal.

Relevant Specifications MONASM Language, Monsoon instruction set, MOC. [To be defined: error message format.]

Discussion The output of MONASM is MOC, and so is loadable by the same loader that loads Id Compiler output. MONASM code, however, does not necessarily conform to the conventions of Id Object Code Format; that is up to the author of each MONASM program.

1.5.2 MONASM Debugger

Function Allows the user to interactively examine and modify the state of Monsoon after a Monsoon program has terminated, either due to normal termination or due to an error. It presents the state in a manner meaningful to a MONASM programmer; that is, it takes a macro-architecture oriented view of Monsoon's memories and instruction set.

Inputs None.

Outputs Manipulates Monsoon machine state through MMI.

Interaction User dialog for examining and modifying machine state.

Relevant Interfaces MMI.

Discussion This is a very simple debugger, supporting only the basic "peek and poke" operations in a variety of formats. As such, it may be used as a low-level debugger for any Monsoon program, not just that produced by MONASM. But because it is so low-level, it is only likely to be usable by experts.

1.6 Microcode, Scan Rings, and MIS Internals

The box labeled "Monsoon Interface Software" in Figure 1.1 actually has several subcomponents. The interaction with the MMI really involves two subcomponents: the Unix device driver that actually interfaces to the hardware, and a layer of subroutines that convert MMI calls into lower-level device driver calls, called the "MMI Hardware Driver," or MHD. These two layers correspond to two levels of architectural detail: the device driver exposes the entire micro-architecture, while the MHD only exposes the macro-architecture.

The MHD is not the only subcomponent of the MIS built on top of the device driver. There is a microcode loader, which initializes the microcode memories and other configuration registers

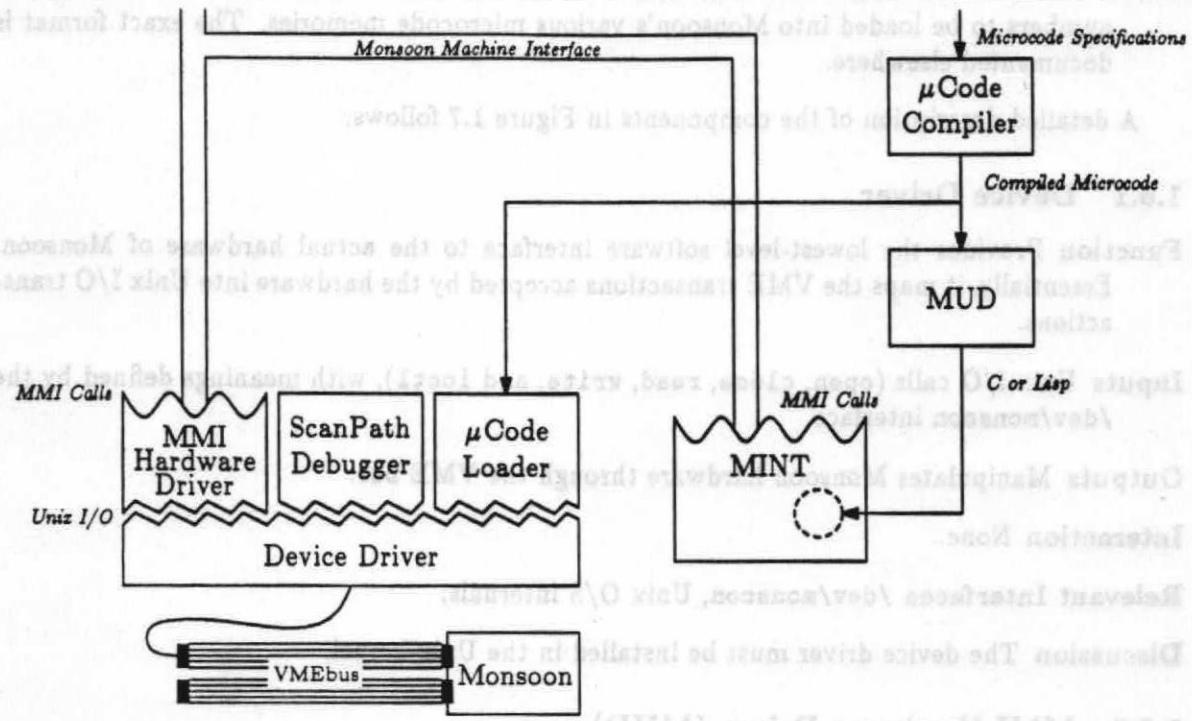

Figure 1.7: Internal Details of MIS and Related Software

of a freshly powered-up Monsoon. There is also a Scan Path Debugger, for debugging Monsoon at the micro-architecture level (recall that the MONASM Debugger debugs Monsoon at the macro-architecture level). The Scan Path Debugger is only likely to be used for diagnosing hardware faults.

Figure 1.7 illustrates the internal details of MIS; MIS corresponds to the large block in the lower left, consisting of the Device Driver, the MHD, the Scan Path Debugger, and the Microcode Loader. Also shown in the figure is MINT, and also two other components related to microcode. The Microcode Compiler takes a description of microcode and compiles it into flat microcode tables that can be loaded into Monsoon. The Monsoon Microcode Decompiler, or MUD, takes microcode tables and produces the core of an emulator for the instruction set defined by that microcode. The MINT program, therefore, is mostly written by the MUD program, not by a human programmer.

The new interfaces shown in Figure 1.7 are as follows.

/dev/monsoon This is the Unix interface to the Device Driver, which is installed into the Unix kernel. Unix defines the general shape of all device driver interfaces; they are accessed through the O/S calls `open`, `close`, `read`, `write`, and `ioctl`. The meaning of these calls when operating upon the Monsoon Device Driver is the definition of this interface, and is documented elsewhere.

Microcode Specification Language The microcode for Monsoon is expressed in a powerful microcode description language, based on Common Lisp-style macros. It is documented elsewhere.

Compiled Microcode Format The output of the Microcode Compiler is basically a file of

numbers to be loaded into Monsoon's various microcode memories. The exact format is documented elsewhere.

A detailed description of the components in Figure 1.7 follows.

1.6.1 Device Driver

Function Provides the lowest-level software interface to the actual hardware of Monsoon. Essentially, it maps the VME transactions accepted by the hardware into Unix I/O transactions.

Inputs Unix I/O calls (`open`, `close`, `read`, `write`, and `ioctl`), with meanings defined by the `/dev/monsoon` interface.

Outputs Manipulates Monsoon hardware through the VME bus.

Interaction None.

Relevant Interfaces `/dev/monsoon`, Unix O/S internals.

Discussion The device driver must be installed in the Unix kernel.

1.6.2 MMI Hardware Driver (MHD)

Function Converts MMI calls into low-level Device Driver calls.

Inputs MMI calls.

Outputs Unix calls to the `/dev/monsoon` driver.

Interaction None.

Relevant Interfaces MMI, `/dev/monsoon`.

Discussion The MHD may have two modes of operation: bootstrap mode and normal mode. In bootstrap mode the processor is always halted while processing MMI calls, and all access to machine state is through the scan rings. In normal mode the Monsoon processor is running a special MMI "operating system," which interacts with the MHD through the I/O mechanism provided by Monsoon (*i.e.*, VME interrupts and direct VME data memory access). It is quite likely that the first working version of MHD will not distinguish between bootstrap and normal mode, effectively always operating in bootstrap mode. See Section 5.9 for more details of the bootstrap process.

1.6.3 Scan Path Debugger

Function Allows the user to interactively examine and modify the micro-architectural machine state of Monsoon. This includes access to all of the scan rings, as well as the VME interface registers and other accessible parts of the hardware.

Inputs None.

Outputs Manipulates Monsoon through the Device Driver.

Interaction User dialog for examining and modifying machine state.

Relevant Interfaces `/dev/monsoon`.

Discussion In addition to an interactive mode, this program should also accept a "script" of commands to be executed in sequence. This makes it possible to use the program as a general purpose initialization tool, in addition to an interactive debugger. If this tool can parse compiled microcode files and load them, then there will be no need for a separate microcode loader program.

1.6.4 Microcode Loader

Function Loads compiled microcode into the control store of Monsoon.

Inputs Compiled microcode file.

Outputs Manipulates Monsoon through the Device Driver.

Interaction None.

Relevant Interfaces Compiled microcode format, `/dev/monsoon`.

Discussion This may not actually be a separate program, but rather one of the functions performed by the Scan Path Debugger.

1.6.5 Microcode Compiler

Function Compiles a description of Monsoon's microcode into actual microcode tables.

Inputs Microcode description language.

Outputs Microcode in compiled microcode format.

Interaction None.

Relevant Interfaces Microcode language, compiled microcode format.

1.6.6 Monsoon Microcode Decompiler (MUD)

Function Takes a template for generating emulation code along with compiled microcode and generates the core of an efficient emulator for the instruction set expressed by the microcode.

Inputs Emulation template, compiled microcode.

Outputs C or Lisp code comprising the core of an emulator (MINT).

Interaction None.

Relevant Interfaces Compiled microcode format, emulation template language, internal interfaces of MINT.

Discussion A more detailed discussion of MUD, the emulation template language, and the internal interfaces of MINT are described elsewhere.

Chapter 2

The Monsoon Machine Interface

This chapter defines an interface between a Monsoon computer equivalent (the "monsoon side") and a program which controls a Monsoon computer equivalent (the "user side"). Examples of a Monsoon computer equivalent are the Monsoon hardware prototype and the Monsoon software emulator. Examples of a program which controls Monsoon are a loader, a debugger, Id World, *etc.* The interface is called the Monsoon Machine Interface, or MMI.

The primary role of this interface is to provide access to and control of Monsoon's machine state. The view that the MMI takes of machine state is that the entire machine is partitioned into units called PE's, indexed by an integer. Each PE has a similar set of memory spaces; the memory spaces on all PE's comprise the Monsoon machine state.

It is important to understand that as far as the MMI is concerned, a "PE" is just a set of memory spaces to which access is provided. A given MMI PE may actually be a Processing Element board, but could also be an I-structure board, or an I/O processor, or a software emulation of a processor, or even a file on Monsoon's host processor. A key role of the MMI, therefore, is to provide a uniform presentation of these "PE's" to programs on the user side. The central design principle of the MMI is to achieve a balance: it must be general enough to accomodate the diversity of what is a PE, while not being so general as to be vacuous.

It is also important to understand that the MMI views Monsoon at the macro-architectural level, the level of the instruction set architecture. The only machine state accessible through the MMI is machine state that is part of the instruction set architecture—the same set of state that is accessible by a program executing on Monsoon itself. Things like microcode memory and other hardware registers are not part of the instruction set architecture, are not accessible to a running Monsoon program, and therefore are not accessible through the MMI.

The memory spaces accessed via the MMI are the following:

Data Memory (DM) Each PE has a single data memory space, addressed by consecutive unsigned integers beginning with zero. Each word has three fields: presence, type, and data.

Instruction Memory (IM) Each PE has a single instruction memory space, addressed by consecutive unsigned integers beginning with zero. Each word only has one field (which usually contains an instruction).

Token Queues Each PE has several token queues, indexed by consecutive unsigned integers beginning with zero. Each queue consists of a memory and a set of pointers. The memory is addressed by consecutive unsigned integers beginning with zero, and each word has four fields: tag-type, tag-data, value-type, and value-data. The type and data fields are

of the same size and hold the same sorts of values as do the type and data fields of data memory. The queue pointers are a group of words, indexed by consecutive unsigned integers beginning with zero, which hold pointers to queue memory and other per-queue state.

Statistics Registers Each PE has several statistics registers, indexed by consecutive unsigned integers beginning with zero. Each register holds a non-negative integer, typically used to count various events during execution.

Again, the MMI is general enough to accomodate any sort of PE; any given PE may only provide a subset of the spaces above. Some examples:

- A Processing Element board has a DM, IM, two token queues, and 16 statistics registers.
- An I-structure board has a DM and a token queue (?).
- A software emulation of a processing element may have a large number of queues depending on the mode of operation (finite latency, *etc.*).

The interface is designed to accomodate any sort of Monsoon computer equivalent, whether hardware or software, single-pe or multi-pe. It tries to minimize the impact of changing such architectural specifications as the number of bits in a word of memory, the size of fields of tags and instructions, floating point representation, and the like. But because all of these characteristics are observable by a running Monsoon program, *it is not possible to abstract away from them* (see also the discussion in Section 2.2). All of these characteristics should therefore be considered a part of the specification of this interface, incorporated by reference. All implementations of both sides of the interface must agree on a single set of these characteristics, and all must be changed simultaneously. The interface tries to minimize the impact of such changes by, for example, including facilities for decoding tags into fields, so that many user programs can be oblivious to the size of those fields.

While characteristics of the *architecture* are part of the interface, characteristics of the *configuration* are not. The distinction is that configuration characteristics could be different each time the Monsoon system is initialized, or different between different installations of Monsoon. Architectural characteristics stay fixed until someone changes the hardware or software specifications. For example, the following are characteristics of the architecture: the kinds of memory spaces that are found on a PE, the width of the presence, type, and data fields of data memory, the width of instruction memory, the ways the hardware parses data and instructions, *etc.* These characteristics are built into the MMI, either in the form of compile-time constants or by the functions the MMI provides. In contrast, the following are characteristics of the configuration: the number of PE's, the kind of PE (processor, I-structure, *etc.*) represented by a given PE number, the size of data memory and instruction memory on a given PE, *etc.* These characteristics are not built into the MMI, but are recorded in a data structure built at system initialization time called the *Proto-Memory Manager* (see Chapter 3), which is kept at a conventional place within Monsoon's memories.

2.1 Hardware Data Types

Each word of Monsoon data memory has three logical fields: presence bits, type bits, and data bits (for Version 2 of the hardware the sizes of these fields are 3 bits, 8 bits, and 64 bits,

respectively). While the memory hardware of Monsoon imposes no interpretation on these bits, the ALU and other stages of the pipeline do. Specifically, presence bits of 00 have a special meaning for the (hypothetical) cache hardware, the instruction fetch unit decodes data and type bits representing instructions in a particular way, and the ALU interprets data bits variously as IEEE double-precision floating point numbers, tags, *etc.* Microcode imposes additional interpretation on the presence and type bits. These interpretations can be fully observable to a Monsoon program; for example, it is possible to write a Monsoon program which decodes the bit-level representation of a floating point number. Thus, every implementation of the Monsoon side of the interface must agree on the bit-level representation of every data type, type code, and presence code.

A similar situation exists for instruction memory: the hardware usually interprets its contents as instructions, but it is also possible for the ALU to read a word of instruction memory and use the contents as an integer.

Because bit-level representations are observable by Monsoon code, it is sufficient to provide functions which manipulate the three fields of data memory (presence, type, and data) as bit strings, or integers. Doing so does not violate any abstractions, for the observability property implies that it is *not possible* to abstract away from, for example, the representation of floating point numbers. Nevertheless, programs on the user side of the interface may not find the bit string representation most convenient in all situations. The MMI therefore defines several ways of representing the contents of both instruction memory and the data field of data memory.

There are four representations for the data field of data memory:

Bits An unsigned integer, of at most N bits, where N is the size of the data field (currently 64 bits).

Integer A two's complement signed integer, in the range $-2^{N-1} \leq x \leq 2^{N-1} - 1$, where N is the size of the data field (currently 64 bits).

Float A floating point number, with precision and magnitude range at least as great as that provided by the Monsoon hardware (currently the same as IEEE double precision: base 2, 11 bit excess-1024 exponent, 52 bit mantissa plus hidden bit).

Tag A compound object consisting of the five fields PORT, MAP, IP, PE, and FP, where each field is an unsigned integer whose maximum magnitude corresponds to the size of these fields as interpreted by the hardware (currently 1 bit, 7 bits, 24 bits, 10 bits, and 22 bits, respectively).

Similarly, there are two ways of representing the contents of instruction memory:

Bits An unsigned integer, of at most N bits, where N is the size of an instruction memory word (currently 32 bits).

Instruction A compound object consisting of the three fields OPCODE, F1, and F2, where each field is an unsigned integer whose maximum magnitude corresponds to the size of these fields as interpreted by the hardware (currently 12 bits, 10 bits, and 10 bits, respectively).

It is entirely the responsibility of the user side to choose an appropriate representation depending on the data that is actually present in the Monsoon memory. Because a user could, for example, store a word as a float then immediately read it back as a tag, there must be a

conversion defined between all pairs of representations that is consistent across implementations of the Monsoon side. This is consistent with the observability property.

Every implementation of the user side is required to provide the "bits" representation for both instruction memory and the data field of data memory, and may optionally provide any or all of the others. For the Common Lisp implementation, all representations are provided, with the following implementation as Common Lisp objects:

Data Memory Bits A non-negative integer, potentially a bignum.

Integer An integer, potentially a bignum.

Float A double precision floating point number. (This really is specific to the Symbolics and TI implementations of Common Lisp, where double precision is known to be as precise and magnitudinous as Monsoon. We should double check Lucid and Allegro. Other implementations may need something different.)

Tag A **struct** of five slots, each a non-negative integer.

Instruction Memory Bits A non-negative integer, potentially a bignum.

Instruction A **struct** of three slots, each a non-negative integer.

Rationale: There was some discussion of whether we should use a tuple of integers for "data memory bits" and "integer" to avoid the possibility of bignums. It seems entirely possible that the effort expended by a user program to manipulate such a tuple could exceed the benefit of avoiding bignums. Therefore, we should use potential bignums for now, and change it later if we can show that there is a superior alternative. Using a single integer is certainly going to be simpler in terms of software complexity on the user side.

For the C implementation of the MMI, all representations are provided, with the following implementation as C objects. The C implementation of the MMI requires the **scalar** and **int64** libraries of the MCRC C Language Support package. The terms "integer type" and "floating point type" are defined in the documentation of the **scalar** library, while the term "**INT64** type" is defined in the documentation of the **int64** library.

Data Memory Bits An unsigned **INT64** type.

Integer A signed **INT64** type.

Float An opaque floating point type.

Tag A **struct** of five slots, each an opaque integer type (see Section 2.6 for more details).

Instruction Memory Bits An opaque, unsigned integer type.

Instruction A **struct** of three slots, each an opaque integer type (see Section 2.6 for more details).

Sections 2.1.1 through 2.4 define the Common Lisp implementation of the MMI interface. The C implementation is summarized in Section 2.6; nearly all of the C functions are direct counterparts of the Common Lisp functions.

Compatibility Note: The above description of data formats predates the macro-architecture specification (*The Monsoon Macro-Architecture Reference Manual*). The Macro-Architecture defines six formats instead of four: it does not have the Bits format, it has a single-precision float format SFloat (the "Float" format is called Dfloat), and instead of Tag it has Pointer, Continuation, and Request. The MMI provides Bits so that an MMI User can get at data bits without any interpretation. That SFloat is missing is indeed an MMI deficiency, which will be corrected in the future if single-precision floating point numbers ever become important. It is unfortunate that the MMI views Pointers, Continuations, and Requests all as instances of Tags. The correspondence is given in the *Macro-Architecture Reference Manual*, and it is recommended that MMI users that make heavy use of these types define their own abstractions on top of the Tag type. A future version of the MMI will undoubtedly have Pointers, Continuations, and Requests instead of Tags.

All of this is rather unfortunate, but it reflects the ever-growing understanding of what is Monsoon!

2.1.1 Types and Sizes of Fields

The following constants are provided for code which depends on the current size of various machine-interpreted data. Also defined are type names for various quantities; many of these are specialized versions of type **integer**, which may be useful in generating efficient code.

`mmi:dm-bits` [*Type*]
`mmi:dm-bits-limit` [*Constant*]

The type `mmi:dm-bits` is the return type of functions that return the "bits" representation of Monsoon data memory contents, as well as the type of arguments which receive that representation. The value of `mmi:dm-bits-limit` is a non-negative integer that is the upper exclusive bound on values of type `mmi:dm-bits`; that is, a value of that type is non-negative and strictly less than the value of `mmi:data-bits-limit`.

`mmi:dm-integer` [*Type*]
`mmi:most-positive-dm-integer` [*Constant*]
`mmi:most-negative-dm-integer` [*Constant*]

The type `mmi:dm-bits` is the return type of functions that return the "integer" representation of Monsoon data memory contents, as well as the type of arguments which receive that representation. Values of that type are greater than or equal to `mmi:most-negative-dm-integer` and less than or equal to `mmi:most-positive-dm-integer`.

`mmi:dm-float` [*Type*]
`mmi:most-positive-dm-float` [*Constant*]
`mmi:most-negative-dm-float` [*Constant*]
`mmi:dm-float-epsilon` [*Constant*]
`mmi:dm-float-negative-epsilon` [*Constant*]

The type `mmi:dm-float` is the return type of functions that return the "float" representation of Monsoon data memory contents, as well as the type of arguments which receive that representation. The constants are analogous to the Common Lisp constants `most-positive-single-float`, `most-negative-single-float`, `single-float-epsilon`, and `single-float-negative-epsilon`, but describe instead the characteristics of Monsoon data memory floating point numbers, and

therefore of the type `mmi:dm-float`. See *Common Lisp—The Language*, page 232, for more details.

`mmi:tag` [*Type*]
`mmi:port` [*Type*]
`mmi:map` [*Type*]
`mmi:ip` [*Type*]
`mmi:pe` [*Type*]
`mmi:fp` [*Type*]
`mmi:port-limit` [*Constant*]
`mmi:map-limit` [*Constant*]
`mmi:ip-limit` [*Constant*]
`mmi:pe-limit` [*Constant*]
`mmi:fp-limit` [*Constant*]

The type `mmi:tag` is the return type of functions that return the "tag" representation of Monsoon data memory contents, as well as the type of arguments which receive that representation. The types `mmi:port`, `mmi:map`, `mmi:ip`, `mmi:pe`, and `mmi:fp` are the types of the values found in the slots of a `mmi:tag`. Each constant is the upper exclusive bound on one of those types.

`mmi:type` [*Type*]
`mmi:type-limit` [*Constant*]

The type `mmi:type` is the return type of functions that return the type field of Monsoon data memory, as well as the type of arguments which receive values for that field. The value of `mmi:type-limit` is the upper exclusive bound on the the type `mmi:type`.

`mmi:presence` [*Type*]
`mmi:presence-limit` [*Constant*]

This type and constant are analogous to `mmi:type` and `mmi:type-limit`.

`mmi:im-bits` [*Type*]
`mmi:im-bits-limit` [*Constant*]

The type `mmi:im-bits` is the return type of functions that return the "bits" representation of Monsoon instruction memory contents, as well as the type of arguments which receive that representation. The constant `mmi:im-bits-limit` is the upper exclusive bound on the type `mmi:im-bits`.

`mmi:instruction` [*Type*]
`mmi:opcode` [*Type*]
`mmi:f1` [*Type*]
`mmi:f2` [*Type*]
`mmi:opcode-limit` [*Constant*]
`mmi:f1-limit` [*Constant*]
`mmi:f2-limit` [*Constant*]

The type `mmi:instruction` is the return type of functions that return the "instruction" representation of Monsoon instruction memory contents, as well as the type of arguments which receive that representation. The types `mmi:opcode`, `mmi:f1`, and `mmi:f2` are the types of the values found in the slots of a `mmi:instruction`. Each constant is the upper exclusive bound on one of those types.

There are alternative views of the fields of instructions, used when interpreting instructions in a particular instruction format. The relevant types and limit constants are defined in Section 2.1.3.

Discussion: The descriptions of these constants are worded as if they are characteristics of a particular implementation of the Monsoon Machine Interface. In fact, the values of these constants are determined by the current Monsoon *architecture*. As the characteristics of an implementation of the interface are constrained by the architecture, the constants will also describe the interface implementation as well (assuming the interface is implemented according to the specification!).

Compatability Note: The wire-wrap prototype of Monsoon has a slightly different instruction format than Version 2. The wire-wrap can be cast in terms of this interface, however, by calling the R field F1, and the combination of the PORT and S fields F2. The sizes of OPCODE, F1, and F2 will be different for the wire-wrap prototype: 10 bits, 10 bits, and 12 bits, respectively, as compared to 12, 10, and 10 for Version 2.

There are no constants describing the limits of token queue memory, because these characteristics are determined by data memory characteristics. Specifically, each entry in a token queue consists of two type/data pairs, each having the same bounds and data types as the type and data fields of data memory. Statistics registers are discussed in Section 2.4.

2.1.2 Constructors and Selectors

This section outlines functions which manipulate the compound objects representing the Monsoon hardware data types "tag" and "instruction."

`mmi:tag` [*Type*]

The type of the object used to pass tags across the MMI.

`mmi:make-tag` *port map ip pe fp* [*Function*]

Returns a tag with the given fields.

`mmi:tag-port` *tag* [*Function*]
`mmi:tag-map` *tag* [*Function*]
`mmi:tag-ip` *tag* [*Function*]
`mmi:tag-pe` *tag* [*Function*]
`mmi:tag-fp` *tag* [*Function*]

Returns the appropriate component of *tag*. Note that these *cannot* be used with **setf**; tag objects are *immutable*.

Rationale: There are just too many potential problems if there can be multiple pointers to a given tag object and someone bashes one of the slots. Better to do it this way and worry later if there is some issue of recycling Lisp storage. Remember that a software emulator of Monsoon is not constrained to use tag objects to represent the tag field of tokens in its emulated token queues, so long as it does the appropriate conversions in its implementation of the functions in Section 2.3.

`mmi:make-new-tag` *old-tag* `&key port map ip pe fp` [*Macro*]

Like calling `mmi:make-tag` where *old-tag* is used to supply the values of any field for which no keyword argument is given. Note that if no keyword arguments are given it just copies *old-tag*.

This is a macro rather than a function so that the overhead of using keyword arguments may be eliminated at compile-time.

`mmi:tagp` *thing* [*Function*]
Returns true if *thing* is of type `mmi:tag`, otherwise returns false.

`mmi:instruction` [*Type*]
The type of the object used to pass instructions across the MMI.

`mmi:make-instruction` *opcode f1 f2* [*Function*]
`mmi:instruction-opcode` *instruction* [*Function*]
`mmi:instruction-f1` *instruction* [*Function*]
`mmi:instruction-f2` *instruction* [*Function*]
`mmi:make-new-instruction` *old-instruction* `&key opcode f1 f2` [*Macro*]
`mmi:instructionp` *thing* [*Function*]
Entirely analogous to the functions for tags. The same comments about immutability apply.

2.1.3 Parsing Instruction Fields

The F1 and F2 fields of both kinds of instructions are often interpreted as a combination of an unsigned integer called PORT and a signed integer called S.[1] The following constants define the sizes of these quantities. Note that F1 and F2 may be of different sizes (although for the current architecture, they are the same size). On the other hand, the PORT subfields of F1 and F2 always have the same size as the PORT field of a tag.

`mmi:s-1` [*Type*]
`mmi:s-2` [*Type*]
`mmi:most-positive-s-1` [*Constant*]
`mmi:most-negative-s-1` [*Constant*]
`mmi:most-positive-s-2` [*Constant*]
`mmi:most-negative-s-2` [*Constant*]
The types `mmi:s-1` and `mmi:s-2` are the return types of functions returning the S subfields of an instruction's F1 and F2, respectively, as well as the types of arguments to functions accepting those subfields. The values of `mmi:most-positive-s-1` and `mmi:most-negative-s-1` are inclusive bounds on the type `mmi:s-1`; the other two constants are analogous, but apply to `mmi:s-2`.

There are no special types or constants defined for the PORT subfields of F1 and F2, as they are required to be the same as the types and constants defined for the PORT field of tags.

`mmi:f1-port` *f1* [*Function*]
`mmi:f1-s` *f1* [*Function*]
`mmi:make-f1` *port s* [*Function*]
`mmi:f2-port` *f2* [*Function*]
`mmi:f2-s` *f2* [*Function*]

[1] In other Monsoon documentation, PORT is simply called P. The term PORT is used consistently throughout the software interface because P might erroneously suggest a predicate.

`mmi:make-f2` *port s* [*Function*]

The functions `mmi:f1-port` and `mmi:f1-s` extract the PORT and S quantities, respectively, from a value taken from the F1 field of an "instruction" `struct`. The function `mmi:make-f1` creates a value suitable for the F1 field of an "instruction" `struct` from the quantities PORT and S. The remaining three functions are analogous, but apply to the F2 field.

The F1 and F2 fields can also be interpreted together as a large R value. The following constants and functions deal with this.

`mmi:long-r` [*Type*]
`mmi:long-r-limit` [*Constant*]

The type MMI:LONG-R is the return type of functions returning a long R value from an instruction, as well as the types of arguments to functions accepting such a value. The value of `mmi:long-r-limit` is the upper exclusive bound on the type `mmi:long-r`.

`mmi:f1-f2-long-r` *f1 f2* [*Function*]
`mmi:long-r-f1` *r* [*Function*]
`mmi:long-r-f2` *r* [*Function*]

The function `mmi:f1-f2-long-r` converts the values taken from the F1 and F2 fields of an "instruction" `struct` into a long R value. The functions `mmi:long-r-f1` and `mmi:long-r-f2` take a long R value and extract values suitable for the F1 and F2 fields, respectively, of an "instruction" `struct`.

`mmi:instruction-port-1` *instruction* [*Function*]
`mmi:instruction-s-1` *instruction* [*Function*]
`mmi:instruction-port-2` *instruction* [*Function*]
`mmi:instruction-s-2` *instruction* [*Function*]
`mmi:instruction-long-r` *instruction* [*Function*]

These functions are provided for convenience and are equivalent to combinations of instruction selectors and field destructurers, for example:

```
(mmi:instruction-port-1 i) ≡ (mmi:f1-port (mmi:instruction-f1 i))
(mmi:instruction-long-r i) ≡ (mmi:f1-f2-long-r (mmi:instruction-f1 i)
                                               (mmi:instruction-f2 i))
```

`mmi:make-instruction-long-r` *opcode r* [*Function*]
`mmi:make-new-instruction-long-r` *old-instruction* `&key opcode r` [*Function*]

These functions are also provided for convenience and are equivalent to combinations of instruction constructors and field constructors, for example:

```
(mmi:make-instruction-long-r op r) ≡ (mmi:make-instruction op
                                                           (mmi:long-r-f1 r)
                                                           (mmi:long-r-f2 r))
```

Finally, the F1 and F2 fields of instructions are sometimes interpreted as the single unsigned integers R and REG. The following synonyms are provided for those situations:

`mmi:r` [*Type*]

`mmi:reg` [*Type*]
`mmi:r-limit` [*Constant*]
`mmi:reg-limit` [*Constant*]
`mmi:instruction-r` *instruction* [*Function*]
`mmi:instruction-reg` *instruction* [*Function*]

```
mmi:r                    ≡ mmi:f1
mmi:reg                  ≡ mmi:f2
mmi:r-limit              ≡ mmi:f1-limit
mmi:reg-limit            ≡ mmi:f2-limit
(mmi:instruction-r i)    ≡ (mmi:instruction-f1 i)
(mmi:instruction-reg i)  ≡ (mmi:instruction-f2 i)
```

2.1.4 Conversion Functions

These functions convert between different representations of machine data types.

`mmi:bits-to-integer` *bits* [*Function*]
`mmi:bits-to-float` *bits* [*Function*]
`mmi:bits-to-tag` *bits* [*Function*]
`mmi:integer-to-bits` *integer* [*Function*]
`mmi:integer-to-float` *integer* [*Function*]
`mmi:integer-to-tag` *integer* [*Function*]
`mmi:float-to-bits` *float* [*Function*]
`mmi:float-to-integer` *float* [*Function*]
`mmi:float-to-tag` *float* [*Function*]
`mmi:tag-to-bits` *tag* [*Function*]
`mmi:tag-to-integer` *tag* [*Function*]
`mmi:tag-to-float` *tag* [*Function*]

Each function converts from one representation of the data field of data memory into another.

`mmi:bits-to-instruction` [*Function*]
`mmi:instruction-to-bits` [*Function*]

Each function converts from one representation of an instruction memory word into another.

2.2 Reading and Writing Memory

Here we provide functions for reading and writing the memory of the Monsoon processor; namely, "data memory" and "instruction memory."

The functions that read and write data memory take an *address* argument; the type of this argument is always `mmi:fp`. The limit `mmi:fp-limit` is also a limit on this argument. Similarly, the type of the *address* argument to the functions that read and write instruction memory is always `mmi:ip`, and `mmi:ip-limit` applies to that argument. This reflects the architectural constraint that the sizes of data and instruction memory are ultimately limited by the sizes of the FP and IP fields of a tag. Note that this is an *architectural* limit, the maximum address permitted by the design of the hardware and software. It does *not* reflect how much data memory any particular PE has; that information is recorded in the Proto-Memory Manager data structure when the system is initialized (see Chapter 3). The user must consult the PMM to decide if an address argument actually points to existing memory.

2.2.1 Reading and Writing Single Words

The following functions read the data field of data memory:

`mmi:read-dm-data-as-bits` *pe address* [*Function*]
`mmi:read-dm-data-as-integer` *pe address* [*Function*]
`mmi:read-dm-data-as-float` *pe address* [*Function*]
`mmi:read-dm-data-as-tag` *pe address* [*Function*]

Each returns the contents of the data field at location *address* in the data memory of PE *pe*, in the indicated representation.

The type field of data memory is read by:

`mmi:read-dm-type` *pe address* [*Function*]

Returns the contents of the type field at location *address* in the data memory of PE *pe*, as a non-negative integer.

And the presence bits of data memory is read by:

`mmi:read-dm-presence` *pe address* [*Function*]

Returns the contents of the presence bits field at location *address* in the data memory of PE *pe*, as a non-negative integer.

`mmi:write-dm-data-as-bits` *pe address integer* [*Function*]
`mmi:write-dm-data-as-integer` *pe address integer* [*Function*]
`mmi:write-dm-data-as-float` *pe address float* [*Function*]
`mmi:write-dm-data-as-tag` *pe address tag* [*Function*]
`mmi:write-dm-type` *pe address integer* [*Function*]
`mmi:write-dm-presence` *pe address integer* [*Function*]

Each writes the given field contents in the appropriate field of location *address* in the data memory of PE *pe*. Note that using, for example, `mmi:write-dm-data-as-float` does not imply anything about how the Monsoon side of the interface actually stores the data; the hardware obviously will convert the float argument into bits, while the software emulator may choose to leave it as a float or to convert it into anything it wishes. The value returned from each of these functions is `eq` to the last argument, and each is also available by using `setf` with the corresponding `mmi:read-dm-` function.

`mmi:read-im-data-as-bits` *pe address* [*Function*]
`mmi:read-im-data-as-instruction` *pe address* [*Function*]
`mmi:write-im-data-as-bits` *pe address integer* [*Function*]
`mmi:write-im-data-as-instruction` *pe address instruction* [*Function*]

These are analgous to the `mmi:read-dm-` and `mmi:write-dm-` functions, except they operate on instruction memory rather than data memory. Note that instruction memory lacks type and presence fields (the word "data" is included in the names of these functions for uniformity). Notice, too, that the integers used with `mmi:read-im-data-as-bits` and `mmi:write-im-data-as-bits` are bounded by `mmi:im-bits-limit`, which may or may not be the same as `mmi:dm-bits-limit`.

Rationale: Instead of having both data memory and instruction memory functions, one could have a single set of functions together with a convention that interprets certain

addresses as data memory and others as instruction memory. The current feeling is that Monsoon will continue to have separate instruction and data memory, and furthermore they may have different characteristics (*e.g.*, instruction memory may lack presence bits), so two sets of functions seems the best choice.

[An issue: do we want "broadcast" versions of the `mmi:write-dm-` and `mmi:write-im-` functions that write the same thing in corresponding addresses on several PE's simultaneously?]

`mmi:read-dm-dtp-as-bits` *pe address* [*Function*]
`mmi:read-dm-dtp-as-integer` *pe address* [*Function*]
`mmi:read-dm-dtp-as-float` *pe address* [*Function*]
`mmi:read-dm-dtp-as-tag` *pe address* [*Function*]

Each returns three values: the data, type, and presence fields of location *address* on PE *pe*. The first value is returned in the indicated representation. `mmi:read-dm-dtp-as-integer`, for example, is equivalent to:

```
(values (mmi:read-dm-data-as-integer pe address)
        (mmi:read-dm-type pe address)
        (mmi:read-dm-presence pe address))
```

`mmi:write-dm-dtp-as-bits` *pe address data type presence* [*Function*]
`mmi:write-dm-dtp-as-integer` *pe address data type presence* [*Function*]
`mmi:write-dm-dtp-as-float` *pe address data type presence* [*Function*]
`mmi:write-dm-dtp-as-tag` *pe address data type presence* [*Function*]

Each writes *data*, *type*, and *presence* in the data, type, and presence fields of location *address* on PE *pe*, using the indicated representation to interpret the *data* argument. Each returns zero values. `mmi:write-dm-dtp-as-integer`, for example, is equivalent to:

```
(progn (mmi:write-dm-data-as-integer pe address data)
       (mmi:write-dm-type pe address type)
       (mmi:write-dm-presence pe address presence)
       (values))
```

There are no analogous functions for instruction memory, as instruction memory has no type or presence bits.

2.2.2 Block Transfers

`mmi:blt-from-dm-data-as-bits` *pe start-address end-address to-array* `&key` [*Function*]
`:to-start :to-end`
`mmi:blt-from-dm-data-as-integer` *pe start-address end-address to-array* `&key` [*Function*]
`:to-start :to-end`
`mmi:blt-from-dm-data-as-float` *pe start-address end-address to-array* `&key` [*Function*]
`:to-start :to-end`
`mmi:blt-from-dm-data-as-tag` *pe start-address end-address to-array* `&key` [*Function*]
`:to-start :to-end`
`mmi:blt-from-dm-type` *pe start-address end-address to-array* `&key :to-start` [*Function*]
`:to-end`

`mmi:blt-from-dm-presence` *pe start-address end-address to-array* `&key` `:to-start :to-end` [*Function*]

These are block transfer functions: data (or type or presence) fields from consecutive words of PE *pe*'s memory, from *start-address* inclusive to *end-address* exclusive, are written into consecutive locations of *to-array*, at indices *to-start* inclusive to *to-end* exclusive. The defaults for *to-start* and *to-end* are zero and the length of *to-array*, respectively. If the ranges do not delimit sequences of the same length, the smaller range determines how many elements are transfered; the extra elements at the end of the longer range are not involved in the operation. The number of elements copied may therefore be expressed as:

`(min (-` *end-address start-address*`) (-` *to-start to-end*`))`

The type of elements transfered to *to-array* corresponds to the type that would be returned by the corresponding `mmi:read-dm-` function. Each of these functions returns *to-array*. [An issue here: do we want to require that the arrays for the presence and type functions be of type `(unsigned-byte 2)` and `(unsigned-byte 8)`? This may be a non-issue for the Common Lisp implementation since Common Lisp will do the right thing automatically!]

`mmi:blt-to-dm-data-as-bits` *from-array from-start from-end pe to-start* `&key` `to-end` [*Function*]
`mmi:blt-to-dm-data-as-integer` *from-array from-start from-end pe to-start* `&key to-end` [*Function*]
`mmi:blt-to-dm-data-as-float` *from-array from-start from-end pe to-start* `&key` `to-end` [*Function*]
`mmi:blt-to-dm-data-as-tag` *from-array from-start from-end pe to-start* `&key` `to-end` [*Function*]
`mmi:blt-to-dm-type` *from-array from-start from-end pe to-start* `&key to-end` [*Function*]
`mmi:blt-to-dm-presence` *from-array from-start from-end pe to-start* `&key` `to-end` [*Function*]

These functions are like the `mmi:blt-to-dm-` functions, but transfer in the opposite direction: from the given array to data memory. The default for *to-end* is the size of Monsoon's data memory (currently ??). Each of these functions returns *from-array*.

`mmi:blt-from-im-as-bits` *pe start-address end-address to-array* `&key :to-start` `:to-end` [*Function*]
`mmi:blt-from-im-as-instruction` *pe start-address end-address to-array* `&key` `:to-start :to-end` [*Function*]
`mmi:blt-to-im-as-bits` *from-array from-start from-end pe to-start* `&key to-end` [*Function*]
`mmi:blt-to-im-as-instruction` *from-array from-start from-end pe to-start* `&key` `to-end` [*Function*]

Like the `mmi:blt-from-dm-` and `mmi:blt-to-dm-` functions, but deal with instruction memory instead of data memory.

[More experience is needed with the interface before we'll know whether these are adequate block transfer functions. For example, we may want version which deal with subdomain interleaving.]

2.3 Token Queues

Tokens are represented as four values corresponding to the type and data subfields of each of the tag and value parts of a token. The representation of type subfields is the same as with `mmi:read-dm-type` above, and the representation of the tag data subfield is the same as with `mmi:read-dm-data-as-tag`. Any of the four representations may be used for the value data subfield.

The software interface provides for a more general set of queues than the hardware actually provides. One of the ways the software emulator will be used is to gather idealized profiles, finite-processor/latency profiles, Maa-style per-codeblock profiles, *etc.* These are implemented by using a more elaborate queueing system which may comprise many queues. The software interface, therefore, provides functions for reading and writing queues selected by a *queue number*. The number of queues provided for a given mode of operation depends on which queueing system is in use at that time.

Note that in this context, the term "queue" is used rather loosely to denote any sort of data structure the queueing system might use to hold tokens. For example, a software queueing system for obtaining hardware timings will have an 8-token pipeline for simulating pipeline latency; this could be made available through the software interface by considering the pipeline as a small "queue." While each queue might be a data structure of arbitrary objects, when viewed through the MMI it must appear as a contiguously-addressed array of tokens. No queueing system is required to provide access to all of its data structures through this interface, though it is highly recommended to do so.

There is a convention which assigns queue numbers to the actual queues found on the hardware; both the hardware driver and the software emulator for the hardware must agree on this convention. The convention is as follows.

Queue Number 0 The system queue.

Queue Number 1 The user queue.

Queue Number 2 (Only present in some emulators, and not in hardware.) An eight element queue used to emulate pipeline latency.

Queue Number 3 (Only present in some emulators, and not in hardware.) A one element queue used to hold a token for recirculation.

The first two of these queues each have three pointers, with meaning described later.

The selection of a queueing system also determines how the software interface controls the execution of the machine (hardware or software). The overall model is that a control program first selects a queueing system and queueing mode for that system, and then makes one or more calls to `mmi:run`. The procedure `mmi:run` causes machine execution to proceed up to some stopping point, where the stopping point depends in part on the queueing system and mode chosen. At this point the control program might examine statistics registers or other machine state, and then make further calls to `mmi:run`. Alternatively, it could then reset the machine, select a new queueing mode, and begin the process from the beginning. For the software emulator `mmi:run` simply transfers control to the emulator program, while for the hardware `mmi:run` starts the machine through a hardware interface and waits for the hardware to reach a stopping point. This actually a simplification of the behavior of `mmi:run`; Section 2.5 takes up `mmi:run` and other functions for machine control in greater detail.

Argtype Value	*Argument's Lisp Type*	*Argtype Value*	*Argument's Lisp Type*
mmi:dm-bits	mmi:dm-bits	mmi:s-1	mmi:s-1
mmi:dm-integer	mmi:dm-integer	mmi:s-2	mmi:s-2
mmi:dm-float	mmi:dm-float	mmi:long-r	mmi:long-r
mmi:tag	mmi:tag	mmi:r	mmi:r
mmi:port	mmi:port	mmi:reg	mmi:reg
mmi:map	mmi:map	mmi:qn	mmi:qn
mmi:ip	mmi:ip	mmi:qp	mmi:qp
mmi:pe	mmi:pe	mmi:qpn	mmi:qpn
mmi:fp	mmi:fp	mmi:statistic-n	mmi:statistic-n
mmi:type	mmi:type	mmi:statistic-value	mmi:statistic-value
mmi:presence	mmi:presence	:uint32	(integer 0 (2^{32}))
mmi:im-bits	mmi:im-bits	:sint32	(integer -2^{31} (2^{31}))
mmi:instruction	mmi:instruction	:dflonum	[float in IEEE dp range]
mmi:opcode	mmi:opcode	:character	character
mmi:f1	mmi:f1	:string	string
mmi:f2	mmi:f2	:keyword	keyword

Figure 2.1: Queueing Mode Argument Types (Lisp Version)

The Monsoon side of the MMI may provide a number of different queueing systems, and each such queueing system may provide a number of different operating modes. An example of a queueing system is the "Monsoon hardware" system, available when the Monsoon side of the MMI is actual Monsoon hardware, or when it is an emulator designed to mimic the hardware precisely. The queueing modes available in the Monsoon hardware queueing system would include "grand total" mode, where `mmi:run` executes a program to completion, "single step" mode, where each call to `mmi:run` executes one instruction, "*n* step" mode, where each call to `mmi:run` executes *n* instructions, "breakpoint token" mode, where each call to `mmi:run` executes instructions until a token matching a given breakpoint value enters the pipeline, and "approximate idealized" mode, where each call to `mmi:run` executes tokens in one queue, with resulting tokens accumulated in another queue. A software emulator will undoubtedly provide many more queueing systems, for example, an "idealized" queueing system, with modes for finite processor and finite latency simulations. The modes described in this paragraph are for illustration only; for more information, consult the documentation of the individual queueing systems. For a general discussion of queueing systems and statistics collection, see the MINT White Paper.[2]

The functions below specify queueing systems and queueing modes by their names, represented as keywords. In addition, functions referring to queueing modes take a set of arbitrary parameters, whose interpretation is completely up to the specified queueing system. For example, the "*n* step mode" of the "Monsoon hardware" queueing system described above would have a parameter to indicate *n*, the number of instructions to execute. In the functions below, *mode-args* is a list of parameters that is queueing mode dependent. This may not be an arbitrary list: its elements must each be a member of one of the types given in the "Lisp Type" columns of Figure 2.1. Accompanying *mode-args* is a list *mode-arg-types*, which is the same length as *mode-args*, and each of whose elements gives the type of the corresponding element

[2]K. R. Traub, *MINT White Paper*, Motorola Cambridge Research Center Technical Report MCRC-TR-2, October, 1989.

of *mode-args*. The legal values for the elements of *mode-arg-types* are given in the "Argtype Value" columns of Figure 2.1.

As an example, the following is a legal call to `mmi:select-queueing-mode`:

```
(mmi:select-queueing-mode :example '(mmi:ip :string) '(34 "hi"))
```

while the following call is not:

```
(mmi:select-queueing-mode :example '(mmi:presence list) '(34 ("hi" "there")))
```

This call is illegal because 34 is not of type `mmi:presence` (it is too large), and because `list` is not a type listed in Figure 2.1.

The documentation of a queueing system must indicate the types and meanings of *mode-args* for each available mode. The preferred programming methodology is for the author of each queueing system to provide some higher-level functions that sit on the MMI, hiding the encoding of parameters into the *mode-arg-types* and *mode-args* lists. For example, the "Monsoon hardware" system might come with this function:

```
(defun select-n-step-mode (n)
  (mmi:select-queueing-mode :n-step '(:uint32) (list n)))
```

Rationale: The MMI is designed so that the User and Monsoon sides may be executing on different machines, or in different programming languages, possibly with a network protocol between them. This means that a given type may have a different representation within the language environment of the User side as compared with the Monsoon side. Inclusion of type information allows translation to take place, if necessary.

The set of types in Figure 2.1 includes all MMI-defined types, as well as some common scalar types. More complex types can be passed by flattening them into the types provided.

`mmi:select-queueing-system` *name mode mode-arg-types mode-args* [*Function*]

Selects a queueing system named *name* (a keyword); subsequent interaction with the queues or the `mmi:run` procedure will use the selected queueing system. The queueing system is reset to a well-defined initial state (determined by the queueing system selected), and the queueing system is placed into a mode indicated by *mode* (a keyword) and *mode-args* (the interpretation of these is also determined by the queueing system selected). The restrictions on *mode-args* and the encoding of *mode-arg-types* is as described earlier. `mmi:select-queueing-system` returns `t` if *name* is a supported queueing system and if *mode* and *mode-args* are legal for that queueing system, `nil` otherwise. In the latter case, the queueing system and mode are unchanged, and the queueing system is not reset.

The behavior of all the remaining functions in this section is potentially altered after each call to `mmi:select-queueing-system`.

`mmi:select-queueing-mode` *mode* `&rest` *mode-args* [*Function*]

Changes the mode of operation of the current queueing system to be that indicated by *mode* and *mode-args* (the interpretation of these is determined by the current queueing system). The restrictions on *mode-args* and the encoding of *mode-arg-types* is as described earlier. Changing the mode does *not* necessarily reset the state of the queueing system; the definition of any queueing system should indicate how the state changes when the mode is changed. For example,

the hardware queueing system might have a queueing mode which causes **mmi:run** to execute a given number of pipeline steps; changing the mode (to change the number of steps, for example) should not reset the contents of the queues. This function is implicitly called each time **mmi:select-queueing-system** is called. **mmi:select-queueing-mode** returns **t** if *mode* and *mode-args* are legal for the current queueing system, **nil** otherwise. In the latter case, the queueing mode is unchanged, and the queueing system is not reset.

mmi:current-queueing-mode [*Function*]

Returns four values: the name of the current queueing system, the current queueing mode, and the *mode-arg-types* and *mode-args* lists for the current mode. Thus, the following code

```
(multiple-value-bind (name mode arg-types args)
    (mmi:current-queueing-mode)
  (mmi:select-queueing-system name mode arg-types args))
```

leaves the current queueing mode unchanged, except that the queueing system is reset.

mmi:reset-queueing-system [*Function*]

Resets the current queueing system to a well-defined initial state (determined by the current queueing system). This function is implicitly called each time **mmi:select-queueing-system** is called. The contents of the queues, as returned by the **mmi:read-queue-as-**x family of functions, are not necessarily preserved across calls to **mmi:reset-queueing-system**.

mmi:run [*Function*]

Causes execution of the machine according to the current queueing mode, until a stopping point determined in part by the current queueing mode is reached. A status value is returned that indicates, among other things, whether **mmi:run** returned because a queueing system defined timestep was completed, or for some other reason. See Section 2.5 for more details.

mmi:advance-timestep [*Function*]

After **mmi:run** returns a status indicating that a queueing system defined timestep was completed, this call may be issued to prepare the queueing system for the next timestep. Normally this is done after statistics registers are read. The exact behavior of **mmi:advance-timestep** depends on the current queueing system and mode; each queueing system should include as part of the documentation of each mode what **mmi:advance-timestep** does.

The following types and functions are used to read and write the state of the current queueing system.

mmi:qn [*Type*]
mmi:qp [*Type*]
mmi:qpn [*Type*]

The type **mmi:qn** is the type of queue numbers, the unsigned integers that identify which queue is to be manipulated. The queues of a given PE are always numbered consecutively from zero. An address of a token in a particular queue is called a queue pointer, of type **mmi:qp**. These addresses are used as arguments to functions that read and write tokens in queues, and always run consecutively from zero. In addition, each queue has associated with it a small number of queue pointers that keep track of where the head and tail are (for example). These

special pointers are indexed by a queue pointer number, of type **mmi:qpn**; these indexes also run consecutively from zero.

mmi:qn-limit *pe* [*Function*]
mmi:qp-limit *pe qn* [*Function*]
mmi:qpn-limit *pe qn* [*Function*]

These functions allow the size characteristics of the current queueing system to be determined. **Mmi:qn-limit** returns the number of queues on PE *pe*, **mmi:qp-limit** returns the exclusive upper bound on the address of queue *qn* on PE *pe*, and **mmi:qpn-limit** returns the number of special queue pointers associated with queue *qn* on PE *pe*. These are all functions, rather than constants, because they can vary from queueing system to queueing system, and indeed from PE to PE. Note that **mmi:qp-limit** does not indicate how many locations within the given queue actually contain valid tokens; this is determined by the queue pointers associated with that queue, in a manner that depends on how that queue is defined by the current queueing system.

mmi:read-queue-as-bits *pe queue address* [*Function*]
mmi:read-queue-as-integer *pe queue address* [*Function*]
mmi:read-queue-as-float *pe queue address* [*Function*]
mmi:read-queue-as-tag *pe queue address* [*Function*]

Returns the token at address *address* in the token queue whose index is *queue* on PE *pe*'s, as four values. In order, the four values returned are tag-type, tag-data, value-type, and value-data. Each of the four functions returns the value-data using a different representation.

mmi:write-queue-as-bits *pe queue address tag-type tag-data value-type value-data* [*Function*]
mmi:write-queue-as-integer *pe queue address tag-type tag-data value-type value-data* [*Function*]
mmi:write-queue-as-float *pe queue address tag-type tag-data value-type value-data* [*Function*]
mmi:write-queue-as-tag *pe queue address tag-type tag-data value-type value-data* [*Function*]

Writes a token into a token queue. Note that these cannot be accessible through **setf**, since four values are required. These functions return no values.

mmi:inject-token-as-bits *pe tag-type tag-data value-type value-data* [*Function*]
mmi:inject-token-as-integer *pe tag-type tag-data value-type value-data* [*Function*]
mmi:inject-token-as-float *pe tag-type tag-data value-type value-data* [*Function*]
mmi:inject-token-as-tag *pe tag-type tag-data value-type value-data* [*Function*]

Injects a token into the queueing system for the given PE. The meaning of injecting a token depends on the queueing system; the documentation of each queueing system should specify this. For most modes of operation of the Monsoon queueing system, injecting a token enqueues the token at the end of the system queue. A queueing system may choose to ignore the *pe* argument, if its notion of injecting is not parameterized by PE.

Rationale: Four values are preferable to a token structure of four slots because of the issue of how to represent the value-data subfield; would we need four types of structure? Also, there is the storage efficiency issue in a software emulator: a token queue as four arrays is preferable to a single array pointing to four-element structures.

For implementations of the interface in langauges which do not allow the return of multiple values, we'll need separate functions for each token subfield (a total of seven `mmi:read-queue-` functions).

The next two functions read and write queue pointers. The meaning of the pointer for a given queue depends on the type of queue; typically it indicates at what address the next token is to be enqueued or dequeued. A queue may have several pointers; the *pointer* argument to the functions below indicates which one is to be read or written. Common types of queues are a LIFO queue (stack), which has only one pointer, and a FIFO queue, which has two. More elaborate kinds of "queues" are accomodated by this interface.

While it is completely up to each queueing system to define the meaning of its queue pointers, the following conventions are recommended where appropriate:

Queue Pointer 0 (Count) Not actually a pointer, but instead gives the number of active elements in the queue.

Queue Pointer 1 (Tail) For both FIFO and LIFO queues, points at the next token to be dequeued; to dequeue, remove the token at Tail and then increment Tail and decrement Count. For LIFO queues, it points at one greater than the next empty slot where a token can be enqueued; to enqueue LIFO, decrement Tail and store the token there, then increment Count.

Queue Pointer 2 (Head) For FIFO queues, points at the next empty slot where a token can be enqueued; to enqueue FIFO, store the token at Head and then increment Head and Count.

Queue Pointer 3 (Mark) May be used by some queues as a placeholder, for example to indicate the last token to be processed as part of a timestep.

`mmi:read-queue-pointer` *pe queue* `&optional` *(pointer 0)* [*Function*]

The function `mmi:read-queue-pointer` returns the current pointer for queue number *queue* on PE *pe*.

`mmi:write-queue-pointer` *new-value pe queue* `&optional` *(pointer 0)* [*Function*]

Changes the value of the queue pointer *pointer* of queue number *queue* on PE *pe* to be *new-value*, and returns *new-value*. Also available by using `setf` with `mmi:read-queue-pointer` (the order of arguments is inconsistent with the other "write" functions of this interface because of the optional argument; the use of `setf` is strongly encouraged).

2.4 Statistics Registers

Monsoon is equipped with several *statistics registers*, each of which holds an unsigned integer. Certain instructions executed by Monsoon affect the contents of these registers, under microcode control. In addition, a software emulator for Monsoon may have a very large number of them, and alter their contents in arbitrary ways. Statistics registers are identified by a non-negative integer.

`mmi:statistic-limit` *pe* [*Function*]

Returns the number of statistics registers on PE *pe*; any non-negative integer strictly less than this value is a valid *statistic* argument to the functions described below.

mmi:statistic-value-limit *pe statistic* [*Function*]

Returns the exclusive upper bound on the value of statistic register *statistic* on PE *pe*, or `nil` if there is no upper bound. (The latter case may arise in a software emulator, for example, if the value can be a bignum.)

Rationale: These limits are allowed to vary from PE to PE because they are not essential charcteristics of the architecture; *i.e.*, the values of these limits cannot affect the behavior of an executing Monsoon program (other than a program specifically designed to examine the registers, such as a program analysis tool running on Monsoon itself). In particular, the limits might be different between PE's if the PE number were used to select between the hardware and a software emulator.

---

mmi:read-statistic *pe statistic* [*Function*]

Returns the current value of statistic register *statistic* on PE *pe*.

mmi:write-statistic *pe statistic new-value* [*Function*]

Stores *new-value* in statistic register *statistic* on PE *pe*, and returns *new-value*. Also available by using **setf** with **mmi:read-statistic**.

mmi:clear-statistics *pe* **&optional** *(from-statistic 0) (to-statistic (statistic-limit pe))* [*Function*]

Stores zero in each of the statistics registers between *from-statistic*, inclusive, to *to-statistic*, exclusive, on PE *pe*. Returns no values. Equivalent to the following:

```
(do ((s from-statistic (+ s 1)))
    ((>= s to-statistic) (values))
  (mmi:write-statistic pe s 0))
```

except that `mmi:clear-statistics` may be more efficient in certain cases (*e.g.*, if a hardware reset line is available).

2.5 Machine Control

The functions in this section provide control over the execution of Monsoon; they start and stop execution, and deal with asynchronous events such as interrupts. These functions view the entire Monsoon as a unit, and are not parameterized by PE.

The basic primitive for machine control is **mmi:run**, which initiates execution of Monsoon, returning when Monsoon halts. **mmi:run** returns a value that indicates why Monsoon halted; this could be due to its executing a halt instruction, or because the queueing system reached the end of a timestep, or due to an exceptional condition such as a statistic overflow or parity error, or because of an asynchronous request to halt from the User side of the MMI. In addition, certain events called *interrupts* can cause **mmi:run** to return with Monsoon continuing to execute in the background. The user side may then make MMI calls (from a restricted subset), and then call **mmi:run** to wait for the next interrupt or halt.

As far as machine control is concerned, at any point in time the MMI is in one of three states:

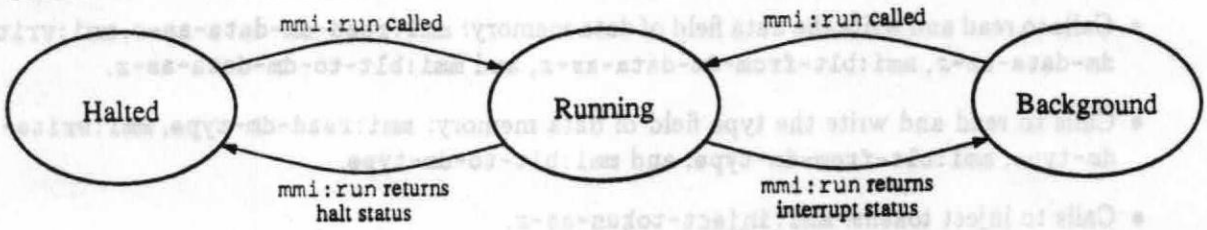

Figure 2.2: MMI State Transition Diagram

Running A call to `mmi:run` has been made, but `mmi:run` has not yet returned. When the MMI is in this state, Monsoon is generally executing code. This is not necessarily the case, however: just after `mmi:run` is called and just before it returns, it is possible that Monsoon is not actually executing.

Halted No call to `mmi:run` is in progress, and Monsoon is not executing code. This state is entered when `mmi:run` returns with a halt status value.

Background No call to `mmi:run` is in progress, but Monsoon is potentially executing code. This state is entered when `mmi:run` returns with an interrupt status value.

Figure 2.2 shows the relationship between these states. It should be emphasized that these are states of the MMI, not of Monsoon. Monsoon itself may be halted but the MMI is still in the running state, for example, just after the machine halts but just before `mmi:run` actually returns to its caller.

The User side of the MMI knows what state the MMI is in based on the value returned from the last call to `mmi:run`. When in the halted state, Monsoon is definately not executing code. When in the background state, Monsoon may or may not be executing code. This is because Monsoon continues to execute when `mmi:run` returns an interrupt status. It may subsequently execute a halt instruction or halt for some other reason, before `mmi:run` is called again. In that case, the next call to `mmi:run` will return immediately with a halt status value. More generally, when `mmi:run` returns an interrupt status, Monsoon continues to execute in the background, potentially generating a arbitrary number of interrupt events and possibly a halt event. While the MMI is in the background state, interrupts and halts are queued, and subsequent calls to `mmi:run` will receive the corresponding status values, one call being needed to get the status from each of the queued events. In other words, the sequence of values returned from a series of calls to `mmi:run` does not depend on the real time elapsed between calls.

While interrupts are queued while the MMI is in background mode, it should be noted that Monsoon program-generated interrupts will often have associated with them some other information, which the User side of the MMI can obtain by examining data memory. If a fixed area of data memory is reserved for this purpose, a higher-level protocol must be implemented to insure that Monsoon does not generate interrupts faster than the User side of the MMI can process them.

The state of the MMI determines what MMI calls the User side of the MMI is allowed to make. In the halted state, all MMI calls may be made. In the background state, only calls from a restricted subset may be made. In the running state, the only legal calls are `mmi:force-halt` and `mmi:force-interrupt`. The set of legal calls in the background state are:

- Calls that do not actually involve the Monsoon side of the MMI interface: all functions in Sections 2.1.2, 2.1.3, and 2.1.4.

- Calls to read and write the data field of data memory: `mmi:read-dm-data-as-`x, `mmi:write-dm-data-as-`x, `mmi:blt-from-dm-data-as-`x, and `mmi:blt-to-dm-data-as-`x.
- Calls to read and write the type field of data memory: `mmi:read-dm-type`, `mmi:write-dm-type`, `mmi:blt-from-dm-type`, and `mmi:blt-to-dm-type`.
- Calls to inject tokens: `mmi:inject-token-as-`x.
- Machine control: `mmi:run`, `mmi:force-halt`, and `mmi:force-interrupt`.

Rationale: The calls permitted in background mode reflect the operations that may be done to the Monsoon hardware while it is executing. Technically, the hardware limitations do not preclude the MMI from supporting other operations in the background state; such operations would be performed on the hardware by halting the hardware, scanning out all scannable state, using the scan rings to perform the MMI operation, scanning the original state back in, and restarting Monsoon. This would be transparent to the User side of the MMI (at least, if the User side program doesn't look at a real time clock!). We can add this functionality later if we find it desirable; until then, it seems like a good idea to limit background operations to what can be performed efficiently.

While Monsoon may be executing when the MMI is in the background state, in general the User side of the MMI may only depend on Monsoon making progress when in the running state. Suppose the MMI enters the background state as a result of a Monsoon program generated interrupt. The User side services the interrupt by reading out some information from data memory, writing some other information back into data memory, and finally injecting a token. The User side may not depend on the injected token being executed, or even that the executing program sees the data written in data memory, until the User side makes another call to `mmi:run`. Another way of putting this is that when the MMI is in the background state, Monsoon may be executing infinitely slowly. This restriction is necessary to accomodate the software configuration where the User side of the MMI is directly linked with MINT (see Section 1.4.2).

The only calls that can be made when the MMI is in the *running* state are `mmi:force-halt` and `mmi:force-interrupt`. Note that when they from the running state, they are *nested* within an in-progress call to `mmi:run`. The only way `mmi:force-halt` and `mmi:force-interrupt` could be called in the running state is if they are called from an asynchronous event handler of some sort, such as a "control-c" handler. In fact, `mmi:force-halt` and `mmi:force-interrupt` are allowed to be called in the running state for exactly this purpose. The MMI does *not* support user sides that do arbitrary multi-tasking; in particular, when `mmi:force-halt` and `mmi:force-interrupt` are called in the running state, their execution must preempt all other computation on the user side of the MMI. This restriction is necessary for correct implementation of the MMI client protocol on Unix.

`mmi:run` [*Function*]

Calling `mmi:run` puts the MMI in the running state, returning when either an interrupt event is generated, or when Monsoon halts. A status code is returned indicating what event caused the interrupt or halt. If called from the background state, and interrupt and/or halt events have occured since the last event returned by `mmi:run`, `mmi:run` returns immediately with the status code corresponding to the oldest of those events (that is, the first event that happened after the event returned by the previous call to `mmi:run`).

`Mmi:run` actually returns three values: a status code, an integer of type **`mmi:pe`** *pe*, and an integer *info*. The status code returned from **`mmi:run`** is a keyword, indicating either a halt reason or an interrupt reason. The values *pe* and *info* give additional information for certain halt and interrupt reasons. The halt reasons are as follows; the MMI is in the halted state after one of these codes is returned. Unless otherwise specified, the returned values *pe* and *info* are always zero.

`:halt-instruction` Monsoon executed an explicit halt instruction.

`:halt-forced` A call to **`mmi:force-halt`** was made.

`:halt-timestep-finished` The queueing system determined that the current timestep has finished. The circumstances that cause this kind of halt depend on the current queueing mode; the documentation for each queueing system should indicate what those circumstances are for each mode it supports. The *info* value is an integer giving the reason for the timestep having finished, for those queueing systems which support this. The documentation for each queueing system should define the meanings of *info* values for those modes that use it.

`:halt-machine-idle` There are no tokens left to process.

`:halt-statistic-overflow` A statistic register has overflowed. The offending register will contain zero. The *pe* value indicates which PE had the overflow. (If the user wants to detect which register overflowed, he must have initialized all registers to something other than zero.)

`:halt-queue-full` One of the queueing system's queues has overflowed. On the Monsoon hardware, this halt actually happens when the offending queue has a small number of empty slots left, so that the token causing the overflow is not lost. Other queueing systems should define what happens when this halt occurs. The values *pe* and *info* give the PE number and queue number, repsectively, of the queue that overflowed.

`:halt-machine-check` A hardware fault such as a parity error has occured. The value *pe* indicates which PE had the fault; *info* is a code identifying the kind of fault. For the hardware, the possible values of *info* are 1 for a data path parity error and 2 for a presence bit parity error.

The interrupt reasons are as follows; the MMI is in the background state after one of these codes is returned. Unless otherwise specified, the returned values *pe* and *info* are always zero.

`:intr-instruction` Monsoon executed an explicit interrupt instruction. The value *pe* indicates which PE caused the interrupt.

`:intr-forced` A call to **`mmi:force-interrupt`** was made.

`:intr-input-fifo` The input FIFO of the token network interface is full. The network continues to operate properly; this interrupt is for informational purposes only. The value *pe* indicates which PE caused the interrupt.

`:intr-output-fifo` The output FIFO of the token network interface is full. The network continues to operate properly; this interrupt is for informational purposes only. The value *pe* indicates which PE caused the interrupt.

`:intr-parc-counter` A statistics counter on a network PARC chip has reached half of its maximum value.

Some implementations of the Monsoon side of the MMI may not use all of these codes; in particular, a software emulator is unlikely to return `:halt-machine-check`, `intr-input-fifo`, `:intr-output-fifo`, or `:intr-parc-counter`.

Execution of `mmi:run` must not be thrown out of or otherwise terminated prematurely; `mmi:force-halt` or `mmi:force-interrupt` may be used to terminate `mmi:run` in an orderly fashion.

`mmi:force-halt` [*Function*]
`mmi:force-interrupt` [*Function*]

A call to `mmi:force-halt` causes Monsoon to halt immediately. The MMI will eventually enter the halted state, with `mmi:run` returning `:halt-forced`. Exactly when `mmi:run` returns `:halt-forced` depends on the state of the MMI when the call to `mmi:force-halt` was made. If made in the halted state, then the next call to `mmi:run` will return immediately with `:halt-forced`. If the MMI was in the background state, then there might be pending interrupt and/or halt events; subsequent calls to `mmi:run` will return the codes associated with those events, but eventually a call will return `:halt-forced`. If the MMI was in the running state, then either the in-progress call to `mmi:run` or some subsequent call will return `:halt-forced`, again, depending on whether there were pending interrupt and/or halt events.

In all cases, `mmi:force-halt` causes the machine to halt as soon as possible, and eventually `:halt-forced` will be returned from `mmi:run`. Once a call is made to `mmi:force-halt`, however, no calls to `mmi:force-halt` or `mmi:force-interrupt` should be made until some call to `mmi:run` returns `:halt-forced`; the returning of `:halt-forced` should be considered an acknowledgement of the call to `mmi:force-halt`. After a call to `mmi:force-halt`, other calls to `mmi:force-halt` or `mmi:force-interrupt` *may or may not be ignored*, until `mmi:run` returns `:halt-forced`.

A call to `mmi:force-interrupt` does exactly the same thing as a call to `mmi:force-halt`, with two differences: execution of Monsoon is *not* stopped, if it was running, and `:intr-forced` is eventually returned from `mmi:run` instead of `:halt-forced`. Note that `mmi:force-interrupt` may be called even when the MMI is in the halted state; if it is, the next call to `mmi:run` will immediately return `:intr-forced`, the MMI will be in the background state, but Monsoon will not be executing. As with `mmi:force-halt`, after calling `mmi:force-interrupt` no subsequent call to `mmi:force-halt` or `force-interrupt` should be made until `mmi:run` returns `:intr-forced`.

2.6 C Interface

The C version of the MMI interface is very similar to the Common Lisp version. Most of the changes are necessary because of characteristics of the C programming language. All differences are summarized below:

- MMI functions and types in the C interface are uniformly prefixed with `MMI_` or `mmi_`, as appropriate; in the Common Lisp interface they are prefixed instead by `mmi:`.
- The limits of various integer quantities are given slightly different names, reflecting different conventions used by C programmers. In the Common Lisp interface, an unsigned

quantity X would be described by an upper exclusive bound **mmi:X-limit**, while in the C interface it is described by an upper *inclusive* bound **MMI_X_MAX**. In both the Common Lisp and C interfaces, a signed quantity Y is described by upper and lower inclusive bounds, but while in the Common Lisp interface they are called **mmi:most-positive-Y** and **mmi:most-negative-Y**, in the C interface they are called **MMI_Y_MAX** and **MMI_Y_MIN**. Note that in the C interface, limits always have the same type as the quantities they describe (in the Common Lisp interface, the limit on unsigned quantities falls outside the type, since it is an exclusive bound).

These remarks apply not only to limits described by compile-time constants, but also to functions such as **mmi_qn_max** and **mmi_statistic_max**.

- In the C interface the limits on type types **MMI_DM_BITS** and **MMI_DM_INTEGER** are given by functions of no arguments, rather than by compile-time constants. This is because C does not permit compile-time constants of type **MMI_DM_BITS** or **MMI_DM_INTEGER**.
- The data structures representing tags and instructions have constructors and selectors analogous to the Common Lisp interface, but they may also be constructed and selected using C field name syntax. The C interface has no equivalent to the **mmi:make-new-X** functions or the **mmi:Xp** predicates in the Common Lisp interface.
- In the C interface, the **mmi_write_dm_** and **mmi_write_im_** functions, as well as **mmi_write_queue_pointer** and **mmi_write_statistic** are all of type **VOID**. In the Common Lisp interface, they each return their data argument, so that they may be accessed via **setf**. In the C interface, there is no construct comparable to Common Lisp's use of **setf** with the read functions.
- The C interface's version of the **mmi:read-dm-dtp-** functions is slightly different because multiple values can only be returned through pointer arguments.
- The C interface's block transfer functions are slightly different owing to the lack of keyword arguments in C (a lack that is partially compensated by the ability to pass pointers to the middle of arrays in C).
- In the C interface, two sets of functions are provided for reading token queues. The functions in the first set return the four components of a token as four return arguments. The second set provides separate functions for reading each of the four components of a token.
- The representation of the *mode-arg-types* and *mode-args* arguments to the queueing mode manipulation functions is different.
- The method of encoding the return value from **mmi_run** is different.

The following sections document the C version of the MMI interface. The MMI is written in draft proposed ANSI C, as described in *C: A Reference Manual*,[3] and requires the **scalar** and **int64** packages from the MCRC C Language Support Libraries. The terms "integer type" and "floating point type" are defined in the documentation of the **scalar** library, while the term "**INT64** type" is defined in the documentation of the **int64** library.

[3] S. P. Harbison and G. L. Steele Jr., *C: A Reference Manual*, Prentice-Hall, Englewood Cliffs NJ, 1987.

2.6.1 Types and Sizes of Fields

`MMI_DM_BITS` [*Type*]
`MMI_DM_BITS mmi_dm_bits_max ()` [*Function*]

The type `MMI_DM_BITS` is the type used to pass the "bits" representation of data memory words across the MMI. It is an unsigned `INT64` type, as defined in the documentation of the MCRC C Language Support Library `int64`. As such, it requires special functions for performing arithmetic, provided by the `int64` library. The function `mmi_dm_bits_max` returns the greatest (inclusive) value of an `MMI_DM_BITS` (this is a function rather than a compile-time constant because C does not permit the definition of compile-time `INT64` constants).

`MMI_DM_INTEGER` [*Type*]
`MMI_DM_INTEGER mmi_dm_integer_max ()` [*Function*]
`MMI_DM_INTEGER mmi_dm_integer_min ()` [*Function*]

The type `MMI_DM_INTEGER` is the type used to pass the "integer" representation of data memory words across the MMI. It is a signed `int64` type, as defined in the documentation of the MCRC C Language Support Library `int64`. As such, it requires special functions for performing arithmetic, provided by the `int64` library. The functions `mmi_dm_integer_max` and `mmi_dm_integer_min` return the greatest and least (inclusive) values, respectively, of an `MMI_DM_INTEGER` (these are functions rather than compile-time constants because C does not permit the definition of compile-time `int64` constants).

`MMI_DM_FLOAT` [*Type*]
`MMI_DM_FLOAT MMI_DM_FLOAT_MAX` [*Constant*]
`MMI_DM_FLOAT MMI_DM_FLOAT_MIN` [*Constant*]
`MMI_DM_FLOAT MMI_DM_FLOAT_EPSILON` [*Constant*]
`MMI_DM_FLOAT MMI_DM_FLOAT_NEGATIVE_EPSILON` [*Constant*]

The type `MMI_DM_FLOAT` is the type used to pass the "integer" representation of data memory words across the MMI. It is an opaque floating point type, as defined by the `scalar` package. The four constants have the same meaning as for the Common Lisp interface.

`MMI_TAG` [*Type*]
`MMI_PORT` [*Type*]
`MMI_MAP` [*Type*]
`MMI_IP` [*Type*]
`MMI_PE` [*Type*]
`MMI_FP` [*Type*]
`MMI_PORT MMI_PORT_MAX` [*Constant*]
`MMI_MAP MMI_MAP_MAX` [*Constant*]
`MMI_IP MMI_IP_MAX` [*Constant*]
`MMI_PE MMI_PE_MAX` [*Constant*]
`MMI_FP MMI_FP_MAX` [*Constant*]

The type `MMI_TAG` is the type used to pass the "tag" representation of data memory words across the MMI. It is a `struct` of five fields, with constructors and selectors defined in the next section. The types `MMI_PORT`, `MMI_MAP`, `MMI_IP`, `MMI_PE`, and `MMI_FP` are the types of the values found in the slots of a `MMI_TAG`. The (inclusive) range of each of these types is zero through the value of `MMI_PORT_MAX`, `MMI_MAP_MAX`, `MMI_IP_MAX`, `MMI_PE_MAX`, or `MMI_FP_MAX`,

as appropriate. MMI_PORT and MMI_MAP are opaque, unsigned integer types, as defined by the scalar package. MMI_IP, MMI_PE, and MMI_FP are opaque, *signed* integer types; they are signed types despite the fact that they range only over non-negative integers, to make it easier to use them in arithmetic expressions involving other signed types.

```
MMI_TYPE                                        [Type]
MMI_TYPE MMI_TYPE_MAX                           [Constant]
MMI_PRESENCE                                    [Type]
MMI_PRESENCE MMI_PRESENCE_MAX                   [Constant]
```

The type MMI_TYPE is the return type of functions that return the type field of Monsoon data memory, as well as the type of arguments which receive values for that field. MMI_PRESENCE is analogous. Both MMI_TYPE and MMI_PRESENCE are opaque, unsigned integer types, and their (inclusive) range is zero through the value of MMI_TYPE_MAX or MMI_PRESENCE_MAX, as appropriate.

```
MMI_IM_BITS                                     [Type]
MMI_IM_BITS MMI_IM_BITS_MAX                     [Constant]
```

The type MMI_IM_BITS is the type used to pass the "bits" representation of instruction memory words acrosss the MMI. It is an opaque, unsigned integer type, with (inclusive) range zero through the value of MMI_IM_BITS_MAX.

```
MMI_INSTRUCTION                                 [Type]
MMI_OPCODE                                      [Type]
MMI_F1                                          [Type]
MMI_F2                                          [Type]
MMI_OPCODE MMI_OPCODE_MAX                       [Constant]
MMI_F1 MMI_F1_MAX                               [Constant]
MMI_F2 MMI_F2_MAX                               [Constant]
```

The type MMI_INSTRUCTION is the type used to pass the "instruction" representation of instruction memory words across the MMI. It is a **struct** of three fields, with constructors and selectors defined in the next section. The types MMI_OPCODE, MMI_F1, and MMI_F2 are the types of the values found in the slots of an MMI_INSTRUCTION. All three of these types are opaque, unsigned integer types, with (inclusive) range zero through the value of MMI_OPCODE_MAX, MMI_F1_MAX, or MMI_F2_MAX, as appropriate.

Additional constructors and selectors for viewing subfields of F1 and F2 are defined in the next section.

2.6.2 Constructors and Selectors

Tags and instructions are advertised as **structs**, which are always passed by *value* to and from MMI functions. Good style would normally be to use these types immutably. The usual constructor and selector functions are provided to support this style. Unfortunately, the constructor function may entail procedure calling overhead with compilers that do not support inline function substitution, and there is no way of expressing the constructor as a C macro. To help alleviate this deficiency, the field names of the MMI_TAG and MMI_INSTRUCTION **structs** are advertised, so that they can be initialized with assignment statements on individual fields. The *order* of the fields is not advertised, however, so the use of brace-enclosed initializer lists is not permitted with these types.

`MMI_TAG mmi_make_tag (MMI_PORT port, MMI_MAP map, MMI_IP ip, MMI_PE pe, MMI_FP fp)` [*Function*]

`MMI_PORT mmi_tag_port (MMI_TAG tag)` [*Macro*]

`MMI_MAP mmi_tag_map (MMI_TAG tag)` [*Macro*]

`MMI_IP mmi_tag_ip (MMI_TAG tag)` [*Macro*]

`MMI_PE mmi_tag_pe (MMI_TAG tag)` [*Macro*]

`MMI_FP mmi_tag_fp (MMI_TAG tag)` [*Macro*]

The function `mmi_make_tag` returns a tag with the given values as components. Note that since it returns the tag by value, `mmi_make_tag` does no storage allocation. The function `mmi_tag_port` returns the `port` field of a tag; the other four selectors are analogous.

`port` [*Field Name*]

`map` [*Field Name*]

`ip` [*Field Name*]

`pe` [*Field Name*]

`fp` [*Field Name*]

Each is the field name of one of the five slots of a `MMI_TAG` `struct`, and may be used to fetch from or assign to a slot of an `MMI_TAG`. The type of these fields are `MMI_PORT`, `MMI_MAP`, `MMI_IP`, `MMI_PE`, and `MMI_FP`, as appropriate.

`MMI_INSTRUCTION mmi_make_instruction (MMI_OPCODE opcode, MMI_F1 f1, MMI_F2 f2)` [*Function*]

`MMI_OPCODE mmi_instruction_opcode (MMI_INSTRUCTION instruction)` [*Macro*]

`MMI_F1 mmi_instruction_f1 (MMI_INSTRUCTION instruction)` [*Macro*]

`MMI_F2 mmi_instruction_f2 (MMI_INSTRUCTION instruction)` [*Macro*]

`opcode` [*Field Name*]

`f1` [*Field Name*]

`f2` [*Field Name*]

These constructors, selectors, and field names are entirely analogous to those for `MMI_TAG`s, but apply to `MMI_INSTRUCTION`s.

`MMI_S_1` [*Type*]

`MMI_S_2` [*Type*]

`MMI_S_1 MMI_S_1_MAX` [*Constant*]

`MMI_S_1 MMI_S_1_MIN` [*Constant*]

`MMI_S_2 MMI_S_2_MAX` [*Constant*]

`MMI_S_2 MMI_S_2_MIN` [*Constant*]

`MMI_LONG_R` [*Type*]

`MMI_LONG_R MMI_LONG_R_MAX` [*Constant*]

`MMI_R` [*Type*]

`MMI_REG` [*Type*]

`MMI_R MMI_R_MAX` [*Constant*]

`MMI_REG MMI_REG_MAX` [*Constant*]

The types `MMI_S_1`, `MMI_S_2`, `MMI_LONG_R`, `MMI_R`, and `MMI_REG` are exactly as defined in the Common Lisp interface: they are various views of the `F1` and `F2` fields of instructions. `MMI_S_1` and `MMI_S_2` are opaque, signed integer types, with inclusive ranges given by the four constants as shown. `MMI_REG` is an opaque, unsigned integer type, with inclusive range from

zero through the value of MMI_REG_MAX. MMI_LONG_R and MMI_R are opaque, signed integer types, with inclusive ranges from zero through the value of the appropriate constant. MMI_LONG_R and MMI_R are signed types, despite their ranging only over non-negative integers, to make it easier to use them in arithmetic expressions with other signed types.

```
MMI_PORT mmi_f1_port (MMI_F1 f1)                                        [Function]
MMI_S_1 mmi_f1_s (MMI_F1 f1)                                            [Function]
MMI_F1 mmi_make_f1 (MMI_PORT port, MMI_S_1 s)                           [Function]
MMI_PORT mmi_f2_port (MMI_F2 f2)                                        [Function]
MMI_S_2 mmi_f2_s (MMI_F2 f2)                                            [Function]
MMI_F2 mmi_make_f2 (MMI_PORT port, MMI_S_2 s)                           [Function]
MMI_F1 mmi_long_r_f1 (MMI_LONG_R r)                                     [Function]
MMI_F2 mmi_long_r_f2 (MMI_LONG_R r)                                     [Function]
MMI_PORT mmi_instruction_port_1 (MMI_INSTRUCTION instruction)           [Function]
MMI_S_1 mmi_instruction_s_1 (MMI_INSTRUCTION instruction)               [Function]
MMI_PORT mmi_instruction_port_2 (MMI_INSTRUCTION instruction)           [Function]
MMI_S_2 mmi_instruction_s_2 (MMI_INSTRUCTION instruction)               [Function]
MMI_LONG_R mmi_f1_f2_long_r (MMI_F1 f1, MMI_F2 f2)                      [Function]
MMI_LONG_R mmi_instruction_long_r (MMI_INSTRUCTION instruction)         [Function]
MMI_INSTRUCTION mmi_make_instruction_long_r (MMI_OPCODE opcode,         [Function]
                                             MMI_LONG_R r)
MMI_R mmi_instruction_r (MMI_INSTRUCTION instruction)                   [Function]
MMI_REG mmi_instruction_reg (MMI_INSTRUCTION instruction)               [Function]
```

These functions for viewing the subfields of the F1 and F2 fields of instructions are exactly as defined in the Common Lisp interface.

```
MMI_DM_INTEGER mmi_bits_to_integer (MMI_DM_BITS bits)                   [Function]
MMI_DM_FLOAT mmi_bits_to_float (MMI_DM_BITS bits)                       [Function]
MMI_TAG mmi_bits_to_tag (MMI_DM_BITS bits)                              [Function]
MMI_DM_BITS mmi_integer_to_bits (MMI_DM_INTEGER integer)                [Function]
MMI_DM_FLOAT mmi_integer_to_float (MMI_DM_INTEGER integer)              [Function]
MMI_TAG mmi_integer_to_tag (MMI_DM_INTEGER integer)                     [Function]
MMI_DM_BITS mmi_float_to_bits (MMI_DM_FLOAT f)                          [Function]
MMI_DM_INTEGER mmi_float_to_integer (MMI_DM_FLOAT f)                    [Function]
MMI_TAG mmi_float_to_tag (MMI_DM_FLOAT f)                               [Function]
MMI_DM_BITS mmi_tag_to_bits (MMI_TAG tag)                               [Function]
MMI_DM_INTEGER mmi_tag_to_integer (MMI_TAG tag)                         [Function]
MMI_DM_FLOAT mmi_tag_to_float (MMI_TAG tag)                             [Function]
MMI_INSTRUCTION mmi_bits_to_instruction (MMI_BITS bits)                 [Function]
MMI_IM_BITS mmi_instruction_to_bits (MMI_INSTRUCTION instruction)       [Function]
```

These conversion functions are exactly as defined in the Common Lisp interface.

2.6.3 Reading and Writing Memory

The functions for reading and writing memory are for the most part exactly the same as the corresponding functions in the Common Lisp interface. The "dtp" functions as well as the block transfer functions have slightly different interfaces, owing to differences between C and Lisp.

```
MMI_DM_BITS mmi_read_dm_data_as_bits (MMI_PE pe, MMI_FP address)                [Function]
MMI_DM_INTEGER mmi_read_dm_data_as_integer (MMI_PE pe,                          [Function]
                                            MMI_FP address)
MMI_DM_FLOAT mmi_read_dm_data_as_float (MMI_PE pe, MMI_FP address)              [Function]
MMI_TAG mmi_read_dm_data_as_tag (MMI_PE pe, MMI_FP address)                     [Function]
MMI_TYPE mmi_read_dm_type (MMI_PE pe, MMI_FP address)                           [Function]
MMI_PRESENCE mmi_read_dm_presence (MMI_PE pe, MMI_FP address)                   [Function]
VOID mmi_write_dm_data_as_bits (MMI_PE pe, MMI_FP address,                      [Function]
                                MMI_DM_BITS integer)
VOID mmi_write_dm_data_as_integer (MMI_PE pe, MMI_FP address,                   [Function]
                                   MMI_DM_INTEGER integer)
VOID mmi_write_dm_data_as_float (MMI_PE pe, MMI_FP address,                     [Function]
                                 MMI_DM_FLOAT f)
VOID mmi_write_dm_data_as_tag (MMI_PE pe, MMI_FP address,                       [Function]
                               MMI_TAG tag)
VOID mmi_write_dm_type (MMI_PE pe, MMI_FP address, MMI_TYPE integer)            [Function]
VOID mmi_write_dm_presence (MMI_PE pe, MMI_FP address,                          [Function]
                            MMI_PRESENCE integer)
MMI_IM_BITS mmi_read_im_data_as_bits (MMI_PE pe, MMI_IP address)                [Function]
MMI_INSTRUCTION mmi_read_im_data_as_instruction (MMI_PE pe,                     [Function]
                                                 MMI_IP address)
VOID mmi_write_im_data_as_bits (MMI_PE pe, MMI_IP address,                      [Function]
                                MMI_IM_BITS integer)
VOID mmi_write_im_data_as_instruction (MMI_PE pe, MMI_IP address,               [Function]
                                       MMI_INSTRUCTION instruction)
```

These functions are exactly as described in the Common Lisp interface.

```
VOID mmi_read_dm_dtp_as_bits (MMI_PE pe, MMI_FP address,                        [Function]
                              return MMI_DM_BITS *data,
                              return MMI_TYPE *type,
                              return MMI_PRESENCE *presence)
VOID mmi_read_dm_dtp_as_integer (MMI_PE pe, MMI_FP address,                     [Function]
                                 return MMI_DM_INTEGER *data,
                                 return MMI_TYPE *type,
                                 return MMI_PRESENCE *presence)
VOID mmi_read_dm_dtp_as_float (MMI_PE pe, MMI_FP address,                       [Function]
                               return MMI_DM_FLOAT *data,
                               return MMI_TYPE *type,
                               return MMI_PRESENCE *presence)
VOID mmi_read_dm_dtp_as_tag (MMI_PE pe, MMI_FP address,                         [Function]
                             return MMI_TAG *data,
                             return MMI_TYPE *type,
                             return MMI_PRESENCE *presence)
VOID mmi_write_dm_dtp_as_bits (MMI_PE pe, MMI_FP address,                       [Function]
                               MMI_DM_BITS data, MMI_TYPE type,
                               MMI_PRESENCE presence)
```

```
VOID mmi_write_dm_dtp_as_integer (MMI_PE pe, MMI_FP address,          [Function]
                                  MMI_DM_INTEGER data, MMI_TYPE type,
                                  MMI_PRESENCE presence)
VOID mmi_write_dm_dtp_as_float (MMI_PE pe, MMI_FP address,            [Function]
                                MMI_DM_FLOAT data, MMI_TYPE type,
                                MMI_PRESENCE presence)
VOID mmi_write_dm_dtp_as_tag (MMI_PE pe, MMI_FP address,              [Function]
                              MMI_DM_TAG data, MMI_TYPE type,
                              MMI_PRESENCE presence)
```

These functions are exactly as described in the Common Lisp interface. But because C does not permit multiple return values, the three results returned by the `mmi_read_dm_dtp_` functions are returned by side-effecting the variables pointed to by the last three arguments.

```
VOID mmi_blt_from_dm_data_as_bits (MMI_PE pe, MMI_FP start_address,   [Function]
                                   MMI_FP end_address,
                                   MMI_DM_BITS to_array[])
VOID mmi_blt_from_dm_data_as_integer (MMI_PE pe,                      [Function]
                                      MMI_FP start_address,
                                      MMI_FP end_address,
                                      MMI_DM_INTEGER to_array[])
VOID mmi_blt_from_dm_data_as_float (MMI_PE pe, MMI_FP start_address,  [Function]
                                    MMI_FP end_address,
                                    MMI_DM_FLOAT to_array[])
VOID mmi_blt_from_dm_data_as_tag (MMI_PE pe, MMI_FP start_address,    [Function]
                                  MMI_FP end_address,
                                  MMI_TAG to_array[])
VOID mmi_blt_from_dm_type (MMI_PE pe, MMI_FP start_address,           [Function]
                           MMI_FP end_address, MMI_TYPE to_array[])
VOID mmi_blt_from_dm_presence (MMI_PE pe, MMI_FP start_address,       [Function]
                               MMI_FP end_address,
                               MMI_PRESENCE to_array[])
VOID mmi_blt_to_dm_data_as_bits (MMI_DM_BITS from_array[], MMI_PE pe, [Function]
                                 MMI_FP start_address,
                                 MMI_FP end_address)
VOID mmi_blt_to_dm_data_as_integer (MMI_DM_INTEGER from_array[],      [Function]
                                    MMI_PE pe, MMI_FP start_address,
                                    MMI_FP end_address)
VOID mmi_blt_to_dm_data_as_float (MMI_DM_FLOAT from_array[],          [Function]
                                  MMI_PE pe, MMI_FP start_address,
                                  MMI_FP end_address)
VOID mmi_blt_to_dm_data_as_tag (MMI_TAG from_array[], MMI_PE pe,      [Function]
                                MMI_FP start_address,
                                MMI_FP end_address)
VOID mmi_blt_to_dm_type (MMI_TYPE from_array[], MMI_PE pe,            [Function]
                         MMI_FP start_address, MMI_FP end_address)
VOID mmi_blt_to_dm_presence (MMI_PRESENCE from_array[], MMI_PE pe,    [Function]
                             MMI_FP start_address,
                             MMI_FP end_address)
```

Argtype Value	*Argument's C Type*	*Argtype Value*	*Argument's C Type*
MMI_QMA_DM_BITS	MMI_DM_BITS	MMI_QMA_S_1	MMI_S_1
MMI_QMA_DM_INTEGER	MMI_DM_INTEGER	MMI_QMA_S_2	MMI_S_2
MMI_QMA_DM_FLOAT	MMI_DM_FLOAT	MMI_QMA_LONG_R	MMI_LONG_R
MMI_QMA_TAG	MMI_TAG	MMI_QMA_R	MMI_R
MMI_QMA_PORT	MMI_PORT	MMI_QMA_REG	MMI_REG
MMI_QMA_MAP	MMI_MAP	MMI_QMA_QN	MMI_QN
MMI_QMA_IP	MMI_IP	MMI_QMA_QP	MMI_QP
MMI_QMA_PE	MMI_PE	MMI_QMA_QPN	MMI_QPN
MMI_QMA_FP	MMI_FP	MMI_QMA_STATISTIC_N	MMI_STATISTIC_N
MMI_QMA_TYPE	MMI_TYPE	MMI_QMA_STATISTIC_VALUE	MMI_STATISTIC_VALUE
MMI_QMA_PRESENCE	MMI_PRESENCE	MMI_QMA_AUINT32	auINT32
MMI_QMA_IM_BITS	MMI_IM_BITS	MMI_QMA_ASINT32	asINT32
MMI_QMA_INSTRUCTION	MMI_INSTRUCTION	MMI_QMA_DFLOAT	dFLONUM
MMI_QMA_OPCODE	MMI_OPCODE	MMI_QMA_CHARACTER	CHARACTER
MMI_QMA_F1	MMI_F1	MMI_QMA_STRING	STRING
MMI_QMA_F2	MMI_F2	MMI_QMA_KEYWORD	KEYWORD

Figure 2.3: Queueing Mode Argument Types (C Version)

```
VOID mmi_blt_from_im_as_bits (MMI_PE pe, MMI_IP start_address,          [Function]
                              MMI_IP end_address,
                              MMI_IM_BITS to_array[])
VOID mmi_blt_from_im_as_instruction (MMI_PE pe, MMI_IP start_address, [Function]
                                     MMI_IP end_address,
                                     MMI_INSTRUCTION to_array[])
VOID mmi_blt_to_im_as_bits (MMI_IM_BITS from_array[], MMI_PE pe,        [Function]
                            MMI_IP start_address, MMI_IP end_address)
VOID mmi_blt_to_im_as_instruction (MMI_INSTRUCTION from_array[],        [Function]
                                   MMI_PE pe, MMI_IP start_address,
                                   MMI_IP end_address)
```

These block transfer functions are exactly as described in the Common Lisp interface, except in how the number of words to be transfered is specified. For all functions above, the number of words transferred is:

```
end_address - start_address + 1
```

The words are transferred to or from the given array beginning with element zero, but note that

```
mmi_blt_from_dm_data_as_bits(pe, start, end, &(a[5]));
```

has the effect of transferring data into a beginning with element five. This capability in C largely compensates for the simpler interface compared to the Common Lisp block transfer functions.

2.6.4 Token Queues

```
MMI_QMODE_ARGTYPE                                                       [Type]
```

```
MMI_QMODE_ARG                                                    [Type]
BOOLEAN mmi_select_queueing_system (KEYWORD name, KEYWORD mode,  [Function]
                                    asINT32 n_args,
                                    MMI_QMODE_ARGTYPE argtypes[],
                                    MMI_QMODE_ARG mode_args)
BOOLEAN mmi_select_queueing_mode (KEYWORD mode, asINT32 n_args,  [Function]
                                  MMI_QMODE_ARGTYPE argtypes[],
                                  MMI_QMODE_ARG args)
```

These functions have the same behavior as in the Common Lisp interface, but the methods of passing arguments is slightly different. To specify the queueing mode parameters to **mmi_select_queueing_system** and **mmi_select_queueing_mode**, three arguments are passed. **n_args** gives the number of arguments, **args** is an array of the parameters, and **argtypes** is an array each of element of which indicates the type of the corresponding parameter. Hence, the **argtypes** and **args** arrays are analogous to the *mode-arg-types* and *mode-args* arguments in the Common Lisp interface. Each element of **argtypes** is of type **MMI_QMODE_ARGTYPE**, an enumeration type whose values are given in the "Argtype Value" columns of Figure 2.3. Each element of **args** is a of type **MMI_QMODE_ARG**, a union of the types given in the "C Type" columns of Figure 2.3. The union tags of each arm of the **MMI_QMODE_ARG** type may be found by removing the **MMI_QMA_** prefix from the corresponding **MMI_QMODE_ARGTYPE** value, and converting to all lowercase. For example, if **argtypes[3]** is **MMI_QMA_ASINT32**, then the corresponding argument is **(args[3]).asint32**, which is of type **asINT32**. If **argtypes[4]** is **MMI_QMA_DM_BITS**, then the argument is **(args[4]).dm_bits**, of type **MMI_DM_BITS**.

The implementation of **mmi_select_queueing_system** and **mmi_select_queueing_mode** is not permitted to keep pointers to the **argtypes** and **args**, so that the caller of these functions may deallocate these arrays after the calls return.

As in the Common Lisp interface, it is recommended that the author of each queueing system provide higher-level functions that sit on the MMI, hiding the encoding of parameters into the **argtypes** and **args** arrays. For example, the **select-n-step-mode** example given in the documentation of the Common Lisp interface would be rendered in C as follows:

```
VOID select_n_step_mode (auINT32 n) {
  KEYWORD mode = name_keyword("N-STEP");
  MMI_QMODE_ARGTYPE argtypes[1] = {MMI_QMA_AUINT32};
  MMI_QMODE_ARG args[1];
  (args[0]).auint32 = n;
  return(mmi_select_queueing_mode(mode, 1, argtypes, args));
}
```

While this example shows the **argtypes** array being contructed each time **select_n_step_mode** is called, an alternative implementation could construct this array once and use it each time **select_n_step_mode** is called.

```
VOID mmi_reset_queueing_system ()                                [Function]
```

This function is the same as in the Common Lisp implementation.

```
asINT32 mmi_current_queueing_mode (asINT32 max_args,                    [Function]
                                   return KEYWORD *name,
                                   return KEYWORD *mode,
                                   return MMI_QMODE_ARGTYPE argtypes[],
                                   return MMI_QMODE_ARG args[])
```

This function has the same behavior as in the Common Lisp interface, but the methods of receiving results is different. It returns the current queueing system name and the current mode name by storing into the variables pointed to by the **name** and **mode** arguments, and stores the current **argtype** and **args** values into the arrays given. The argument **max_args** limits the number of values stored: if the actual number of parameters for the current queueing mode is greater than **max_args**, then only the first **max_args** types and arguments will be stored into **argtypes** and **args**. If the actual number of parameters is fewer than **max_args**, however, then elements of **argtypes** and **args** beyond those parameters will remain unmodified. The value returned from **mmi_current_queueing_mode** indicates the actual number of parameters for the current queueing mode; the number of elements of **argtypes** and **args** modified will be the lesser of this number and **max_args**. Either of **name** or **mode** may be the null pointer to suppress the returning of those values.

A **max_args** value of zero may be given in conjunction with null pointers for the remaining arguments to find out how many parameters the current queueing mode has, in preparation for a second call to **mmi_current_queueing_mode**. To illustrate, the following code fragment leaves the current queueing mode unchanged, except that the queueing system is reset.

```
{
  asINT32 n_args = mmi_current_queueing_mode(0, NULL, NULL, NULL, NULL);
  KEYWORD name;
  KEYWORD mode;
  MMI_QMODE_ARGTYPES *argtypes =
    (MMI_QMODE_ARGTYPES *)malloc(n_args * sizeof(MMI_QMODE_ARGTYPE));
  MMI_QMODE_ARGS *args =
    (MMI_QMODE_ARGS *)malloc(n_args * sizeof(MMI_QMODE_ARG));
  mmi_current_queueing_mode(n_args, &name, &mode, argtypes, args);
  select_current_queueing_mode(name, mode, n_args, argtypes, args);
}
```

```
MMI_EVENT mmi_run ()                                                    [Function]
VOID mmi_advance_timestep ()                                            [Function]
```

These functions have the same behavior as in the Common Lisp interface, and are described in Section 2.6.6.

```
MMI_QN                                                                  [Type]
MMI_QP                                                                  [Type]
MMI_QPN                                                                 [Type]
QN mmi_qn_max (MMI_PE pe)                                               [Function]
QP mmi_qp_max (MMI_PE pe, MMI_QN qn)                                    [Function]
QPN mmi_qpn_max (MMI_PE pe, MMI_QN qn)                                  [Function]
```

These types and functions are the same as in the Common Lisp interface. All three types are opaque, signed integer types. They are signed types, despite their ranging only over non-

negative integers, to make it easier to use them in arithmetic expressions with other signed types.

```
VOID mmi_read_queue_as_bits (MMI_PE pe, MMI_QN qn, MMI_QP address,      [Function]
                             return MMI_TYPE *tag_type,
                             return MMI_TAG *tag_data,
                             return MMI_TYPE *value_type,
                             return MMI_DM_BITS *value_data)
VOID mmi_read_queue_as_integer (MMI_PE pe, MMI_QN qn, MMI_QP address,  [Function]
                                return MMI_TYPE *tag_type,
                                return MMI_TAG *tag_data,
                                return MMI_TYPE *value_type,
                                return MMI_DM_INTEGER *value_data)
VOID mmi_read_queue_as_float (MMI_PE pe, MMI_QN qn, MMI_QP address,    [Function]
                              return MMI_TYPE *tag_type,
                              return MMI_TAG *tag_data,
                              return MMI_TYPE *value_type,
                              return MMI_DM_FLOAT *value_data)
VOID mmi_read_queue_as_tag (MMI_PE pe, MMI_QN qn, MMI_QP address,      [Function]
                            return MMI_TYPE *tag_type,
                            return MMI_TAG *tag_data,
                            return MMI_TYPE *value_type,
                            return MMI_TAG *value_data)
```

These functions are exactly as described in the Common Lisp interface. But because C does not permit multiple return values, the four results are returned by side-effecting the variables pointed to by the last four arguments.

```
MMI_DM_BITS mmi_read_queue_value_data_as_bits (MMI_PE pe, MMI_QN qn,    [Function]
                                               MMI_QP address)
MMI_DM_INTEGER mmi_read_queue_value_data_as_integer (MMI_PE pe,        [Function]
                                                     MMI_QN qn,
                                                     MMI_QP address)
MMI_DM_FLOAT mmi_read_queue_value_data_as_float (MMI_PE pe,            [Function]
                                                 MMI_QN qn,
                                                 MMI_QP address)
MMI_TAG mmi_read_queue_value_data_as_tag (MMI_PE pe, MMI_QN qn,        [Function]
                                          MMI_QP address)
MMI_TYPE mmi_read_queue_value_type (MMI_PE pe, MMI_QN qn,              [Function]
                                    MMI_QP address)
MMI_TAG mmi_read_queue_tag_data (MMI_PE pe, MMI_QN qn,                 [Function]
                                 MMI_QP address)
MMI_TYPE mmi_read_queue_tag_type (MMI_PE pe, MMI_QN qn,                [Function]
                                  MMI_QP address)
```

These are alternative functions for reading token queues that do not return values by side-effecting arguments. Instead, separate functions are provided for reading each of the four components of a token.

```
VOID mmi_write_queue_as_bits (MMI_PE pe, MMI_QN qn, MMI_QP address,          [Function]
                              MMI_TYPE tag_type, MMI_TAG tag_data,
                              MMI_TYPE value_type,
                              MMI_DM_BITS value_data)
VOID mmi_write_queue_as_integer (MMI_PE pe, MMI_QN qn,                       [Function]
                                 MMI_QP address, MMI_TYPE tag_type,
                                 MMI_TAG tag_data,
                                 MMI_TYPE value_type,
                                 MMI_DM_INTEGER value_data)
VOID mmi_write_queue_as_float (MMI_PE pe, MMI_QN qn, MMI_QP address,         [Function]
                               MMI_TYPE tag_type, MMI_TAG tag_data,
                               MMI_TYPE value_type,
                               MMI_DM_FLOAT value_data)
VOID mmi_write_queue_as_tag (MMI_PE pe, MMI_QN qn, MMI_QP address,           [Function]
                             MMI_TYPE tag_type, MMI_TAG tag_data,
                             MMI_TYPE value_type, MMI_TAG value_data)
VOID mmi_inject_token_as_bits (MMI_PE pe, MMI_TYPE tag_type,                 [Function]
                               MMI_TAG tag_data, MMI_TYPE value_type,
                               MMI_DM_BITS value_data)
VOID mmi_inject_token_as_integer (MMI_PE pe, MMI_TYPE tag_type,              [Function]
                                  MMI_TAG tag_data,
                                  MMI_TYPE value_type,
                                  MMI_DM_INTEGER value_data)
VOID mmi_inject_token_as_float (MMI_PE pe, MMI_TYPE tag_type,                [Function]
                                MMI_TAG tag_data,
                                MMI_TYPE value_type,
                                MMI_DM_FLOAT value_data)
VOID mmi_inject_token_as_tag (MMI_PE pe, MMI_TYPE tag_type,                  [Function]
                              MMI_TAG tag_data, MMI_TYPE value_type,
                              MMI_TAG value_data)
```

These functions are exactly as described in the Common Lisp interface.

```
MMI_QP mmi_read_queue_pointer (MMI_PE pe, MMI_QN qn, MMI_QPN pointer)        [Function]
VOID mmi_write_queue_pointer (MMI_QP new_value, MMI_PE pe, MMI_QN qn,        [Function]
                              MMI_QPN pointer)
```

These functions are exactly as described in the Common Lisp interface.

2.6.5 Statistics Registers

```
MMI_STATISTIC_N                                                              [Type]
MMI_STATISTIC_VALUE                                                          [Type]
MMI_STATISTIC_N mmi_statistic_max (MMI_PE pe)                                [Function]
MMI_STATISTIC_VALUE mmi_statistic_value_max (MMI_PE pe,                      [Function]
                                             MMI_STATISTIC_N statistic)
MMI_STATISTIC_VALUE mmi_read_statistic (MMI_PE pe,                           [Function]
                                        MMI_STATISTIC_N statistic)
```

`VOID mmi_write_statistic (MMI_PE pe, MMI_STATISTIC_N statistic, MMI_STATISTIC_VALUE new_value)` [*Function*]

`VOID mmi_clear_statistics (MMI_PE pe)` [*Function*]

These types and functions are exactly as described in the Common Lisp interface. `MMI_STATISTIC_N` is an opaque, signed integer type. It is signed, despite its ranging only over non-negative integers, to make it easier to use in arithmetic expressions with other signed types. `MMI_STATISTIC_VALUE` is an unsigned `INT64` type; arithmetic on values of this type must be done using the functions provided by the `int64` package.

2.6.6 Machine Control

The function `mmi_run` has identical behavior and restrictions as its Common Lisp counterpart. The only difference is in how it returns its results: rather than returning three values, it returns a three-component `struct` by value.

`MMI_EVENT` [*Type*]

The return type of `mmi_run`, describing a halt or interrupt event. It is a three component `struct`, whose components are the event type, the PE (when appropriate), and an additional value (when appropriate). These components correspond to the status code, *pe*, and *info* returned by the Common Lisp function `mmi:run`.

`MMI_EVENT mmi_make_event (MMI_EVENT_TYPE event_type, MMI_PE pe, asINT32 info)` [*Function*]

`MMI_EVENT_TYPE mmi_event_event_type (MMI_EVENT event)` [*Macro*]

`MMI_PE mmi_event_pe (MMI_EVENT event)` [*Macro*]

`asINT32 mmi_event_info (MMI_EVENT event)` [*Macro*]

`event_type` [*Field Name*]

`pe` [*Field Name*]

`info` [*Field Name*]

The function `mmi_make_event` returns an `MMI_EVENT struct` with the given values as components. The three selector functions each return a component of such a `struct`. The field names may also be used to fetch from or assign to an `MMI_EVENT`.

`MMI_EVENT_TYPE` [*Type*]

The type `MMI_EVENT_TYPE` is an enumeration type, whose members are analogous to the keywords that can be returned from `mmi_run` in the Common Lisp interface. The values that can be returned are: `MMI_HALT_INSTRUCTION`, `MMI_HALT_FORCED`, `MMI_HALT_TIMESTEP_FINISHED`, `MMI_HALT_MACHINE_IDLE`, `MMI_HALT_STATISTIC_OVERFLOW`, `MMI_HALT_QUEUE_FULL`, `MMI_HALT_MACHINE_CHECK`, `MMI_INTR_INSTRUCTION`, `MMI_INTR_FORCED`, `MMI_INTR_INPUT_FIFO`, `MMI_INTR_OUTPUT_FIFO`, and `MMI_INTR_PARC_COUNTER`. They have the same meaning as in the Common Lisp interface, and imply the same interpretations of the other components of the `MMI_EVENT`.

`MMI_EVENT mmi_run ()` [*Function*]

`VOID mmi_force_halt ()` [*Function*]

`VOID mmi_force_interrupt ()` [*Function*]

The functions `mmi_run`, `mmi_force_halt`, and `mmi_force_interrupt` are exactly as in the Common Lisp interface, with the same restrictions and interrelationships.

Chapter 3

Proto-Memory Manager

There are several components of the complete Monsoon software system that compete for Monsoon's memory. For example, the loader needs an area in which to load code, while the run time system controls space used for activation frames and I-structures. This division of memory cannot be a permanent part of these programs, as it will change depending on the machine configuration (not to mention the software configuration!). This chapter defines a data structure and protocol, called the *Proto-Memory Manager* (PMM), that allows the initial set of Monsoon resources to be divvied up dynamically among the various software components that compete for them. Hence, the PMM serves not only to allocate machine resources to software components, but also is the chief means the software programs have for determining the machine configuration.

The simplest possible PMM would be a single pointer to the lowest unused memory location; then a program (loader, run time system, *etc.*) could obtain some memory by simply advancing this pointer by as many words as were needed. Unfortunately, the PMM for Monsoon is a bit more complex, due in part to the following considerations:

- There are multiple PE's, and hence multiple address spaces.
- Even within a single PE there may be several spaces (*e.g.*instruction memory and data memory). Moreover, different spaces may have different characteristics, and may not be uniformly distributed among PE's (*e.g.*, the spaces found on I-structure boards have different characteristics than those on PE boards).
- A logical area may be composed of many physical segments. For example, the I-structure area managed by the run time system is comprised of the memories on all I-structure boards.
- The segments composing a logical area may be interrelated. For example, the segments of instruction memory comprising the code area managed by the loader must have synchronized IP addresses.

The Proto-Memory Manager, therefore, is designed to manage multiple segments in multiple memory spaces.

It is important to understand that the Proto-Memory Manager is not truly a piece of software. More accurately, it is a data structure which describes the current status of all memory spaces under its control. A program such as the loader or run time system which desires a new memory segment obtains it by modifying the contents of this data structure.

Another view is that there are several proto-memory allocators, each a part of some other program such as the loader or run time system. They may run on Monsoon itself or on the host, but they share a single data structure. Note that the PMM is expected to be used only at "boot" or system initialization time, so that conflict between the users of the data structure is not an issue (at least in the present view of the total system). Also for this reason, the PMM data structure is designed for simplicity, rather than to optimize the performance of allocation algorithms.

Throughout this chapter, the term "PMM user" is used to denote a program such as the loader or run time system which obtains or releases resources through the PMM.

3.1 Spaces, Areas, and Regions

There are three entities described by the PMM data structure: *spaces*, *areas*, and *regions*. A space is a contiguous resource under management by the PMM. A region is a contiguous subdivision of a space, either allocated to a program or not yet allocated. An area is a collection of regions allocated to some program, which treats them as a logical unit. The PMM also associates an area with each space, containing all free regions in that space.

The maximum number of spaces, areas, and regions are each fixed at the time the PMM is initialized, and thereafter cannot be changed.

Rationale: This restriction allows the PMM data structure to be of fixed size. Note, however, that the maximum number of spaces, areas, and regions is not wired into any program. These numbers will likely vary with machine configuration, not to mention as experience is accumulated.

3.1.1 Spaces

A space is a contiguous region of a single kind of memory located on a single processor. Examples of a space include: the instruction memory of a PE, the portion of the data memory accessible through absolute-FP addressing, the memory on an I-structure board, *etc.* More generally, a space is any resource consisting of elements addressed by consecutive unsigned integers, such that all elements are accessed through the same mechanism (*e.g.*, all elements are accessed through `read-dm-` and `write-dm-` calls). While spaces cannot span processors or the address spaces within a processor, the PMM does not impose any restrictions on how address spaces within one processor are subdivided into PMM spaces; for example, the data memory of PE 3 could be described as a single PMM space, or many (disjoint) PMM spaces. (A potentially useful example of this would be to divide the data memory of each PE into two PMM spaces: absolute-FP addressable and not absolute-FP addressable.)

A space is characterized by a *type* and a *pe* value; each space described by the PMM has a unique *type/pe* pair. The *type* says what kind of memory or resource the space describes. Examples might include instruction memory, absolute-FP addressable data memory, host memory, *etc.* The *pe* says which of many such memories or resources the space describes. Typically, this is a processor number, hence the name "*pe*."

The *type/pe* pair determines how a space is accessed through the Monsoon Machine Interface (MMI). The *type* determines which set of procedures is to be used; *e.g.*, `read-dm-` and `write-dm-` *vs.* `read-im-` and `write-im-`. Most sets of MMI procedures require a *pe* argument, and the *pe* attribute of a space supplies the appropriate value. (The other argument required by MMI procedures is, of course, an address, and the description of a PMM space also includes

the bounds on the address for the space.) From the most general point of view, then, it is only by convention that *type* and *pe* are interpreted as memory type and processor ID: they are really just parameters which indicate how to access the space through the MMI.

The *type* attribute can convey more than just which MMI procedures are applicable. For example, processor data memory and I-structure board memory will probably both be accessed via `read-dm-` and `write-dm-` calls, but will carry different *type* attributes in the PMM so that the loader and run time system can distinguish them. In general, two spaces will be assigned the same *type* attribute if and only if they are interchangable from the point of view of all programs which use the PMM. The values allowed for *type* and their meanings, therefore, must be established by convention and agreed upon by all PMM users.

3.1.2 Regions

A region is simply a contiguously addressed segment drawn from a single space, with the following characteristics:

- Each location within the addressing limits of a space belongs to exactly one region at any given time.
- The locations in a given region are either entirely allocated to some PMM user, or entirely unallocated. That is, a region either belongs to some program, or is part of the free list of a space.

Allocating a free part of a space to a PMM user involves finding an appropriate free region for that space and either allocating all of it or splitting it into an allocated region and a new (smaller) free region. Notice that while the total number of locations managed by the PMM cannot change (because the number of spaces cannot), the total number of regions can as spaces become fragmented. The maximum number of regions, which is fixed when the PMM is initialized, must be chosen to anticipate the degree of PMM-level fragmentation expected.

3.1.3 Areas

Regions that have been allocated to a given program for a given purpose are grouped together into areas. An area is simply a set of regions; there are no restrictions on how many regions may belong to a given area, nor on whether the regions are part of the same space or different spaces. Each region, however, is always part of exactly one area. An example of an area is an area under the control of the run time system, consisting of regions drawn from the I-structure memory of all I-structure boards in the system. Each space in the PMM has an associated free area, consisting of all unallocated regions in the space.

When the PMM is initialized, the only areas which exist are the free areas for each space, and another area containing one region: the region containing the PMM data structure itself! As programs like the loader and run time system are initialized, they will establish other areas for their own purposes. The maximum number of areas, which is fixed when the PMM is initialized, must be chosen to anticipate the needs of all PMM users. Regions may be added to or removed from an area at any time after the area is created.

3.2 The PMM Data Structure

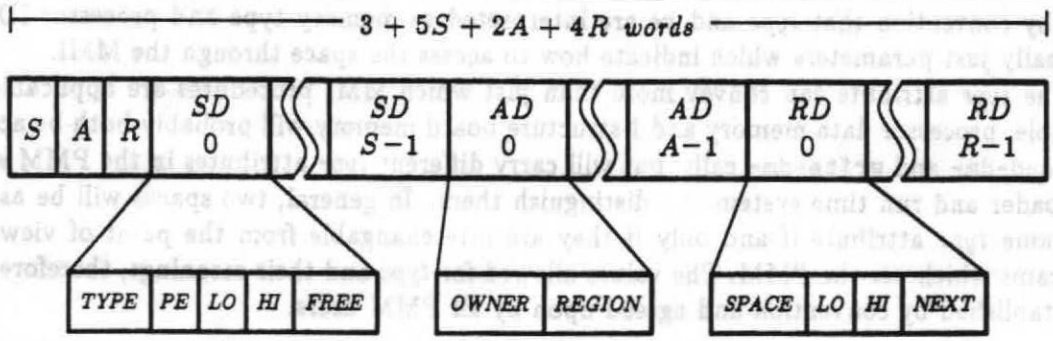

Figure 3.1: The PMM Data Structure

The PMM data structure describes all spaces, areas, and regions under control of the PMM. The maximum number of spaces, areas, and regions are each fixed at the time the PMM is initialized; call these S, A, and R, respectively. The PMM data structure consists of three words containing S, A, and R, followed by S space descriptors, A area descriptors, and R region descriptors (see Figure 3.1). Note that the number of spaces is a function of machine configuration and does not change once the system is brought up. PMM allocation and deallocation operations, however, may add or remove areas and/or regions from use. At any given time, therefore, some area and region descriptors actually describe areas and regions, while others will be unused. An allocation operation which attempts to use more areas or regions than the maximum values A or R will fail. Space, area, and region descriptors are separately numbered beginning with zero.

A space descriptor consists of five words:

type Says what kind of memory or resource this space describes. The values allowed for *type* and their meanings must be agreed upon by all PMM users; the current values are tabulated in Section 3.5.

pe Says which of many spaces of similar type this space describes.

low The smallest legal address for this space.

high The largest legal address for this space.

free The descriptor number of the free area for this space. (There is a convention that dictates the free area number for each space; see below.)

An area descriptor consists of two words:

owner This word contains zero for an unused area descriptor, in which case the remaining word in the descriptor is random. Otherwise, it is some nonzero value indicating to what PMM user the area belongs. The nonzero values of *owner* and their meanings are agreed upon by all PMM users; the current values are tabulated in Section 3.5. The main use of nonzero *owner* values is for debugging.

region The regions comprising an area are indicated by a linked list of region descriptors; *region* is the descriptor number of the first region in the list. If the area has no regions, *region* contains the value −1.

A region descriptor consists of four words:

space Contains the descriptor number of the space of which this region is a part, or -1 if this is an unused region descriptor. In the latter case, the remaining words in the descriptor are random.

low The smallest address in the region.

high The largest address in the region.

next The descriptor number of the next region belonging to the same area, or -1 if there are no more regions.

The following constraints on the PMM data structure must be observed:

- It must be located in a contiguous region of a single space (it occupies $3 + 5S + 2A + 3R$ words).
- It must be located in a space accessible by all PMM users.
- All words of the PMM data structure are integers. The *region* words of area descriptors and the *next* words of region descriptors are 2's complement signed integers, while all other words are unsigned. (Note that *low* and *high* values are integers, not Monsoon pointer (tag) values.)
- It must be located in a space whose word size accomodates:
  - Unsigned integers in the range $minlow \leq i \leq maxhigh$, where *minlow* is the smallest *low* value over all spaces and *maxhigh* is the largest *high* value.
  - Signed integers in the range $-1 \leq i < max(S, A, R)$.

  It is *not* required, however, that the space accomodate the range $-1 \leq i \leq maxhigh$.

Rationale: This allows the PMM data structure to be kept in n-bit memory even when *maxlow* is zero and *maxhigh* is $2^n - 1$. It is for the same reason that the *high* words of space and region descriptors are inclusive bounds, even though exclusive bounds would be more consistent with Common Lisp and C programming practice as well as with other interfaces defined in this document.

The areas used by the PMM as free areas for each space must obey the following restrictions:

- The regions in a free area must all be from the same space (obviously).
- The regions must be linked in ascending address order.
- The regions must be maximal; *i.e.*, there are no abutting regions.

These restrictions are imposed to simplify the allocation process for PMM users. There are no similar restrictions on the regions comprising areas other than PMM free areas; however, each PMM user is free to impose his own conventions for the areas under his control.

By convention, the area numbers for the spaces' free areas are chosen as follows. If there are a total of S spaces and A areas, then the free area for space s is $A - S + s$. In other words, the free areas are the S highest numbered areas, appearing in the same order as the spaces. The reason

for this convention is that certain areas participating in the bootstrap process need to have area numbers established by convention, independent of PMM size. Hence, these areas need to be assigned the lowest area numbers. See Figure 5.1 for a summary of areas established to support the Id programming environment, and Section 5.9 for details of the bootstrap procedure.

PMM users need to know the address of the PMM. By convention, the PMM is kept in the data memory of PE zero, and word zero of that data memory contains an unsigned integer (not a Monsoon pointer) that is the address of the first word of the PMM data structure. The words of the PMM itself will have presence bits **read-only** and type bits **unsigned** (see Section 5.10 for the actual numeric values). The memory occupied by the PMM itself must be accounted for in the PMM data structure. By convention, area zero is the area containing the PMM data structure, and only the PMM data structure.

Rationale: The data memory of PE zero is accessible both to Monsoon programs and to programs running on the front end, and has sufficiently wide words. Having word zero contain the address of the PMM rather than being the first word of the PMM itself allows the PMM to be placed in high memory without fixing how much physical memory PE zero must contain. High memory is desirable because low memory is a more precious resource, owing to the limitations of absolute-FP addressing.

3.3 PMM Operations

Conceptually, the PMM provides three kinds of operations: requests to determine machine configuration (by examining space descriptors), requests to create areas and allocate regions to them, and requests to return regions to free areas (there are no plans to use the latter at present). But because the PMM is a data structure, not a program, there are no procedures defined as part of the PMM interface. Instead, each PMM user may manipulate the PMM data structure as it pleases, subject to the constraints outlined above. As currently defined, the loader accepts a very general directive to allocate PMM areas; this allows a program that would otherwise be a PMM user to obtain its areas strictly through "configuration" files loaded by the loader. See Section 4.1.

Rationale: PMM allocation requests tend to be rather complicated, as PMM users may need regions with synchronized origins, or may have complicated formulas for deciding how much space to allocate based on configuration, *etc.* As experience is accumulated, we may define a standard procedural interface to the PMM data structure to support the kinds of requests we need. The need for this sort of standard will increase as we move toward a system where the PMM plays more than just a "boot time" role.

3.4 Multiple PMMs

The PMM data structure is capable of supporting a hierarchy of PMM's. That is, one could allocate a large area containing regions drawn from many spaces, and then consider those regions as the spaces of a subordinate PMM. Each PMM user would take as an argument the address of a PMM; it would be oblivious as to whether it is the master PMM or a subordinate one. Currently, there are no plans to exploit this feature—perhaps in the future it will be used to partition the machine or support multiple users.

3.5 Numeric Values of Memory Types and Owners

The following tabulates all of the values that currently may be found in the *type* field of space descriptors (the *name* column is for documentation purposes only).

Name	*Value*	*Description*
`pe-dm-short-r`	2	Those locations in PE Board Data Memory accessible through the "short-r" absolute addressing mode ($0 \leq addr < 2^{10}$).
`pe-dm-long-r`	4	Those locations in PE Board Data Memory accessible through the "long-r" absolute addressing mode but not through the "short-r" mode ($2^{10} \leq addr < 2^{20}$).
`pe-dm-no-r`	8	Those locations in PE Board Data Memory not accessible through any absolute addressing mode ($addr \geq 2^{20}$).
`is-dm`	16	I-Structure Board Data Memory.
`mmi-dm`	32	Data memory simulated by and accessible through the MMI, but not accessible from Monsoon itself (normally used only for the `loader-internals` area, as described in Section 4.7.2).
`pe-im`	3	PE Board Instruction Memory.

These values have been chosen to accomodate the area definition commands in MOC (see Section 4.1). The MOC commands allow the user to request an area allocated from any of a number of spaces, by adding together one or more *type* values given above. In the PMM itself, however, only the values given above will be found in the *type* field of space descriptors.

The values that may be found in the *owner* field of area descriptors include the following:

Value	*Description*
0	An unused area descriptor.
1	The free area for a space.
2	The area containing the PMM itself.
3	An internal area of the loader (*not* a "loader area").
4	A loader area (Section 4.1).
5	An area controlled by the Id Run Time System.

Other values of the *owner* field are available for future use.

Chapter 4

Monsoon Object Code Format

This chapter describes the Monsoon Object Code format (MOC). The object code format has some knowledge of Monsoon hardware—limited to number of bits per word, the fields in tags, instructions, *etc.* The loader has zero knowledge of Id and runtime environment; hence, it supports any kind of Monsoon programming. It is able to encode arbitrary memory images: code, data, *etc.* The loader must support dynamic linking across modules so that the programmer may have interactive compile-test-debug sessions without long waits for linking and loading.

The largest unit in MOC is an object module, the unit normally generated by an invocation of the compiler. A module will often be an implicitly delimited entity, consisting of all the records written to a file or stream. A module consists of a *version object* followed by a sequence of *records*; records either give data to be loaded into Monsoon memory (*data records*) or are directives to the loader (*command records*).

There are four kinds of data records; these are by far the most common kind of record. *Absolute*, *local*, and *global* data records simply give the contents of a contiguous region of memory. They differ in how they specify the address where the data is to be loaded: an absolute record gives a physical address, while local and global records allow the loader to choose the address, within limits. The address chosen by the loader for a local or global record is made available to other records via a *name*; any word of a data record may have a reference to a name, which the loader replaces by the address of the corresponding record. A variety of *reference methods* are provided so that such addresses may be inserted into any field of a referencing word. The difference between local and global records is that the associated name is valid across all modules for a global record but only within the defining module for a local record.

The remaining kind of data record is the *table entry* record. This is similar to an ordinary data records, but its data becomes part of a data structure called a *table*. Tables allow information generated at different times, and possibly appearing in different MOC object modules, to be collected into a single structure available to programs other than the loader. A typical example would be a table summarizing all Id procedures currently loaded, for use by the Execution Manager.

A typical object code file is composed primarily of local data records, with a smattering of global data records and table entry records for linkage between procedures. Absolute data records tend only to be used for system initialization purposes.

There are six kinds of command records. A `define-table` record causes the loader to establish a new table for the use of subsequent table entry records. A `define-area` record

provides a simple interface to the Proto-Memory Manager (Chapter 3) by creating an area with certain parameters and making its number available to other records. Areas created by define-area records are to be managed by other programs. Two specialized versions of this, `define-broadcast-area` and `define-interleaved-area` records, create areas which are managed by the loader. Every data record must indicate an area created by one of these two commands; this is how data records are directed to program or data memory, for example. Broadcast and interleaved loader areas differ in how the loader treats them when they encompass multiple PE's. All of these command records are typically found only in system initialization files.

The remaining two command records, **provide** and **require** records, implement a simple mechanism for ensuring compatability between modules loaded at different times.

Exsym Tables: In a departure from the TTDA Id World, Exsym Tables in the Monsoon system will be kept in separate compiler information files. Including exsym tables in MOC modules would compromise the modularity of the Big Picture—it would mean having two-way communication between the loader and the compiler, and it means keeping to the Lisp machine model of one big address space for all modules and processes. At least for a while, we will have to think in the style of standard operating systems, where each program has its own memory. This also keeps us from having to encode exsyms in MOC until someone writes Id Compiler Version 3 in Id.

Note, however, that we *do* expect that debugging information will be encoded in MOC. More accurately, we expect that debugging information will be included in the records of object modules; it will become part of instruction memory (or data memory) just as the executable part of object code will. MOC, therefore, has no special knowledge of debugging information—it looks like any other data record.

4.1 Loader Areas

The Proto-Memory Manager (PMM) data structure (Chapter 3) partitions the resources of Monsoon into *areas*, each of which is a set of resources under the management of some program. The loader provides an interface to the PMM through the **define-area** record, which simply creates an area for use by some other program. But the loader itself manages some areas; these are called *loader areas* and are created by **define-broadcast-area** and **define-interleaved-area** records. Local and global MOC data records are always loaded into memory contained in a loader area; each such record gives the name of a loader area, whereupon the loader is free to choose an address for the record within that area.

Each of the three area defining records specifies a name for that area. The name may appear as a symbolic reference within a data record, in which case the name stands for the PMM area number of the area. Additionally, the names of loader areas are found in the headers of local and global data records, indicating in what area the data records are to be loaded.

In the PMM, an area consists of a set of regions, each of which is a contiguously addressed portion of a resource with some PE number. When an area is created by a MOC command an ordering on these regions is established for use when the area is referenced by name. Specifically, there is a reference method which calls for "the PE number of the ith region of the area named a." In this method, "ith" is with respect to the region ordering established by the loader. Each of the area-creating commands discussed below defines what that region ordering is. Furthermore, the loader will link the PMM region descriptors according to the ordering when it creates the area.

4.1.1 Non-Loader Areas

`define-area` *area-name mtype owner n-pes start-pe end-pe size start end alignment syncp* [*MOC Record*]

A `define-area` record creates a new area in the PMM, and makes its number available to other records via the name *area-name*. The area will consist of *n-pes* regions drawn from spaces of type *mtype* on consecutively numbered PE's, one region for each PE number. Each region will be *size* words long, and if *syncp* is true all regions will have the same starting (and therefore ending) address. *Start-pe* (inclusive) and *end-pe* (exclusive) constrain the range of PE numbers from which the regions may be drawn; either constraint may be omitted by supplying -1 in its place. Similarly, *start* (inclusive) and *end* (exclusive) constrain the range of addresses each region will occupy, and may also be omitted by supplying -1. It is an error if either $end\text{-}pe - start\text{-}pe < n\text{-}pes$ or $end - start < size$, when those constraints are supplied. The starting addresses for each region will be a multiple of *alignment* (a value of 1 may be supplied if alignment is not important). The ordering on the regions for the purpose of reference methods is consecutive, smallest PE first.

The argument *mtype* may take on more values than are tabulated in Section 3.5. If *mtype* is even, it is taken to be the sum of one or more of the five data memory values `pe-dm-short-r`, `pe-dm-long-r`, `pe-dm-no-r`, `is-dm`, and `mmi-dm`. If *mtype* is odd, it must be the instruction memory value `pe-im` (although in the future, there might be more instruction memory types, and an odd *mtype* would be taken as the sum of one or more of them). If *mtype* is the sum of more than one value, it means that the area may be allocated from spaces of any or all of those types. For example, an *mtype* value of 14 (the sum of `pe-dm-short-r`, `pe-dm-long-r`, and `pe-dm-no-r`) indicates that any PE data memory may be allocated.

Despite the abundance of options, please note that `define-area` is not the most general interface to the PMM possible. Specifically, an area created by `define-area` can only have one region per PE, all regions must be of the same size and type, and the PE numbers must be consecutive. None of these restrictions are imposed by the PMM itself.

Warning: Due to the foregoing, `define-area` is subject to change in the future. The intent is to provide enough functionality that things like the Id Run Time System can be initialized by loading a MOC file. More experience is needed to see whether `define-area` together with other MOC features is truly adequate for this purpose.

4.1.2 Loader Areas

There are two kinds of loader managed areas: *broadcast* and *interleaved*. When a data record is loaded into a broadcast area, copies are loaded on all PE's in the area, at like addresses. When a data record is loaded into an interleaved area, consecutive words are directed to consecutive PE's, following the same interleaving technique as supported by the Monsoon hardware. Typically, objects such as user code and symbol information are stored in broadcast areas, while constant data structures may be stored in interleaved areas.

Because the loader must manage the free space in loader areas, the loader may reserve a few words out of each loader area to hold free pointers and similar information. Thus, if a loader area of 1000 words is created, a smaller number (*e.g.*, 998 words) may actually be available to hold data records. Unpredictable results may occur if an `absolute` record is loaded into a loader area, as the loader will not recognize that free space has been consumed.

Broadcast Areas

A broadcast area must satisfy two criteria:

- The area must consist of regions of identical size and type.
- Each region must start (and therefore end) at the same address.

These constraints imply that every region of a broadcast area has a unique PE number. As discussed earlier, there must be an ordering on these regions for use by certain reference methods.

When a local or global record is loaded into a broadcast area, it is loaded into each region of the area at the same address. The name of such a record maps to the following values:

Area(*name*) The PMM area number of the area into which the record is loaded.

Addr(*name*) The absolute address of the first word of the record in each region. (Note that the address is the same for all regions.)

A special case of broadcast area is called an *aliased* broadcast area; in addition to the above criteria an aliased broadcast area must satisfy the following:

- The area must consist of 2^K equal size regions of type data memory, where K is a non-negative integer.
- The PE numbers of the regions must be consecutive, and must start with a multiple of 2^K (*i.e.*, $i2^K \leq \mathrm{PE} < (i+1)2^K$ for some i).

The names of records loaded into aliased broadcast areas also map to the following values, in addition to those described above:

Map(*name*) The value $K + 32$, where there are 2^K PE's in the area.

PE(*name*) The value $i2^K$, that is, the number of the lowest numbered PE in the area.

FP(*name*) The same as *Addr*(*name*).

Note that the triple $\langle Map(Name), PE(name), FP(name)\rangle$ is a Monsoon pointer to the first word in the record, using the aliased-FP mapping strategy.

`define-broadcast-area` *area-name mtype n-pes start-pe end-pe size start end* [*MOC Record*] *alignment aliasp*

A `define-broadcast-area` record establishes a broadcast area, into which subsequent MOC data records may be loaded. The parameters have exactly the same meaning as they do for the `define-area` record. Notice, though, that there is no *syncp* parameter, as broadcast areas are required to be synchronized. The ordering on the regions for the purpose of reference methods is consecutive, smallest PE first.

If *aliasp* is true, then the following restrictions also apply:

- *N-pes* must be a power of two (2^K).
- *Start-pe* and *end-pe* (when specified) must span a range that includes *n-pes* PE's starting with a multiple of 2^K.
- *Mtype* must be a data memory type.

In this case, the PE's from which the regions are drawn will have numbers beginning with a multiple of 2^K, so that the area will qualify as an aliased broadcast area.

The name *area-name* maps to values *Area*(*area-name*), *etc.*, as would a record loaded into the first location in the area. Thus, *area-name* may be used to refer to the beginning of the area.

Limitation: As with **define-area**, **define-broadcast-area** is somewhat more restrictive than is required by the criteria for broadcast areas. In particular, the area created by **define-broadcast-area** is always on consecutively numbered PE's. We may decide to enhance this in the future.

Note that other systems may request broadcast areas, but manage the areas themselves.

Interleaved Areas

Interleaved areas must satisfy the following criteria:

- The area must consist of 2^K equal size regions of type data memory, where K is a non-negative integer.
- Each region must start (and therefore end) at the same address.
- The PE numbers of the regions must be consecutive, and must start with a multiple of 2^K (*i.e.*, $i2^K \leq \text{PE} < (i+1)2^K$ for some i).

The ordering on regions for an interleaved area is always consecutive, smallest PE first.

When a local or global data record is loaded into an interleaved area, it is loaded at some address in the area, but consecutive words in the record are interleaved across the different regions of the area (and therefore different PE's). The interleaving follows the same scheme as that supported by the Monsoon hardware. That is, if word j of a record is loaded at address a on PE p, then the next word will be loaded at address a on PE $p+1$ unless PE $p+1$ is outside the range of PE's in the area, in which case the word will be loaded at address $a+1$ on PE $p+1-2^K$.

Let an interleaved area consist of 2^K PE's starting with PE p_0, where each region starts at address a_0, and let the first word of a record loaded into such an area be record be loaded at address a on PE p. Then the name of a record maps to the following values:

Area(*name*) The PMM area number of the area.

Addr(*name*) The value $2^K(a-a_0)+(p-p_0)$, that is, the logical displacement of the start of the record from the start of the area.

Map(*name*) The value K.

PE(*name*) The value p.

FP(*name*) The value a.

Note that the triple $\langle Map(Name), PE(name), FP(name)\rangle$ is a Monsoon pointer to the first word in the record, using the interleaved-FP mapping strategy. Note too that *Addr*(*name*) for an interleaved area is a *relative* value (relative to the start of the area), while for a broadcast area it is an *absolute* quantity.

Rationale: An absolute address does not make much sense for an interleaved area unless it has only one region, in which case *FP(name)* can be used to obtain the absolute address. For both interleaved and broadcast areas, however, two *Addr(name)* quantities from the same region can be subtracted to get a "distance between" value.

`define-interleaved-area` *area-name mtype n-pes start-pe end-pe size start end alignment* [*MOC Record*]

A `define-interleaved-area` establishes an interleaved area, into which subsequent MOC data records may be loaded. The parameters have exactly the same meaning as they do for the `define-area` record, except that:

- *N-pes* must be a power of two (2^K).
- *Start-pe* and *end-pe* (when specified) must span a range that includes *n-pes* PE's starting with a multiple of 2^K.
- *Mtype* must be a data memory type.

Notice, too, that there is no *syncp* parameter, as interleaved areas are required to be synchronized. As mentioned above, the ordering on the regions for the purpose of reference methods is consecutive, smallest PE first.

The name *area-name* maps to values *Area(area-name)*, *etc.*, as would a record loaded into the first location in the area. Thus, *area-name* may be used to refer to the beginning of the area.

Note that another system may request an interleaved area, but manage the area itself An example of this would be the heap manager asking for an interleaved area for the heap—it may load a single record to initialize the heap, but manage the heap afterwards.

4.2 Data Records

A data record consists of a *header*, *contents*, and *references*. The header of a record tells where to load the record and what name (if any) to associate with the address where the record is loaded. The contents of a record defines a contiguous block of instruction or data memory, possibly broadcast or interleaved among several PE's. The references say how to fix up the contents of the record when they refer to names defined by other records, either in the same module or in other modules.

4.2.1 Headers

The format of the contents and references part of a data record is the same for all kind of data records. In the description of data records below, it is understood that the contents and references follow the header parameters.

`absolute` *dm-or-im pe address* [*MOC Record*]

An `absolute` record says that the contents are to be loaded in consecutive locations beginning with *address* on PE *pe*. The *dm-or-im* parameter indicates data memory or instruction memory. The locations into which the record is loaded must not be part of a loader area; they normally

should be part of a PMM area that is not a loader area. Loading an absolute record into a loader area may confuse the loader's free space management.

`local` *dm-or-im record-name area-name* [*MOC Record*]

A `local` record says that the contents are to be loaded in some previously unused portion of memory contained in the loader area named *area-name*. If that area is a broadcast area, the contents are loaded on all PE's in the area, if an interleaved area, they are interleaved among them. Other records within the module may have references to *record-name*, which will map to the address of the first word of the record. It is an error for *record-name* to be the name of any other record in the current module, and for *record-name* to be the name of a global record in previously loaded modules. The *dm-or-im* parameter indicates data memory or instruction memory; the value of this parameter must be consistent with the type of memory from which the area named *area-name* was allocated.

Rationale: The *dm-or-im* parameter of the `local` record is redundant information, as the value of *area-name* will determine whether the contents of the record should be data memory contents or instruction memory contents. It is included anyway, because it is impossible to parse the contents portion of the record without knowing the type of memory. Including the *dm-or-im* parameter explicitly makes the MOC format somewhat more robust in the event that the wrong area name is specified.

These same comments apply not only to the `local` record, but also to the `global`, `aligned-local`, `aligned-global`, and `table-entry` records as well.

`global` *dm-or-im record-name area-name* [*MOC Record*]

A `global` record says that the contents are to be loaded in a portion of memory contained in the loader area named *area-name*. If that area is a broadcast area, the contents are loaded on all PE's in the area, if an interleaved area, they are interleaved among them. Other records within the current module and in other modules may have references to *record-name*, which will map to the address of the first word of the record. If no record bearing the name *record-name* has been encountered by the loader in any module since its initialization, a new portion of area *area-name* will be allocated. Otherwise, the contents will overwrite what already exists at the address to which *record-name* maps. In the latter case, it is an error for the number of words in the contents to be greater than the number of words in previously encountered records with the same name. The *dm-or-im* parameter indicates data memory or instruction memory; the value of this parameter must be consistent with the type of memory from which the area named *area-name* was allocated.

`aligned-local` *dm-or-im record-name area-name alignment* [*MOC Record*]
`aligned-global` *dm-or-im record-name area-name alignment* [*MOC Record*]

These records are like `local` and `global` records, but further specify that the starting address of the record relative to the beginning of the area must be a multiple of *alignment*. More precisely, if loaded into a broadcast area, the value *Addr*(*record-name*) − *Addr*(*area-name*) must be a multiple of *alignment*, while if loaded into an interleaved area, the value *Addr*(*record* − *name*) must be a multiple of *alignment*. Often, aligned records will be loaded into areas which are themselves aligned, so that the *absolute* addresses of the records will be aligned.

`table-entry` *dm-or-im record-name table-name* [*MOC Record*]

The `table-entry` record is similar to the `local` record, except that its contents are loaded into a table data structure rather than in an arbitrary location in memory. See Section 4.3 for details on how the loading of table-entry records differs from ordinary data records. The mapping of the *record-names*, however, is exactly analogous to ordinary data records: *record-name* maps to the address where the first word of the contents is loaded. The scope of the name is like a `local` record: it is defined only for the current module (why this is not a limitation is discussed in Section 4.3). The *dm-or-im* parameter indicates data memory or instruction memory; the value of this parameter must be consistent with the type of memory containing the table named *table-name*.

There are two key differences between local and global data records:

1. Global names are defined across module boundaries and loader invocations, while local names are defined only within a module.
2. Global records may overwrite a previously loaded record, while local records are always loaded into freshly allocated memory.

A typical use of a global record would be to hold a fixed-size structure describing a top-level user procedure, with a record name derived from the name of the procedure. Each time a module is loaded containing a new definition of the procedure, a new descriptor record replaces the old one in Monsoon memory, thus accomplishing dynamic linking. Note that a global record which replaces another must be of the same size or smaller, implying that global records are unsuitable for holding the actual code of a user procedure (which may change in size). The "meat" of an object module—code for user procedures, user data structures, and the like—is usually found in local records, with global records only containing fixed-size objects with references to the local records. Table-entry records (discussed in Section 4.3) will often have pointers to global records, as loader tables are typically used to aggregate the objects found in global records into a single summary data structure.

4.2.2 Contents

The *contents* part of a record says what to load. The general format is the number of words, followed by defaults for type bits, presence bits, and the like, followed by the actual words themselves, in ascending address order. For the purposes of identifying a word within a record, the first word is called word zero and the last word $N - 1$, where N is the number of words in the record.

The actual format of data words depends on whether the record is to be loaded into data memory or instruction memory. For data memory, each word consists of presence bits, type bits, data bits, and a *holep* bit (described below). For instruction memory, each word just has data bits and a *holep* bit. Because it is often the case that all words in a record will have similar presence, type and *holep* bits, each data word may appear in a short format which takes these values from a default which immediately follows the number of words. Again, the format of the default depends on whether the record is to be loaded into data memory or instruction memory. The data bits themselves may appear in any of several different encodings, corresponding to the different encodings found in the MMI, on a word-by-word basis. *The format of data words (DM or IM) must agree with the memory type of the area into which the record is loaded.*

The *holep* bit, when true, suppresses the loading of the corresponding presence, type, and data bits under certain circumstances. For an `absolute` data record, a true *holep* always

suppresses the word. For `global` and `table-entry` data records, *holep* suppresses the word only when the record is replacing an earlier record, as opposed to when the record is loaded into newly-allocated memory. For a `local` data record, all *holep* bits are ignored. By "suppressing a word" it is meant that the presence, type, and data bits in the record are ignored, and the corresponding word in Monsoon memory is left unchanged. The *holep* feature is primiarly useful with `global` records, where it allows initialization of words of an object which should not subsequently change when the object is replaced.

The detailed format of the contents part may be found in Section 4.5.

4.2.3 References

The *references* part of a record says how to adjust certain words of the record, based on the values of names of this or other records. The references part is just a sequence of references, each of which tells how to fix up one word of the record contents. There may be several references fixing up the same word, in which case they are applied in the order in which they appear.

A *reference* consists of an *offset* and a *reference expression*. The offset says which word of the record to adjust (zero being the first word) while the reference expression says how to compute the value to be stored there. The expression may specify the adjustment of the entire word or just a subfield, but in any event a reference only acts on data bits, never on presence or type bits.

If *offset* refers to a word for which loading was suppressed by the *holep* mechanism, the reference is ignored.

The reference expression is in postfix notation. Operationally, one can think of a stack of values, with the reference expression being a sequence of operators which push and pop values from the stack. The stack is initially empty. The last operator in any reference expression is a command to store the value or values remaining on the stack into some field or fields of the word being adjusted. Such operators also implicitly delimit the end of a reference, hence each reference expression performs exactly one storing operation. All calculations in reference methods are performed using arbitrary-precision integer arithmetic.

`number` *n* [*Reference Operator*]

Pushes *n* onto the stack.

`area` *name* [*Reference Operator*]

Pushes *Area*(*name*) onto the stack.

`addr` *name* [*Reference Operator*]

Pushes *Addr*(*name*) onto the stack.

`map` *name* [*Reference Operator*]

Pushes *Map*(*name*) onto the stack. Valid only when *Map*(*name*) is defined, that is, when *name* is the name of a data record loaded into an an interleaved or aliased broadcast area, or the name of such an area itself.

`pe` *name* [*Reference Operator*]

Pushes *PE*(*name*) onto the stack. Valid only when *PE*(*name*) is defined, that is, when *name* is the name of a data record loaded into an an interleaved or aliased broadcast area, or the name of such an area itself.

`fp` *name* [*Reference Operator*]

Pushes $FP(name)$ onto the stack. Valid only when $FP(name)$ is defined, that is, when *name* is the name of a data record loaded into an an interleaved or aliased broadcast area, or the name of such an area itself.

`plus` [*Reference Operator*]
`minus` [*Reference Operator*]
`times` [*Reference Operator*]
`floor` [*Reference Operator*]
`ceiling` [*Reference Operator*]
`truncate` [*Reference Operator*]
`round` [*Reference Operator*]

The `plus` operator pops two values off the stack, v_2 followed by v_1, and then pushes the value $v_1 + v_2$ onto the stack. The other operators are analogous. For example, the following sequence:

```
number 5 number 3 minus
```

has the effect of pushing the number 2 onto the stack (*not* -2). The operators `floor`, `ceiling`, `truncate`, and `round` are all forms of integer division, differing in their treatment of non-integral results. `floor` rounds towards negative infinity, `ceiling` rounds towards positive infinity, `truncate` rounds towards zero, and `round` rounds to the nearest integer, or to the nearest even integer if the result is exactly between two integers.

`ptr` *name* [*Reference Operator*]

Pushes the three values $Map(name)$, $PE(name)$, and $FP(name)$ onto the stack, in that order. Valid only when these quantities are defined, that is, when *name* is the name of a data record loaded into an an interleaved or aliased broadcast area, or the name of such an area itself. Using the `ptr` operator is exactly equivalent to the sequence:

`map` *name* `pe` *name* `fp` *name*

but is provided for convenience. It is most often used in conjunction with the `ptrinc` and `tag-ptr` reference operators described below.

`ptrinc` [*Reference Operator*]

Pops the four values Δ, *fp*, *pe*, and *map*, in that order, from the stack, then pushes the values *map'*, *pe'*, and *fp'*, in that order, as defined by:

$$\langle map', pe', fp' \rangle = PointerInc(\langle map, pe, fp \rangle, \Delta)$$

where *PointerInc* is the Monsoon pointer increment operation. In other words, the sequence

`ptr` *name* `number` Δ `ptrinc`

has the effect of pushing a pointer which is at offset Δ from the pointer to which *name* maps, under the appropriate interleaving strategy.

`ithpe` [*Reference Operator*]

Pops the values i and A, in that order, from the stack, where A is the PMM area number of a loader area. It then pushes the PE number of the ith PE in the region ordering of area A,

interpreting i modulo the number of regions in the area. It is an error for A to be other than the PMM number of some loader area.

`thislpe` [*Reference Operator*]

Pushes the logical PE number of the word being adjusted onto the stack. The "logical PE number" is the PE's position within the region ordering of the area into which the current record is being loaded; it is an integer between zero inclusive and the number of regions in the area, exclusive. For an interleaved area, the number pushed onto the stack depends on where the word being adjusted ended up according to the interleaving. For a broadcast area, the number pushed is different for each copy of the record loaded. The latter case is the only situation in which a reference can cause a different value to be loaded for each copy of a record in a broadcast area. In general, if the `thislpe` operator occurs in a reference expression for a record loaded into a broadcast area, the reference expression will need to be recomputed for each copy.

`signed` *size position* [*Reference Operator*]

The `signed` operator pops the top value off the stack and stores it in a subfield of the word being adjusted. The value is stored as a two's complement signed integer, in a field *size* bits wide, and with the least significant bit *position* bits to the left of the least significant bit of the word. *Size* must be positive, *position* must be non-negative, and *size* + *position* must be no greater than the number of data bits in the word to be adjusted (which may be different between data memory and instruction memory). The value to be stored must fall in the range $2^{size-1} \leq x < 2^{size-1}$. This operator implicitly indicates the end of the reference expression.

`unsigned` *size position* [*Reference Operator*]

The `unsigned` operator is similar to the `signed` operator, except that the field is to be treated as a unsigned. The same interpretations of and restrictions on *size* and *position* apply, while the permissible range for the value is $0 \leq x < 2^{size}$. This operator implicitly indicates the end of the reference expression.

`tag-pe` [*Reference Operator*]
`tag-ip` [*Reference Operator*]
`tag-fp` [*Reference Operator*]
`tag-map` [*Reference Operator*]
`tag-port` [*Reference Operator*]

These may only be used with a record loaded into a data memory area. Each is equivalent to an `unsigned` operator, specifying the subfield of a data word corresponding to a component of a tag. All of these operators implicitly indicate the end of the reference expression.

`tag-ptr` [*Reference Operator*]

The `tag-ptr` operator pops three values *fp*, *pe*, and *map*, in that order, and stores them in the subfields of the word being adjusted corresponding to the appropriate components of a tag. This operator implicitly indicates the end of the reference expression.

`instruction-opcode` [*Reference Operator*]
`instruction-f1` [*Reference Operator*]

`instruction-f2` [*Reference Operator*]

These may only be used with a record loaded into an instruction memory area. Each is equivalent to an `unsigned` operator, specifying the subfield of a data word corresponding to a component of an instruction. All of these operators implicitly indicate the end of the reference expression.

`instruction-s1` [*Reference Operator*]
`instruction-s2` [*Reference Operator*]

These may only be used with a record loaded into an instruction memory area. Each is equivalent to an `signed` operator, specifying the subfield of a data word corresponding to the '*s*' subfield of a destination component of an instruction. All of these operators implicitly indicate the end of the reference expression.

`instruction-r` [*Reference Operator*]

Equivalent to the `instruction-f1` operator. This operator implicitly indicates the end of the reference expression.

`instruction-long-r` [*Reference Operator*]

Equivalent to an `unsigned` operator, specifying the subfield which is a combination of '*f1*' and '*f2*', used by instructions as a long '*r*' value. This operator implicitly indicates the end of the reference expression.

Non-Limitation: While all of the storing operators currently defined refer to contiguous bit fields, there is no reason in principle why this must be the case. Indeed, future changes to the format of machine data types may require some of the `tag-` or `instruction-` operators to refer to concatenations of several contiguous fields. Currently, however, none of them do, nor is there any general way of specifying an arbitrary concatenated field.

Rationale: A fairly restricted syntax and semantics of reference methods is used to give the loader maximum freedom in dealing with forward references. It is fairly easy for the loader to figure out which references expressions contain forward references, and which references adjust overlapping fields of the same word. If a more elaborate reference expression language were allowed, say one in which names were first-class objects or in which more than one word could be adjusted from within the same expression, the loader might have to store a great deal more information while forward references were pending.

4.3 Tables

A `define-table` record establishes a table data structure in some area, consisting of two-word pairs (called *indicator* and *value*). A name is given to the table, whose scope is across invocations (like a global data record), mapping to the first word in the entire table data structure. This name allows other programs to gain access to the entire table through the reference mechanism.

Initially a table has no pairs; subsequent `table-entry` records can add pairs (none are ever removed). A `table-entry` record always consists of exactly two words, which are entered as an indicator/value pair in the table. The pair replaces an existing table entry if the new indicator is bit-for-bit the same as an existing indicator, otherwise the pair is added to the table. Within

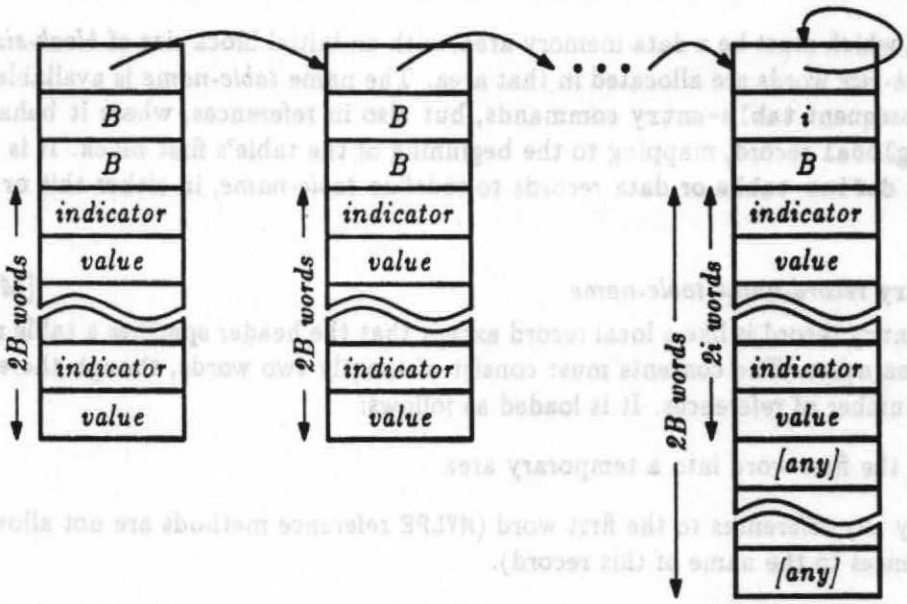

Figure 4.1: Structure of a Loader Table

a given table, therefore, indicators are unique. A `table-entry` record may have references on both content words; a special algorithm is used for applying them.

The structure of a table is a linked list of blocks of indicator/value pairs, as illustrated in Figure 4.1. Each block consists of $3 + 2B$ words, where B is the *block size* of that block, the maximum number of indicator/value pairs in that block. The first three words are:

1. A pointer to the next block, or a pointer to this block if it is the last. A pointer for this purpose is a tag whose PORT and IP fields are zero, and whose MAP, PE, and FP fields are as appropriate given whether the table is in an interleaved or aliased broadcast area. If in a non-aliased broadcast area, MAP is zero and PE is the first PE in the region ordering.

2. The number of indicator/value pairs in this block, as an integer. This is always equal to B unless this is the last block in the table.

3. The size of this block (B), as an integer.

[To be determined: what type and presence bits to use?]

Rationale: A list of blocks was used rather than a list of pairs for three reasons. One, it consumes less memory (unless the block size is terribly inappropriate). Two, it leads to less fragmentation. Three, it is a nicer representation if used as a fixed-size table. The convention for the end-of-list pointer is a bit weird, but it avoids having to choose some representation for `nil`—remember that MOC is supposed to be independent of language, and anyway the language's version of `nil` may not exist at the time the very first table is created!

`define-table` *table-name area-name block-size* [*MOC Record*]

A `define-table` command establishes a new table named *table-name*, suitable for use by subsequent `table-entry` records in this or other modules. The table is established in area

area-name, which must be a data memory area, with an initial block size of *block-size*; a total of $3+2\times$ *block-size* words are allocated in that area. The name *table-name* is available for use not only in subsequent **table-entry** commands, but also in references, where it behaves like the name of a **global** record, mapping to the beginning of the table's first block. It is an error for subsequent **define-table** or data records to redefine *table-name*, in either this or subsequent modules.

table-entry *record-name table-name* [*MOC Record*]

A **table-entry** record is like a local record except that the header specifies a table name rather than an area name. The contents must consist of exactly two words, though there may be an arbitrary number of references. It is loaded as follows:

1. Load the first word into a temporary area.
2. Apply any references to the first word (MYLPE reference methods are not allowed, nor are references to the name of this record).
3. If the first word (updated) matches, bit-for-bit, an indicator in the table, let the name of this record map to that pair. Otherwise, add a new entry using the updated first word as the indicator, and let the name of this record map to it. If the last block of the table was full, a new block of the same block size is allocated, in the same area, and linked in.
4. Load the second word into the value word of this entry, replacing the old value if the indicator matched an existing entry in Step 3.
5. Apply any references to the second word (no restrictions).

Note that the name of a table-entry maps to the address where the indicator ended up. While this name is scoped like a local record, this is not a limitation: if a subsequent module wants to refer to a previous **table-entry** record, it may include its own **table-entry** record with the same indicator, and a value that has the *holep* bit turned on.

4.4 Require and Provide Records

A MOC module is generated with certain assumptions about the run time system, microcode, and system configuration. **Require** records allow these assumptions to be made explicit—if some facility named by the **require** record has not been loaded, then the loader signals an error. To indicate that a particular facility has been loaded, a module includes a **provide** record, which names a facility along with version numbers. A module may include as many **provide** and **require** records as it wishes.

The facilities **required** by a module are not necessarily loaded through MOC modules, although they should have associated MOC modules that do the **provide**. An example of this is microcode, for which no provision has been made in MOC.

When the loader signals an error due to an unsatisfied **require**, it will normally suspend itself and allow the user to take corrective action. For example, the user may at that point load whatever module is required and then restart the loader.

The require/provide mechanism was inspired by a similar facility in Common Lisp.

provide *facility-name major-version minor-version* [*MOC Record*]

A **provide** record specifies that any **require** of facility *facility-name* and version *major-version.minor-version* will be satisfied. If the current module is compatible with more than one version of the facility, a **provide** record should be loaded for each supported version.

require *facility-name major-version minor-version* [*MOC Record*]

This record specifies that version *major-version.minor-version* of facility *facility-name* is required in order to use the current module.

Facility-name is a name denoting some sort of software facility or subsystem, and the two version numbers are integers. The **provide/require** mechanism supports a major and minor version number, but in fact it is up to each facility to decide how to interpret the two numbers. As far as MOC is concerned, the two version numbers are just values to be matched for equality, with no further ordering or interpretation.

4.5 Record Encodings

A MOC file (or stream) is merely an encoding of a MOC module. It contains a version object followed by a series of MOC records. The MOC format is built on top of CIOBL, Version 2, described in Chapter 7. MOC format requires only a Level I implementation of CIOBL, and makes heavy use of fixed-length objects in order to minimize the space required by MOC files.

The grammar gives the complete encoding of MOC into CIOBL objects. The only CIOBL objects found in MOC files are Unsigneds, Signeds, Doubles, Integers, and Keywords, which appear below as *n*`bit-unsigned`, *n*`bit-signed`, `double`, `integer` and `keyword`, respectively. The notation `<absolute>` indicates a 6-bit unsigned; a table giving the numerical values of these words follows the grammar.

Rationale: Things like `<absolute>` are used as punctuation to indicate record types and the like. 6-bit unsigneds are used instead of keywords because they take only one byte in both the Binary and Compressed encodings, while keywords take at least two. Signeds and Unsigneds are used for most data because there is no reason to expect that a significant fraction of them will require fewer bits than their maximum, implying variable-length integers would take more space. The sole exception is the argument of the number reference operator, which is a variable-length integer.

```
MODULE          →  VERSION-OBJECT {RECORD}*
VERSION-OBJECT  →  keyword 8bit-unsigned 8bit-unsigned
RECORD          →  DATA-RECORD | COMMAND-RECORD
```

All data records have a common syntax for their CONTENTS and REFERENCES parts. The `1bit-unsigned` in the definition of the headers is zero for a data memory record, one for instruction memory.

```
DATA-RECORD     →  DATA-HEADER CONTENTS REFERENCES
DATA-HEADER     →  ABSOLUTE-HEADER | LOCAL-HEADER |
                   GLOBAL-HEADER | TABLE-ENTRY-HEADER |
                   ALIGNED-LOCAL-HEADER |
                   ALIGNED-GLOBAL-HEADER
```

```
ABSOLUTE-HEADER          → <absolute> 1bit-unsigned PE ADDRESS
LOCAL-HEADER             → <local> 1bit-unsigned NAME NAME
GLOBAL-HEADER            → <global> 1bit-unsigned NAME NAME
ALIGNED-LOCAL-HEADER     → <aligned-local> 1bit-unsigned NAME NAME ADDRESS
ALIGNED-GLOBAL-HEADER    → <aligned-global> 1bit-unsigned NAME NAME ADDRESS
TABLE-ENTRY-HEADER       → <table-entry> 1bit-unsigned NAME NAME
```

The CONTENTS part varies depending on whether the record is intended for a data memory area or an instruction memory area, which is always indicated as part of the header. The 12-bit unsigned PRESENCE-TYPE-HOLEP contains type bits in the least eight significant bits, presence bits in the next most significant three bits, and *holep* in the most significant bit. The 1-bit unsigned HOLEP contains only the *holep* bit. A *holep* value of one indicates that *holep* processing should take place (that is, it indicates the presence of a "hole"), zero that it should not. The order of the last five fields in the DM-TAG and DM-SHORT-TAG word specifiers is PORT, MAP, IP, PE, and FP. The order of the last three fields in the IM-INSTRUCTION and IM-SHORT-INSTRUCTION word specifiers is OPCODE, F1, and F2.

```
CONTENTS                 → DM-CONTENTS | IM-CONTENTS
DM-CONTENTS              → 32bit-unsigned PRESENCE-TYPE-HOLEP {DM-WORD}*
IM-CONTENTS              → 32bit-unsigned HOLEP {IM-WORD}*

DM-WORD                  → DM-BITS | DM-INTEGER | DM-FLOAT | DM-TAG |
                           DM-SHORT-BITS | DM-SHORT-INTEGER |
                           DM-SHORT-FLOAT | DM-SHORT-TAG
DM-BITS                  → <dm-bits> PRESENCE-TYPE-HOLEP 64bit-unsigned
DM-INTEGER               → <dm-integer> PRESENCE-TYPE-HOLEP 64bit-signed
DM-FLOAT                 → <dm-float> PRESENCE-TYPE-HOLEP double
DM-TAG                   → <dm-tag> PRESENCE-TYPE-HOLEP 1bit-unsigned
                           8bit-unsigned 23bit-unsigned 10bit-unsigned
                           22bit-unsigned
DM-SHORT-BITS            → <dm-short-bits> 64bit-unsigned
DM-SHORT-INTEGER         → <dm-short-integer> 64bit-signed
DM-SHORT-FLOAT           → <dm-short-float> 64bit-double
DM-SHORT-TAG             → <dm-short-tag> 1bit-unsigned 8bit-unsigned
                           23bit-unsigned 10bit-unsigned 22bit-unsigned
PRESENCE-TYPE-HOLEP      → 12bit-unsigned

IM-WORD                  → IM-BITS | IM-INSTRUCTION |
                           IM-SHORT-BITS | IM-SHORT-INSTRUCTION
IM-BITS                  → <im-bits> HOLEP 32bit-unsigned
IM-INSTRUCTION           → <im-instruction> HOLEP 12bit-unsigned
                           10bit-unsigned 10bit-unsigned
IM-SHORT-BITS            → <im-short-bits> 32bit-unsigned
IM-SHORT-INSTRUCTION     → <im-short-instruction> 12bit-unsigned
                           10bit-unsigned 10bit-unsigned
HOLEP                    → 1bit-unsigned
```

The references part starts with a number indicating how many references there are. The end of a reference expression is implicitly delimited by the last reference operator. The 6-bit

Unsigned which follows the operators **signed** and **unsigned**, indicating the size of a field to be stored, is actually encoded as one less than the size. For example, if the field is to be five bits wide, the number four appears in the file.

```
REFERENCES                  →  32bit-unsigned {REFERENCE}*
REFERENCE                   →  32bit-unsigned REFERENCE-EXPRESSION
REFERENCE-EXPRESSION        →  {REFERENCE-OPERATOR}* TERMINATING-REFERENCE-OPERATOR
REFERENCE-OPERATOR          →  <number> integer | <area> NAME | <addr> NAME |
                               <map> NAME | <pe> NAME | <fp> NAME |
                               <plus> | <minus> | <times> |
                               <floor> | <ceiling> | <truncate> | <round> |
                               <ptr> NAME | <ptrinc> | <ithpe> | <thislpe>
TERMINATING-REFERENCE-OPERATOR  →
                               <signed> 6bit-unsigned 6bit-unsigned |
                               <unsigned> 6bit-unsigned 6bit-unsigned |
                               <tag-pe> | <tag-ip> | <tag-fp> |
                               <tag-map> | <tag-port> | <tag-ptr> |
                               <instruction-op> | <instruction-f1> | |
                               <instruction-f2> |
                               <instruction-s1> | <instruction-s2> |
                               <instruction-r> | <instruction-long-r>
```

Command records are straightforward.

```
COMMAND-RECORD              →  DEFINE-TABLE-RECORD | DEFINE-AREA-RECORD |
                               DEFINE-BROADCAST-AREA-RECORD |
                               DEFINE-INTERLEAVED-AREA-RECORD |
                               PROVIDE-RECORD | REQUIRE-RECORD
DEFINE-TABLE-RECORD         →  <define-table> NAME NAME 16bit-unsigned
DEFINE-AREA-RECORD          →  <define-area> NAME MTYPE 32bit-unsigned N-PE
                               OPT-PE OPT-PE ADDRESS OPT-ADDRESS OPT-ADDRESS
                               ADDRESS BOOL
DEFINE-BROADCAST-AREA-RECORD  →  <define-broadcast-area> NAME MTYPE
                               N-PE OPT-PE OPT-PE ADDRESS OPT-ADDRESS
                               OPT-ADDRESS ADDRESS BOOL
DEFINE-INTERLEAVED-AREA-RECORD  →  <define-interleaved-area> NAME MTYPE
                               N-PE OPT-PE OPT-PE ADDRESS OPT-ADDRESS
                               OPT-ADDRESS ADDRESS
PROVIDE-RECORD              →  <provide> NAME 8bit-unsigned 8bit-unsigned
REQUIRE-RECORD              →  <require> NAME 8bit-unsigned 8bit-unsigned
```

NAMEs are encoded simply as CIOBL keywords. The values for MTYPE are defined by the Proto-Memory Manager (Chapter 3). The value of an N-PE, which indicates how many PE's an area should span, is encoded as one less than the number of PE's desired. For example, if an area is to be defined across four PE's, the number three actually appears in the file. The values OPT-PE and OPT-ADDRESS are used for arguments to various area definition commands that expect either a PE (or address), or -1 to indicate an unspecified value. OPT-PE and OPT-ADDRESS are signed to accomodate the -1 value.

Punctuation	*Value*	*Punctuation*	*Value*
<absolute>	1	<ceiling>	29
<local>	2	<truncate>	30
<global>	3	<round>	31
<aligned-local>	4	<ptr>	32
<aligned-global>	5	<ptrinc>	33
<table-entry>	6	<ithpe>	34
<dm-bits>	7	<thislpe>	35
<dm-integer>	8	<signed>	36
<dm-float>	9	<unsigned>	37
<dm-tag>	10	<tag-pe>	38
<dm-short-bits>	11	<tag-ip>	39
<dm-short-integer>	12	<tag-fp>	40
<dm-short-float>	13	<tag-map>	41
<dm-short-tag>	14	<tag-port>	42
<im-bits>	15	<tag-ptr>	43
<im-instruction>	16	<instruction-op>	44
<im-short-bits>	17	<instruction-f1>	45
<im-short-instruction>	18	<instruction-f2>	46
<number>	19	<instruction-s1>	47
<area>	20	<instruction-s2>	48
<addr>	21	<instruction-r>	49
<map>	22	<instruction-long-r>	50
<pe>	23	<define-table>	51
<fp>	24	<define-area>	52
<plus>	25	<define-broadcast-area>	53
<minus>	26	<define-interleaved-area>	54
<times>	27	<provide>	55
<floor>	28	<require>	56

Figure 4.2: Numeric Values of MOC Punctuation

```
NAME        → keyword
PE          → 10bit-unsigned
N-PE        → 10bit-unsigned
OPT-PE      → 11bit-signed
ADDRESS     → 31bit-unsigned
OPT-ADDRESS → 32bit-signed
MTYPE       → 16bit-unsigned
```

The numeric values of punctuation items such as <absolute> are given in Figure 4.2.

4.6 Restricted Mode

During initialization of the Monsoon software system, it is necessary to load certain data into data and instruction memory before the PMM and the loader's symbol tables have been ini-

tialized (see Section 5.9 for details of the bootstrap process). To support this need, the loader has a *restricted* mode of operation, in which it accepts only a subset of the MOC language.

A MOC file in the restricted subset has the same syntax as the full MOC language, except that the only record type allowed is the **absolute** data record. Furthermore, those **absolute** records must all have zero references. Thus, the restricted subset only allows absolute data to be loaded into absolute locations. The loader can load such records without the existence of the PMM, or of its internal symbol tables.

The loader has a special command-line directive that instructs it to initialize its internal tables, rather than load a module. This directive is usually given during the bootstrap process, after the PMM has been established. Once this directive has been given, the loader may be used in unrestricted mode.

4.7 Loader Structure

This section describes some of the internal structure of the loader, and is only pertinent to people interesting in actually writing a MOC loader.

4.7.1 Memory Management for Loader Areas

It is the loader's responsibility to keep track of free space within loader areas, and allocate this space as data records are loaded into these areas. Note that the run time system is loaded by the loader, and therefore may not be used by the loader (besides which, the loader must operate independent from any particular run time system, as run time systems are specific to a particular programming language environment).

Currently, we do not expect the loader to recycle storage within loader areas. In other words, once a data record is loaded into a loader area, that space is never reused for other records (of course, a global record may overwrite a previously loaded global record of the same name).

The reason we cannot recycle space in loader areas is that the loader cannot tell whether pointers to data records have been stored in other areas outside the loader's control. On the other hand, normal garbage collection of the machine cannot recycle loader managed space, because the garbage collector is not expected to have access to or knowledge of the loader's own storage management data structures.

Given all the foregoing, it is likely that the loader will manage storage by simply keeping a free pointer for each area. A more complex scheme could be used if a large number of aligned records are leading to wasted storage with a simple free pointer scheme. In any event, the loader is free to keep free pointers and other information about the free space of an area in the area's memory itself.

4.7.2 Loader Internal Data Structures

Abstractly speaking, the loader must keep the following data structures:

Global Name Table This table maps global names to the five values *Area*(*name*), *Addr*(*name*), *Map*(*name*), *PE*(*name*), and *FP*(*name*) (the latter three are not relevant if the area is a non-aliased broadcast area). It also maps each name to the size of the first record with that name loaded, so that an error may be signalled if an attempt is made to redefine a global name with a record larger than the original record for that name.

The Global Name Table also records the names of areas defined by the **define-area**, **define-broadcast-area**, and **define-interleaved-area** commands, as well as those defined by the **define-table** command.

Local Name Table This table is similar to the Global Name Table, but keeps track of local names instead of global names. It is cleared before loading each MOC module.

Provide Table This table records the names and versions declared by **provide** commands, to perform subsequent verification of **require** commands.

Pending References This data structure keeps track of reference expressions that have not yet been computed and stored, because they refer to one or more names that have not yet been defined. Expressions containing one or more local names must be identifiable, so that an error may be signalled if any such expressions are outstanding after a module has been completely read (such expressions could never be satisfied, since local names are defined only within one module). A special loader command may be used to find out the names of all records that have pending references, as well as the names that those references are waiting for. This allows the Execution Manager to inform the user if the program he is trying to run is not completely loaded.

Of these data structures, all but the Local Name Table must persist across invocations of the loader, as they contain information that is valid across all modules ever encountered. If the loader runs outside of Monsoon, therefore, these tables must be kept in persistent storage. The most logical place to put these tables is in Monsoon's memory itself, since the tables describe the state of other parts of Monsoon's memory. The **loader-internal** area is reserved for this purpose (see Chapter 5). Note that the data in this area does not need to be accessible by a running Monsoon program, and so the **loader-internal** area can be allocated from memory accessible only through the MMI. The MMI can simulate a data memory of a fictitious PE number, so that the loader may still be written as if the data structures were kept in Monsoon memory itself.

The Local Name Table, as it does not need to persist across invocations of the loader, may simply be an ordinary data structure within the loader program.

The exact format of the persistent data structures as kept in Monsoon's memory will be defined at a later date.

Chapter 5

Id Object Code Format

Id Object Code Format is a blanket term which refers to a collection of conventions obeyed by Id programs when compiled for the Monsoon architecture. This is more than simply the layout of compiled code, as it encompasses many aspects which are run-time, as opposed to load-time, characteristics. These conventions represent agreement between the compiler, execution manager, debugger, and run-time system. Other components of the software system, including the loader, statistics viewer, and Monsoon itself (including interface software and the MINT emulator) have no built-in knowledge of these conventions. The topics which comprise Id Object Code Format are enumerated below; this list is also an outline of this chapter. We indicate which aspects must be respected by which components of the software system.

Areas The memory of Monsoon is partitioned into various areas, each containing different sorts of objects. Some of these areas reflect architectural restrictions; for example, there is an area of words that can be addressed with "short-R" absolute addressing. Others reflect a difference between load-time and run-time objects, or between different garbage-collection policies. Still others hold linkage information.

Heap Management The Id language requires heap storage for many of its datatypes. Heap management routines provide facilities for allocating and deallocating (explicitly or via garbage collection) heap storage; from the point of view of these routines, heap objects are just a contiguous group of words, of various sizes. Conventions are established for placing these objects in heap along with size indications, where necessary. These conventions are necessary for the garbage collector to be able to find objects, and also for the debugger to sensibly interpret the contents of memory. The run-time system provides the heap management routines which must obey these conventions. To the extent that data objects are included in object code, the compiler must also obey these conventions.

Object Representation During execution, an Id program manipulates data representing objects found in the Id programming language. These conventions say how each datatype is represented in Monsoon as a type/data bits pair. When an object uses heap storage, these conventions also say what is found in each word of the heap object it occupies. These conventions are necessary for the garbage collector to distinguish pointers from non-pointers, for the execution manager to properly format input data and interpret output data, and for the debugger to present the contents of memory in a format meaningful to the Id programmer. The compiler must obey these conventions by producing code which builds and interprets objects according to them.

Dynamic Linking Id is a separately compiled, dynamically linked language. This means that a user can change a single top-level Id definition, load it into the system, and have it

replace the old definition, with all callers automatically linked to the new version. The MOC loader langauge provides primitive mechanisms for accomplishing this; the dynamic linking conventions specify exactly how these mechanisms are employed for Id, and therefore give the layout of object code and how different Id procedures refer to one another. Dynamic linking conventions also allow different code blocks to share literal constants loaded into frame store. These conventions are necessary for the execution manager and debugger to identify and invoke various Id procedures, and also establish how the run time system obtains certain necessary information about code blocks, such as frame size. The compiler must obey these conventions when formatting a compiled procedure into MOC records.

Frame Layout and Calling Conventions Associated with each dynamic invocation of a code block is an activation frame, containing local data for that invocation. Conventions are established for the placement of certain information in activation frames, particularly argument/result linkage information. These conventions, therefore, are intimately tied to the calling convention used within Id programs. These conventions help the debugger provide a meaningful trace of activity in a halted Id program, and also are necessary for the garbage collector to identify the root set of active heap objects. The compiler must obey these conventions by producing code which builds and uses activation frames according to them. The execution manager must also obey these conventions when constructing the initial activation frame.

Debugging Information The source debugger for Id tries to present the contents of Monsoon's memory in a format most easily understood by the Id programmer. This means the debugger must associate Id identifiers with memory locations, identify procedure names, reconstruct detailed type signatures for objects, give pointers to source code, *etc.* Much of this information is not necessary for correct execution of the program, but is additional information supplied by the compiler specifically for debugging. These conventions are used only by the debugger, and must be obeyed by the compiler in the code it produces (when the inclusion of such information is enabled by the user).

Some of these conventions, particularly those related to debugging information and storage management, are still topics of research. Everything in this chapter should be considered subject to change.

It is important to understand the difference between the conventions of IOCF and the MOC loader language. MOC is a general-purpose language of loader directives, which instruct the loader to store given data in Monsoon's memory, and also to resolve the use of data by one object module that is defined in another. As MOC is general purpose, it has no knowledge of conventions specific to the Id language system, though it does have some knowledge of architectural details of Monsoon. Those aspects of IOCF which are reflected in compiler output (object code) will determine what MOC constructs the compiler emits, but none of those conventions are part of the MOC language or built into the loader.

5.1 Areas

Different kinds of objects are kept in different areas in memory, because different objects require memory with differing characteristics, and because there are different constraints on how the objects are created and used. For instance, the instruction format constrains some literals to be in the low kiloword (2^{10} words) of a processor's data memory, others to be in the low megaword (2^{20} words). Instructions must be in a different kind of memory than data. Also, data structures are grouped according to the type of use and type of resource management. For example, information about code blocks is kept in one area because this information is

Area Name	*Area Number*	*Storage Management*	*Organization*	*Memory Type*	*Restrictions*
`pmm`	0	Static	Broadcast	DM	PE 0 only
`loader-internal`	1			QDM	(see text)
`static-constants`		Static	A. Broadcast	DM	Must include PE 0
`static-code`		Static	Broadcast	IM	
`code-block`		Loader	Broadcast	IM	
`short-literal`		Loader	A. Broadcast	PEDM	$addr < 2^{10}$
`long-literal`		Loader	A. Broadcast	PEDM	$addr < 2^{20}$
`code-block-descriptor`		Loader	Broadcast	DM	
`identifier-descriptor`		Loader	Broadcast	DM	
`dynamic-link-table`		Loader	Interleaved	DM	
`load-time-pair`		Loader	Interleaved	DM	
`load-time-headered`		Loader	Interleaved	DM	
`run-time-pair`		RTS	Interleaved	DM	
`run-time-headered`		RTS	Interleaved	DM	
`static-frame`		Static		PEDM	
`frame`		RTS		PEDM	

DM = Data Memory, PE or I/S
PEDM = Data Memory, PE only
IM = Instruction Memory
QDM = Data Memory or Quasi-Data Memory (see text)

Figure 5.1: Summary of Areas in IOCF

maintained by the loader. Data structures generated by a program's execution are kept in another area, because they are allocated and deallocated at run time.

The Proto-Memory Manager (PMM), described in Chapter 3, is the data structure and general conventions for partitioning memory into areas. We describe here the particular partitioning required by IOCF. While IOCF imposes many constraints on what areas must be allocated and where, there is great freedom in choosing the exact sizes and placement of the various areas. The primary function of system initialization is to establish these areas, and therefore define the resources available to the Id system. The allocation of areas will therefore depend on the system configuration, and will incorporate experience and expectations of system performance, *e.g.*, the most desirable ratio of activation frame memory to heap memory.

IOCF defines a total of sixteen areas, summarized in Figure 5.1. Each area has a name, known to the loader, which may be used in MOC records and references. Each area also has a number, the index of that area's descriptor in the PMM data structure. Most of these numbers are assigned arbitrarily by the loader when it loads the MONASM file which sets these areas up (see Section 5.9), but the numbers of the `pmm` and `loader-internal` areas are required to be zero and one (see page 70). The storage management column indicates who manages the contents of the area: "static" means that the exact layout of the area is established by convention (and described elsewhere in this chapter) and loaded using absolute MOC records; "loader" means that objects are placed into the area using local and global MOC records, so that the loader manages the layout; and "RTS" means that the Id Run Time System manages the area. The organization column indicates the addressing strategy; see Section 4.1 for more details ("A. Broadcast" stands for "aliased broadcast"). The memory type column indicates

what kind of memory comprises the area; in PMM terms, it indicates the memory type of the spaces from which the area's regions are drawn. Finally restrictions on where the area may be allocated are noted in the last column.

It should be noted that Figure 5.1 does not give the whole story of how various areas should be allocated. For example, the `code-block`, `short-literal`, and `long-literal` areas must be allocated on the same set of PE's, as instructions loaded into the `code-block` area will have absolute R fields which refer to addresses of words in `short-literal` and `long-literal` area. More detailed information is given in the descriptions of the individual areas, which follows.

`pmm` This area holds the PMM data structure, described in Chapter 3. That chapter stipulates that the PMM be located in the data memory of PE 0. Typically this area would be carved out of high memory, as low memory is needed by `short-literal` and `long-literal` areas. Even in a multi-node system, this area only has one region, on PE 0. It needn't be larger than the PMM data structure itself (which does not change size).

`loader-internal` This is the area where the loader keeps its internal symbol tables and lists of pending references. Note that these are completely internal to the loader: their format is not part of the MOC specification, nor is there any way for another component of the software system to directly examine them. (The loader's internal symbol tables should not be confused with the tables explicitly established and managed through the MOC `define-table` and `table-entry` directives, explained in Section 4.3.) While the contents of this area must persist between invocations of the loader, it does not actually have to be located in Monsoon's memory at all, if the loader is not a Monsoon program. This is why the memory type is given as "QDM" in Figure 5.1; the memory for this area may be something like a disk file, accessed through the MMI as if it were Monsoon memory. On the other hand, it could be placed in real data memory, and would have to be if the loader actually ran on Monson.

`static-constants` This data memory area holds constants at addresses established by convention, so that the various components of the software system can communicate with one another. The layout of this area is completely described in Section 5.6. This area is always found at the lowest addresses in data memory of PE 0, as well as all other PE's in the system; the contents of the area is identical on all PE's. The reason it has to include the low memory of PE 0 is because the PMM specification stipulates that the address of the PMM data structure be stored in data memory location 0 on PE 0, and the reason it includes *all* PE's is so that the static constants can be read using absolute-FP addressing.

`static-code` This instruction memory area holds (1) a block of 256 instructions comprising the repertoire of I-structure emulation instructions; (2) a block of 256 instructions which are the entry points for exception handlers (the first of these is always the pipeline idle instruction); and (3) any other code that must be loaded before the loader can run. The latter would include, for example, instructions that must be there for the scan-ring implementation of the MMI to completely function. Note that these instructions might be loaded before the PMM is actually established; see Section 5.9 for a more detailed discussion. This area is a broadcast area encompassing all PE's, since all PE's will need these instructions to function. It is statically allocated not only because the loader is not completely functional when this area is written, but also because the absolute IP's of most of the instructions involved are established by convention. In particular, the I-structure instructions must be at IP's zero through 255, because of the way the PE

generates I-structure request tokens, and so this area will always include those addresses. While the exception entry points can be anywhere, it makes most sense to put them at IP's 256 through 511, with the remaining instructions in this area starting at IP 512. The first exception entry point is always the idle instruction because of the way idles are generated by the PE board.

`code-block` This instruction memory area normally includes all of the instruction memory of all PE's, excluding what belongs to the `static-code` area. All code is loaded in this area, including both user Id code and nearly all of the code that is part of the run time system. Current policy makes this a broadcast area (so that all code is loaded into the same addresses on all PE's).

`short-literal` This data memory area holds Id objects and other constants which are accessed using short-R absolute addressing. Because of the addressing mode, this area must be on PE data memory, within the first 2^{10} locations. This area is a broadcast area, consistent with the policy for code blocks. Literals are discussed in Section 5.4.1.

`long-literal` This data memory area holds Id objects and other constants which are accessed using long-R absolute addressing. Because of the addressing mode, this area must be on PE data memory, within the first 2^{20} locations. This area is a broadcast area, consistent with the policy for code blocks. Literals are discussed in Section 5.4.1.

`code-block-descriptor` This data memory area holds code block descriptors, described in Section 5.4.2. Code block descriptors have to go in a separate area because of the way closures are encoded (see Section 5.3.6). This area could be either an aliased broadcast area or an interleaved area, as it is always accessed using split-phase transactions. As a broadcast area it would save network traffic, as an interleaved area it would minimize total memory usage. Currently, IOCF requires this area to be a broadcast area; changing it to an interleaved area would entail a slight adjustment in the way references to closures are encoded in MOC (Section 5.5.2).

`identifier-descriptor` This data memory area holds identifier descriptors, described in Section 5.4.3. This area could be either an aliased broadcast area or an interleaved area, as it is always accessed using split-phase transactions. As a broadcast area it would save network traffic, as an interleaved area it would minimize total memory usage. Currently, IOCF requires this to be a broadcast area.

Identifier descriptors do not necessarily have to be in a separate area; they could be put in the `load-time-headered` area, for example, if they were given proper headers.

`dynamic-link-table` This data memory area holds two MOC tables: one mapping symbols to code block descriptors, and one mapping symbols to identifier descriptors. These tables are discussed in Section 5.4.4. This area could be either an aliased broadcast area or an interleaved area, as it is always accessed using split-phase transactions.

`load-time-pair` This data memory area holds Id aggregate objects which are (1) created at load time, and (2) have an active area less than or equal to two. See Section 5.3.

`load-time-headered` This data memory area holds Id aggregate objects which are (1) created at load time, and (2) have an active area greater than two. See Section 5.3.

run-time-pair This data memory area holds Id aggregate objects which are (1) created at run time, and (2) have an active area less than or equal to two. See Section 5.3.

run-time-headered This data memory area holds Id aggregate objects which are (1) created at run time, and (2) have an active area greater than two. See Section 5.3.

static-frame This data memory area is used as activation frames by code which runs before the Run Time System's frame area has been initialized. In particular, the program which intializes the Run Time System uses this area.

frame This data memory area holds the activation frames managed by the Run Time System. As different frame management policies are tried out, we may find that we need several frame areas, corresponding to different allocation strategies for different size frames. In any event, the **frame** area must be distinct from the **run-time-headered** area because frames have different restrictions on processor assignment and interleaving.

5.2 Heap Management

The heap contains Id objects that are aggregates: objects which contain other objects. When such an object is created, a new region of data memory must be allocated to contain it. When that object is no longer accessible from a running program, the heap space can be reclaimed, either by the program explicitly deallocating the storage, or by an automatic garbage collector that finds and reclaims all inaccessible objects.

This section defines the basic layout of the areas comprising the heap. We try to do this in a way that does not take a position on the algorithms and data structures used for managing the heap; for example, we want the freedom to use a first fit allocation algorithm or a buddy system algorithm, the freedom to have explicit deallocation or automatic garbage collection. What we do take a position on is the layout of those words of heap which have been allocated to objects—the layout described in this section must be obeyed no matter what allocation/deallocation system is used. We describe the strategy for identifying the boundaries of an object.

5.2.1 Basic Heap Operations

The operation *allocate*(n) allocates n contiguous[1] words of heap. The program making such a request receives a pointer to the first of those n words, and can rely on the remaining words being at logically consecutive locations. On the other hand, the program cannot rely on any relationship between the addresses returned by two calls to *allocate*, regardless of their arguments or timing relationship. The n words beginning with the pointer returned are called the *active area* of the object. The allocator may also allocate additional words before and after the active area; together with the active area they comprise the *total area* of the object. The total area is also required to be contiguous. Words outside the active area may be used by the heap manager for various overheads, but the program calling *allocate* cannot depend on their existence or non-existence.

A program has two ways of returning previously allocated storage to the heap for subsequent re-use. The first is an explicit *deallocate* operation, which takes a pointer to previously allocated

[1] In all discussions of heap storage, the word "contiguous" means *logically contiguous* with respect to the interleaving strategy of the area containing that heap storage. That is, the addresses of contiguous words are obtained by applying Monsoon's pointer increment operation, using the appropriate "map" value.

heap and returns it for use. It is the program's responsibility not to read or write deallocated storage, except as such storage is later returned to the program by another call to *allocate*.

The second way of returning previously allocated storage is by automatic garbage collection, which deallocates all objects to which the program cannot possibly refer. Typically, garbage collection begins with a "root set" of objects known to be non-garbage, then scans these objects to find other pointers to objects, which are then deemed non-garbage. The transitive closure of this process therefore finds all accessible objects, and the rest are assumed to be garbage and available for deallocation. This is not to say that the garbage is returned to use by calling *deallocate* on each garbage object; more often, the garbage collection process transforms the entire state of the heap at once into one with more available storage.

Either method of garbage collection can result in *relocation* of non-garbage objects, meaning that the addresses of allocated objects change. Explicit deallocation may relocate objects to reduce fragmentation. Garbage collection often copies non-garbage objects as it scans them, also resulting in relocation. If deallocation results in relocation, all pointers to relocated objects must be adjusted accordingly.

One of the main topics of research on Monsoon focuses on heap management policy. We therefore do not want to specify what storage allocation algorithm must be used, nor whether deallocation is explicit, automatic, or some combination of the two. What we do specify here, though, is how an allocated object is laid out in memory: what it would look like if a snapshot of memory were taken. The specification guarantees the following two properties:

- Given the address of some word within the total area of an allocated object, the boundaries of the total area can be determined. Thus the address of any word within an object's total area may be used to name an object for explicit deallocation, or to indicate a non-garbage object to be scanned for other non-garbage objects.
- A word outside the active area of an object can never be mistaken for an active word containing a reference to another object. Thus the total area may be scanned for object references if the active area is not known.

There are two other properties which are necessary for certain heap management systems, but require the cooperation of both the heap system and the conventions for representing Id objects:

- Given the root set of the program, all heap objects to which the root set refers can be identified, and given a non-garbage object, all its references to other heap objects can be identified. This is necessary if automatic garbage collection is used.
- All references to heap objects anywhere in the program's state can be identified. This is necessary if deallocation (explicit or automatic) causes relocation.

These properties will be discussed again when the conventions for representing Id objects are presented in Section 5.3. That section also defines the exact representation of a pointer.

Deficiency: As will be pointed out in Section 5.3, the current conventions for object representation do not guarantee the second property, and so do not accomodate relocation. This is a topic under investigation.

5.2.2 Layout

An object created during execution of a program is allocated from one of two areas, the `run-time-pair` area or the `run-time-headered` area, depending on the size of its active area.

The `run-time-pair` area holds objects whose active area is one or two. The total area of an object in this area is always exactly two, and begins on an even boundary (*i.e.*, a location that is an even number of locations away from the first word of the whole area).

The `run-time-headered` area holds objects whose active area is greater than two. If the active area is n, the total area is always at least $n + 1$. The first word of the total area has presence bits `read-only`, type bits `header`, and data bits which give the *total* area as an unsigned integer. (Section 5.3.4 discusses the various presence bit values found in aggregates. Section 5.10 gives the actual numeric values for presence and type fields, which are referred to by name throughout this chapter.) The active area begins with the second word of the total area. The program is not permitted to store a word with type bits `header` anywhere within the active area.

In both the `run-time-pair` area and the `run-time-headered` area, any inactive words following the active area have presence bits `unused`. Thus, a word outside the active area cannot be mistaken for an active word containing a reference to another object, as such a word will either have presence bits `unused` or type bits `header`.

Rationale: Objects in `run-time-headered` area are allowed to have total area greater than $n + 1$ to allow for storage allocation algorithms which can't allocate arbitrary size objects. For example, the buddy system can only allocate powers of two.

Caveat: IOCF does not guarantee that `unused` is a distinct presence bit value from `empty`. See Section 5.10.

An algorithm for finding the boundaries of the total area of an object given a pointer within the total area is as follows:

1. If the pointer p is into the `run-time-pair` area, the boundaries are $2\lfloor p/2 \rfloor$ and $2\lfloor p/2 \rfloor + 1$.

2. If the pointer is into the `run-time-headered` area, scan backward until a word with type bits `header` is found; this word is the beginning of the total area. The data bits of this word indicate the size of the total area, and therefore the end of the total area.

Note that it is necessary to identify a pointer's area to find the boundaries of the corresponding object. This suggests that it may be a good idea to choose the boundaries of the areas to minimize the complexity of this test.

The `load-time-pair` and `load-time-headered` areas are laid out exactly as the `run-time-pair` and `run-time-headered` areas, but hold objects created at load time rather than at run time. Storage in these areas is allocated by the loader when it encounters MOC records that specify those areas; such records must obey the layout rules described above. While objects in the load-time areas will be scanned during garbage collection, it is not feasible to deallocate them because the loader is managing the storage. The number of garbage load-time objects is likely to be very small, so this is not expected to be a problem.

5.3 Representation of Datatypes

5.3.1 Immediates versus Aggregates

The view of objects and their representations taken in IOCF is the "object reference" view. This view is explained by Moon[2]:

[2]David A. Moon, "Architecture of the Symbolics 3600," in *Proceedings of the 12th Annual International Symposium on Computer Architecture*, IEEE, June, 1985.

> The fundamental form of data ... is an *object reference*. The values of variables, the arguments to functions, the results of functions, and the elements of lists are all object references. An object reference designates a conceptual object. There can be more than one reference to a given object. Copying an object reference makes a new reference to the same object; it does not make a copy of the object.
>
> A typical object reference contains the address of the representation in storage of the object. There can be several object references to a particular object, but it has only one stored representation. Side-effects to an object, such as [storing into] an array, are implemented by modifying the stored representation. All object references address the same stored representation, so they all see the side-effect.
>
> In addition to such *object references by address*, it is possible to have an *immediate object reference*, which directly contains the entire representation of the object. The advantage is that no memory needs to allocated when creating such an object. the disadvantage is that copying an immediate object reference effectively copies the object; thus immediate object references can only be used for object types that are not subject to meaningful side-effects, have a small representation, and need very efficient allocation of new objects. [Integers and floating point numbers] are examples of such types.

We will use the term *immediate object* to denote an object represented by an immediate object reference, and *aggregate object* to denote an object represented by an object reference by address. Thus, "aggregate object" is synonymous with "heap-allocated object."

In Monsoon, object references consist of type bits and data bits (currently eight type bits and 64 data bits), and are carried on tokens and stored in data memory.

5.3.2 Data Bits

The Monsoon architecture views data bits in one of five machine data formats: integer (a two's complement signed integer), Dfloat (an IEEE double precision floating point number), Sfloat (an IEEE single precision floating point number), pointer (a four tuple of NODE, OFFSET, MAP, and INFO), continuation (a four tuple of PE, FP, IP, and PORT). (A sixth type, request (a three tuple of NODE, OFFSET, and OPCODE) is used internally by the instruction set in the implementation of split-phase transactions.) What IOCF describes is how objects and object references are encoded into these five machine data formats. Generally speaking, all object references by address are encoded as pointers, and different immediates are encoded into integers, Dfloats, pointers, and continuations. Sfloats are not used by IOCF. A more detailed discussion is found in the sections describing the representation of each Id datatype; a general discussion of the hardware formats and type system is found in the *Monsoon Macro-Architecture Reference Manual.*

5.3.3 Type Bits

In the current definition of IOCF, type bits serve two purposes. First, they allow the garbage collector to distinguish between immediate object references and object references by address (*i.e.*, between non-pointers and pointers). Second, they give additional information about immediates which can help a low-level debugger print them in a meaningful way. Id is strongly typed, so the type bits are *not* relied upon to implement the semantics of the language. Also, type bits are not adequate to recover the complete Id type of an arbitrary object, and so they do not adequately support the needs of a high-level debugger for the Id programming language.

Major Type	*Data Format*	*Minor Type*
INT	Integer	void signed boolean enum* character* packed-character* header*
SFLT	SFloat	[Not used by IOCF]
DFLT	DFloat	dp-float
PTR	Pointer	internal external head* closure*
CONT	Continuation	[Encodes PRIV and COLOR]
FUTR	Continuation	[Not used by IOCF]

*Defined by IOCF.

Figure 5.2: Type Bits in IOCF

Future Improvement: Eventually, the IOCF specification will be extended to fully support the needs of high-level debugging of Id programs. It is not yet known, however, to what extent the type bits will assist in allowing such a debugger to determine complete Id datatypes. It is likely that the final specification will use a combination of type bits, address conventions, and information stored in the data bits of objects to achieve this effect.

The instruction set of Monsoon provides the overall structure of the type bits system. The instruction set divides the type bits into *major type* and *minor type*. The major type identifies a class of related types, and all types within a given major type will have data bits in one particular format. All exceptions taken on the basis of type bits are sensitive only to the major type. The minor type further qualifies the major type. Some of the minor types within each major type are defined by the instruction set, others are "user defined" from the instruction set's point of view, and are defined by IOCF. Figure 5.2 summarizes the major and minor types, indicating which ones are defined by IOCF.

The following values of type bits are found on object references that may be pointers of one kind or another: `head`, `internal`, `external`, and `closure`. The meanings of the first three are discussed in Section 5.3.5 on the general format of aggregates, while closures are described in Section 5.3.6. The type bits `header` is not found on any object references, but has a special role in storage management described in Section 5.2. All other values of type bits are found on

immediates, their values helping to indicate how an immediate should be printed by a low-level debugger. (The actual numeric values of type bits are given in Section 5.10.)

5.3.4 Presence Bits

The representation of an aggregate determines the range of presence bit values that each word in heap can assume. Basically, every word in heap can either have *I-structure semantics*, or *read-only semantics*. Read-only semantics are used for words whose values are stored when the object is created, while I-structure semantics are used for words that are stored asynchronously with respect to the creation of the object.

Words with read-only semantics can have one of two values of presence bits: `empty` and `read-only`. The code which creates an aggregate must insure that no attempt is made to read an empty read-only slot, thereby allowing code that does read those slots to rely on never having those reads deferred. Typically, the creator of an aggregate will first obtain empty storage from the storage allocator, then fill in the read-only slots (simultaneously changing each word's presence bits from empty to read-only), and only then releasing the object reference to potential readers of the data structure. Read-only slots, therefore, are only used for data which is guaranteed known at object creation time, such as the access coefficients for an array. The advantage of read-only slots is potentially cheaper access, since readers of those slots do not need to worry about the possibility of deferred reads.

Words with I-structure semantics can have one of five values of presence bits: `empty`, `present`, `deferred`, `delayed`, or `lock-deferred`. The meanings of these values are as follows:

`empty` The word has neither been written nor read; the data and type bits are unspecified. For a location used as a lock, it may also mean that the location had been written but was subsequently taken.

`present` The word has been written: the data and type bits contain an object reference.

`deferred` The word has been read at least once but not yet written: when the value is eventually written, a copy should be sent in the data part of a token, with the tag part of the token given by the current contents of the word (data and type bits). Sending this token may have the effect of satisfying several deferred readers.

`delayed` The word has not been read and the computation which is to write the word has been delayed. When the word is read, the data and type bits are sent to the thunk manager, which will interpret them as a pointer to a thunk representing the delayed computation which, when executed, will write the word.

`lock-deferred` The word has been taken at least once but never written, or taken at least once after a prior take had emptied the word. When the value is eventually written (put), it will be sent to one of the deferred takers.

Words with either read-only or I-structure semantics may also have presence bits `error`, indicating that an inappropriate transition has taken place. This can only occur if the hardware malfunctions, or if there are bugs in system code.

Figure 5.3 summarizes the state transitions that can take place on I-structure words. Omitted from the figure are transitions to the `error` state: any operation for which no transition is depicted in the figure causes a transition to the `error` state. Also, there are operations provided for writing an arbitrary presence value, regardless of current state. The operations and

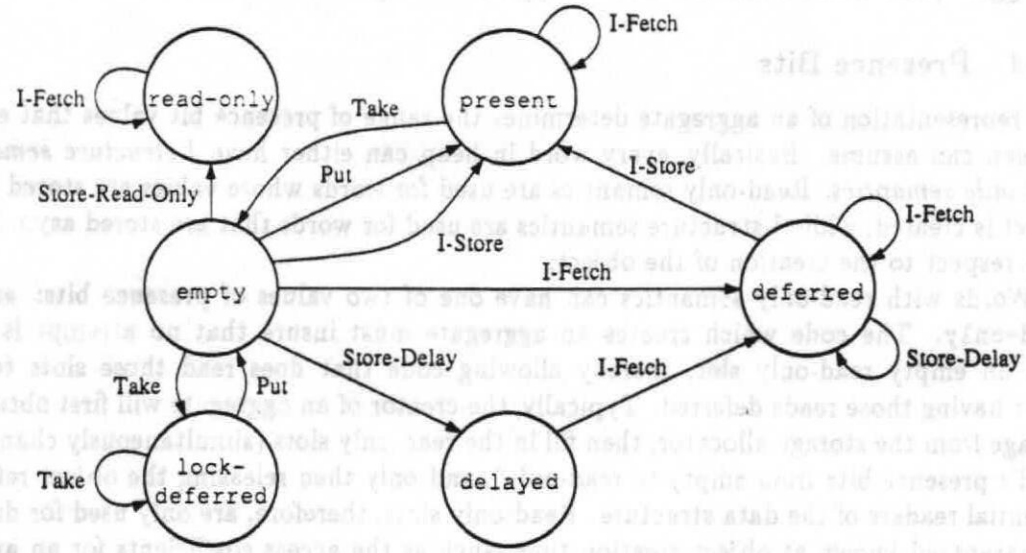

Figure 5.3: I-Structure State Transition Diagram

transitions are presented only to assist in understanding the meaning of each presence value; technically, the meaning of I-structure operations and their associated transitions is outside the scope of the IOCF specification.

The actual numeric values of presence bits are given in Section 5.10.

5.3.5 Aggregates: General Policy

To describe the representation of an object as an aggregate we must say how the object is laid out in heap and how the object reference is encoded. In this section we give the general rules that all aggregate encodings follow. Note that an Id program only makes use of the active area of an aggregate, and in general has no information on what may or may not exist outside the active area.

References to Aggregates

An object reference to an aggregate always has type bits **head**, and data bits which are formatted as a "pointer" machine data type. The INFO field is zero, and the MAP, NODE, and OFFSET fields are a Monsoon pointer to the first word of the active area of the aggregate. That is, the first word of the aggregate is found in word OFFSET of data memory on node number NODE. Successive words of the aggregate are found at logically successive addresses, by using MAP to control the Monsoon pointer increment operation.

The **head** object reference is the usual reference to an aggregate; it corresponds to a denotable value in Id. There are two other types of object references that do not truly denote an object, but nevertheless arise as intermediate values during manipulation of object references. An *internal pointer* is a reference to an object that points to some word of that object, though not necessarily the first. An *external pointer* is a reference to an object that may not even point at a word within the object. Both have the same format as pointers, except that the type bits are **internal** or **external**, respectively. Because the program has no knowledge of what may lie outside the active area, internal pointers will only be created for words within the active

area, even though the storage manager could deal with a pointer to any word within the *total* area just as well.

Head pointers, internal pointers, and external pointers obey the following rules, important to a garbage collector. If a head pointer to an object is accessible from the root set, then that object is not garbage. Similarly, if an internal pointer is accessible then the object is not garbage, even if there are no head references to that object. On the other hand, an external pointer to an object may not exist unless there is a head pointer or internal pointer accessible from the root set. Thus, a garbage collector may disregard all external pointers when determining which objects are not garbage (the "marking" phase).

The above rules imply that a garbage collector must be able to identify the end of an object given a head pointer to it (the head pointer identifies the beginning). Similarly, given an internal pointer, the garbage collector must be able to find both the beginning and the end. Given an external pointer, however, the garbage collector does not need to be able to infer anything about the object (and in general will not be able). How the boundaries of an object are identified was discussed in Section 5.2.

Rationale: The distinction between internal and external pointer was made because internal pointers are created routinely by the ordinary *increment-pointer* operation. It would be prohibitively expensive to insure that a pointer to an object exists whenever an internal pointer does (consider the case where the last reference to an array is about to be consumed by a *increment-pointer* and *I-fetch* pair, just after the *increment-pointer* executes but before the *I-fetch*. On the other hand, external pointers are generated only through the application of specialized optimizations, particularly the hoisting of invariant code from loops. In these cases, it is not terribly expensive to insert special code to insure a head pointer or internal pointer exists (not expensive in terms of run-time overhead, but it may make life hard for the compiler!).

Deficiency: While the scheme outlined here is adequate to mark the heap (separate the non-garbage from the garbage) in the presence of internal and external pointers, it is not sufficient to enable the proper updating of external pointers if objects are relocated during garbage collection. This is because there is no way to identify to which object an external pointer refers. This is still a topic under investigation.

Active and Total Area

We reiterate here some points about active and total area.

From the Id programmer's point of view, an aggregate consists only of other object references. In fact, there are two kinds of overhead associated with aggregates which consume additional words beyond that needed for subcomponents' object references. One overhead is words needed by the Id language to record attributes of an object, such as the dimensions of an array, or the disjunct number of a member of an algebraic data type. The other overhead is words used by the heap manager to keep track of the boundaries of objects. In IOCF, these two overheads are kept separate from each other and from the subcomponents.

To account for these overheads, we distinguish between the *total area* and the *active area* of an object. The *total area* of an object is all the words consumed by that object, including language overhead and storage management overhead. Thus, if the total area of some object is six words, then creating it will remove exactly six words from the heap, and deallocating it will return exactly six words for use by other objects. The *active area* of an object includes the subcomponents and language overhead, but excludes storage management overhead. For example, the active area of a one-dimensional, three word array is (currently) five: three words

for the array elements, and two words for the access coefficient and bounds. Notice that the active area of an object is completely determined by the compiler; it is independent of the storage allocation strategy used. The total area of an object is always at least as large as the active area.

With the above in mind, we elaborate on our earlier description of pointer types:

- The total area of an aggregate is logically contiguous.
- The active area of an aggregate is also logically contiguous, and is contained within the total area.
- A head reference to an aggregate always points to the first word of the *active* area.
- A internal reference to an aggregate always points to some word in the *active* area, which word may or may not be the first.
- An external reference may point to any word, which word may be in the active area, the total area, or completely outside the object.
- The storage management system must be able to determine the boundaries of an object's total area given a head pointer or internal pointer.

The words belonging to the total area but not to the active area are entirely the responsibility of the storage manager. A request to allocate n words really means allocate n active words, returning a head pointer to the first active word. Whether there are additional words before or after the active area is not of interest to the code making the request. Conversely, the storage allocator does not know what the requestor intends to do with those words, so the number of total words depends only on the number of active words (and not, for instance, on the type of object to be stored in those words).

In describing the representations of Id datatypes, we will only speak of active area. The relationship between active area and total area is discussed in Section 5.2.

5.3.6 Representations of Id Datatypes

The representations of Id datatypes are summarized in Figure 5.4. For immediate datatypes, the type bits of the object reference are given. For aggregate datatypes, the type bits of the object reference are always **head**, or **internal** or **external** when appropriate. (Figure 5.4 is not an exhaustive list of the possible type bits: type bits **header** and **packed-character** are used in the representations of certain aggregates, and type bits **request** are generated by the hardware during split-phase transactions. The figure only lists the type bits found on object references to Id data types.) The *signed*, *void*, and *continuation* datatypes are non-denotable: no Id identifier can have one of these as its value, but they arise internally in code generated by the compiler. Each datatype is described in more detail below.

Characters

A character is an immediate object with type bits **character**. The least significant eight data bits are the character itself, while the most significant fifty-six data bits are all zero.

The character set of Id includes the 94 printing characters of ASCII, plus some special characters. The printing characters are encoded as ASCII codes, with the most significant bit (of eight) zero. The special characters have the following decimal values:

Datatype	*Immediate/ Aggregate*	*Immediate Type Bits*	*Aggregate Active Area*	*Aggregate* INFO
Character	I	`character`		
Number	I	`dp-float`		
Integer *(non-denotable)*	I	`signed`		
Boolean	I	`boolean`		
Void *(non-denotable)*	I	`void`		
Continuation *(non-denotable)*	I	`continuation`		
String	A		$\lceil len/8 \rceil + 1$	0
Symbol	A		$\lceil len/8 \rceil + 1$	0
Array	A		$rank + 1 + \prod dim_i$	0
0-ary Algebraic Disjunct	I	`enum`		
n-ary Algebraic Disjunct	A		n or $n+1$	0
Closure	I or I+A	`closure`	2	$2^{17}r + cbname$

Figure 5.4: Summary of Representations of Id Datatypes

Id Character	*Encoding*
`\backspace`	8
`\tab`	9
`\newline`	10
`\linefeed`	10
`\page`	12
`\return`	13
`\space`	32
`\rubout`	127

Numbers (Floats)

A float is an immediate datatype whose type bits are `dp-float`. The data bits are in the same format as Monsoon's machine data type Dfloat; that is, the data bits are in IEEE double-precision floating point format. Single-precision floating point numbers are not used in the implementation of Id.

Currently, the Id language has only a single numerical datatype called *Number*, which is always represented as a float. Integers only arise internally during certain calculations, described below; the value of an Id identifier or a accessible component of an Id data structure is never represented as an integer.

Potential Change: It is possible that at some point in the future, both floats and integers will be denotable in Id. One possible scheme for doing so involves overloading; use of automatic coercions based on type bits is another possibility. In any event, the representations of floats and integers in IOCF will not change, only the situations in which each can appear.

Integers

An integer is an immediate datatype whose type bits are `signed`. The data bits are in the same format as Monsoon's machine data type integer; that is, the data bits are the integer in 64-bit

two's complement representation.

Integers only arise in the following situations:

1. The access coefficients of arrays (see *Arrays*, below) are integers.
2. Array subscripts are converted to integers prior to subscript linearization calculations.
3. Constants denoting offsets within n-ary algebraic disjuncts are integers.
4. Various pieces of system code that manipulate Monsoon's machine state, or that comprise the run time system, may use integers internally.

Booleans

A boolean is an immediate datatype whose type bits are **boolean**, and represents boolean **true** or **false**. The data bits of **true** are all ones (*i.e.*, the integer -1), and of **false** are all zeros.

Void

Void is an immediate datatype whose type bits are **void** and whose data bits are unspecified.

Void arises during resource management code, especially termination detection; they only carry information that certain events have completed.

Continuations

A continuation is an immediate datatype whose type bits are **continuation**, and is a hardware supported datatype. Continuations represent computational contexts, consisting of a pointer to code and a pointer to data. The minor type of continuations is used by the hardware to encode the PRIV and COLOR fields; their use in IOCF is still under study.

Continuations arise during procedure linkage and during other manipulation of activation frames.

Strings

A string is an aggregate datatype. The first word of the active area is a non-negative integer giving the number of characters in the string. The following words contain the characters of the string, packed eight characters to a word, with each group of eight characters arranged in order from most significant byte to least significant byte, corresponding to left to right. Unused bytes in the last word (if any) may contain anything. The active area, therefore, totals $1 + \lceil len/8 \rceil$ words.

All words of strings have read-only semantics. The type bits of the first word are **signed**, that of suceeding words are **packed-character**.

Symbols

A symbol is an aggregate datatype. Symbols have a representation identical in every respect to that of strings, with the additional property that if two object references refer to symbols with the same name, then the object references will be bit-for-bit identical. In other words, a symbol with a given name will only occur once in the entire heap. This allows a quick equality comparison on symbols. The Id language does not allow new symbols to be created at run-time; all symbols are created at load time. The encoding of symbols into MOC ensures that the pointer equality property is preserved.

Arrays

An array is an aggregate datatype. The first word of the active area is (an object reference to) the bounds tuple. The next r words are access coefficients, where r is the rank of the array; each coefficient is an integer, with type bits **signed**. The remaining words are the elements of the array. The bounds tuple and access coefficient words have read-only semantics, while the remaining words have I-structure semantics.

The access coefficients are as follows. For an array of rank r, the offset o of the element referenced by $\mathtt{a}[x_1, \ldots, x_n]$ is given by the following:

$$o = c_0 + c_1 x_1 + \cdots + c_r x_r$$

The actual address of that location is given by $PTRINC(p, o)$, where *PTRINC* increments p, a pointer to the first word of the array's active area, by o according to the interleaving specified in the MAP component of the pointer.

Let the lower and upper inclusive bounds on a subscript x_i be l_i and u_i, respectively. Furthermore, define $n_i = u_i - l_i + 1$. Then the values of the coefficients are given by:

$$\begin{aligned} c_0 &= (r+1) - \sum_{j=1}^{r} l_i \prod_{k=1}^{j-1} n_k \\ c_1 &= 1 \\ c_i &= \prod_{j=1}^{i-1} n_i \end{aligned}$$

These equations give the coefficients for column-major order. The term $r + 1$ in the equation for c_0 accounts for the space occupied by the bounds tuple and the coefficients. Note that the value of c_1 is always one, and so there is no need to store it with the array. The coefficients are stored in the order $c_0, c_2, c_3, \ldots, c_r$.

Algebraic Disjuncts

Algebraic datatypes provide a general facility for discriminated unions of fixed-length, heterogeneous data structures. Lists and n-tuples in Id are implemented as special cases of algebraic datatypes (these special cases are discussed at the end of this section).

In its most general form, an algebraic datatype in Id is a type declared by a statement of the form:

type *typename formal-1 ... formal-k* =
d-1 elt-type-1-1 ... elt-type-1-n_1 | ... | *d-m elt-type-m-1 ... elt-type-m-*n_m

This declares a new type *typename*. Each *d-i* is the name of a *disjunct* of the new type; each disjunct *d-i* has arity n_i, $n_i \geq 0$. A disjunct of arity n_i is essentially an n_i-tuple, with component types given by the expressions *elt-type-i-1* through *elt-type-i-*n_i, where these expressions may be in terms of the type variables *formal-1* through *formal-k*. An identifier of type *typename* may be bound to any of the m disjuncts, and so it must be possible to distinguish among the disjuncts of a given algebraic type at run time.

As 0-ary disjuncts have no components, they are represented as immediates. The type bits of a 0-ary disjunct are **enum**, and the data bits are a non-negative integer indicating which

disjunct the object represents. The correspondence between these integers and the disjunct names is given below.

An n-ary disjunct, for positive n, is an aggregate. The format of this aggregate, however, depends on the total number of positive-ary disjuncts in the type. If an algebraic type has only one n-ary disjunct other than 0-ary disjuncts, then there is no need for the n-ary disjunct to carry an indication of which disjunct it is; a reference to it is distinguishable from all other disjuncts of the type since it is the only one with major type PTR. In this case, the active area of the aggregate is n, each word containing (a reference to) a component, with successive components corresponding to the left-to-right order in the **type** declaration. All words have I-structure semantics.

If an algebraic type has more than one n-ary disjunct other than 0-ary disjuncts, then every disjunct must carry an indication of which disjunct it is. In this case, the active area of the aggregate is $n + 1$, where the first word contains an indication of which disjunct, and the remaining words are the components, in left-to-right order. The first word has type bits **enum**, and a non-negative integer as data bits. The correspondence between this integer and disjunct names is given below. The first word has read-only semantics, while the remaining words have I-structure semantics.

The integers used as **enum** values are assigned as follows. First, all 0-ary disjuncts are assigned consecutive values from zero, in left-to-right order as declared in the **type** statement. Then, all positive-ary disjuncts are assigned consecutive values from zero, also in left-to-right order. The rationale is that in doing a case dispatch on an object reference to a member of an algebraic datatype, a dispatch on type is first necessary to distinguish 0-ary disjuncts (immediates) from positive-ary disjuncts (aggregates). Assigning numbers to the two subsets separately yields two dense dispatches on **enum** values, rather than two sparse ones. Note that if there is only one positive-ary disjunct, instances of that disjunct do not actually contain an **enum** value. For example, for the following type:

```
type status = Yes | Excuse string | No
```

the object `Yes` is represented by an immediate with type bits **enum** and data bits equal to zero, while `No` has data bits equal to one. For the following type:

```
type intermediate_status = Yes | In_progress string | Excuse string | No
```

the objects `Yes` and `No` again have type bits **enum** and data bits equal to zero and one, respectively, while the aggregates for `In_progress` and `Excuse` have as their first words enums with data bits equal to zero and one, respectively.

Tuples in Id (obtained through the comma (`,`) infix syntax) are special cases of algebraic datatypes. They are represented as if the following definitions existed:

```
type 2_tuple *0 *1       = Two_tuple   *0 *1
type 3_tuple *0 *1 *2    = Three_tuple *0 *1 *2
type 4_tuple *0 *1 *2 *3 = Four_tuple  *0 *1 *2 *3
```

etc.

Thus, an n-tuple is represented as an aggregate with an active area of n, with consecutive words corresponding to the components in left-to-right order.

Lists in Id (obtained through the colon (`:`) infix syntax) are also a special case of an algebraic datatype. They are represented as if the following definition existed:

```
type list *0 = Nil | Cons *0 (list *0)
```

Thus, `Nil` is an immediate with type bits **enum** and data bits equal to zero, while a cons is an aggregate of two words where the first word is any object, and the second word is either (a reference to) another cons or the immediate `Nil`.

There are two special cases of algebraic datatypes that are not denotable: code block descriptors and identifier descriptors. These are only created at load-time, never at run-time, and are described in Section 5.4.

First-Class Functions (Closures)

An object representing a function in Id is always a partial application of a top-level function to some of its arguments, specifically, to the first i arguments where i is less than the function's arity. The name of a top-level function used directly as a value is a special case of a partial application to zero arguments, while internal functions (functions defined at lexical levels more inner than top-level) are converted to top-level functions during compilation by the *lambda-lifting* transformation. The details of lambda-lifting, and therefore the identity and number of extra formals introduced during lambda-lifting, are not specified by IOCF.

Partial applications of top-level functions are called *closures*. Conceptually, a closure consists of the name of a procedure, the number of arguments remaining (the procedure's arity minus the number of arguments to which it has already been applied), and a linked list containing the arguments to which the procedure has already been applied. Portions of the linked list may be shared among closures; this arises in the case where the same partial application is later applied to two different arguments.

The representation of closures is unique, in that it contains both immediate data (the procedure name and number of arguments remaining) and aggregate data (the linked list of arguments). The type bits of a closure are always **closure**. The data bits are formatted as a pointer, where the MAP, NODE, and OFFSET fields comprise a pointer to the linked list of arguments, the most significant 7 bits of INFO are the number of arguments remaining minus one, and the least significant 17 bits of INFO is *cbname*, described below.

A true pointer to the linked list can be constructed from the MAP, NODE, and OFFSET fields of the closure by setting the INFO field to zero and the type bits to **head**. The linked list is an aggregate with an active area of two, where the first word contains the argument most recently applied, and the second word is a linked list of the arguments remaining after that one. Both words have I-structure semantics. The last pair in the chain has the null pointer (a pointer whose MAP, NODE, and OFFSET fields are all zero). Note that this is different from an Id list, where the last pair would have an **enum** with data bits zero instead of a null pointer. If a closure has no arguments in the chain (*i.e.*, the number of remaining arguments is the same as the procedure's arity), the MAP, NODE, and OFFSET fields of the closure are all zero. Hence, when adding another argument to the front of a chain, the second component of the new pair can always be obtained by extracting a pointer from the closure.

The value *cbname* is a compact representation of the address of a code block descriptor (see Section 5.4.2). The `code-block-descriptor` area contains contiguous code block descriptors; if you think of the descriptors as being numbered consecutively from zero, then *cbname* is the number of a particular descriptor. Thus, a head pointer to the code block descriptor can be constructed as follows:

$$\begin{aligned} \text{MAP} &= \mathit{Map}(\texttt{code-block-descriptor-area}) \\ \text{NODE} &= [\text{Any}] \\ \text{OFFSET} &= \mathit{cbsize} \times \mathit{cbname} + \mathit{FP}(\texttt{code-block-descriptor-area}) \end{aligned}$$

INFO = 0

where *cbsize* is the number of words in a code block descriptor. Alternatively, we could have described this pointer as

$$PTRINC(Ptr(\texttt{code-block-descriptor-area}), cbsize \times cbname)$$

Note that when a closure is applied to another argument, the new INFO is obtained by subtracting 2^{17} from the old INFO. When only one argument remains, INFO is equal to *cbname* (this is why the upper bits of INFO contain the number of arguments remaining minus one).

Alternative: Again, the MAP, NODE, and OFFSET fields would form a pointer to an argument chain, but INFO would be an entry point to some closure manipulation code specific to the procedure being applied. To apply the closure to an argument, one allocates a small frame (at FP'), sends the closure itself to ⟨INFO,FP'⟩. This code will either return a new closure (with a new entry point) or it will allocate a context and perform a tail call to the real entry point of the procedure (and unwind the arguments from the argument chain). This method may require *a* entry points for an arity *a* procedure, but it may require less overhead for general applies than the current scheme.

Delayed Objects (Thunks)

[The mechanism for thunks was still under development as this document went to press. Please check for any addenda that have been issued.]

5.4 Dynamic Linking I: Literals, Code Blocks, and Identifiers

Literals, code blocks, and identifiers are not denotable objects in the Id language, but are the framework which binds together the various components of an Id program into a single programming environment. *Identifiers* represent the top-level identifiers in the Id program, the identifiers at the outermost lexical scope. A reference to an identifier occurs when an identifier defined by one top-level definition appears in the body of the definition of another; such a reference receives the value to which the identifier is bound, which may be any denotable Id value. *Code blocks* are contiguous segments of executable code, loaded into instruction memory, and correspond to the unit of code for which activation frames are allocated at run time. References to code blocks occur when the compiler breaks a function definition into several code blocks (usually for reasons related to resource management): the main code block of a function will invoke inferior code blocks, and so on. As an optimization, the compiler also converts a reference to an identifier into a reference to a code block, when the identifier is known to be bound to a particular function, and is applied to a complete set of arguments. *Literals* are constants compiled as part of a code block and stored in data memory. References to literals occur in constructs like `x + 1`, where the integer one is effectively built into the object code. Procedure calling conventions also result in literals that refer to code blocks and identifiers.

All references to an identifier or code block with a given name indirect through a common data structure, a *identifier descriptor* or *code block descriptor*, as appropriate. This allows an individual procedure to be changed, recompiled, and reloaded, with all callers automatically seeing the change, thus supporting a highly interactive development environment. Thus, for

each top-level identifier in an Id program there is loaded an identifier descriptor, and for each code block there is an associated code block descriptor.

In addition, a table of all identifier descriptors and a table of all code block descriptors are maintained; these tables are useful to the Execution Manager and the Debugger.

While identifiers and code blocks are related, they are not in one-to-one correspondence. While the top-level definition of a function `f` will define an identifier named `f` and also generate a code block named `f`, the compiler may choose to generate several code blocks to implement `f`. On the other hand, a definition like

```
def pi = 3.1415927;
```

could define an identifier named `pi`, but no code blocks at all. Even in the case where an identifier is bound to a procedure, its value is not the code block for that procedure; rather, its value is a closure of that procedure over zero arguments. The value of an identifier is always a denotable value in Id. Code blocks are not denotable, but closures (which are the representations of first-class procedures) are. The various possibilities for the value of an identifier is covered in the section on identifiers, Section 5.4.3.

The remainder of this section covers the representation of identifier descriptors and code block descriptors, and explains the linking conventions for literals, code blocks, identifiers, and how those conventions are implemented through MOC.

5.4.1 Literals

The Monsoon architecture does not allow constant data to be imbedded directly into instruction memory. Instead, a compile-time constant, also called a *literal*, must be stored in data memory, with the referencing instruction having the data memory address in its R field. The data memory word contains an ordinary object reference, with presence bits **read-only**.

An important consideration is the architectural restriction on the R field of instructions. Monsoon provides two instruction formats: one with a 10-bit R field, and one with a 20-bit R. When compiling a reference to a particular literal, the compiler must decide whether to generate an instruction which uses the 10-bit format or the 20-bit format. The 10-bit format generally results in fewer total instructions, because the other 10 bits may be used for a destination, but the architecture does not permit more than 1024 (2^{10}) different literals to be referenced in this format.[3] IOCF allows the compiler to choose, for each literal reference, whether to use a 10-bit or a 20-bit reference. The literal will be placed in either **short-literal** area or **long-literal** area; these areas are placed in memory to insure that their addresses fit within 10 bits or 20 bits, respectively (see Section 5.1).

Clarification: It is solely the responsibility of the compiler to decide whether to use a short or long literal for a given compile-time constant. If the compiler generates too many different short literals, when the program is loaded the loader will eventually fill up the short literal area, and the program will not load. The current thinking on this is that all literals will be long literals except those on a short list built into the compiler, which would probably include such things as zero, one, **nil**, *etc.* The compiler could also provide an annotation to allow the programmer to request that a certain literal should be compiled as short; it would then be the programmer's responsibility to not use this feature too heavily.

---

[3] Actually, the limit is somewhat smaller in practice because some of the low memory locations are occupied by static constants (see Section 5.6).

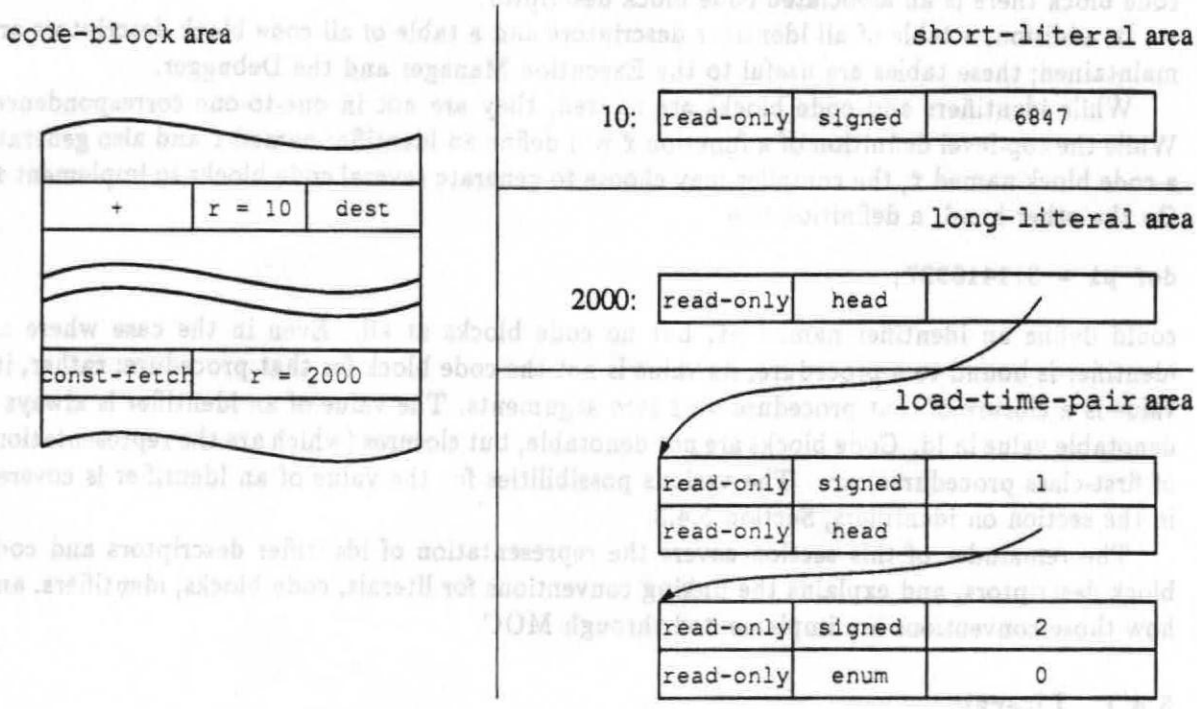

Figure 5.5: A Code Block with Two Literals

A code block with two literals is illustrated in Figure 5.5. The first literal is the integer 6847, placed in **short-literal** area. The second literal is the list **1:2:nil**, placed in **long-literal** area. The literal word contains an object reference exactly as described in Section 5.3, with presence bits **read-only**. As the list literal illustrates, the heap storage for literals that are aggregates is allocated out of **load-time-pair** and/or **load-time-headered** area, and all words in the load time heaps receive presence bits **read-only**.

In theory, each compile-time constant in every code block needs its own word in literal area. In practice, IOCF is designed so that multiple references to the same constant share the same word in literal area, even if the references are in different code blocks defined in different object modules loaded at different times. This kind of sharing is done for literals that are characters, floats, integers, booleans, voids, 0-ary disjuncts, symbols, identifier descriptors, and code block descriptors. Literals that are continuations, strings, arrays, and *n*-ary disjuncts are not necessarily shared; each reference gets a new literal word.[4] The remaining datatypes, closures and thunks, never appear in literals (although they do appear as components of load-time aggregates).

Rationale: Sharing of literal words is desirable as it saves memory, especially given the severe architectural limit on the size of **short-literal** area. Accomplishing this sharing, however, requires the values of such literals to be encoded in the names of the MOC records which load the literal words, as explained in Section 5.5. This is not feasible for arrays and *n*-ary disjuncts because of their complex structure. Continuation literals, which are quite rare, have a slightly different problem in that they are not likely to be compile-time constants, but instead will have some linking done to them as they

[4] The compiler may choose to share such literals within an object module, but no MOC mechanism is employed to obtain this sharing *between* object modules, as is done for the other literal types.

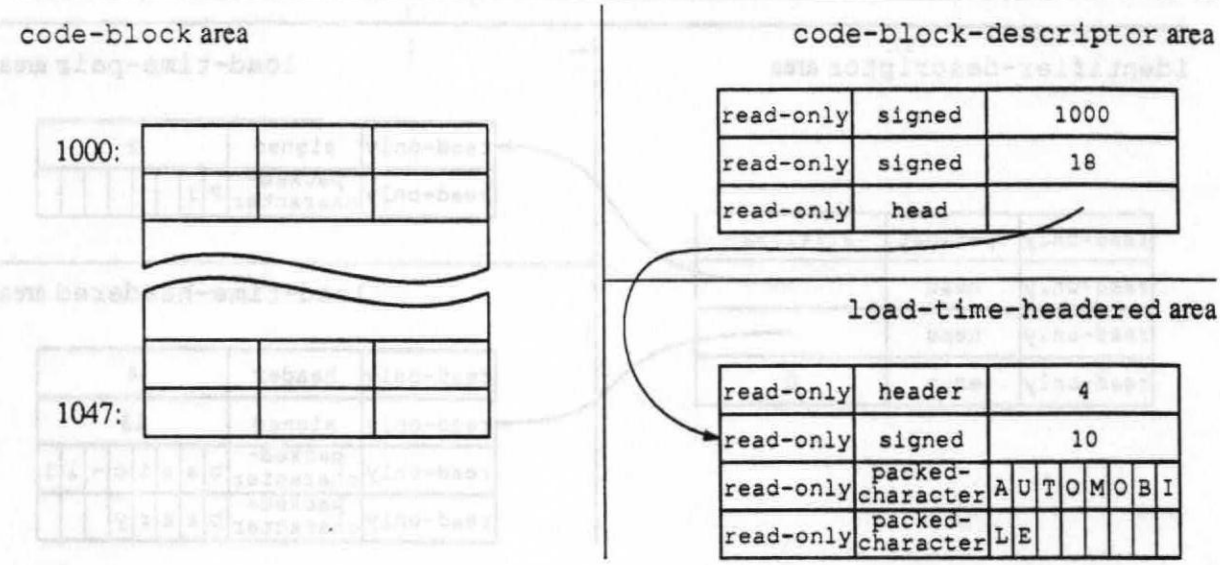

Figure 5.6: A Code Block and its Descriptor

are loaded. String literals could be shared in the same way as are symbols; experience will show whether this is really necessary.

5.4.2 Code Block Descriptors

The layout of a code block descriptor is illustrated in Figure 5.6. The code block descriptor is a data structure of three words, stored in the **code-block-descriptor** area. Only code block descriptors are stored in this area, and they are always stored at multiple-of-three boundaries, so that the address of a code block descriptor may be efficiently encoded into a closure (see Section 5.3.6). Notice that there are no header words in this area. The type and data bits of the three slots are as follows; all words have presence bits **read-only**.

Code Pointer An unsigned integer giving the address in instruction memory (the IP) of the first word of the code block (recall that the code block itself is in **code-block** area; as this is a broadcast area the IP is the same for all processors). The type bits are **signed**.

Frame Size An unsigned integer giving the size of activation frames created for invocations of this code block. The type bits are **signed**.

Name An object reference to a symbol giving the name of the code block, as assigned by the compiler.

In the illustration, the code block name is **AUTOMOBILE** and the frame size is 18.

5.4.3 Identifier Descriptors

Each top-level identifier in an Id program has a corresponding identifier descriptor. An identifier descriptor is a four word data structure, whose slots are as follows (all presence bits are **read-only**):

Value An object reference to the value of the identifier, with the appropriate type bits.

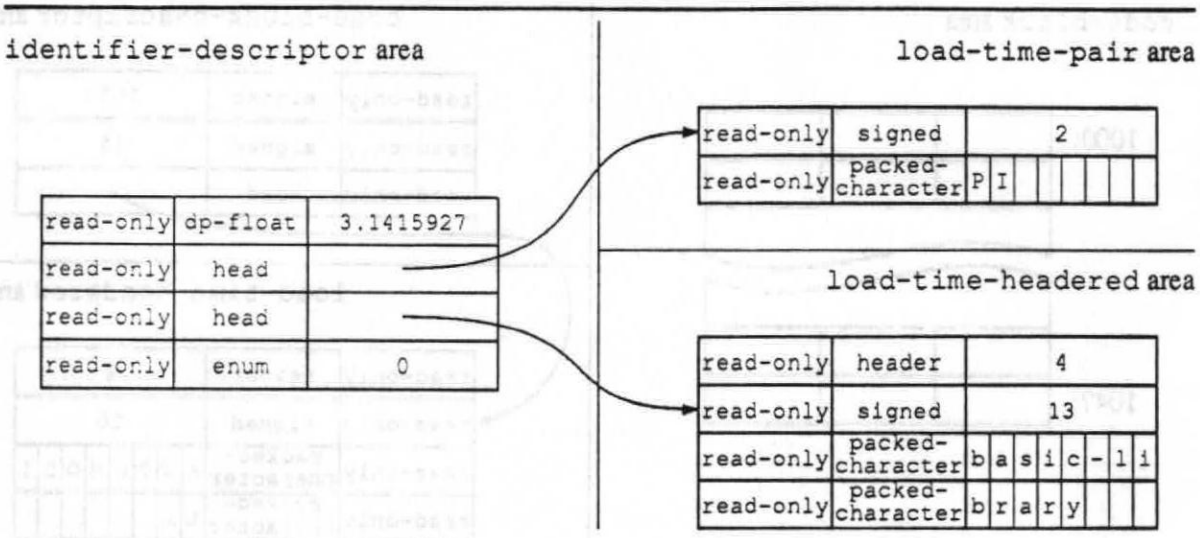

Figure 5.7: An Identifier Descriptor for a Compile-time Constant

Name An object reference to a symbol giving the identifier's name.

Filename An object reference to a string giving the fully qualified name of the MOC file where the identifier was defined. The compiler will normally generate MOC such that all identifiers from the same file defined in the same object module will share this string. Note that the Execution Manager can use the MOC file name to derive the name of the associated exsym file, which will have source pointers for each identifier as well as the name of the MOC file, as a cross-check.

Thunk This slot holds an object reference to the thunk when the identifier is defined by an expression computed at run time. See the discussion below.

Identifier descriptors generated by the Id Compiler fall into one of three cases. The first is for an identifier defined as a compile time constant, for example:

```
def pi = 3.1415927;
```

In this case, the *value* slot of the identifier descriptor just holds an object reference, and the *thunk* slot is `nil`, as illustrated in Figure 5.7.

The second case is an identifier defined as a top-level function, for example:

```
def quad a b c = {...};
```

Again the *value* slot holds an object reference to the value of quad, and the *thunk* slot is `nil`. The value of quad, in this case, is a closure of code block quad over zero arguments. See Figure 5.8.

The last case is an identifier defined as an arbitrary expression that is not a constant at compile time. For example:

```
def intvec =
  {vector (1,10)
     | [i] = i || i <- 1 to 10};
```

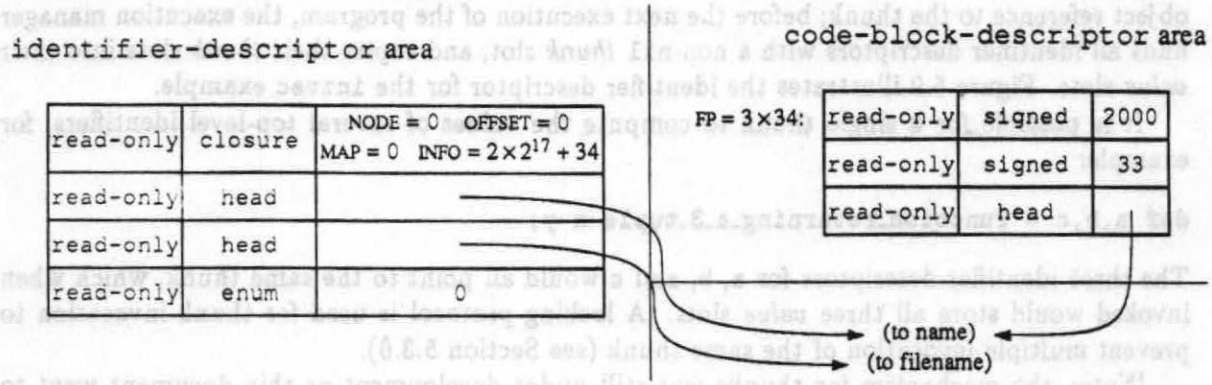

Figure 5.8: An Identifier Descriptor for a Top-level Function

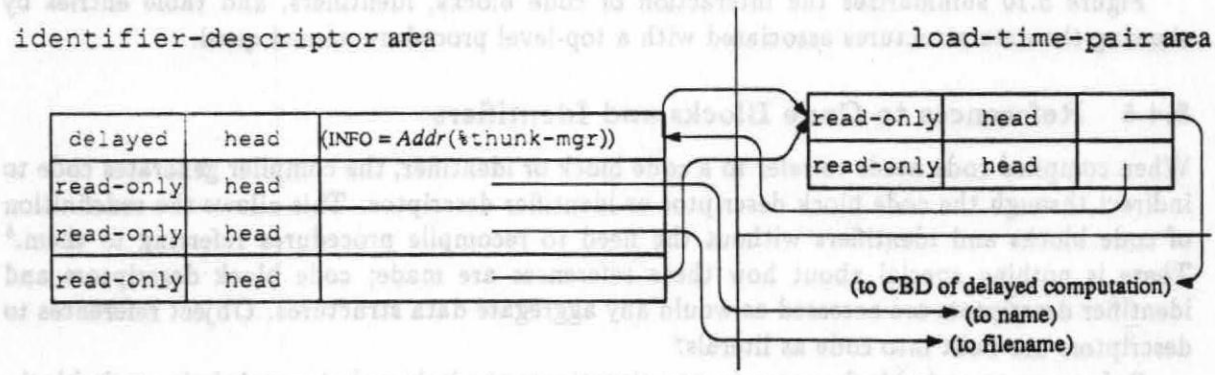

Figure 5.9: An Identifier Descriptor for an Arbitrary Expression (Before Execution)

Initially, the *value* slot holds an object reference to a thunk for the expression. When the first attempt to fetch the value of `intvec` is made, the thunk's code block will be invoked, which will ultimately store the value of `intvec` back into the *value* slot of the identifier descriptor. Before the next execution of the program, the *value* slot must somehow be reset to contain an object reference to the thunk. The *thunk* slot of the identifier descriptor, therefore, always holds an object reference to the thunk; before the next execution of the program, the execution manager finds all identifier descriptors with a non-`nil` *thunk* slot, and copies their *thunk* slots into their *value* slots. Figure 5.9 illustrates the identifier descriptor for the `intvec` example.

It is possible for a single thunk to compute the values of several top-level identifiers, for example:

```
def a,b,c = function_returning_a_3_tuple x y;
```

The three identifier descriptors for `a`, `b`, and `c` would all point to the same thunk, which when invoked would store all three *value* slots. A locking protocol is used for thunk invocation to prevent multiple invocation of the same thunk (see Section 5.3.6).

[Note: the mechanism for thunks was still under development as this document went to press. The above description may not be accurate. Please check for any addenda that have been issued.]

5.4.4 The Code Block and Identifier Tables

Two loader tables (see Section 4.3) are maintained in the `dynamic-link-table` area. The *code block table* maps symbols to code block descriptors, while the *identifier table* maps symbols to identifier descriptors. Each entry in the code block table consists of two words: the first is an object reference to a symbol, and the second an object reference to the corresponding code block descriptor. The identifier table is similarly structured.

The code block and identifier tables are primarily kept for the benefit of the debugger, although the execution manager also needs the identifier table so that it can find all identifier that need their thunks reset.

Figure 5.10 summarizes the interaction of code blocks, identifiers, and table entries by showing the data structures associated with a top-level procedure named `quad`.

5.4.5 References to Code Blocks and Identifiers

When compiled code needs to refer to a code block or identifier, the compiler generates code to indirect through the code block descriptor or identifier descriptor. This allows the redefinition of code blocks and identifiers without the need to recompile procedures referring to them.[5] There is nothing special about how these references are made; code block descriptors and identifier descriptors are accessed as would any aggregate data structures. Object references to descriptors are built into code as literals.

References to code blocks occur in two situations: the linkage between inferior code blocks (code blocks split from the main code block for a procedure because they contain loops, or for other reasons), and the optimized calling convention when a known procedure is applied to a complete set of arguments. In both cases, the caller must read the code block descriptor to obtain the IP of the code block, and to obtain the activation frame size. The caller will

[5] That is, unless the compiler has performed certain optimizations which bypass the fully dynamic linking convention. For example, the compiler may take advantage of the type or arity of procedure `f` when compiling procedure `g` which refers to `f`; in that case `g` would need to be recompiled if the type or arity of `f` changed.

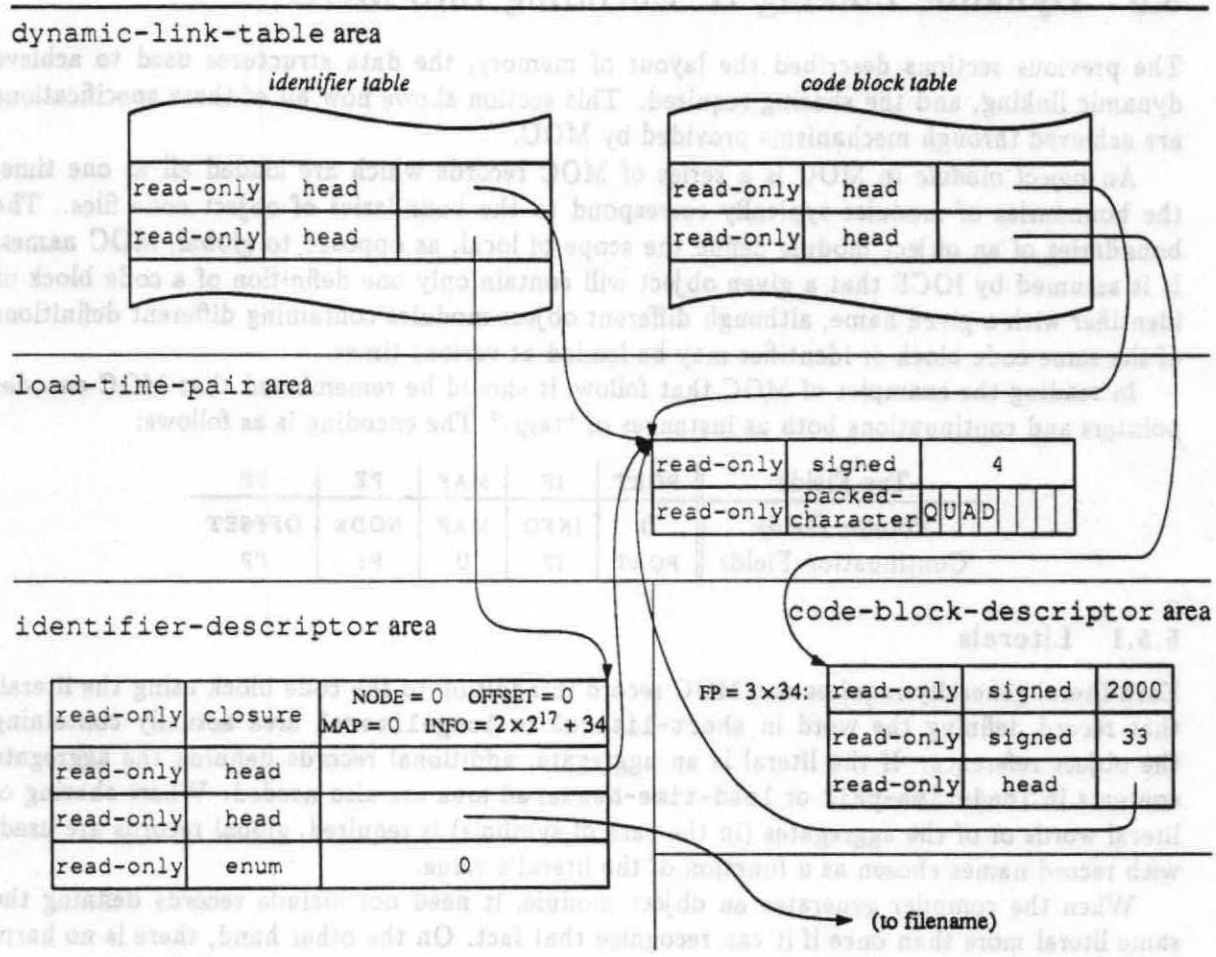

Figure 5.10: Code Block Descriptor, Identifier Descriptor, and Table Entries for a Top-Level Procedure

have a literal that is an object reference to the code block descriptor, and will simply fetch the descriptor's first and second words.

References to identifiers occur when a top-level variable appears as part of an expression. The referencing code block has a literal that is an object reference to the code block descriptor, and fetches the descriptor's first word to obtain the value. It is completely transparent to that code block whether the descriptor contained an actual object or a thunk.

5.5 Dynamic Linking II: Encoding Into MOC

The previous sections described the layout of memory, the data structures used to achieve dynamic linking, and the sharing required. This section shows how all of these specifications are achieved through mechanisms provided by MOC.

An *object module* in MOC is a series of MOC records which are loaded all at one time; the boundaries of modules typically correspond to the boundaries of object code files. The boundaries of an object module define the scope of local, as opposed to global, MOC names. It is assumed by IOCF that a given object will contain only one definition of a code block or identifier with a given name, although different object modules containing different definitions of the same code block or identifier may be loaded at various times.

In reading the examples of MOC that follow, it should be remembered that MOC encodes pointers and continuations both as instances of "tags." The encoding is as follows:

Tag Fields	PORT	IP	MAP	PE	FP
Pointer Fields	0	INFO	MAP	NODE	OFFSET
Continuation Fields	PORT	IP	0	PE	FP

5.5.1 Literals

Each literal generally requires one MOC record in addition to the code block using the literal, that record defining the word in `short-literal` or `long-literal` area actually containing the object reference. If the literal is an aggregate, additional records defining the aggregate contents in `load-time-pair` or `load-time-headered` area are also needed. Where sharing of literal words or of the aggregates (in the case of symbols) is required, global records are used, with record names chosen as a function of the literal's value.

When the compiler generates an object module, it need not include records defining the same literal more than once if it can recognize that fact. On the other hand, there is no harm in including a literal record every time a reference is made. It is not permissible, however, for an object module to refer to a literal that it does not define itself, as there is no guarantee that some other module will eventually define that literal. The sharing conventions insure that no extra storage is consumed if the same literal is defined in two different object modules (for shared literals).

The record defining the literal word will always be one word long, and will have a name chosen according to the rules in Figure 5.11. Unshared literals are defined by local records, and their names are simply chosen so as not to conflict with other records in the same object module. Shared literals, however, are defined by global records, and their names are derived as a function of the literal's value. This causes repeated definitions of the same literal to occupy the same storage.

The records defining the contents of aggregates are almost always local records, the sole exception being the records defining the contents of symbols. Records defining the contents of

Datatype	*Record Type*	*Record Name*	
Character	Global	`LLIT$CHARACTER$`*x*	*x* is the eight bits of the character, as a two-digit hexidecimal string.
Float	Global	`LLIT$FLOAT$`*x*	*x* is the 64 data bits of the float (IEEE double precision representation), as a 16-digit hexidecimal string.
Integer	Global	`LLIT$INTEGER$`*x*	*x* is the 64 data bits of the integer (two's complement representation), as a 16-digit hexidecimal string.
Boolean	Global	`LLIT$BOOLEAN$`*x*	*x* is either **TRUE** or **FALSE** according to the value of the boolean.
Void	Global	`LLIT$VOID`	There is only one void.
Continuation	Local	`LLIT$`*x*	*x* is a serial number unique to the current object module.
String	Local	`LLIT$`*x*	*x* is a serial number unique to the current object module.
Symbol	Global	`LLIT$SYMBOL$`*x*	*x* is the name of the symbol.
Array	Local	`LLIT$`*x*	*x* is a serial number unique to the current object module.
0-ary Disjunct	Global	`LLIT$ENUM$`*x*	*x* is the 64 data bits of the enumeration tag, as a 16-digit hexidecimal string.
n-ary Disjunct	Local	`LLIT$`*x*	*x* is a serial number unique to the current object module.
Code Block Desc.	Global	`LLIT$CBD$`*x*	*x* is the name of the code block.
Identifier Desc.	Global	`LLIT$ID$`*x*	*x* is the name of the identifier.

Note: The records defining short literals have names beginning with **SLIT** rather than **LLIT**.

Figure 5.11: Rules for Naming Literal Records

symbols have record name SYMBOL$x, where x is the name of the symbol. All other aggregate records have record name HEAP$x, where x is a serial number unique to the current object module.

Various literals as encoded into MOC are illustrated below.

Shared Immediate Literal

Here is the MOC code for a code block with the integer literal 6847, as a long literal:

```
GLOBAL CB$EXAMPLE Area = CODE-BLOCK
n words
     ...
  34 INST Opcode = const-fetch F1 = 0 F2 = 0
     ...
m references
  34 ADDR LLIT$INTEGER$0000000000001ABF INSTRUCTION-LONG-R

GLOBAL LLIT$INTEGER$0000000000001ABF Area = LONG-LITERAL
1 words          Presence    Type              Data
  0 INTEGER      read-only   signed  6847
0 references
```

Short literals are similar, except that the area is short-literal, the record name of the literal begins with SLIT instead of LLIT, and of course the reference expression for the instruction is different (this case is also illustrated in Figure 5.5):

```
GLOBAL CB$EXAMPLE Area = CODE-BLOCK
n words
     ...
  34 INST Opcode = const-fetch F1 = [dest] F2 = 0
     ...
m references
  34 ADDR SLIT$INTEGER$0000000000001ABF INSTRUCTION-R

GLOBAL SLIT$INTEGER$0000000000001ABF Area = SHORT-LITERAL
1 words          Presence    Type              Data
  0 INTEGER      read-only   signed  6847
0 references
```

Symbol Literals

Here is a code block with a long literal whose value is the symbol AUTOMOBILE:

```
GLOBAL CB$EXAMPLE Area = CODE-BLOCK
n words
     ...
  68 INST Opcode = const-fetch F1 = 0 F2 = 0
     ...
m references
  68 ADDR LLIT$SYMBOL$AUTOMOBILE INSTRUCTION-LONG-R
```

```
GLOBAL LLIT$SYMBOL$AUTOMOBILE Area = LONG-LITERAL
1 words     Presence   Type                    Data
  0 TAG     read-only  head  Port = R IP = 0 Map = 0 PE = 0 FP = 0
1 references
  0 PTR SYMBOL$AUTOMOBILE TAG-PTR

GLOBAL SYMBOL$AUTOMOBILE Area = LOAD-TIME-HEADERED
4 words     Presence        Type                    Data
  0 BITS    read-only       header                  4
  1 BITS    read-only       signed                  10
  2 BITS    read-only       packed-character        2^56 char-code(A) + 2^48 char-code(U) + ···
  3 BITS    read-only       packed-character        2^56 char-code(L) + 2^48 char-code(E)
0 references
```

Code Block Descriptor Literals

Here is a code block with a reference to some other code block via its descriptor, as a long literal:

```
GLOBAL CB$EXAMPLE Area = CODE-BLOCK
n words
      ...
  47  INST  Opcode = const-fetch F1 = 0 F2 = 0
      ...
m references
  47  ADDR LLIT$CBD$AUTOMOBILE INSTRUCTION-LONG-R

GLOBAL LLIT$CBD$AUTOMOBILE Area = LONG-LITERAL
1 words     Presence   Type                    Data
  0 TAG     read-only  head  Port = R IP = 0 Map = 0 PE = 0 FP = 0
1 references
  0 PTR CBD$AUTOMOBILE TAG-PTR
```

The MOC defining the code block descriptor itself is discussed in Section 5.5.2.

Identifier Descriptor Literals

Here is a code block with a reference to an identifier via its descriptor, as a long literal:

```
GLOBAL CB$EXAMPLE Area = CODE-BLOCK
n words
      ...
  23  INST  Opcode = const-fetch F1 = 0 F2 = 0
      ...
m references
  23  ADDR LLIT$ID$AUTOMOBILE INSTRUCTION-LONG-R

GLOBAL LLIT$ID$AUTOMOBILE Area = LONG-LITERAL
1 words     Presence   Type                    Data
  0 TAG     read-only  head  Port = R IP = 0 Map = 0 PE = 0 FP = 0
1 references
  0 PTR ID$AUTOMOBILE TAG-PTR
```

The MOC defining the identifier descriptor itself is discussed in Section 5.5.2.

Unshared Aggregate Literals

Here is a code block with the literal list `1:2:nil` (this is illustrated in Figure 5.5):

```
GLOBAL CB$EXAMPLE Area = CODE-BLOCK
n words
    ...
  47  INST   Opcode = const-fetch F1 = 0 F2 = 0
    ...
m references
  47  ADDR LLIT$2691 INSTRUCTION-LONG-R

LOCAL LLIT$2691 Area = LONG-LITERAL
1 words     Presence   Type                    Data
  0  TAG   read-only   head   Port = R IP = 0 Map = 0 PE = 0 FP = 0
1 references
  0  PTR HEAP$2692 TAG-PTR


LOCAL HEAP$2692 Area = LOAD-TIME-PAIR
2 words          Presence    Type                      Data
  0  INTEGER    read-only   signed   1
  1  TAG        read-only   head     Port = R IP = 0 Map = 0 PE = 0 FP = 0
1 references
  1  PTR HEAP$2693 TAG-PTR


LOCAL HEAP$2693 Area = LOAD-TIME-PAIR
2 words          Presence    Type     Data
  0  INTEGER    read-only   signed   2
  1  BITS       read-only   enum     0
0 references
```

5.5.2 Code Blocks

The MOC records for the code block shown in Figure 5.6 along with its associated descriptor and table entry are illustrated below.

First, the code block itself. The code block is a local record, because if it is redefined it might be a different size, and therefore cannot overwrite the previous definition. The name is always `CB$`x, where x is the name of the code block; this cannot conflict with any other name in the same object module because we assume that code blocks are not redefined within the same object module.

```
LOCAL CB$AUTOMOBILE Area = CODE-BLOCK
48 words
  0  INST   Opcode = plus F1 = 7 F2 = 10
    ...
0 references
```

The code block descriptor. This is a global record, and furthermore aligned on a multiple of three boundary. The alignment is necessary because of the representation of closures. It is a global record to achieve dynamic linking of references to code blocks, which indirect through the descriptors. Its name is always `CBD$`x, where x is the name of the code block.

```
ALIGNED-GLOBAL CBD$AUTOMOBILE Area = CODE-BLOCK-DESCRIPTOR Align = 3
6 words     Presence     Type                    Data
  0  BITS   read-only    signed   0
  1  BITS   read-only    signed   18
  2  TAG    read-only    head     Port = r IP = 0 Map = 0 PE = 0 FP = 0
2 references
  0  ADDR CB$AUTOMOBILE UNSIGNED 32 0
  2  PTR SYMBOL$AUTOMOBILE TAG-PTR
```

The table entry record. The name of the record is CBTE$$x$, where x is the name of the code block, though this is not terribly important.

```
TABLE-ENTRY CBTE$AUTOMOBILE Table = CODE-BLOCK-TABLE
2 words     Presence     Type                    Data
  0  TAG    read-only    head     Port = r IP = 0 Map = 0 PE = 0 FP = 0
  1  TAG    read-only    head     Port = r IP = 0 Map = 0 PE = 0 FP = 0
2 references
  0  PTR SYMBOL$AUTOMOBILE TAG-PTR
  1  PTR CBD$AUTOMOBILE TAG-PTR
```

The contents of the symbol AUTOMOBILE. This would not have to be included if it already appeared elsewhere in the same object module.

```
GLOBAL SYMBOL$AUTOMOBILE Area = LOAD-TIME-HEADERED
4 words     Presence        Type                  Data
  0  BITS   read-only     header                4
  1  BITS   read-only     signed                10
  2  BITS   read-only     packed-character      2^56 char-code(A) + 2^48 char-code(U) + ···
  3  BITS   read-only     packed-character      2^56 char-code(L) + 2^48 char-code(E)
0 references
```

The illustration above does not show any literals associated with the code block, but those records would of course be included, too.

Identifiers

The MOC records for the identifier shown in Figure 5.7 along with its associated descriptor and table entry are illustrated below.

The identifier descriptor. It is a global record to achieve dynamic linking of references to identifiers, which indirect through the descriptors. Its name is ID$$x$, where x is the name of the identifier.

```
GLOBAL ID$PI Area = IDENTIFIER
4 words     Presence     Type                    Data
  0  FLOAT  read-only    dp-float  3.1415927
  1  TAG    read-only    head      Port = r IP = 0 Map = 0 PE = 0 FP = 0
  2  TAG    read-only    head      Port = r IP = 0 Map = 0 PE = 0 FP = 0
  3  BITS   read-only    enum      0
2 references
  1  PTR SYMBOL$PI TAG-PTR
  2  PTR HEAP$1122 TAG-PTR
```

The contents of the symbol PI. This would not have to be included if it already appeared elsewhere in the same object module.

```
GLOBAL SYMBOL$PI Area = LOAD-TIME-PAIR
2 words     Presence        Type                Data
  0 BITS    read-only       signed              2
  1 BITS    read-only       packed-character    2^56 char-code(P) + 2^48 char-code(I) + ···
0 references
```

The table entry record. The name of the record is **IDTE$**$x$, where x is the name of the code block, though this is not terribly important.

```
TABLE-ENTRY IDTE$PI Table = IDENTIFIER-TABLE
2 words     Presence    Type        Data
  0 TAG     read-only   head    Port = r  IP = 0  Map = 0  PE = 0  FP = 0
  1 TAG     read-only   head    Port = r  IP = 0  Map = 0  PE = 0  FP = 0
2 references
  0 PTR SYMBOL$PI TAG-PTR
  1 PTR ID$PI TAG-PTR
```

The contents of the string "basic-library". This would normally be shared among all identifiers in the object module, and so would only appear once. It is a local record, however, and so is not shared across modules.

```
LOCAL HEAP$1122 Area = LOAD-TIME-HEADERED
4 words     Presence        Type                Data
  0 BITS    read-only       header              4
  1 BITS    read-only       signed              13
  2 BITS    read-only       packed-character    2^56 char-code(b) + 2^48 char-code(a) + ···
  3 BITS    read-only       packed-character    2^56 char-code(b) + 2^48 char-code(r) + ···
0 references
```

The MOC records for the identifier shown in Figure 5.8 are pretty much the same as shown above, except that a rather unusual reference method is needed to construct the closure which occupies the *value* slot. The record for the identifier descriptor is as follows:

```
GLOBAL ID$QUAD Area = IDENTIFIER
4 words     Presence    Type        Data
  0 TAG     read-only   closure  Port = r  IP = 2 × 2^17  Map = 0  PE = 0  FP = 0
  1 TAG     read-only   head     Port = r  IP = 0  Map = 0  PE = 0  FP = 0
  2 TAG     read-only   head     Port = r  IP = 0  Map = 0  PE = 0  FP = 0
  3 BITS    read-only   enum     0
3 references
  0 FP CBD$FOO FP CODE-BLOCK-AREA MINUS NUMBER 3 FLOOR UNSIGNED 17 32
  1 PTR SYMBOL$QUAD TAG-PTR
  2 PTR HEAP$1122 TAG-PTR
```

The first word in the data section defines the "arguments remaining" subfield of the closure's IP field, while the reference expression for that word computes the code block number from the descriptor's address and stores it in the "cbname" subfield.

The other records that would accompany this identifier are not shown.

The MOC records for the identifier shown in Figure 5.9 are also similar to what is shown above, except that the *value* and *thunk* slots both point to a thunk.

```
GLOBAL ID$INTVEC Area = IDENTIFIER
4 words     Presence    Type                    Data
  0  TAG    read-only   head   Port = r IP = 0 Map = 0 PE = 0 FP = 0
  1  TAG    read-only   head   Port = r IP = 0 Map = 0 PE = 0 FP = 0
  2  TAG    read-only   head   Port = r IP = 0 Map = 0 PE = 0 FP = 0
  3  TAG    read-only   head   Port = r IP = 0 Map = 0 PE = 0 FP = 0
6 references
  0  PTR HEAP$1534 TAG-PTR
  0  ADDR %THUNK-MGR TAG-IP
  1  PTR SYMBOL$QUAD TAG-PTR
  2  PTR HEAP$1535 TAG-PTR
  3  PTR HEAP$1534 TAG-PTR
  3  ADDR %THUNK-MGR TAG-IP
```

Notice the reference methods used to get the proper IP field for the thunk references. The MOC for the thunk:

```
LOCAL HEAP$1534 Area = LOAD-TIME-PAIR
2 words     Presence    Type                    Data
  0  TAG    read-only   head       Port = r IP = 0 Map = 0 PE = 0 FP = 0
  1  TAG    read-only   internal   Port = r IP = 0 Map = 0 PE = 0 FP = 0
2 references
  0  PTR CBD$INTVEC-THUNK-0 TAG-PTR
  1  PTR ID$INTVEC TAG-PTR
```

Notice that the second location is an internal pointer naming the first word of the identifier. The first location is an object reference to the code block descriptor for the code block computing intvec's value. The MOC for that code block is not shown.

[Note: the mechanism for thunks was still under development as this document went to press. The above description may not be accurate. Please check for any addenda that have been issued.]

5.5.3 Summary of Record Names

The following table summarizes the conventions on record names established by IOCF:

Record Name	*Record Type*	Use
LLIT$*x*	Local/Global	Long literal—see Figure 5.11
SLIT$*x*	Local/Global	Short literal—see Figure 5.11
SYMBOL$*x*	Global	Contents of symbol named *x*.
CB$*x*	Global	Code block named *x*.
CBD$*x*	Global	Code block descriptor for code block named *x*.
ID$*x*	Global	Identifier descriptor for identifier named *x*.
CBTE$*x*	Local	Table entry into the code block table for code block named *x*.
IDTE$*x*	Local	Table entry into the identifier table for identifier named *x*.
HEAP$*x*	Local	Aggregate contents; *x* is a serial number unique within the current object module.

5.6 Layout of Static Constants

The static-constants area contains short literals whose existence is pre-defined, such as area numbers for important areas, manager entry points, *etc.* The value of some of these literals will be defined before any code is run on Monsoon. All of these literals have presence bits read-only.

5.6.1 PMM Constants

The PMM will reside in its own area on processor 0. The address of the first word of the PMM is stored in location 0 on that processor.

pmm-address [*Static Constant*]
This literal contains the FP of the first word of the PMM data structure, with type bits signed. The definition of the PMM requires this literal to be at location 0 of PE 0's data memory.

pmm-pointer [*Static Constant*]
This is similar to pmm-address but is a Monsoon pointer to the first word of the PMM, rather than just the FP. Unlike pmm-address, this constant is initialized when all the other constants are initialized, not when the PMM is established.

5.6.2 Area Numbers

The following literals contain the area numbers of important areas. All of them have presence bits read-only and type bits signed.

pmm-area-number [*Static Constant*]
loader-internal-area-number [*Static Constant*]
static-constants-area-number [*Static Constant*]
static-code-area-number [*Static Constant*]

`code-block-area-number` [*Static Constant*]
`short-literal-area-number` [*Static Constant*]
`long-literal-area-number` [*Static Constant*]
`code-block-descriptor-area-number` [*Static Constant*]
`identifier-descriptor-area-number` [*Static Constant*]
`dynamic-link-table-area-number` [*Static Constant*]
`load-time-pair-area-number` [*Static Constant*]
`load-time-headered-area-number` [*Static Constant*]
`run-time-pair-area-number` [*Static Constant*]
`run-time-headered-area-number` [*Static Constant*]
`static-frame-area-number` [*Static Constant*]
`frame-area-number` [*Static Constant*]

5.6.3 Configuration Parameters

`my-pe-number` [*Static Constant*]

The value of `my-pe-number` is the PE number of the processor in whose data memory this word resides. Note that unlike all other static constants, the value is different on each PE that is part of the `static-constants` broadcast area.

5.6.4 Dynamic Link Table Pointers

`code-block-table-pointer` [*Static Constant*]
`identifier-table-pointer` [*Static Constant*]

The `code-block-table-pointer` constant holds a Monsoon pointer to the first word of the code block table, contained in the `dynamic-link-table` area. `Identifier-table-pointer` is analogous, but points to the identifier table.

5.6.5 Manager Entry Points

These literals contain the entry points (IP values) for various manager operations. All have presence bits `read-only` and type bits `signed`.

`boot-code-block-entry-point` [*Static Constant*]

This literal contains the IP of the entry point of the boot code block. The execution manager will presumably start execution by dropping one or more tokens targeted at the boot code block. The arguments will have been stored as constants in the boot frame (a static frame). The results and termination signal will also be stored in the boot frame when the procedure finishes.

`context-initialization-entry-point` [*Static Constant*]
`get-context-entry-point` [*Static Constant*]
`return-context-entry-point` [*Static Constant*]

Before invoking a procedure, the execution manager will invoke the context initialization procedure, which will set up the run time frame area. During program execution, frames will be allocated and deallocated by making calls to `get-context` and `return-context`. These managers also may provide hooks for the debugger (*e.g.*, `trace`).

`thunk-mgr-entry-point` [*Static Constant*]

This contains the entry point to the thunk manager, which is where a thunk is sent when it is forced. This manager will allocate a context in which the thunk body will execute.

`pair-initialization-entry-point` [*Static Constant*]
`allocate-pair-entry-point` [*Static Constant*]
`clear-pair-entry-point` [*Static Constant*]
`deallocate-pair-entry-point` [*Static Constant*]

`headered-initialization-entry-point` [*Static Constant*]
`allocate-headered-entry-point` [*Static Constant*]
`clear-headered-entry-point` [*Static Constant*]
`deallocate-headered-entry-point` [*Static Constant*]

These literals contain the entry points to the manager procedures that initialize the run-time pair and headered storage areas, and allocate, clear and deallocate from the run-time pair and headered storage areas.

5.6.6 Manager Parameters

These literals are used by the managers when performing actions such as allocating contexts, heap storage, *etc.* These should be static literals so that they may be changed by the execution manager.

`n-initial-frames` [*Static Constant*]
`fixed-size-frame-size` [*Static Constant*]
`frame-area-address` [*Static Constant*]

`n-initial-frames` tells the context-initialization routine how many frames to allocate. It will put this many frames onto the frame-free-list, each of size `fixed-size-frame-size`.

`run-time-pair-area-address` [*Static Constant*]
`run-time-headered-area-address` [*Static Constant*]

These literals contain the addresses of the run-time pair and headered storage areas.

5.7 Frame Layout and Calling Conventions

In order to support the use of functions as first-class objects in ID, as well as to facilitate debugging, the generated code for a procedure must obey certain constraints, loosely known as the *Id calling convention.* These constraints include the use of the temporary frame storage, references to global values, and the manner in which arguments are passed in and results passed out. A code block has several *inputs*, sometimes referred to as the *arguments*, as well as several *outputs* or *return values.*

5.7.1 Frame Layout

For the storage of values temporarily during execution, each procedure has access to a portion of local memory called the *frame.* The amount of frame storage used by a given procedure is fixed at compile time, and is allocated by the calling procedure. The normal mechanism is

to allocate a fixed size frame; unusually large procedures thus use the standard frame storage temporarily while they allocate their own sufficiently large frame storage.

At offset 0 in every frame is stored the *return address*, to which results are returned (see below). This is the first location written and the last location cleared in any frame. Thus, an active frame is easy to identify and link into the call tree.

Frame slots may be distinguished by the matching mode used on them; that is, by the values their presence bits may take on as execution progresses. Dyadic slots take on the presence values `empty`, `left-present`, and `right-present`. Constant slots take on the values `empty` and `read-only`. "Count-down" slots take on any of the eight possible values; the presence bits for those slots are used as a 3-bit counter. At some point in the future, we may allocate data structures directly in frames as opposed to the heap; in that case, some frame slots may have I-structure presence values (see Section 5.3.4). The constant slots always precede all other frame slots except slot 0.

5.7.2 Parameter Passing Conventions

To call a known procedure using the standard calling convention, we need to know the number of inputs and outputs, as well as the address of the code block. The number of inputs and outputs is known at compile time; the address of the code block is found by consulting the code block descriptor. The following steps are then taken:

1. A *frame* (probably fixed size, see above) is obtained for the storage of temporaries local to the called procedure.
2. The address of this procedure and the frame are combined to produce a *context* C.
3. A *return address* R is generated, to which the returned values will be sent.
4. The return address is sent to C.
5. Argument i ($1 \leq i \leq n$, where n is the number of arguments expected by this procedure) is sent to *Adjust-Offset*(C, i), where *Adjust-Offset*(C, i) is the context obtained by adding i to the IP of C.

When the procedure returns, the m values returned are sent to *Adjust-Offset*(R, j), for $1 \leq j \leq m$. In addition, when the called procedure is finished with the temporary frame storage, it returns a context to R. In general, however, the context returned to R *may not be the same* as the original context C.

This is the standard calling convention; certain code blocks may have non-standard calling conventions. These conventions have yet to be fully determined, and will be documented later.

5.8 Debugging Information

[This section is omitted until a strategy for source-level debugging can be devised. It will explain what additional information is placed in code blocks, activation frames, and descriptors to aid the source-level debugger in presenting a memory image in a way related to the source text.]

5.9 Bootstrapping Monsoon

This section describes how a freshly powered-up single-node Monsoon is brought to the point where user Id programs may be loaded and run. The process for bringing up a multi-node Monsoon is essentially the same, but may include some initialization of the token network which is yet to be determined.

The bootstrap process has three main phases. At the start of the bootstrap phase, only the micro-architecture has been defined (by the actual circuitry of the processor); there is no instruction set, or access to the machine in a micro-architecture-independent way. The bootstrap process then proceeds as follows:

Macro-Architecture I This phase is responsible for transforming the hardware into a working instruction set dataflow architecture. At the end of this phase, microcode has been loaded (thus defining an instruction set) and the VME hardware interface initialized. The MMI is operative in bootstrap mode, providing access (albeit somewhat inefficient) to the macro-architecture, or instruction set architecture. The loader may be used in restricted mode (no use of symbols).

Macro-Architecture II This phase is responsible for setting up the infrastructure needed to allocate machine resources at the PMM level (*i.e.*, at a very gross level), and to make full use of the capabilities of the loader. At the end of this phase, the PMM data structure is initialized, as are the loader's internal tables. The MMI is operative in normal mode, providing efficient access to the macro-architecture. If the MMI's normal mode involves code actually running on Monsoon, then that code has been loaded and is running.

Id This phase is responsible for loading and initializing all the support for the Id programming language, including the resource managers. At the end of this phase, all areas and other data structures specified by IOCF have been created and initialized, and the Id Run Time System (resource managers) has been loaded and initialized. At this point, the user may load and run ordinary Id programs.

The Macro-Architecture I phase is always required to use Monsoon. The Macro-Architecture II phase is always required if the PMM and MOC loader are to be used—we expect that any use of Monsoon would want these facilities, but it is possible to imagine a software system that does not. Perhaps a diagnostic program would not need the PMM or full use of the loader, and could be loaded and run without going through the Macro-Architecture II phase. The Id phase is specific to Id, and is the only phase that is influenced by the conventions in IOCF. If some other programming language environment is to be used with Monsoon, some other bootstrapping phase would replace the Id phase.

Because the MINT emulator does not mimic the micro-architecture, the Macro-Architecture I phase is different for MINT. Instead of loading microcode and VME registers, this phase consists of calling the initialization function provided by MINT, establishing upper limits on the amount of emulated memory, and other like parameters. The end result of the Macro-Architecture I phase is the same, however: the emulated machine has an instruction set, the MMI is operative in bootstrap mode, and the loader may be used in restricted mode. The Macro-Architecture II and Id phases are also performed on MINT, and have exactly the same purpose and implementation as for the hardware.

The following sections describe in detail the steps taken during each phase of bootstrapping. Diagnostics are excluded from the discussion: although it is likely that one or more phases will include running diagnostics, all the details have yet to be determined.

5.9.1 The Macro-Architecture I Phase

Prior to the Macro-Architecture I phase, the machine has been freshly powered up and reset. No microcode or other machine state has been initialized.

Load Microcode and VME Registers The first-level decode, type map, presence map, second-level decode, and type propagation maps are loaded from a microcode file (generated by the microcode compiler). Various other machine registers (*e.g.*, the base IP for exception tokens) as well as VME registers (*e.g.*, the interrupt vector for VME interrupts) are also initialized.

While there may be a separate microcode loader program, in all likelihood these tasks will be accomplished by giving a command script to the Scan Path Debugger.

All internal machine state, including microcode, has now been loaded. The machine has a definite instruction set.

Load Static Instructions Proper operation of Monsoon requires that special instructions be loaded into instruction memory at particular IP addresses. These instructions include the Idle instruction, as well as the instructions that emulate I-structure operations, and those that handle the second phases of instructions for reading and writing instruction memory from the instruction stream. All of these instructions fall in the `static-code` area defined by IOCF, although at this point none of the infrastructure associated with areas and IOCF have yet been established.

This task is again accomplished by giving a command script to the Scan Path Debugger.

The machine now conforms completely to the specification of its macro-architecture; it is now truly a Monsoon processor. The MMI is now usable in bootstrap mode, providing a path for manipulating the macro-architecture machine state (data memory, instruction memory, token queues, and statistics registers). While the loader has not yet initialized its symbol table management data structures, it is possible to use the loader in "restricted mode," which allows the loading of absolute data into absolute addresses, with no references to symbols. Because the MMI is still in bootstrap mode, access to this machine state may be somewhat inefficient, as in all likelihood it is accessed exclusively through the scan path at this point.

This completes the Macro-Architecture I phase.

5.9.2 The Macro-Architecture II Phase

Prior to the Macro-Architecture II phase, the Macro-Architecture I phase must have been completed (MINT starts up already in this state).

Initialize PMM The PMM data structure, which describes the available machine resources (*e.g.*, the number and sizes of data memory), and records their gross allocation, is initialized according the the machine configuration. At this point, only one PMM area is established: the area that holds the PMM itself. The address of the PMM has been recorded in location zero of node zero's data memory; this location will eventually become part of the `static-constants` area defined by IOCF, although at this point none of IOCF's areas have been established.

This task may be accomplished by a special PMM initializer program, or simply by loading a file (created by MONASM) using the restricted mode of the loader. If a special

PMM initializer program is used, it may be able to automatically determine some of the sizes of machine resources by probing memory locations; otherwise the information must be precomputed and recorded in file.

At this point, the PMM data structure has been established, making the total machine resources known to other software components, and also recording how those resources have been divided up among the components.

Initialize Loader Tables All tables maintained by the loader for managing symbolic references are initialized. The loader first creates a new PMM area, called the **loader-internal** area, in which it keeps its global symbol table (the table of loader names that persist across invocations of the loader) and other persistent information. After creating the area, it initializes the table and any other data structures it keeps there.

This task is performed by the loader itself, directed to do so by invoking it with a special "initialize" option.

At this point, the loader is fully operational, and may load any arbitrary MOC file.

Load and Initialize MMI Support (This task is omitted if the normal mode of the MMI is the same as its bootstrap mode.) The ultimate version of the MMI will have a normal mode that allows access to machine state while Monsoon is running. This requires some MMI support code, written in MONASM, to be loaded into Monsoon's instruction memory and executing; this code supervises the transactions requested via the MMI. That code is now loaded and initiated.

The code is loaded by the loader, and a special MMI call is made to start the support code on Monsoon and switch the MMI from bootstrap mode to normal mode.

At this point, the PMM and loader are fully initialized; the PMM controls gross allocation of machine resources into areas. Only two areas exist at this point, the **pmm** area and the **loader-internal** area, but the loader now provides commands for establishing new areas. The full capabilities of the loader are available, including the use of symbolic names and the associated dynamic linking mechanisms. The MMI operates in normal mode, providing efficient access to the macro-architecture machine state. Ultimately, normal mode will involve code running on Monsoon to supervise MMI transactions, and this code is up and running at this point.

This completes the Macro-Architecture II phase.

5.9.3 The Id Phase

Prior to the Id phase, the Macro-Architecture II phase has been completed.

Load IOCF Configuration File A MOC file (produced by MONASM) is loaded, creating all remaining areas defined by IOCF. Various static constants defined by IOCF are also initialized at this point (one constant and some static code were already loaded in previous bootstrap phases; this phase must take care to preserve them).

This task is performed by the loader.

At this point, the basic infrastructure defined by IOCF has been established, and all the areas and data structures to support dynamic linking of Id programs have been initialized.

Load Id Run Time System The MOC file (likely produced by the Id Compiler, possibly with some routines from MONASM) containing the object code of the Id Run Time System is loaded. This includes all of the code for the resource managers invoked during execution of a user's Id program, as well as an initialization program to be executed in the next step of the bootstrap.

This task is performed by the loader.

At this point, the code for the Id Run Time System is in place, although its data structures have not been initialized.

Initialize the Run Time System The "initialize" entry point of the Run Time System is invoked, causing all of the data structures associated with the Run Time System to be initialized. In particular, the **run-time-pair**, **run-time-headered**, and **frame** heap areas are set up and cleared in whatever manner is appropriate for the storage managers associated with those areas.

This task is performed by the initialization code of the Run Time System. This code runs on Monsoon itself, and is invoked by the Execution Manager.

At this point, the Id language development system is fully operational, and the user may load and run his own programs.

This completes the Id phase, and the entire bootstrap process.

5.10 Numeric Values of IOCF Constants

Throughout the documentation of IOCF, we have used symbolic names for presence values and the like. We tabulate below the actual numerical values of these names.

5.10.1 I-Structure Presence Bits

These are the values of presence bits that appear in words of heap storage. Their semantics are described in Section 5.3.4.

Name	*Value*
`empty`	0
`present`	2
`read-only`	3
`deferred`	4
`lock-deferred`	5
`delayed`	6
`error`	7

5.10.2 Activation Frame Presence Bits

These are the values of presence bits that appear in "constant" and "dyadic" words of frame store, as discussed in Section 5.7.1.

Name	*Value*
`empty`	0
`left-present`	1
`right-present`	2
`read-only`	3

5.10.3 Type Bits

These are the values of the type bits on object references. The situations in which these arise are discussed in Section 5.3.

The instruction set views the eight type bits as having three subfields:

Type Bits		
MARK	Major Type	Minor Type
1	3	4

IOCF always sets the MARK bit to zero, although in the future it may be used by garbage collectors. The major types and some of the minor types are defined by the instruction set; see the *Monsoon Macro-Architecture Reference Manual.* The values of all minor types used by IOCF are given below (these numbers are seven-bit numbers, as they include both major and minor bits).

Name	*Hexadecimal Value*	*Decimal Value*
`void`	00	0
`signed`	01	1
`boolean`	02	2
`enum`	03	3
`character`	04	4
`packed-character`	05	5
`header`	06	6
`dp-float`	20	32
`internal`	30	48
`external`	31	49
`head`	32	50
`closure`	33	51
`continuation`	40	64

Note that the minor type of continuations is interpreted by the hardware as PRIV and COLOR. All values of type bits not tabulated here are not used by IOCF. This includes the major types SFLT (major type 1) and FUTR (major type 5).

5.10.4 Port Bits

These are the values that can appear in the PORT field of tags and instructions (when an F1 or F2 field is used as a destination).

Name	*Value*
`left`	0
`right`	1

Chapter 6

Statistics Format

This chapter defines the file format for recording raw data derived from instrumented execution of Monsoon, whether a hardware or software implementation. Typically, such a file would be written by the Execution Manager and read and processed later by the Statistics Analyzer/Viewer. The statistics file format is built on top of CIOBL, Version 2, described in Chapter 7. Statistics format requires only a Level I implementation of CIOBL, and primarily uses variable-length objects.

The format provides for two simple kinds of datasets: *tables*, which are ordered pairs of keywords and values, and *profiles*, which are essentially bar graphs. Each table or profile holds data describing a particular measurement, for example, a table of opcodes and their frequency of execution, or a profile of ALU operations versus time. Often, the same kind of measurement is made individually for each processor or memory unit in the system; these are grouped into *table sets* or *profile sets*, as appropriate. The aggregate of table and profile sets from a given execution of a program is called a *run*.

A statistics file consists of a series of runs, each of which is the collection of statistics describing a single execution of a program. There is not necessarily any relationship between the runs found in a given file, although often they represent data from a single experiment, such as executing the same program on different numbers of processors. Each run has a property list, zero or more table sets, and zero or more profile sets. The property list is used to give information about the run as a whole: its name, the time when it took place, *etc.* Each table set or profile set is data resulting from a particular type of measurement. The precise format of run is the property list, followed by the number of table sets, followed by the number of profile sets, followed by the table sets, followed by the profile sets.

```
STAT-FILE  →  {RUN}*
RUN        →  PLIST integer(t) integer(p) {TABLE-SET}^t {PROFILE-SET}^p
PLIST      →  integer(n) {keyword anyvar}^n
```

A table is a set of ordered pairs, where the first element in each pair is a keyword and the second is an integer. No two pairs in a given table will have the same keyword, and the order of the pairs' appearence in the table is immaterial. Tables representing the same measurement on different processors are grouped into sets, and each table set carries a keyword identifying which measurement it represents.

```
TABLE-SET  →  keyword integer(n) {TABLE}^n
TABLE      →  integer(m) {keyword integer}^m
```

A profile is a bar graph: a list of ordered pairs (x_i, y_i), where both x_i and y_i are unsigned integers, and where the x_i are consecutive. Because the x_i are consecutive, only the endpoints x_1 and x_N are encoded into the file, followed by the y_i in order from lowest x_i to highest.

To conserve space, *run length encoding* is used: a sequence of k identical y_i values, where $k \geq 2$, may be replaced by the two integers $-k$ and y_i. The total number of integers required to encode the y's, therefore, may be fewer than N; this number is called M, and is encoded into the file for the convenience of programs which read statistics files. The statistics format does not require that the k's be maximal, although it is obviously desirable.

Profiles representing the same measurement on different processors are grouped into sets, and each profile set carries a keyword identifying which measurement it represents.

PROFILE-SET → **keyword integer**(n) {PROFILE}^{n}
PROFILE → **integer**(x_1) **integer**(x_N) **integer**(M) {INTEGER}^{M}

Chapter 7

CIOBL

CIOBL stands for "Common Input/Output Base Language," and is pronounced "CHO-bul," as if it were an Italian word. It refers to a language for representing various types of data in files, either for communication between two programs or for communication between different invocations of the same program. Because CIOBL makes a file appear to contain objects from a variety of datatype, it provides a higher level of interface than simply character or byte I/O. It is designed to be flexible enough to accommodate three broad categories of files, distinguished by the sorts of objects they contain:

1. Files composed of a series of simple, atomic objects, which have universal meaning across a range of programs and programming languages. Such objects include integers, characters, strings, *etc.* Example: Monsoon Object Code files produced by the Id Compiler. Monsoon object code files are built on a grammar that defines their contents as a series of integers.

2. Files composed of atomic objects along with structured objects, which also have universal meaning across a range of programs and programming languages. The structured objects are lists and arrays of other objects. Example: GITA statistics files. The main parts of these have a particular representation as arrays of data, but arrays are a simple enough data type that they can reasonably be manipulated within any programming language.

3. Files composed of atomic objects, structured objects, and objects which do not have universal meaning across a range of programs and programming languages. Example: Id Compiler internal program graph files. These files contain data structures that are represented by objects defined within the Id Compiler code, and are not meant to be (easily) read by programs other than the Id Compiler.

Throughout this chapter, these three applications will be referred to Category I, Category II, and Category III. CIOBL is a *layered* specification—a given implementation of CIOBL may only provide support up to a given level:

Level I Provides support only for Category I objects.

Level II Provides support only for Category I and Category II objects.

Level III Provides support for all objects of Category I and Category II, along with some objects of Category III. The Category III objects supported will vary from implementation to implementation, as Level III implementations may be customized for a particular application program.

A specification defining a file format using the CIOBL language should state what level of support is required, and if Level III support is required, it should also state which Level III objects are required. For example, Monsoon Object Code format requires only Level I support.

The main feature of CIOBL is that it provides three different *encodings* for files, each useful for different applications. These encodings are:

Standard An encoding which uses only "standard" characters (the 94 printing characters of ASCII, plus space and newline), and which is readable enough to be edited manually by humans.

Compressed An encoding which also uses only standard characters, but uses a variety of tricks to greatly reduce the number of characters required, at the expense of human readability.

Binary An encoding which uses the same compression techniques as the Compressed encoding, but which is composed of 8-bit bytes rather than standard characters. This makes it even more compact than the Compressed encoding.

The Standard and Compressed encodings are useful for transmission over media which only transmit standard characters, such as electronic mail. The Standard encoding is also useful for making manual adjustments to CIOBL files.

The CIOBL software provides a uniform interface between files and programs by defining a set of *CIOBL objects* which can appear in CIOBL files. Programs that do I/O deal only with CIOBL objects, and the CIOBL software attends to the details of how the objects are represented in each of the three encodings. A program need only be aware of the encodings when opening a file, for at that time it must select which of the three encodings is to be used.

Compatibility Note: This chapter describes Version 2 of CIOBL, which is *incompatible* with Version 1 used in the TTDA Id World system. The differences are:

- Fixed-length objects have been added to Version 2.
- Non-generic read and write functions have been added to Version 2.
- The Compressed and Binary encondings of floats has changed.
- The syntax of arrays has changed.
- Run length encoding has been removed from the Compressed and Binary encodings in Version 2 (of course, a format designer may explicity do his own run length encoding).
- The specification of Version 2 has been divided into layers.

7.1 CIOBL objects

A CIOBL file consists of a sequence of CIOBL objects. CIOBL objects are divided into two classes: *fixed-length* and *variable-length*. Variable-length objects are the most commonly used: they include all data types known to CIOBL, including numbers, characters, strings, *etc.* Variable-length objects offer the greatest flexibility as the size of these objects—how many bits in an integer, how many characters in a string, and so forth—is encoded directly into the CIOBL file. A reader of the file, therefore, need only instruct the CIOBL software to read the next object, without knowing how large it is, or even what type of object it is. Variable-length

objects can be subdivided into *atomic* objects and *compound* objects. As the name suggests, atomic objects are indivisible; examples include numbers, characters, strings, and keywords. An atomic object is always read or written through a single call to CIOBL software. In contrast, compound objects are composed of other CIOBL objects; examples include lists and arrays. A compound object may be read or written in one call, or by many calls which read or write its components.

While variable-length objects offer the greatest flexibility, they incur a certain amount of space overhead because length and type information must accompany them in the CIOBL file. Fixed-length objects provide an alternative when greater control of space overhead is required, at the expense of flexibility. The only fixed-length objects are numbers and characters, and when a fixed length object is read or written the CIOBL software must be informed of its size; *e.g.*, how many bits in an integer. Because length and type information does not accompany fixed-length objects, it is the responsibility of the user to read such objects through calls which exactly match the calls which wrote those objects. Fixed-length objects can offer a space savings when many objects of known size are to be represented; for example, a series of 8-bit integers. On the other hand, if a series of numbers is to be written where the largest is 2^{31} but the majority are less than 2^8, variable-length integers will generally be more space-efficient because the space required for length and type information will be offset by the smaller average space for data. Section 7.6 gives information for estimating the space required for CIOBL files.

7.1.1 Variable-Length Objects

The variable-length objects available in Level I implementations of CIOBL are the following:

Integers These are distinct from floats which happen to have integral values.

Floats A "float" is a floating point number. As described later, CIOBL uses a representation which allows arbitrary magnitude and precision.

Characters The only characters allowed in CIOBL files are standard characters, which consist of the 94 printing characters of ASCII plus space and newline. (This restriction to standard characters has nothing to do with the fact that the Standard and Compressed encodings use only standard characters. Instead, it stems from the need to make sure that CIOBL character objects have a representation in all implementations of CIOBL.)

Strings A string is a (possibly empty) character string. The characters of a string are limited to the 96 standard characters.

Keywords A keyword is a name, composed of standard characters.

CIOBL places no practical limit on the magnitude of integers and floats, or on the length of strings.

Keywords seem superficially similar to strings, but are usually used for different purposes. One reason is that an implementation of CIOBL often represents strings and keywords differently: while strings are generally represented as arrays of characters, keywords are usually mapped into addresses or serial numbers, with all keywords with the same name being mapped into the same address or serial number. Strings are often decomposed into their component characters, while keywords rarely are. Instead, the names of keywords are important only insofar as they distinguish between keywords that are the same and those that are different. Finally, the Binary and Compressed encodings of CIOBL have a particularly efficient representation for multiple appearences of the same keyword, while no such effort is expended on strings.

In addition to all objects from Level I, the objects available in Level II are the following:

Lists A list is a (possibly empty) delimited sequence of CIOBL objects (which could themselves be lists). The components of a list need not all be the same type of CIOBL object, although they all must be variable length.

Arrays Like a list, an array is a sequence of CIOBL objects. Unlike a list, an array carries an indication of how many objects it contains. Also, arrays may be multidimensional, up to seven dimensions (there must be at least one dimension). Each dimension has subscripts running from zero up to but not including the size of the dimension. The components of an array need not all be the same type of CIOBL object, although they all must be variable length.

In designing a file format, the choice between lists and arrays is fairly arbitrary. One consideration is that they will generally have different representations within the program which makes CIOBL calls. Another has to do with knowing the number of objects the list or array contains. Lists have the advantage when writing a file that the contents may be written without knowing the length of the list. When reading a file, this aspect may be a disadvantage.

In addition to all objects from Level II, the objects available in a Level III implementation can include any of the following:

Symbols A symbol is an ordered pair of names, where the first name is called the *package* name and the second is just called the "name." In Level III implementations of CIOBL which support symbols, symbols subsume keywords in that a keyword is a symbol whose package name is `KEYWORD`. Symbols in their full generality are primarily used by Common Lisp implementations of CIOBL.

Dotted Lists A dotted list is like a list, but also includes a special object at the end. Dotted lists are mainly used in Common Lisp implementations of CIOBL; they correspond to non-`NIL` terminated lists in Common Lisp.

User Defined Objects Arbitrary compound objects may be defined by user applications.

The exact set of objects supported by a Level III implementation may vary, although they will always include all objects from Level II. This variation is acceptable because Category III files are primarily those read and written by the same program; each such program will generally have a different set of user defined objects corresponding to its own internal data types.

Of the objects described above, lists, arrays, dotted lists, and user defined objects are compound objects. All the others are atomic objects.

7.1.2 Fixed-Length Objects

The fixed-length objects are described below. All fixed-length objects are part of Level I.

Unsigneds An "unsigned" is an integer x in the range $0 \leq x < 2^N$, for some $N > 0$. The value of N must be specified both when writing and reading an unsigned.

Signeds A "signed" is an integer x in the range $-2^{N-1} \leq x < 2^{N-1}$, for some $N > 0$. The value of N must be specified both when writing and reading a signed.

Singles A "single" is an IEEE single precision floating point number.

Doubles A "double" is an IEEE double precision floating point number.

Chars A "char" is a standard character.

The difference between a char (a fixed-length object) and a character (a variable-length object) is only that characters carry type information with them. That is, while characters do not vary in length, they are like other variable-length objects in that they carry type information, so that a call to a generic "read variable-length object" call can distinguish a character from some other type of variable-length object. Chars carry no such information.

Note that the components of compound objects cannot be fixed-length objects. On the other hand, there is nothing preventing a user from using fixed-length objects to construct compound entities; they will just not be objects from CIOBL's point of view.

7.2 CIOBL Tokens and Compound Objects

The contents of a CIOBL file can be viewed from three different levels of abstraction. At the lowest level, a CIOBL file is just a sequence of characters (or 8-bit bytes, in the case of Binary encoding). At the highest level, a CIOBL file is a sequence of CIOBL objects, as previously discussed. At the middle level, a CIOBL file is a sequence of *CIOBL tokens.*

CIOBL tokens are the smallest indivisible components of CIOBL files, and are independent of the encoding chosen. The set of CIOBL tokens consists of all of the fixed-length and atomic variable-length CIOBL objects, as well as some *punctuation tokens* which serve to identify compound objects. Defining a token layer allows the portions of a CIOBL implementation which understand the various encodings to be separated from the portions which understand how to put together and pick apart compound objects.

Compound CIOBL objects are composed from punctuation tokens and variable-length objects. Some of these variable-length objects may themselves be compound objects, and so the syntax of compound objects in terms of tokens is recursive. Ultimately, though, a compound CIOBL object ends up as a series of atomic variable-length tokens and punctuation tokens. The syntax of each type of compound object is described below. The names of punctuation tokens appear in small caps (*e.g.*, LIST-BEGIN).

Lists (Level II) A CIOBL list appears as the token LIST-BEGIN, followed by the CIOBL objects comprising the list (or none if the list is empty), followed by the token LIST-END.

Dotted Lists (Level III) A dotted list appears as LIST-BEGIN, followed by one or more CIOBL objects, followed by LIST-DOT, followed by exactly one CIOBL object, followed by LIST-END.

Arrays (Level II) A CIOBL array appears as the token ARRAY-BEGIN, followed by the rank (an integer), then the dimensions (a series of integers), then the elements of the array (each a variable-length object). There is no terminating punctuation token. The rank of the array is the number of integers in the dimension series; each dimension has subscripts from zero, inclusive, to the integer given in the dimensions, exclusive. The elements of the array appear in row-major order. Here is an example of a 3×2 two dimensional array:

ARRAY-BEGIN 2 3 2 $a_{0,0}$ $a_{0,1}$ $a_{1,0}$ $a_{1,1}$ $a_{2,0}$ $a_{2,1}$

User Defined Objects (Level III) A user-defined object appears as the token USER-DEFINED-BEGIN, followed by a symbol, followed by some number of variable-length CIOBL objects, followed by the token USER-DEFINED-END. The symbol immediately after the USER-DEFINED-BEGIN indicates which user defined object is being represented; when reading such an object, the symbol is used to dispatch to user-written code which reads the remaining objects up to the USER-DEFINED-END.[1] In Level III implementations which do not support full symbols, the object following the USER-DEFINED-BEGIN token is a keyword.

There is another kind of token that is not associated with any object. Called the version token, it indicates what version of CIOBL software was used to write the file in which it appears. It also indicates in which of the three encodings the file is expressed. Whenever a CIOBL file is opened for writing, a version token is immediately written. Thus, every file has a version token at the beginning. A file may have more version tokens within, if the file was ever opened for appending, but if that is the case then the file must contain only variable-length objects. When a file is read, the version token is used to make sure the file uses the expected encoding, and that the file was written with a compatible version of CIOBL.

Rationale: The restriction that appended files may only contain variable-length objects is necessary because it is not possible to distinguish the version token from a fixed-length object.

To summarize, there are up to 17 kinds of CIOBL tokens in a Level III implementation: the five atomic variable-length CIOBL objects, the five fixed-length CIOBL objects, the six punctuation tokens (LIST-BEGIN, LIST-DOT, LIST-END, ARRAY-BEGIN, USER-DEFINED-BEGIN, and USER-DEFINED-END), and the version token. (There are five atomic objects, not six, because keywords are a special case of symbols.) For Level II there are 14 kinds of tokens (no LIST-DOT, USER-DEFINED-BEGIN, or USER-DEFINED-END), and for Level I there are only eleven (no LIST-BEGIN, LIST-END, or ARRAY-BEGIN).

7.3 Encodings

In the previous section, the translation between CIOBL tokens and CIOBL objects was described. Here, we describe how the 17 CIOBL tokens are encoded in the three types of encodings. Of course, not all 17 are necessarily relevant to a particular implementation, according to what level of the specification it conforms.

7.3.1 Standard Encoding

Standard encoding is the most easily read by humans. In fact, it is closely related to the syntax for Common Lisp objects used by the Lisp reader and printer. Beware, however, for CIOBL Standard encoding is not compatible with Lisp. Each contains objects not present in the other. Furthermore, CIOBL Standard encoding represents some objects differently, in order that the job of CIOBL input routines may be made easier.

Here is how the seventeen CIOBL tokens appear in the Standard encoding.

Unsigneds Unsigneds appear exactly as do integers, below.

---

[1] The USER-DEFINED-END token is not strictly necessary, but is included to help catch errors in user-written handler code.

Signeds Signeds appear exactly as do integers, below.

Singles Singles appear exactly as do floats, below.

Doubles Doubles appear exactly as do floats, below.

Chars Chars appear exactly as do characters, below.

Integers Integers are represented as they are written in base 10: a sequence of digits, at least one digit long, and immediately preceded by a hyphen if negative. Leading zeros are never present (except when the integer is itself zero).

Floats Floats are represented as a sequence of digits containing exactly one period, with at least one digit on each side of the period. This may optionally be preceded by a hyphen, and optionally followed by the letter `E` (never lowercase `e`) and a sequence of digits, possibly with a hyphen appearing between the `E` and the digits. The number after the `E` indicates by what power of ten the number preceding the `E` should be multiplied.

Characters Characters are represented as the two-character sequence `#\` followed immediately by the name of the character. For the 94 printing characters, the name of the character is just the character itself. For example, the character `A` appears as `#\A`. The names of Space and Newline are `Space` and `Newline`, respectively.

Strings Strings are represented as the two-character sequence `#"` followed immediately by the characters of the string. For example, the string `Hello` appears as `#"Hello`. Notice that there is no closing quotation mark.

Symbols Symbols are represented as the package name, followed immediately by a single colon (`:`), followed immediately by the symbol name. Any colon appearing within the package name or the symbol name must be preceded by a backslash to prevent its being interpreted as the separator between symbol name and package name. Furthermore, the first character of a symbol must be preceded by a backslash if it is a digit or a hyphen.

The empty string may be used in place of `KEYWORD` as a package name. Thus, the symbol `A` in the keyword package can appear as `:A`. If the symbol is accessible in the `DFCS` package, the package name and the colon may be dropped entirely. Hence, the symbol `NIL` appears as `NIL`. The exception to this rule is that the symbol in the `DFCS` package whose name is the empty string always appears as `DFCS:`, since there would be nothing left if the package name and colon were dropped.

Notice that this differs from Lisp syntax in several respects. First, in Standard encoding lower case letters are not converted to upper case (implying that most symbols will appear in upper case in Standard encoding files). Second, a double colon is never used. Third, vertical bars are never used.

For implementations not supporting full Level III symbols, the only type of symbol allowed is a keyword. Keywords follow the same rules above, with the empty string as the package name. That is, a keyword appears as a colon followed by the keyword's name, with the same rules as above for inserting backslashes.

`LIST-BEGIN` The token `LIST-BEGIN` appears as a left parenthesis (`(`).

`LIST-END` The token `LIST-END` appears as a right parenthesis (`)`).

LIST-DOT The token LIST-DOT appears as a the two-character sequence `#.` (pound sign followed by period).

ARRAY-BEGIN The token ARRAY-BEGIN appears as the two-character sequence `#A`.

USER-DEFINED-BEGIN The token USER-DEFINED-BEGIN appears as as the two-character sequence `#[`.

USER-DEFINED-END The token USER-DEFINED-END appears as the two-character sequence `#]`.

Version The version token is a four-character sequence: `#`, followed by `V`, followed by a character indicating the version, followed by `S`. A CIOBL version number is an integer from 1 through 63, inclusive, and is represented in the Standard encoding using the *value* column of Section 7.7. The fourth character indicates that this is a Standard encoding file.

Whitespace characters (Space and Newline) may appear between any adjacent pair of CIOBL tokens, and is required in certain situations described shortly.

Some terminology is introduced to explain the Standard encoding more precisely. The *whitespace characters* are Space and Newline. The *terminating characters* are open parenthesis (`(`), close parenthesis (`)`), and pound sign (`#`). The *escape character* is backslash (`\`). The remaining 90 standard characters are the *constituent characters.* A *constituent sequence* is an uninterrupted sequence of constituent characters, up to but not including a terminating character, whitespace character, or end of file. Any character, constituent or not, may be included in a constituent sequence by preceding it with an escape character. Thus, the sequence `A!\C\\d\ 5#` is a six-character constituent sequence followed by a pound sign, where the six characters of the constituent sequence are `A`, `!`, `C`, `\`, `d`, Space, and `5`.

Integers, floats, the names of characters, the contents of strings, and symbols are all constituent sequences. This means that the rules for escape characters apply. For example, the string "`String #1`" appears as `#"String\ \#1`, and the character `\` appears as `#\\\`. At least one whitespace character is required between a pair of tokens if either the first token is a integer, float, character, string, or symbol, or the second token is a integer, float, or symbol.

With that in mind, here is how the Standard encoding can be decoded. A constituent sequence is read by accumulating characters up to (but not including) the first non-escaped, non-constituent character. The following algorithm can be used to read a CIOBL token:

1. Skip over any whitespace characters.

2. Take a peek at the next character, which is guaranteed non-whitespace. If an open or close parenthesis, go to Step 3. If a pound sign, go to Step 4. Otherwise, go to Step 5.

3. Read the next character. The token LIST-BEGIN or LIST-END has been read, depending on whether the character was `(` or `)`, respectively.

4. Read the pound sign, and the character following. If the character following was `[`, `]`, `A`, or `.`, the token USER-DEFINED-BEGIN, USER-DEFINED-END, ARRAY-BEGIN, or LIST-DOT has been read. If it was `"`, read the next constituent sequence; this is the contents of a String. If it was `\`, read the next constituent sequence; this is the name of a Character. If it was `V`, read the next two characters; they are the remainder of a version token. If it was anything else, the file is not properly encoded.

5. Read the next constituent sequence, including the character peeked in Step 2. If the first character is an unescaped digit or hyphen, go to Step 6, otherwise go to Step 7.

6. If the sequence contains a period, interpret it as a float, otherwise interpret as an integer. If it cannot be interpreted as either a float or an integer, the file is not properly encoded.

7. The sequence is a symbol. Find an unescaped colon; it serves to separate the package name from the symbol name.

CIOBL takes no position as to what character codes are used in Standard encoding files. This is because such files are character files, so it is assumed that an implementation will use whatever codes for characters are appropriate, and that file transfer programs will take care of any necessary code conversions when transferring Standard encoding files between machines. There must, however, be a unique code for each of the 96 standard characters.

7.3.2 Binary Encoding

Binary encoding is the most compact of the three encodings. Unlike the other encodings, binary encoding is based on 8-bit bytes rather than characters. A variety of techniques are used to reduce the space required by a file.

The following describes how the seventeen CIOBL tokens are represented in the Binary encoding. All tokens except for fixed-length objects begin with a special byte that identifies its type. Throughout the discussion, these bytes are represented by a name enclosed in angle brackets, for example, `<b-pos-integer-8>`. The numerical values of these bytes are given later.

Unsigneds

An unsigned integer of N bits (*i.e.*, in the range $0 \leq x < 2^N$) is represented by $\lceil N/8 \rceil$ bytes, where the first byte is the eight least significant bits of the integer, the next byte is the next eight bits, *etc.* Any unused upper bits in the last byte are zero.

Signeds

A signed integer of N bits (*i.e.*, in the range $-2^{N-1} \leq x < 2^{N-1}$) is represented by $\lceil N/8 \rceil$ bytes, where the first byte is the eight least significant bits of the two's complement representation of the integer, *etc.* More precisely, let $M = \lceil N/8 \rceil$. Then if the integer x is non-negative, it is represented exactly as would be the $8M$-bit unsigned integer x. If negative, it is represented exactly as would be the $8M$ bit unsigned integer $2^{8M} - |x|$.

Singles

Singles are represented as the four bytes comprising the IEEE single precision representation of the number, least significant byte first.

Doubles

Doubles are represented as the eight bytes comprising the IEEE double precision representation of the number, least significant byte first.

Chars

The chars corresponding to the 94 printing standard characters or the space character are represented as would be the 7-bit unsigned integer given by their ASCII codes. The char Space is represented by the 7-bit unsigned integer 32, and the char Newline is represented by the 7-bit unsigned integer 10. See also Section 7.7.

This encoding was chosen because it is exactly the encoding used in some systems (*e.g.*, Unix), and very close to the encoding used in most others (*e.g.*, the Lisp Machine, where Newline is 141). CIOBL implementations on non-ASCII systems will have to explicitly translate when reading or writing Binary encoded files.

Integers

Most integers are represented by a punctuation byte followed by an unsigned that indicates the magnitude of the integer. The punctuation byte indicates the sign of the integer as well as how many bits are in the unsigned that follows. The representations are:

Magnitude Range	*Representation*
$-2^{32} < x \leq -2^{24}$	`<b-neg-integer-32> 32bit-unsigned`
$-2^{24} < x \leq -2^{16}$	`<b-neg-integer-24> 24bit-unsigned`
$-2^{16} < x \leq -2^{8}$	`<b-neg-integer-16> 16bit-unsigned`
$-2^{8} < x < 0$	`<b-neg-integer-8> 8bit-unsigned`
$0 \leq x < 2^{8}$	`<b-pos-integer-8> 8bit-unsigned`
$2^{8} \leq x < 2^{16}$	`<b-pos-integer-16> 16bit-unsigned`
$2^{16} \leq x < 2^{24}$	`<b-pos-integer-24> 24bit-unsigned`
$2^{24} \leq x < 2^{32}$	`<b-pos-integer-32> 32bit-unsigned`

For example, the number −4000 is represented as `<b-neg-integer-16> 160 15`, and zero is represented as `<b-pos-integer-8> 0`. (Throughout, all bytes are given in base ten.)

If the magnitude of the integer is 2^{32} or greater, one of the two bytes `<b-pos-long-integer>` or `<b-neg-long-integer>` is used, followed by an 8-bit unsigned M, followed by an $8M$-bit unsigned giving the magnitude of the integer. For example, five trillion (5×10^{12}) appears as `<b-pos-long-integer> 6 0 80 57 39 140 4`. Integers with magnitudes greater than or equal to 256^{256} cannot be represented (fortunately).

Characters

A character appears as two bytes: `<b-character>` followed by the corresponding char (see the section "Chars," above).

Strings

Strings of length 255 or less are encoded as `<b-string>`, followed by an 8-bit unsigned giving the length of the string, followed by the characters of the string itself, as chars (see the section "Chars," above).

Strings whose length is greater than 255 are encoded as `<b-long-string>`, followed by an 8-bit unsigned M, followed by an $8M$-bit unsigned giving the length of the string, followed by the characters of the string itself, as chars. Strings longer than 256^{256} cannot be represented (not that your file system has room for all those characters, anyway).

Floats

Floats are not encoded into a two's complement or similar representation. Instead, they appear as a pair of integers, b and e, such that the float is equal to $b \times 2^e$. The number of bits in these integers is determined based on the magnitude of the exponent and the precision of the mantissa.

The representation is:

<b-pos-float> n_b n_e n_b**bit-unsigned** n_e**bit-signed**

where n_b is an 8-bit unsigned giving the number of bits in the mantissa, and n_e is an 8-bit unsigned giving the number of bits in the exponent. The example above is for a positive float; for a negative float, the byte **<b-neg-float>** replaces **<b-pos-float>**, and the absolute value of b appears after n_e. Note that the mantissa value will normally be in the range $2^{n_b-1} \leq b < 2^{n_b}$; if b were less than 2^{n_b-1} it would not truly have n_b significant bits. The exponent will be in the range $-2^{n_e-1} \leq e < 2^{n_e}$, and n_e will be one if e is zero.

For example, the number -6.023×10^{-23}, represented with a 24-bit mantissa, appears as **<b-neg-float> 24 8 144 160 145 159**.

By convention, zero is represented with a zero-bit exponent (so that n_e is zero and no bytes follow the mantissa bytes), and with a mantissa of all zeros. While strictly speaking, zero has no significant bits, the number of bits in the mantissa may be used to indicate a zero represented in a particular floating point representation. That is, when the float is read a zero will be constructed in a floating point representation which for non-zero numbers has at least n_b significant bits. Usually, **<b-pos-float>** is used for zero, although **<b-neg-float>** may be used to indicate the "negative zero" available in floating point representations such as IEEE.

For example, an IEEE single-precision zero (24 bits of mantissa) appears as **<b-pos-float> 24 0 0 0 0**.

Symbols

For implementations not supporting full Level III symbols, the only kind of symbol which can appear is a keyword. Keywords are represented by encoding their names in a similar manner as strings (see the section "Strings," above), except that the bytes **<b-keyword-symbol>** and **<b-long-keyword-symbol>** are used instead of **<b-string>** and **<b-long-string>**, respectively.

For implementations which do support Level III symbols, the following may appear in addition to keywords encoded as above.

The symbol `NIL` (not the same as `NIL` in the `KEYWORD` package) is represented by a single byte, **<b-nil>**.

All other symbols are represented by encoding their names in a similar manner as strings, except that the bytes **<b-symbol>** and **<b-long-symbol>** are used instead of **<b-string>** and **<b-long-string>**. The package name is not included in the name that follows the **<b-symbol>** or **<b-long-symbol>** byte. Instead, the package name of a symbol read from a Binary encoded file is taken to be the "current" package name. The current package name is changed by including in the file one of the bytes **<b-set-current-package>** or **<b-long-set-current-package>**, which are used like **<b-string>** and **<b-long-string>** to encode the package name.

For example, if the following sequence appeared in a Standard encoding file:

```
:A  NIL  B:C  :NIL  NIL  B:D  Q:R
```

It would be rendered in a Binary encoding file as:

```
<b-keyword-symbol> 1 65 <b-nil> <b-set-current-package> 1 66
<b-symbol> 1 67 <b-keyword-symbol> 3 78 73 76 <b-nil>
<b-symbol> 1 68 <b-set-current-package> 1 81 <b-symbol> 1 82
```

The first symbol in a Binary encoding file that is not a keyword or NIL is always preceded by a set-current-package directive. Note that the set-current-package directive is not a CIOBL token, but only controls the interpretation of symbols that follow it. Note, too, that the set-current-package directive has no effect on keywords or NIL.

One additional trick is used to encode symbols. If the same symbol (same package name and symbol name) appears more than once in the same file, only the first occurrence is encoded as described earlier. All future occurrences are encoded as an integer which indicates position within the file of its first occurrence. These remarks do not apply to the symbol NIL, which is always encoded as the byte <b-nil>. These remarks *do* apply, however, even when an implementation does not support Level III symbols. Thus, this trick is used even for files that have only keywords.

As a Binary encoded file is written, a table is maintained which associates symbols (or just keywords) and serial numbers. When a non-NIL symbol is to be written, it is looked up in the table. If an entry for that symbol is present, its serial number is written in a format to be described shortly. If an entry is not present, the symbol is written in the format described earlier, preceded by a set-current-package directive if necessary. The symbol is then assigned the next highest serial number, and entered in the table for future reference. The unique non-NIL symbols in a file are assigned consecutive serial numbers beginning with zero. Note that when a symbol is encoded as a serial number, it is never necessary to issue a set-current-package directive, as the serial number identifies both components of the symbol's name.

A symbol encoded by serial number appears as one of the bytes <b-predefined-symbol-8>, <b-predefined-symbol-16>, or <b-predefined-symbol-24>, followed by an 8-bit, 16-bit, or 24-bit unsigned, respectively, giving the serial number.

Another example: suppose the following appeared at the beginning of a Standard encoding file:

```
:D :A  NIL  B:C  :NIL  NIL  Q:R  B:C  B:D
```

It would be rendered in a Binary encoding file as:

```
<b-keyword-symbol> 1 68 <b-keyword-symbol> 1 65 <b-nil>
<b-set-current-package> 1 66 <b-symbol> 1 67
<b-keyword-symbol> 3 78 73 76 <b-nil>
<b-set-current-package> 1 81 <b-symbol> 1 82
<b-predefined-symbol-8> 2 <b-set-current-package> 1 66
<b-symbol> 1 68
```

Only the first 2^{24} different non-NIL symbols in a file can be encoded by serial number.

The Version Token

The version token appears in a Binary file as the four-byte sequence 35 86 *version* 66, where *version* is the version number plus 32. The last byte indicates that this is a Binary file.

When reading a Binary encoding file, 163 (35 plus 128) should also be accepted as the beginning of a version token. In other words, only the lower seven bits are used in recognizing the beginning of the version token. The number 35 was chosen for the version token because it is the ASCII code for #, so that a version will be recognized even when read from a file in the wrong encoding (at least on systems that use ASCII to represent characters). This facilitates early detection of an attempt to read the wrong kind of file.

Other Tokens

The CIOBL tokens LIST-BEGIN, LIST-DOT, LIST-END, ARRAY-BEGIN, USER-DEFINED-BEGIN, and USER-DEFINED-END appear in Binary encoding as the bytes **<b-list-begin>**, **<b-list-dot>**, **<b-list-end>**, **<b-array-begin>**, **<b-user-defined-begin>**, and **<b-user-defined-end>**, respectively.

Values of Binary Punctuation Bytes

The following table gives the values of each of the 30 binary punctuation bytes.

Name	*Value*	*Name*	*Value*
<b-symbol>	0	<b-pos-integer-32>	23
<b-keyword-symbol>	1	<b-pos-long-integer>	24
<b-long-symbol>	2	<b-neg-integer-8>	25
<b-long-keyword-symbol>	3	<b-neg-integer-16>	26
<b-nil>	4	<b-neg-integer-24>	27
<b-set-current-package>	5	<b-neg-integer-32>	28
<b-long-set-current-package>	6	<b-neg-long-integer>	29
<b-predefined-symbol-8>	7	<b-pos-float>	30
<b-predefined-symbol-16>	8	<b-neg-float>	31
<b-predefined-symbol-24>	9	<b-version>	35
<b-string>	10	<b-list-begin>	40
<b-long-string>	11	<b-list-end>	41
<b-character>	15	<b-list-dot>	42
<b-pos-integer-8>	20	<b-user-defined-begin>	50
<b-pos-integer-16>	21	<b-user-defined-end>	51
<b-pos-integer-24>	22	<b-array-begin>	55

7.3.3 Compressed Encoding

Compressed encoding is very similar to Binary encoding, but is based on standard characters rather than 8-bit bytes. This makes it useful for transmission over electronic mail and other media which can only accomodate character files.

The following describes how the seventeen CIOBL tokens are represented in the Compressed encoding. All tokens except for fixed-length objects begin with a special character that identifies its type. Throughout the discussion, these characters are represented by a name enclosed in angle brackets, for example, **<c-pos-integer-6>**. The values of these characters are given later.

Note that most of the descriptions below are almost the same as for the Binary encoding, except that numerical values are broken into groups of six bits, each of which is encoded into a character according to the third column of the table in Section 7.7.

Unsigneds

An unsigned integer of N bits (*i.e.*, in the range $0 \leq x < 2^N$) is represented by $\lceil N/6 \rceil$ characters, where the first character encodes the six least significant bits of the integer, the next character encodes the next six bits, *etc.* If N is not a multiple of six, the upper bits of the most significant group of six are considered to be zero. The encoding of each group of six bits into a character is given in Section 7.7.

Signeds

A signed integer of N bits (*i.e.*, in the range $-2^{N-1} \leq x < 2^{N-1}$) is represented by $\lceil N/6 \rceil$ characters, where the first character is the six least significant bits of the two's complement representation of the integer, *etc.* More precisely, let $M = \lceil N/6 \rceil$. Then if the integer x is non-negative, it is represented exactly as would be the $6M$-bit unsigned integer x. If negative, it is represented exactly as would be the M bit unsigned integer $2^{6M} - x$.

Singles

Singles are represented as would be a 32-bit unsigned whose value is the 32 bits comprising the IEEE single precision representation of the number. Hence, it is represented as 6 characters.

Doubles

Doubles are represented as would be a 64-bit unsigned whose value is the 64 bits comprising the IEEE double precision representation of the number. Hence, it is represented as 11 characters.

Chars

A char simply appears as itself.

Integers

Most integers are represented by a punctuation char followed by an unsigned that indicates the magnitude of the integer. The punctuation char indicates the sign of the integer as well as how many bits are in the unsigned that follows. The representations are:

Magnitude Range	*Representation*
$-2^{24} < x \leq -2^{18}$	`<c-neg-integer-24> 24bit-unsigned`
$-2^{18} < x \leq -2^{12}$	`<c-neg-integer-18> 18bit-unsigned`
$-2^{12} < x \leq -2^{6}$	`<c-neg-integer-12> 12bit-unsigned`
$-2^{6} < x < 0$	`<c-neg-integer-6> 6bit-unsigned`
$0 \leq x < 2^{6}$	`<c-pos-integer-6> 6bit-unsigned`
$2^{6} \leq x < 2^{12}$	`<c-pos-integer-12> 12bit-unsigned`
$2^{12} \leq x < 2^{18}$	`<c-pos-integer-18> 18bit-unsigned`
$2^{18} \leq x < 2^{24}$	`<c-pos-integer-24> 24bit-unsigned`

For example, the number −4000 is represented as <c-neg-integer-12> 6 ^, and zero is represented as <c-pos-integer-6> 0 <space>.

If the magnitude of the integer is 2^{24} or greater, one of the two characters <c-pos-long-integer> or <c-neg-long-integer> is used, followed by a 6-bit unsigned M, followed by an $6M$-bit unsigned giving the magnitude of the integer. For example, one hundred million (10^8) appears as <c-pos-long-integer> % <space> $ >] %. Integers with magnitudes greater than or equal to 64^{64} cannot be represented (fortunately).

Characters

A character appears as two characters: <c-character> followed by the character itself.

Strings

Strings of length 63 or less are encoded as <c-string>, followed by a 6-bit unsigned giving the length of the string, followed by the characters of the string itself.

Strings whose length is greater than 63 are encoded as <c-long-string>, followed by a 6-bit unsigned M, followed by an $6M$-bit unsigned giving the length of the string, followed by the characters of the string itself, as chars. Strings longer than 64^{64} cannot be represented (not that your file system has room for all those characters, anyway).

Floats

Floats are not encoded into a two's complement or similar representation. Instead, they appear as a pair of integers, b and e, such that the float is equal to $b \times 2^e$. The number of bits in these integers is determined based on the magnitude of the exponent and the precision of the mantissa.

The representation is:

<c-pos-float> n_b n_e $n_b$bit-unsigned $n_e$bit-signed

where n_b is a 12-bit unsigned giving the number of bits in the mantissa, and n_e is a 6-bit unsigned giving the number of bits in the exponent. The example above is for a positive float; for a negative float, the character <c-neg-float> replaces <c-pos-float>, and the absolute value of b appears after n_e.

For example, the number -6.023×10^{-23}, represented with a 24-bit mantissa, appears as <c-neg-float> 8 <space> (0 " : D ? ^.

Symbols

For implementations not supporting full Level III symbols, the only kind of symbol which can appear is a keyword. Keywords are represented by encoding their names in a similar manner as strings (see the section "Strings," above), except that the bytes <c-keyword-symbol> and <c-long-keyword-symbol> are used instead of <c-string> and <c-long-string>, respectively.

For implementations which do support Level III symbols, the following may appear in addition to keywords encoded as above.

The symbol NIL (not the same as NIL in the KEYWORD package) is represented by a single byte, <c-nil>.

All other symbols are represented by encoding their names in a similar manner as strings, except that the bytes <c-symbol> and <c-long-symbol> are used instead of <c-string> and <c-long-string>. The package name is not included in the name that follows the <c-symbol> or <c-long-symbol> byte. Instead, the package name of a symbol read from a Compressed encoded file is taken to be the "current" package name. The current package name is changed by including in the file one of the bytes <c-set-current-package> or <c-long-set-current-package>, which are used like <c-string> and <c-long-string> to encode the package name.

For example, if the following sequence appeared in a Standard encoding file:

```
:A  NIL  B:C  :NIL  NIL  B:D  Q:R
```

It would be rendered in a Compressed encoding file as:

```
<c-keyword-symbol> ! A <c-nil> <c-set-current-package> ! B
<c-symbol> ! C <c-keyword-symbol> # N I L <c-nil>
<c-symbol> ! D <c-set-current-package> ! Q <c-symbol> ! R
```

The first symbol in a Compressed encoding file that is not a keyword or NIL is always preceded by a set-current-package directive. Note that the set-current-package directive is not a CIOBL token, but only controls the interpretation of symbols that follow it. Note, too, that the set-current-package directive has no effect on keywords or NIL.

One additional trick is used to encode symbols. If the same symbol (same package name and symbol name) appears more than once in the same file, only the first occurrence is encoded as described earlier. All future occurrences are encoded as an integer which indicates position within the file of its first occurrence. These remarks do not apply to the symbol NIL, which is always encoded as the byte <c-nil>. These remarks *do* apply, however, even when an implementation does not support Level III symbols. Thus, this trick is used even for files that have only keywords.

As a Compressed encoded file is written, a table is maintained which associates symbols (or just keywords) and serial numbers. When a non-NIL symbol is to be written, it is looked up in the table. If an entry for that symbol is present, its serial number is written in a format to be described shortly. If an entry is not present, the symbol is written in the format described earlier, preceded by a set-current-package directive if necessary. The symbol is then assigned the next highest serial number, and entered in the table for future reference. The unique non-NIL symbols in a file are assigned consecutive serial numbers beginning with zero. Note that when a symbol is encoded as a serial number, it is never necessary to issue a set-current-package directive, as the serial number identifies both components of the symbol's name.

A symbol encoded by serial number appears as one of the characters <c-predefined-symbol-6>, <c-predefined-symbol-12>, or <c-predefined-symbol-18>, followed by a 6-bit, 12-bit, or 18-bit unsigned, respectively, giving the serial number.

Another example: suppose the following appeared at the beginning of a Standard encoding file:

```
:D :A  NIL  B:C  :NIL  NIL  Q:R  B:C  B:D
```

It would be rendered in a Compressed encoding file as:

```
<c-keyword-symbol> ! D <c-keyword-symbol> ! A <c-nil>
<c-set-current-package> ! B <c-symbol> ! C
```

```
<c-keyword-symbol> # N I L <c-nil>
<c-set-current-package> ! Q <c-symbol> ! R
<c-predefined-symbol-6> " <c-set-current-package> ! B
<c-symbol> ! D
```

Only the first 2^{18} different non-NIL symbols in a file can be encoded by serial number.

The Version Token

The version token appears in a Compressed file as the four-character sequence **# V** *version* **C**, where *version* is the version number as a 6-bit unsigned. The last character indicates that this is a Compressed file.

Other Tokens

The CIOBL tokens LIST-BEGIN, LIST-DOT, LIST-END, ARRAY-BEGIN, USER-DEFINED-BEGIN, and USER-DEFINED-END appear in Compressed encoding as the characters **<c-list-begin>**, **<c-list-dot>**, **<c-list-end>**, **<c-array-begin>**, **<c-user-defined-begin>**, and **<c-user-defined-end>**, respectively.

Optional Newlines

Newline characters may occur between tokens (but not within them) in the Compressed encoding, and are ignored during reading. It is desirable for an implementation to insert newline characters automatically when writing so that there are no more than 80 or so characters between consecutive newlines. This makes Compressed files more readily transmissible over electronic mail and similar media. Of course, it may not always be possible to have newlines this frequently because tokens may be of arbitrary length.

Values of Compressed Punctuation Characters

The following table gives the values of each of the 30 Compressed punctuation characters.

Name	*Value*	*Name*	*Value*
<c-symbol>	y	<c-pos-integer-24>	3
<c-keyword-symbol>	k	<c-pos-long-integer>	4
<c-long-symbol>	Y	<c-neg-integer-6>	5
<c-long-keyword-symbol>	K	<c-neg-integer-12>	6
<c-nil>	N	<c-neg-integer-18>	7
<c-set-current-package>	p	<c-neg-integer-24>	8
<c-long-set-current-package>	P	<c-neg-long-integer>	9
<c-predefined-symbol-6>	-	<c-pos-float>	f
<c-predefined-symbol-12>	_	<c-neg-float>	F
<c-predefined-symbol-18>	=	<c-version>	#
<c-string>	s	<c-list-begin>	(
<c-long-string>	S	<c-list-end>	)
<c-character>	C	<c-list-dot>	.
<c-pos-integer-6>	0	<c-user-defined-begin>	[
<c-pos-integer-12>	1	<c-user-defined-end>	]
<c-pos-integer-18>	2	<c-array-begin>	A

7.4 CIOBL Functions (Common Lisp)

This section documents the program interface to CIOBL software for Common Lisp. It also serves as a guide to the preferred interface in other languages (for the C interface, see Section 7.5). Unless otherwise stated, all functions are part of Level I of CIOBL, and must therefore be present in all implementations.

The names of all functions are external in the `CIOBL` package. Note that there are three symbols in the `CIOBL` package which shadow symbols in the `LISP` package: **`ciobl:read-char`**, **`ciobl:write-char`**, and **`ciobl:write-string`**. For this reason, the `ciobl` package should not be used by programs that need CIOBL; instead, calls to CIOBL functions should be prefixed with an explicit `ciobl:` qualifier. Alternatively, commonly used CIOBL functions could be individually imported, but this is discouraged.

7.4.1 General CIOBL Stream Functions

`ciobl:make-ciobl-stream` *underlying-stream encoding* [*Function*]

Returns a CIOBL stream such that CIOBL read and write calls performed on that stream will cause corresponding reads and writes to be performed on *underlying-stream*. The value of *encoding* gives the encoding to be used in translating between objects in the CIOBL stream and characters or bytes in the underlying stream; once established, the encoding may not be changed. The value of *encoding* must be one of **`:standard`**, **`:compressed`**, or **`:binary`**. The value of *underlying-stream* must be a stream to which character I/O can be performed if *encoding* is **`:standard`** or **`:compressed`**, or a stream to which 8-bit byte I/O can be performed if *encoding* is **`:binary`**. The underlying stream need not be a bidirectional stream, but if it is unidirectional then so will be the CIOBL stream.

If *underlying-stream* is an output or bidirectional stream, then **`make-ciobl-stream`** will write a version token before returning. If *underlying-stream* is an input or bidirectional stream,

then the first call to a CIOBL reading function will read the first four characters or bytes from the stream and interpret them as a version token.

`ciobl:close-ciobl` *ciobl-stream* [*Function*]

A call to `ciobl:close-ciobl` closes the stream underlying *ciobl-stream*, performing any cleanup operations that calling `close` on the underlying stream would perform. It is imperative that `ciobl:close-ciobl` be called on the CIOBL stream rather than simply closing `close` on the underlying stream, because there may be cleanup operations specific to CIOBL.

`ciobl:with-ciobl-stream` (*stream underlying-stream encoding*) {*declaration*}* {*form*}* [*Macro*]

`ciobl:with-ciobl-stream` evaluates the *forms* of the body (an implicit `progn`) with the variable *stream* bound to the result of calling `ciobl:make-ciobl-stream` on *underlying-stream* and *encoding*. When control leaves the body, either normally or abnormally, `ciobl:close-ciobl` is called on *stream*. Because `ciobl:with-ciobl-stream` always calls `ciobl:close-ciobl`, even when an error exit is taken, it is preferred over `ciobl:make-ciobl-stream` for most applications.

`with-ciobl-file` (*stream underlying-stream encoding* `&rest` *open-options*) {*declaration*}* {*form*}* [*Macro*]

`with-ciobl-file` combines `with-open-file` with `ciobl:with-ciobl-stream`. The arguments *stream*, *underlying-stream*, and *encoding* are as for `with-ciobl-file`. The *open-options* are passed direct to `with-open-file`. The `:element-type` option to `with-open-file` is supplied automatically, based on the value of *encoding*.

`ciobl:ciobl-stream-p` *thing* [*Function*]

Returns true if *thing* is a CIOBL stream, false otherwise.

7.4.2 Reading and Writing Fixed-Length Objects

`ciobl:read-unsigned` *n ciobl-stream* `&optional` *eof-value* [*Function*]

`ciobl:read-unsigned` reads and returns the next n-bit unsigned integer from the CIOBL stream *ciobl-stream*; n must be a positive integer. The value x returned will always be in the range $0 \leq x < 2^n$. Unpredictable results will occur if the stream is not positioned at a place written by a corresponding `ciobl:write-unsigned`, with the same value of n. If the CIOBL stream is at end-of-file before the call, *eof-value* is returned instead of an integer (the default is `nil`). On the other hand, if end-of-file occurs in the middle of reading the unsigned, an error is signalled.

`ciobl:read-signed` *n ciobl-stream* `&optional` *eof-value* [*Function*]

`ciobl:read-signed` reads and returns the next n-bit signed integer from the CIOBL stream *ciobl-stream*; n must be a positive integer. The value x returned will always be in the range $-2^{n-1} \leq x < 2^{n-1}$. Unpredictable results will occur if the stream is not positioned at a place written by a corresponding `ciobl:write-signed`, with the same value of n. The end-of-file treatment is the same as for `ciobl:read-unsigned`.

`ciobl:read-single` *ciobl-stream* `&optional` *eof-value* [*Function*]

`ciobl:read-double` *ciobl-stream* `&optional` *eof-value* [*Function*]
`ciobl:read-char` *ciobl-stream* `&optional` *eof-value* [*Function*]

Reads and returns the next single, double, or char from the CIOBL stream *ciobl-stream*. Unpredicatable results will occur if the stream is not positioned at a place written by a corresponding `ciobl:write-single`, `ciobl:write-double`, or `ciobl:write-char`. The end-of-file treatment is the same as for `ciobl:read-unsigned`.

Implementation Note: Implementations for typed languages such as C will require a different convention for indicating end of file.

`ciobl:write-unsigned` *n unsigned ciobl-stream* [*Function*]

Writes the n-bit unsigned integer *unsigned* on *ciobl-stream*. The value of *unsigned* must be in the range $0 \leq unsigned < 2^n$.

`ciobl:write-signed` *n signed ciobl-stream* [*Function*]

Writes the n-bit signed integer *signed* on *ciobl-stream*. The value of *signed* must be in the range $-2^{n-1} \leq signed < 2^{n-1}$.

`ciobl:write-single` *x ciobl-stream* [*Function*]
`ciobl:write-double` *x ciobl-stream* [*Function*]
`ciobl:write-char` *x ciobl-stream* [*Function*]

Each writes the object x on *ciobl-stream*, as the appropriate fixed-length object.

7.4.3 Reading and Writing Variable Length Objects

`ciobl:read-integer` *ciobl-stream* `&optional` *eof-value* [*Function*]
`ciobl:read-float` *ciobl-stream* `&optional` *eof-value* [*Function*]
`ciobl:read-character` *ciobl-stream* `&optional` *eof-value* [*Function*]
`ciobl:read-string` *ciobl-stream* `&optional` *eof-value* [*Function*]
`ciobl:read-keyword` *ciobl-stream* `&optional` *eof-value* [*Function*]

Each function reads and returns the appropriate variable-length objects from *ciobl-stream*. If the stream was positioned at end-of-file before the call, *eof-value* will be returned instead (the default is `nil`). On the other hand, if end-of-file occurs in the middle of reading the object, an error is signalled. An error will also be signalled if the stream was positioned at a variable-length object other than the one given in the name of the function. Unpredicatable results will occur if the stream was positioned at a fixed-length object.

Implementation Note: Implementations for typed languages such as C will require a different convention for indicating end of file.

`ciobl:write-integer` *x ciobl-stream* [*Function*]
`ciobl:write-float` *x ciobl-stream* [*Function*]
`ciobl:write-character` *x ciobl-stream* [*Function*]
`ciobl:write-string` *x ciobl-stream* [*Function*]
`ciobl:write-keyword` *x ciobl-stream* [*Function*]

Each function writes the object x as the appropriate variable-length object to *ciobl-stream*. It is an error to call one of these functions with the wrong type of argument, *e.g.*, calling `ciobl:write-integer` with a string.

`ciobl:keyword-name` *keyword* [*Function*]
`ciobl:name-keyword` *string* [*Function*]

These functions convert between keywords as returned by **`ciobl:read-keyword`** and accepted by **`ciobl:write-keyword`** and their corresponding names, as strings. **`ciobl:keyword-name`** takes a keyword and returns its name as a string, while **`ciobl:name-keyword`** takes a string and returns the corresonding keyword. Note that the Lisp function **`eql`** or the C function **`==`** may be used to compare keywords for equality, while the same is not true for strings.

Implementation Note: These functions are included primarily for the benefit of implementations in langauges (such as C) which do not have keywords as a primitive data type. Even in langauges which do (such as Common Lisp), a Level I implementation may wish to use a different representation for CIOBL keywords. For Level III Common Lisp implementations, however, the following equivalences hold:

```
(ciobl:keyword-name x) ≡ (symbol-name x)
(ciobl:name-keyword x) ≡ (intern x "KEYWORD")
```

7.4.4 Level II Functions

All of the functions in this section are provided only in Level II and Level III implementations of CIOBL.

Compound Objects

`ciobl:read-list` *ciobl-stream* `&optional` *eof-value* [*Function*]
`ciobl:read-array` *ciobl-stream* `&optional` *eof-value* [*Function*]

Reads and returns a list or array from *ciobl-stream*. If the stream was positioned at end-of-file before the call, *eof-value* will be returned instead (the default is **`nil`**). On the other hand, if end-of-file occurs in the middle of reading the list or array, whether between component objects or within them, an error is signalled. An error will also be signalled if the stream was positioned at a variable-length object other than the one given in the name of the function, or if an improperly formatted list or array is encountered. Unpredicatable results will occur if the stream was positioned at a fixed-length object or if fixed-length objects are encountered before the entire list or array is read.

`ciobl:write-list` *x ciobl-stream* [*Function*]
`ciobl:write-array` *x ciobl-stream* [*Function*]

Each function writes the object *x* as a list or array to *ciobl-stream*. It is an error to call one of these functions with an inappropriate argument, *e.g.*, calling **`ciobl:write-list`** with a string. An error is signalled if **`ciobl:write-array`** is called with an array whose rank is less than one or greater than seven.

`ciobl:read-list-begin` *ciobl-stream* `&optional` *eof-value* [*Function*]
`ciobl:read-list-end` *ciobl-stream* `&optional` *eof-value* [*Function*]
`ciobl:read-array-begin` *ciobl-stream* `&optional` *eof-value* [*Function*]

These functions are provided for the benefit of programs which wish to read compound objects without actually having the CIOBL software construct those objects. Each reads a punctuation

token from *ciobl-stream* and returns `nil`. If the stream was positioned at end-of-file before the call, *eof-value* will be returned instead (the default is `nil`). An error will be signalled if the stream was positioned at an atomic variable-length object or at a punctuation token other than the one given in the name of the function. Unpredicatable results will occur if the stream was positioned at a fixed-length object.

`ciobl:write-list-begin` *ciobl-stream* [*Function*]
`ciobl:write-list-end` *ciobl-stream* [*Function*]
`ciobl:write-array-begin` *ciobl-stream* [*Function*]

These functions are provided for the benefit of programs which wish to write compound objects without first constructing them in memory. Each writes a punctuation token to *ciobl-stream*. It is the responsibility of the user to observe the proper syntax for compound objects when writing them in this manner.

Generic Objects

`ciobl:read-any` *ciobl-stream* `&optional` *eof-value return-opening-punctuation-p return-closing-punctuation-p* [*Function*]

`ciobl:read-any` reads the next variable-length CIOBL object from *ciobl-stream*. Two values are returned: the object read, and a keyword indicating what kind of object was read (see below). If the stream was positioned at end-of-file before the call, the values *eof-value* (default `nil`) and `:eof` will be returned instead. On the other hand, if end-of-file occurs in the middle of reading the object, an error is signalled. Unpredicatable results will occur if the stream was positioned at a fixed-length object, or if fixed-length objects are encountered while reading a compound object.

The arguments *return-opening-punctuation-p* and *return-closing-punctuation-p*, both defaulting to false, control the behavior when the stream is positioned at punctuation tokens. If the stream is positioned at the beginning of a compound object (at a LIST-BEGIN or ARRAY-BEGIN token) and *return-opening-punctuation-p* is false, then the entire list or array is read. If *return-opening-punctuation-p* is true, then only the punctuation token is read, and `nil` is returned. If the stream is positioned at a LIST-END token and *return-closing-punctuation-p* is false, an error is signalled. If *return-closing-punctuation-p* is true, the punctuation token is read. Typically, both arguments will be false when a program does not want to read compound objects manually, and both will be true when it wants to read *all* compound objects that way. On the other hand, if a program is in the middle of reading a compound object manually but does not want to read manually any components which might be themselves compound objects, *return-opening-punctuation-p* will be false and *return-closing-punctuation-p* will be true.

The second value returned by `ciobl:read-any` depends on *return-opening-punctuation-p* and *return-closing-punctuation-p*. If both are false, then the second value is one of `:integer`, `:float`, `:character`, `:string`, `:keyword`, `:list`, `:array`, or `:eof`. If *return-opening-punctuation-p* is true then the second value will never be `:list` or `:array`, but it might be `:list-begin` or `:array-begin`. If *return-closing-punctuation-p* is true then the second value could also be `:list-end`.

`ciobl:write-any` *x ciobl-stream* [*Function*]

The argument *x* must be an integer, float, character, string, keyword, list, or array. It is written as the appropriate variable-length object to *ciobl-stream*.

Implementation Note: Implementations for typed languages (such as C) will require different conventions for indicating end of file and for indicating the type of object returned from `ciobl:read-any` or passed to `ciobl:write-any`.

7.4.5 Level III Functions

All of the functions in this section are provided only in Level III implementations of CIOBL. A Level III implementation is not required to provide all of them, but of course it should not provide a read function without the corresponding write function, or vice versa.

`ciobl:read-symbol` *ciobl-stream* `&optional` *eof-value* [*Function*]
`ciobl:read-dotted-list` *ciobl-stream* `&optional` *eof-value* [*Function*]
`ciobl:write-symbol` *x ciobl-stream* [*Function*]
`ciobl:write-dotted-list` *x ciobl-stream* [*Function*]

Analogous to all the other functions for reading and writing variable-length objects. These are typically provided only in Common Lisp implementations of CIOBL. The argument *x* to `ciobl:write-dotted-list` may be an ordinary list, in which case it is written as such. Note that in CIOBL, `nil` and the empty list are not the same.

`ciobl:read-`*t* *ciobl-stream* `&optional` *eof-value* [*Function*]
`ciobl:write-`*t* *x ciobl-stream* [*Function*]

A Level III implementation may provide functions of this form for reading and writing particular cases of user-defined objects of type *t*. Typically, such functions would only be provided in implementations where the vocabulary of user-defined objects is not user programmable.

`ciobl:read-any` *ciobl-stream* `&optional` *eof-value return-opening-punctuation-p return-closing-punctuation-p* [*Function*]
`ciobl:write-any` *x ciobl-stream* [*Function*]

These are the same as in Level II, extended to handle symbols, dotted lists, and user-defined objects according to what the implementation provides. In implementations where the vocabulary of user-defined objects is user programmable, these functions are typically the only way of reading and writing such objects in one call. In addition to their Level II roles, *return-opening-punctuation-p* governs the behavior when positioned at a USER-DEFINED-BEGIN token, and *return-closing-punctuation-p* governs the behavior when positioned at a LIST-DOT or USER-DEFINED-END token. The second value returned from `ciobl:read-any`, in addition to the values it can assume in Level II, may also be `:dotted-list`, `:symbol`, any symbol that might occur after a USER-DEFINED-BEGIN token, `:user-defined-begin`, `:user-defined-end`, or `:list-dot`. Again, whether some of these can occur depends on *return-opening-punctuation-p* and *return-closing-punctuation-p*.

If `:dotted-list` is returned, it indicates only that a LIST-DOT token was part of the list read, not that the object following the dot is non-`nil`. While `ciobl:write-list` will never write a LIST-DOT token if the object following would be `nil`, such a file could be written manually through calls to `ciobl:write-list-begin`, `ciobl:write-list-dot` and `ciobl:write-list-end`.

`ciobl:read-list-dot` *ciobl-stream* `&optional` *eof-value* [*Function*]
`ciobl:read-user-defined-begin` *ciobl-stream* `&optional` *eof-value* [*Function*]
`ciobl:read-user-defined-end` *ciobl-stream* `&optional` *eof-value* [*Function*]

`ciobl:write-list-dot` *ciobl-stream* [*Function*]
`ciobl:write-user-defined-begin` *ciobl-stream* [*Function*]
`ciobl:write-user-defined-end` *ciobl-stream* [*Function*]

These are analogous to the functions provided in Level II for reading and writing individual punctuation tokens.

`ciobl:defciobl-read` *type* (*stream*) {*declaration*}* {*form*}* [*Macro*]
`ciobl:defciobl-write` *type* (*object stream*) {*declaration*}* {*form*}* [*Macro*]

`ciobl:defciobl-read` and `ciobl:defciobl-write` provide a user programmable implementation of user-defined objects for Common Lisp implementations. When `ciobl:read-any` reads a USER-DEFINED-BEGIN token followed by a symbol with the same name and package as *type*, it executes the *forms* from the `ciobl:defciobl-read` (as an implicit `progn`) with the variable *stream* bound to the CIOBL stream. The forms should read all objects up to (but not including) the matching USER-DEFINED-END, and the last form should return the object. CIOBL then automatically reads the USER-DEFINED-END token, signalling an error if some other token is found instead.

Similarly, when `ciobl:write-any` is passed an object of type *type*, it writes a USER-DEFINED-BEGIN token followed by the symbol *type*, then executes the *forms* from the `ciobl:defciobl-write` (as an implicity `progn`) with the variable *stream* bound to the CIOBL stream and the variable *object* bound to the object to be written. The forms should write the components of the object to *stream*, as variable-length CIOBL objects. When the *forms* are finished, CIOBL automatically writes the USER-DEFINED-END token.

It is the responsibility of the user to insure that the `ciobl:defciobl-read` and `ciobl:defciobl-write` definitions for a given type are compatible.

7.5 CIOBL Functions (C)

The C language interface to CIOBL is currently only defined for Level I of CIOBL. The interface is similar to the Common Lisp interface, but with these differences:

1. Because C does not have arbitrary precision integers, several versions of the integer read and write functions have been created, differing in their argument or result types. These differences *only* constrain the C types of the arguments and results; the formatting of the CIOBL object being read or written is controlled in the same way as in Common Lisp.
2. Because C does not have optional arguments, and because functions are typed, end-of-file is indicated in a different way by the reading functions.
3. Because C does not have a general error-signalling mechanism, a different way of indicating various error conditions is used.

The C interface requires the standard `stdio` library and include file, as well as the `scalar`, `int64`, and `keyword` libraries and include files from the MCRC C support package.

7.5.1 General CIOBL Stream Functions

`CIOBL_STREAM` [*Type*]

The type of CIOBL streams.

`CIOBL_ENCODING` [*Type*]

An enumeration type, used to indicate the encoding desired to **make_ciobl_stream**.

`CIOBL_ERROR` [*Type*]

`CIOBL_ERROR ciobl_error` [*Variable*]

All CIOBL functions set the global variable `ciobl_error` before returning; the type of this variable is `CIOBL_ERROR`, an enumeration type. If no error or exceptional condition occured, it is set to `CIOBL_NO_ERROR` (which has value zero). The other possibilities for **ciobl_error** are as follows:

`CIOBL_OS_ERROR` An operating system error occured during an I/O operation. In this case the standard C variable **errno** is set to indicate the error, according to the conventions established by the operating system.

`CIOBL_LIMITATION` A limitation peculiar to the particular implementation of the C interface, such as a buffer size, was exceeded.

`CIOBL_EOF` A read function was called with the CIOBL stream positioned at end of file.

`CIOBL_INTEOF` During a read function, end of file was reached within an object.

`CIOBL_BADARG` Some argument to a CIOBL function was out of range.

`CIOBL_OVERFLOW` The variable-length object read was too large to be returned properly.

`CIOBL_ROUNDOFF` The variable-length float read was too precise to be returned fully.

`CIOBL_BADVERSION` An improperly formatted version token was read.

`CIOBL_INCOMPATIBLE` A file written with an incompatible version of CIOBL was opened for reading.

Any of the functions in the C interface may raise the `CIOBL_OS_ERROR` or `CIOBL_LIMITATION` conditions. The other conditions can only occur as documented with each CIOBL function, below. With the exception of the `CIOBL_OVERFLOW` and `CIOBL_ROUNDOFF` conditions, the CIOBL stream is left in an undefined state after an error occurs, and so further I/O on that stream is generally precluded.

`CIOBL_STREAM make_ciobl_stream (FILE *underlying_stream, STRING type, CIOBL_ENCODING encoding)` [*Function*]

Returns a CIOBL stream such that CIOBL read and write calls performed on that stream will cause corresponding reads and writes to be performed on **underlying_stream**, where **underlying_stream** is a stream as returned by **fopen** or **fdopen**. The value of **encoding** gives the encoding to be used in translating between objects in the CIOBL stream and characters or bytes in the underlying stream; once established, the encoding may not be changed. The value of **encoding** must be one of `CIOBL_STANDARD`, `CIOBL_COMPRESSED`, or `CIOBL_BINARY`. The value of **underlying_stream** must be a stream to which character I/O can be performed if **encoding** is `CIOBL_STANDARD` or `CIOBL_COMPRESSED`, or a stream to which 8-bit byte I/O can be performed if **encoding** is `CIOBL_BINARY`.

The directionality of the ciobl stream returned is established by **type**: a value of "`r`" yields an input stream, a value of "`w`" or "`a`" yields an output stream, and a value of "`r+`", "`w+`",

or "a+" yields a bidirectional stream. It is the responsibility of the user to ensure that the directionality of underlying_stream is compatible with type. The legal values for type were chosen to be compatible with the standard function fopen, but note that for the purposes of make_ciobl_stream there is no distinction between "w" and "a", nor between "r+", "w+", and "a+".

If type specifies an output or bidirectional stream, then make_ciobl_stream will write a version token onto the stream before returning. If type specifies an input or bidirectional stream, then the first call to a CIOBL reading function will read the first four characters or bytes from the stream and interpret them as a version token.

If type specifies an input or bidirectional stream, then the following two errors can occur if an improper version token is read. If the version token is properly formatted but specifies a version of CIOBL incompatible with the version being used, ciobl_error is set to CIOBL_INCOMPATIBLE. In this case, the stream is still opened for reading and writing (if applicable), but calls to read functions can behave unpredictably. If the version token is not even properly formatted, or indicates an encoding different from encoding, then ciobl_error is set to CIOBL_BADVERSION. In that case, the stream will not be open for input (although it will still be open for output if type specified a bidirectional stream).

`VOID close_ciobl (CIOBL_STREAM ciobl_stream)` [*Function*]

A call to close_ciobl closes the stream underlying ciobl_stream, performing any cleanup operations that calling fclose on the underlying stream would perform. It is imperative that close_ciobl be called on the CIOBL stream rather than simply closing fclose on the underlying stream, because there may be cleanup operations specific to CIOBL.

`CIOBL_STREAM make_ciobl_file (STRING filename, STRING type,`
`CIOBL_ENCODING encoding)` [*Function*]

Make_ciobl_file is equivalent to calling fopen on arguments filename and type, and passing the result to make_ciobl_stream along with type and encoding. If, however, the call to fopen resulted in an error, the call to make_ciobl_stream is not made, and ciobl_error is set to CIOBL_OS_ERROR.

7.5.2 Reading and Writing Fixed-Length Objects

`auINT32 ciobl_read_unsigned_auint32 (auINT16 n,`
`CIOBL_STREAM ciobl_stream)` [*Function*]

`auINT64 ciobl_read_unsigned_auint64 (auINT16 n,`
`CIOBL_STREAM ciobl_stream)` [*Function*]

Both functions read and return the next n-bit unsigned integer from the CIOBL stream ciobl_stream; n must be a positive integer. The value x returned will always be in the range $0 \leq x < 2^n$. Unpredictable results will occur if the stream is not positioned at a place written by a corresponding ciobl_write_unsigned_x, with the same value of n (only the n's have to match; it is permissible to write an integer with ciobl_write_unsigned_auint64 and read it with ciobl_read_unsigned_auint32, for example, as long as both were given the same argument n). If the CIOBL stream is at end-of-file before the call, zero is returned, and ciobl_error is set to CIOBL_EOF. On the other hand, if end-of-file occurs in the middle of reading the unsigned, zero is returned, and ciobl_error is set to CIOBL_INTEOF.

The two functions differ in the type of the result returned, which in turn restricts the range of n allowed: for `ciobl_read_unsigned_auint32`, n must be less than or equal to 32, for `ciobl_read_unsigned_auint64`, n must be less than or equal to 64. If an inappropriate n is given, zero is returned and `ciobl_error` is set to `CIOBL_BADARG`.

```
asINT32 ciobl_read_signed_asint32 (auINT16 n,                    [Function]
                                   CIOBL_STREAM ciobl_stream)
asINT64 ciobl_read_signed_asint64 (auINT16 n,                    [Function]
                                   CIOBL_STREAM ciobl_stream)
```

Both functions read and return the next n-bit signed integer from the CIOBL stream `ciobl_stream`; n must be a positive integer. The value x returned will always be in the range $-2^{n-1} \leq x < 2^{n-1}$. Unpredictable results will occur if the stream is not positioned at a place written by a corresponding `ciobl_write_signed_`x, with the same value of n (only the n's have to match; it is permissible to write an integer with `ciobl_write_signed_asint64` and read it with `ciobl_read_signed_asint32`, for example, as long as both were given the same argument n). The end-of-file treatment and the restrictions on n are the same as for `ciobl_read_unsigned_auint32` and `ciobl_read_unsigned_auint64`.

```
sFLONUM ciobl_read_single (CIOBL_STREAM ciobl_stream)            [Function]
dFLONUM ciobl_read_double (CIOBL_STREAM ciobl_stream)            [Function]
CHARACTER ciobl_read_char (CIOBL_STREAM ciobl_stream)            [Function]
```

Reads and returns the next single, double, or char from the CIOBL stream *ciobl-stream*. Unpredicatable results will occur if the stream is not positioned at a place written by a corresponding `ciobl_write_single`, `ciobl_write_double`, or `ciobl_write_char`. The end-of-file treatment is the same as for `ciobl_read_unsigned_`x.

```
VOID ciobl_write_unsigned_auint32 (auINT16 n, auINT32 u,         [Function]
                                   CIOBL_STREAM ciobl_stream)
VOID ciobl_write_unsigned_auint64 (auINT16 n, auINT64 u,         [Function]
                                   CIOBL_STREAM ciobl_stream)
```

Writes the n-bit unsigned integer `u` on `ciobl_stream`. The value of `u` must be in the range $0 \leq u < 2^n$. Note that the suffix of the function name in no way constrains the value of n; for example, it is permissible to call `ciobl_write_unsigned_auint32` with $n = 48$. The suffix merely indicates the C type of the second argument.

```
VOID ciobl_write_signed_asint32 (auINT16 n, asINT32 s,           [Function]
                                 CIOBL_STREAM ciobl_stream)
VOID ciobl_write_signed_asint64 (auINT16 n, asINT64 s,           [Function]
                                 CIOBL_STREAM ciobl_stream)
```

Writes the n-bit signed integer `s` on `ciobl_stream`. The value of `s` must be in the range $-2^{n-1} \leq s < 2^{n-1}$. Note that the suffix of the function name in no way constrains the value of n; for example, it is permissible to call `ciobl_write_signed_asint32` with $n = 48$. The suffix merely indicates the C type of the second argument.

```
VOID ciobl_write_single (sFLONUM x, CIOBL_STREAM ciobl_stream)   [Function]
VOID ciobl_write_double (dFLONUM x, CIOBL_STREAM ciobl_stream)   [Function]
VOID ciobl_write_char (CHARACTER x, CIOBL_STREAM ciobl_stream)   [Function]
```

Each writes the object x on *ciobl-stream*, as the appropriate fixed-length object.

7.5.3 Reading and Writing Variable-Length Objects

`auINT32 ciobl_read_integer_auint32 (CIOBL_STREAM ciobl_stream)` [*Function*]
`asINT32 ciobl_read_integer_asint32 (CIOBL_STREAM ciobl_stream)` [*Function*]
`auINT64 ciobl_read_integer_auint64 (CIOBL_STREAM ciobl_stream)` [*Function*]
`asINT64 ciobl_read_integer_asint64 (CIOBL_STREAM ciobl_stream)` [*Function*]

Each function reads and returns a variable-length integer from `ciobl_stream`. If the stream was positioned at end-of-file before the call, zero is returned, and `ciobl_error` is set to `CIOBL_EOF`. On the other hand, if end-of-file occurs in the middle of reading the object, zero is returned and `ciobl_error` is set to `CIOBL_INTEOF`. If the stream was positioned at a variable-length object other than an integer, zero is returned and `ciobl_error` is set to `CIOBL_BADOBJ`. Unpredictable results will occur if the stream was positioned at a fixed-length object.

Because the range of the return type is limited, the stream must be positioned at an integer that is within the range representable in the return type of each function. If positioned at some other integer, zero is returned, and `ciobl_error` is set to `CIOBL_OVERFLOW`. This is the only error condition which guarantees the position of `ciobl_stream` afterward: it is positioned immediately after the offending integer.

`sFLONUM ciobl_read_float_sflonum (CIOBL_STREAM ciobl_stream)` [*Function*]
`dFLONUM ciobl_read_float_dflonum (CIOBL_STREAM ciobl_stream)` [*Function*]

Each function reads and returns a variable-length float from `ciobl_stream`. The error conditions `CIOBL_EOF`, `CIOBL_INTEOF`, and `CIOBL_BADOBJ` are raised in analogous circumstances as for `ciobl_read_integer_`x, with zero returned in each case. Unpredictable results will occur if the stream was positioned at a fixed-length object.

If the magnitude of the float read is too large to allow the float to be represented in the return type, zero is returned and `ciobl_error` set to `CIOBL_OVERFLOW`. If the magnitude is not too large but the float has more precision than the return type, it is rounded to the nearest float of the return type, and `ciobl_error` is set to `CIOBL_ROUNDOFF`. This is true even if the extra mantissa bits were all zero. `CIOBL_OVERFLOW` and `CIOBL_ROUNDOFF` are the only errors which guarantee the position of `ciobl_stream` afterward: it is positioned immediately after the offending float.

`CHARACTER ciobl_read_character (CIOBL_STREAM ciobl_stream)` [*Function*]

Reads and returns a variable-length character from `ciobl_stream`. The error conditions `CIOBL_EOF`, `CIOBL_INTEOF`, and `CIOBL_BADOBJ` are raised in analogous circumstances as for `ciobl_read_integer_`x, with an unspecified character returned in each case. Unpredictable results will occur if the stream was positioned at a fixed-length object.

`STRING ciobl_read_string (CIOBL_STREAM ciobl_stream)` [*Function*]
`VOID ciobl_read_string_destructively (auINT32 n, STRING s, CIOBL_STREAM ciobl_stream)` [*Function*]

`Ciobl_read_string` reads and returns a string from `ciobl_stream`; the string returned is newly allocated via `malloc`.

`Ciobl_read_string_destructively` is similar, but instead of `malloc`ing the string it is destructively read into `s`, with a null character terminating. Elements of `s` beyond the null character stored are not modified. No more than `n` elements of `s`, however, will be modified; hence the string read must be no longer than $n-1$ characters in order to be properly stored in `s`

(including the null terminator). If `ciobl_stream` is positioned at a string longer than $n-1$, only the first n characters will be stored in `s`, with no terminating null, and `ciobl_error` will be set to `CIOBL_OVERFLOW`. This is the only error which guarantees the position of `ciobl_stream` afterward: it is positioned immediately after the offending string.

Both functions raise the error conditions `CIOBL_EOF`, `CIOBL_INTEOF`, and `CIOBL_BADOBJ` in analogous circumstances as for `ciobl_read_integer_`x. If any of these conditions is raised, `ciobl_read_string` returns the `NULL` pointer, while `ciobl_read_string_destructively` may perform arbitrary modifications to the first `n` elements of `s`. Unpredictable results will occur if the stream was positioned at a fixed-length object.

`KEYWORD ciobl_read_keyword (CIOBL_STREAM ciobl_stream)` [*Function*]
Reads and returns a keyword from `ciobl_stream`. The error conditions `CIOBL_EOF`, `CIOBL_INTEOF`, and `CIOBL_BADOBJ` are raised in analogous circumstances as for `ciobl_read_integer_`x. Unpredictable results will occur if the stream was positioned at a fixed-length object.

`VOID ciobl_write_integer_auint32 (auINT32 x, CIOBL_STREAM ciobl_stream)` [*Function*]
`VOID ciobl_write_integer_asint32 (asINT32 x, CIOBL_STREAM ciobl_stream)` [*Function*]
`VOID ciobl_write_integer_auint64 (auINT64 x, CIOBL_STREAM ciobl_stream)` [*Function*]
`VOID ciobl_write_integer_asint64 (asINT64 x, CIOBL_STREAM ciobl_stream)` [*Function*]
`VOID ciobl_write_float_sflonum (sFLONUM x, CIOBL_STREAM ciobl_stream)` [*Function*]
`VOID ciobl_write_float_dflonum (dFLONUM x, CIOBL_STREAM ciobl_stream)` [*Function*]
`VOID ciobl_write_character (CHARACTER x, CIOBL_STREAM ciobl_stream)` [*Function*]
`VOID ciobl_write_string (STRING x, CIOBL_STREAM ciobl_stream)` [*Function*]
`VOID ciobl_write_keyword (KEYWORD x, CIOBL_STREAM ciobl_stream)` [*Function*]
Each function writes the object `x` as the appropriate variable-length object to `ciobl_stream`.

7.6 Estimating the Length of CIOBL Files

The designer of a CIOBL file format faces many choices in deciding what CIOBL objects to use to represent the components of the file. In particular, the designer must choose between fixed-length and variable-length numbers, strings and keywords, lists and arrays, and so forth. The following information may prove useful in the event that the designer's main goal is to minimize (or maximize, for that matter!) the space occupied by a Binary or Compressed version of his file.

Object	*Characteristics*	*Size in Binary Encoding*
Integer	$x = 0$	2
	$256^{N-1} \le \lvert x \rvert < 256^N$, $1 \le N \le 4$	$1 + N$
	$256^{N-1} \le \lvert x \rvert < 256^N$, $4 < N < 256$	$2 + N$
Float	b bit mantissa, e bit exponent	$3 + \lceil b/8 \rceil + \lceil e/8 \rceil$
Character		2
String	$len < 256$	$2 + len$
	$256^{N-1} \le len < 256^N$, $2 \le N < 256$	$2 + N + len$

Symbol	NIL	1
	Keyword, first occurence	
	$len < 256$	$2 + len$
	$256^{N-1} \leq len < 256^N$, $2 \leq N < 256$	$2 + N + len$
	Keyword, later occurence	2, 3, or 4 (see below)
	Other symbol, first occurence	
	$len < 256$	$2 + len$ (see below)
	$256^{N-1} \leq len < 256^N$, $2 \leq N < 256$	$2 + N + len$ (see below)
	Other symbol, later occurence	2, 3, or 4 (see below)
Unsigned	n bits	$\lceil n/8 \rceil$
Signed	n bits	$\lceil n/8 \rceil$
Single		4
Double		8
Char		1
List		2 + total bytes in components
Array		2 + total bytes in dimension integers + total bytes in components
Dotted List		2 + total bytes in components
User Defined		2 + total bytes in type symbol + total bytes in components

Subsequent occurences in a Binary file of the same symbol take fewer bytes because of the 'predefined symbol' punctuation byte. Subsequent occurences take two bytes for the first 256 unique symbols, three bytes for the next 65280, and four bytes for the next 16,711,680. The first occurence of symbols other than keywords may require the insertion of a 'set current package' directive, which takes the same number of bytes as given above for a string, where *len* is the number of characters in the package name.

Object	*Characteristics*	*Size in Compressed Encoding*
Integer	$x = 0$	2
	$64^{N-1} \leq \|x\| < 64^N$, $1 \leq N \leq 4$	$1 + N$
	$64^{N-1} \leq \|x\| < 64^N$, $4 < N < 64$	$2 + N$
Float	b bit mantissa, e bit exponent	$4 + \lceil b/6 \rceil + \lceil e/6 \rceil$
Character		2
String	$len < 64$	$2 + len$
	$64^{N-1} \leq len < 64^N$, $2 \leq N < 64$	$2 + N + len$

Symbol	NIL	1
	Keyword, first occurence	
	$len < 64$	$2 + len$
	$64^{N-1} \leq len < 64^N$, $2 \leq N < 64$	$2 + N + len$
	Keyword, later occurence	2, 3, or 4 (see below)
	Other symbol, first occurence	
	$len < 64$	$2 + len$ (see below)
	$64^{N-1} \leq len < 64^N$, $2 \leq N < 64$	$2 + N + len$ (see below)
	Other symbol, later occurence	2, 3, or 4 (see below)
Unsigned	n bits	$\lceil n/6 \rceil$
Signed	n bits	$\lceil n/6 \rceil$
Single		6
Double		11
Char		1
List		2 + total chars in components
Array		2 + total chars in dimension integers + total chars in components
Dotted List		2 + total chars in components
User Defined		2 + total chars in type symbol + total chars in components

Subsequent occurences in a Compressed file of the same symbol take fewer characters because of the 'predefined symbol' punctuation character. Subsequent occurences take two characters for the first 64 unique symbols, three characters for the next 4032, and four characters for the next 258,048. The first occurence of symbols other than keywords may require the insertion of a 'set current package' directive, which takes the same number of characters as given above for a string, where *len* is the number of characters in the package name.

7.7 Character Codes

The following table lists the 96 standard characters. The *byte* column gives the representation (in base 10) for each character in the Binary encoding. The *value* column gives the value assigned to characters when used to encode integers in the Compressed encoding, and when used as part of the version token in both the Compressed and the Standard encoding.

Char.	*Byte*	*Value*	*Char.*	*Byte*	*Value*	*Char.*	*Byte*	*Value*
Space	32	0	@	64	32	‘	96	
!	33	1	A	65	33	a	97	
"	34	2	B	66	34	b	98	
#	35	3	C	67	35	c	99	
$	36	4	D	68	36	d	100	
%	37	5	E	69	37	e	101	
&	38	6	F	70	38	f	102	
’	39	7	G	71	39	g	103	
(	40	8	H	72	40	h	104	
)	41	9	I	73	41	i	105	
*	42	10	J	74	42	j	106	
+	43	11	K	75	43	k	107	
,	44	12	L	76	44	l	108	
-	45	13	M	77	45	m	109	
.	46	14	N	78	46	n	110	
/	47	15	O	79	47	o	111	
0	48	16	P	80	48	p	112	
1	49	17	Q	81	49	q	113	
2	50	18	R	82	50	r	114	
3	51	19	S	83	51	s	115	
4	52	20	T	84	52	t	116	
5	53	21	U	85	53	u	117	
6	54	22	V	86	54	v	118	
7	55	23	W	87	55	w	119	
8	56	24	X	88	56	x	120	
9	57	25	Y	89	57	y	121	
:	58	26	Z	90	58	z	122	
;	59	27	[	91	59	{	123	
<	60	28	\	92	60	|	124	
=	61	29	]	93	61	}	125	
>	62	30	^	94	62	~	126	
?	63	31	_	95	63	Newline	10	