LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

ID Run-time System

Computation Structures Group Memo 311
November 10, 1990

Stephen Brobst

James Hicks

Gregory Papadopoulos

Jonathan Young

This report describes research done at 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.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

ID Run-time System

Stephen Brobst James Hicks Gregory Papadopoulos Jonathan Young

November 10, 1990

Contents

1 ID Run-time System External Specifications

This section describes our first cut at the external specifications for the ID Run-time System (RTS). These specifications are by no means intended to be exhaustive, but rather represent a starting point for development of the RTS. We expect to experiment with a number of different implementations as the RTS matures. We divide the specifications into two types: file system interfaces and storage management interfaces. These are described in the next two subsections.

1.1 File System Interface

The file system interface is modeled after Common Lisp [2] system calls for low level I/O. To open (close) files for reading and writing the standard functions are required. Note that we will not, initially, implement protection modes on the file system interfaces.

open *filename &key :direction :if-exists :if-does-not-exist* ⇒ *fd* [*RTS Operation*]

The *filename* argument is a pointer to a character string containing the name of the file to be opened. The element type for the stream transactions will be defined as buffer of unsigned bytes. The return value of this function is an integer file descriptor for the opened file.

:direction

This argument specifies whether the stream should handle input, output, or both.

- :input

  The result will be an input stream. This is the default.

- :output

  The result will be an output stream.

- :io

  The result will be a bidirectional stream.

:if-exists

This argument specifies the action to be taken if the :direction is :output or :io and a file of the specified name already exists. If the direction is :input, this argument is ignored.

- :error

  Signals an error. This is the default.

- :rename-and-delete

  Renames the existing file to some other name and then deletes it. Then creates a new file with the specified name.

- :overwrite

  Uses the existing file. Output operations on the stream will destructively modify the file. If the :direction is :io, the file is opened in a bidirectional mode that allows both reading and writing. The file pointer is initially positioned at the beginning of the file; however, the file is not truncated back to length zero when it is opened.

- :append

  Uses the existing file. Output operations on the stream will destructively modify the file. The file pointer is initially positioned at the end of the file. If the :direction is :io, the file is opened in a bidirectional mode that allows both reading and writing.

- nil

  Does not create a file or even a stream, but simply returns a file descriptor with the value -1 to indicate failure.

:if-does-not-exist
This argument specifies the action to be taken if a file of the specified name does not already exist.

- :error

  Signals an error. This is the default if the :direction is :input, or if the :if-exists argument is :overwrite or :append.

- :create

  Creates an ampty file with the specified name and then proceeds as if it had already existed (but does not perform any processing directed by the :if-exists argument). This is the default if the :direction is :ouput or :io, and the :if-exists argument is anything but :overwrite or :apend.

- nil

  Does not create a file or even a stream, but simply returns a file descriptor with the value -1 to indicate failure.

`close` *fd* $\Rightarrow$ *void* [*RTS Operation*]

The `CLOSE` operation breaks the connection between the file descriptor *fd* and its associated open file. The file is closed and the file descriptor is freed for re-use with some other file. It is an error to call an I/O function using a file descriptor after it has been freed for re-use.

Note that subsequent calls to `OPEN` may recycle *fd*. As in previous sections, interleaved calls to `OPEN` and `CLOSE` will use a bounded number of file descriptors.

The standard *read* and *write* functions are required as a means of performing the actual I/O functions:

`read` *fd buf n* $\Rightarrow$ *nread* [*RTS Operation*]

This function takes as arguments a file descriptor (*fd*), a pointer to a pre-allocated buffer (*buf*), and the number of bytes to read (n). It then reads n bytes from the file specified by *fd* into the pre-allocated buffer (*buf*). Its return value (*nread*) is an integer indicating the number of bytes read. A return value of zero means end-of-file, a value of -1 means an error of some sort, and a value less than n means that only *nread* bytes were remaining in the file.

`write` *fd buf n* $\Rightarrow$ *nwritten* [*RTS Operation*]

This function takes as arguments an integer file descriptor (*fd*), a pointer to the buffer to be written (*buf*), and the number of character bytes to write (n). It then appends n character bytes to the file

specified by *fd* from the specified buffer (buf). Its return value (*nwritten*) is an integer indicating the number of bytes written. There is generally some kind of error condition if *nwritten* is not equal to *n*.

The `read` and `write` procedures both use buffers for passing character byte-vectors. These buffers have imperative (non-i-structure) semantics. Using imperative buffers allows efficient recycling of buffers, because they do not have to be explicitly cleared between uses — their contents are overwritten. This also makes the implementation of `write` easier because it does not have to defer on empty elements of the buffer.

`ERROR` *string* [*RTS Operation*]

Signals an error to the host. Other threads executing on the machine will not be interrupted; as in the TTDA model, when the machine becomes idle, the execution manager will try to print out the *string* from each error on the console.

1.2 Storage Allocation Interface

The storage allocation system has two classes of interfaces: one for context allocations and deallocations, and a second for heap allocations and deallocations. These interfaces will be described in this section.

The Monsoon PE is a pipelined architecture. Currently, the machine can execute up to eight separate "sequential threads" in parallel. Although the ID language and compiler ensure that no two user threads can interfere with each other, the RTS routines must use locks and careful discipline to avoid interfering with any other thread.

RTS routines are called via asynchronous traps from user code. On the Monsoon PE, asynchronous traps are allocated a special *ephemeral context*. There is a different ephemeral context for each thread. Thus, trap code such as the RTS is constrained to be an uninterrupted thread of instructions unless it allocates another context.

Non-local memory references are *two-phase*, and may take an arbitrary amount of time to transmit and process. Thus, *low-level RTS routines may access only processor-local state.* This restriction is most exacting on the context-management routines; all other traps may simply allocate a context (e.g. make a procedure call) in order to access non-local state.

2 Context Management

A *context* is a continuation (as defined in SWIS) whose IP[1] points to the beginning of a code block and whose Node and FP point to the beginning of a portion of local memory on a processing element, called a *frame*. A *frame* is assumed to be a contiguous, non-interleaved block of storage. Each context also has associated with it a *color* (for statistics) and a *mode* (for system/user distinctions).

A *code block descriptor pointer* (cbdp) points to the beginning of a data structure containing four pieces of information: (1) a code pointer, (2) a symbol pointer, (3) a thunk pointer (if present), and (4) the color and mode of the code block. For the purposes of context management, we are interested only in fields (1) and (4) of this data structure. A *code pointer* (cp) points to the beginning of a code block in instruction memory. When making a procedure call, it is the responsibility of the calling procedure to determine the code block descriptor pointer of the procedure being called.

[1] It is assumed that an IP points to the same instruction on each PE.

Contexts are either *fixed-size* or *variable-size*; the size of a fixed-size context is not specified here. It is expected that fixed-size contexts are more efficiently allocated than variable-sized ones. (On the other hand, no similar performance differential is expected between the allocation of local and non-local contexts.)

Since each context points to storage on one particular PE, we plan to achieve load balancing in the near term by distributing the contexts randomly to all PEs, so each PE manages a collection of contexts from all other PEs. Currently, there is no plan to achieve load balancing by migrating contexts across PEs; a context will always be managed by the same PE.

`GET-CONTEXT` *cbdp* ⇒*context* [*RTS Operation*]

Returns a context for use in calling the procedure described by the code block descriptor pointer *cbdp*. Allocates temporary frame storage on an arbitrary PE and returns a continuation containing the initial IP of the procedure, the color of the procedure, and a pointer to the frame storage. Every word in the active area of the frame is guaranteed to have presence bits **empty**.

`RETURN-CONTEXT` *context* ⇒*void* [*RTS Operation*]

Informs the RTS that all activity has ceased in procedure activation corresponding to *context*, which must be a context which was returned by `GET-CONTEXT`. In particular, every word in the active area of the returned context must have presence bits **empty**. Executing this instruction enables the reuse of the temporary storage allocated.

We do not guarantee that a `GET-CONTEXT` which immediately follows a `RETURN-CONTEXT` will return a context which points to the same portion of frame storage. However, any *interleaved sequence* of calls to `GET-CONTEXT` and `RETURN-CONTEXT` in which each call waits for the previous call to finish[2] and in which each context allocated is returned is guaranteed to use no net resources; the amount of frame storage available at the end of the sequence will be the same as the amount available at the beginning.

It is an error to return a context more than once.

`GET-VARIABLE-CONTEXT` *cp length* ⇒*context* [*RTS Operation*]

Returns a context, fully instantiated, for use in calling the procedure described by the code pointer *cp*. Allocates temporary frame storage of at least size *length* (integer) on an arbitrary PE and returns a continuation containing the initial IP of the procedure, the color of the procedure, and a pointer to the frame storage.

`RETURN-VARIABLE-CONTEXT` *context* ⇒*void* [*RTS Operation*]

Like `RETURN-CONTEXT`.

`GET-LOCAL-CONTEXT` *cp* ⇒*context* [*RTS Operation*]

Same as `GET-CONTEXT` except the PE of the context returned is constrained to be the same as the current PE.

`RETURN-LOCAL-CONTEXT` *context* ⇒*void* [*RTS Operation*]

Like `RETURN-CONTEXT`.

`GET-LOCAL-VARIABLE-CONTEXT` *cp length* ⇒*context* [*RTS Operation*]

Returns a context, fully instantiated, for use in calling the procedure described by the code pointer *cp*. Allocates temporary frame storage of at least size *length* (integer) on the current PE and returns

[2] A token returned by an invocation of an SVC is sufficient to indicate that the call has finished.

a continuation containing the initial IP of the procedure, the color of the procedure, and a pointer to the frame storage.

`RETURN-LOCAL-VARIABLE-CONTEXT` *context* ⇒ *void* [*RTS Operation*]
Like `RETURN-VARIABLE-CONTEXT`.

2.0.1 Heap Management

The ID heap consists of those storage locations which are managed as aggregates by the ID RTS. The heap is a *global* resource; the system makes no effort to allocate objects *near* to a given processor.

Similar to the contract obeyed by the frame management instructions, we guarantee that there is a discipline under which repeated calls to the allocator will not use an unbounded amount of storage.

`GET-AGGREGATE` *length* ⇒ *object* [*RTS Operation*]
Returns an ID aggregate object reference (type "head") to a newly allocated heap object whose active area is of length at least *length* (an integer). Every word in the active area is guaranteed to have presence bits `empty`.

`RETURN-AGGREGATE` *object* ⇒ *void* [*RTS Operation*]
Informs the RTS that no more memory requests will reference *object*, which must be a pointer returned by `GET-AGGREGATE`, and thus that the storage allocated may be reused. It is an error to return an object more than once.

Like the frame manager, we do not guarantee that a `GET-AGGREGATE` which immediately follows a `RETURN-AGGREGATE` will immediately reuse the same storage. However, any interleaved sequence of calls to `GET-AGGREGATE` and `RETURN-AGGREGATE` in which each call waits for the previous call to finish[3] and in which each object allocated is returned is guaranteed to use no net resources; the amount of heap storage available at the end of the sequence will be the same as the amount available at the beginning.

Note that unlike frames, returned aggregates may still have words in the active area which are not `empty`. The `RETURN-AGGREGATE` call ensures that all words become `empty` before this storage is reused.

Do we want GET-LOCAL-OBJECT & RETURN-LOCAL-OBJECT as well?

3 File System Implementation

Using the file system interfaces given in the previous section, along with primitives for sequentializing I/O, we will begin work on the file system in mid-July. However, necessary language support issues are not scheduled to be resolved for a full-blown I/O system in the ID language. Priority will be given to storage management implementation until that time.

4 Storage Management Implementation

This section provides a description of our plan for storage management implementation in the ID run-time system (RTS). The plan is broken into four milestones, as follows:

[3] A token returned by an invocation of an SVC is sufficient to indicate that the call has finished.

1. In-line allocation, no recycling.

2. Supervisor call traps (SVCs) for context allocation, free-list management.

3. Performance optimization.

4. Free-list management of contexts across multiple PEs.

Each milestone represents an enhancement upon its predecessor, based on improved instruction set support or more sophisticated algorithms for storage management.

4.1 Milestone 1: In-line allocation, no recycling

Milestone 1 of the RTS is implemented with IS0, our initial subset of the Monsoon Instruction set. IS0 is limited in that it supports only spin-locks, and does *not* support exceptions or temporaries. Our first prototype of the RTS involves primitives for *get-context* (frame allocation) and *get-aggregate* (heap allocation). The code for these primitives is generated *in-line* by the compiler where needed by the program to perform storage allocation.

Storage is allocated by simply incrementing a pointer into a free space area. Separate pointers at each end of a free space monolith are used to facilitate partitioned allocation of contexts and aggregates. However, since the pointers advance from opposite end of the free space monolith, the size of the partitions allocated to contexts and aggregates may vary dynamically according to program requirements (*i.e.*, there is an overall maximum of storage that can be allocated, but no arbitrary limit is placed on the proportions allocated between contexts and aggregates). Figure 1, below, illustrates this scheme.

In this simple model there is no recycling of storage. When the *context-ptr* and *aggregate-ptr* shown below meet, there is no space left for allocation. In such a case, the behavior of the machine is undefined; it may crash or it may continue executing erroneously by allocating over non-free space.

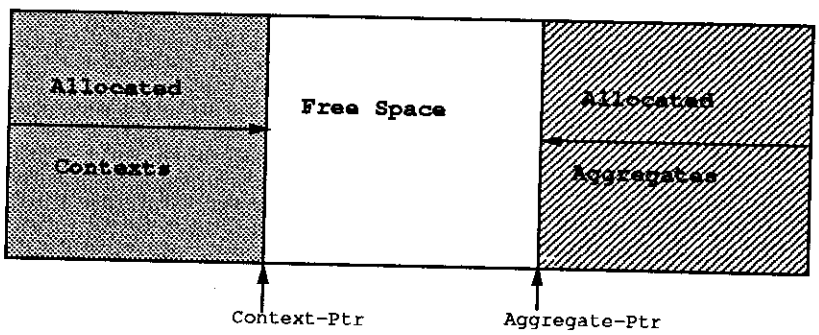

Figure 1: Allocation scheme for *get-context* and *get-aggregate*.

4.2 Milestone 2: SVCs for contexts, free-list management

Milestone 2 of the RTS storage management system is also implemented. It makes use of exceptions as provided by the IS1 subset of the Monsoon instruction set. The milestone 2 storage allocation scheme gets rid of in-line coding for both context and aggregate allocation. Moreover, free-list

management is performed in both cases to facilitate storage recycling. Instead of code in-lining, SVC trap instructions will be generated by the compiler which cause a supervisor call to the appropriate storage management code.

Contexts will be allocated and deallocated from a free list of fixed size frames. Contexts larger than the fixed size are allocated from the heap. Heap management is performed within the supervisor via procedure calls to allocate and deallocate primitives. The free-list management algorithms employed make use of the "true" locks and multiple deferred reads and takes, as provided in the IS1 instruction subset.

4.3 Milestone 3: Performance optimization

Milestone 3 represents a performance tuning of our storage management system. We will experiment with a number of algorithms, including first-fit, quick-fit and the buddy system. The outcome will be to select and optimize a free-list management algorithm which minimizes fast-path code length and keeps its critical section to a minimum. Our goal is to bring storage management overhead down to 20% of instructions executed during a program run. Currently, this overhead is upwards of 45%. Much of this overhead will be trimmed once we go to bulk clearing of memory rather than doing this on a cell-by-cell basis.

To speed up context allocations, a collection of fixed size contexts will be maintained for "quick" context allocation. Whenever size requirements are less than or equal to the fixed size frames, a fast allocation can be made from this free-list. In cases where a larger context is required, the heap mechanism is employed.

In addition, a "caching" scheme will be implemented that allows efficient allocation of fixed size contexts on a thread-local basis. The goal is to avoid interference between threads during context allocation, and to minimize code path lengths. The scheme maintains a small number of contexts which are associated with active threads in the processor.

Fixed size context allocation (and deallocation) under this scheme requires approximately 6 instructions (exception, load cache pointer, return fetched context to caller, increment pointer, store pointer) upon a "cache hit", and approximately 20 instructions when we run out of contexts (or overflow) in the "cache." Statistically, the inefficient case occurs once per 16 requests, so the averaged cost for an allocation (deallocation) is less than 7 instructions.

For a more thorough analysis of tradeoffs and a discussion of various algorithms that have been undertaken in this study, the reader is encouraged to read [5].

4.4 Milestone 4: Storage management across multiple PEs

Although each PE will attempt to allocate contexts from its local store, a request to the "global" store must be issued whenever the local store is exhausted. The global store will be interleaved across multiple PEs in the Monsoon architecture. Moreover, storage allocation requests will be handled by multiple PEs, in parallel.

4.5 Storage Management: Schedule and Status

Our goal is to put an efficient storage allocation system into full production by October 1, 1990. However, free-list management across PEs is not targeted until after the I/O system is put into place. The schedule below lists our start and end dates for each of the four phases in the implementation of the RTS.

Milestone	Target Date

Milestone 1: In-line allocation, no recycling.	Complete
Milestone 2: SVCs for context allocation, free-list management.	Complete
Milestone 3: Performance optimization.	Ongoing
Milestone 4: Free-list management across multiple PEs.	To be determined

4.5.1 Status: Milestone 1

Milestone 1 is complete. The simple allocators have been in use since April. Their purpose is primarily as an interim measure to allow software development in other areas to continue to make progress.

4.5.2 Status: Milestone 2

Milestrone 2 is complete. SVC implementation for context allocation and free-list management have been in place since the summer of 1990.

4.5.3 Status: Milestone 3

Performance optimization of existing free-list algorithms has already begun. A singly-linked list version of the first-fit algorithm has been implemented which reduces both fast path length and resource contention for heap management. Coalescing in this implementation occurs only when there are no free blocks left which are big enough to satisfy a request (as opposed to on a per release basis in the doubly-linked list version of first-fit with boundary tags). This algorithm has been verified on both GITA and Lisp-MINT.

A third version of first-fit which sorts a singly-linked list of free blocks by address will allow coalescing on a per release basis. This algorithm has been implemented and tested on GITA, but is still under debug for Monsoon in the Lisp-Mint environment. Performance analysis of fast path length and parallelism is now underway.

We have also begun analysis on dynamic program profiles of storage allocation and deallocation requests. In particular, we are interested in deriving the size profile of heap requests during program execution to facilitate an efficient selection of storage thresholds for the quick-fit algorithm. The quick-fit has been implemented, but not integrated into the "production" system.

The "caching" scheme for context allocation has been designed, but not implemented. We expect this to be straightforward, however.

4.5.4 Status: Milestone 4

Milestone 4 is on hold until after the I/O system has been implemented. The milestone 3 storage allocator should be sufficient for most large applications. Code has been written for across PE allocations, but not debugged. Further design discussions need to take place in regard to how allocated storage will be interleaved and so on.

5 Implementation of the First Milestone Storage Management System

This section gives the implementation details of the milestone 1 storage management system. This code has been implemented and tested, and has been running successfully on ID World 0 for some

time now.

5.1 Get-Context

This code is inlined wherever a call to *get-context* is performed, because SVC instructions are not implemented in IS0.

This version of Get-context ignores the value on the incoming token. Get-Context takes the free-frame pointer, decrements it by the frame size, stores this frame-pointer back to the free-frame pointer. This new frame pointer is also returned as the pointer to the frame that has been allocated.

Here is the Monasm code for get-context:

```
get_context:
        stake   [FRAME_POINTER]          >
        ap      v, [NEG_FRAME_SIZE]      >
        mov     v, [FRAME_POINTER]
```

5.2 Return-context

This call is not implemented in the milestone 1 storage management system.

5.3 Get-Aggregate

Again, because SVC instructions are not available in IS0, the code for this manager call is inlined directly into ID code wherever *get-aggregate* is called.

Upon entry to the get-aggregate code, v contains the length of the object to be allocated. Get-aggregate *takes* the heap pointer, returns the current contents as the pointer to the object, then increments the contents by the length to be allocated, and stores that pointer back as the new heap pointer.

Here is the Monasm code for get-aggregate.

```
get_aggregate:
        stake   [HEAP_POINTER]      >
        fork    OUTPUT              >
        ap      v, [fp+r]           >
        mov     v, [HEAP_POINTER]   >
        fork    SIGNAL
```

5.4 Return Aggregate

This call is not implemented in the milestone 1 storage management system.

6 Implementation Details for the Second Milestone Storage Management System

This section gives the details of the milestone 2 storage management system. This implementation requires ID World 0 running microcode for Instruction Subset 1[1] plus the ID Compiler v1 and

the restricted loader (described in *Monsoon Software Interface Specifications[4]*). ID World 0 will also have a simple execution manager.

The second milestone storage management system requires a context manager, a heap manager, and execution manager support. The context manager consists of three handlers: get-context, return-context and initialize-free-contexts. Likewise, the heap manager consists of three handlers: get-aggregate, return-aggregate and initialize-heap.

6.1 Context Management

Contexts are managed by a simple free list. Get-Context consists of popping a frame off of the free list and creating a continuation whose IP is the entry point of the procedure to be called and whose FP is the address of the allocated frame. Return-Context consists of pushing the frame to be deallocated onto the free list. Initialize-Contexts builds the initial free frame list.

6.1.1 Get Context

Here is the code that the compiler will generate for `get-context`, expressed in monasm[3].

```
;;; CB-Name enters in v:
get_context:
        fo1     gc.L, mc.L
gc:     svc0    v,[VOID]                    ; send trigger
mc:[r]  mc      vl,vr   || jump dest ; form continuation
```

The handler for the get-context SVC is:

```
svc0_entry:
        mov     v, [fp+0]           >   ; save return continuation
        stake   v, [FREE_POINTER]   >   ; take free-list
        mov     v, [fp+1]           >   ; save frame
        plt     v,                  >   ; take rest of free-list
        mov     v, [FREE_POINTER]   >   ; put new free-list
        mov     [fp+0], v           >   ; return continuation
        aics    v, [fp+1], 1            ; send frame to instruction
                                        ; following SVC0.
```

6.1.2 Return Context

Here is the code that the compiler will generate for `get-context`, expressed in monasm[3].

```
   ;;; Continuation enters in v:
rc:     svc1    v
```

The handler for the return-context SVC is:

```
svc1_handler:
        mov     v, [fp+0]                       ; save return continuation
                || %exc1 xa, v          >       ; and read xa (frame-pointer)
        mov     v, [fp+1]               >       ; save frame
        stake   v, [FREE_POINTER]       >       ; take free-list
        plp     v, [fp+1]               >       ; add frame to free-list
        mov     [fp+1], v               >
        mov     v, [FREE_POINTER]       >       ; store new free-list
        mov     [fp+0], v               >
        aics    v,[VOID]                        ; send ack
```

6.1.3 Initialization of Context Free List

```
define free_pointer_offset, 32
define frame_size, 128.0
define frame_area_lb,  3000x
define frame_area_ub, 10000x

absolute        dm, node, offset {
FREE_POINTER:
        word    pb_read_only,tb_tb_internal,
                <ptrinfo>0 ^ <map>0 ^ <node>0 ^ <offset>free_pointer_offset
FRAME_SIZE:
        word    pb_read_only, tb_float, frame_size
FRAME_AREA_LB:
        word    pb_read_only, tb_signed, frame_area_lb_offset
FRAME_AREA_LEN:
        word    pb_read_only, tb_float, frame_area_ub_offset
}

absolute        im, node, offset {
CLEAR_CONTEXTS:
        mov     [ZERO], v
        mov     v,[fp+0]        ; INDEX
LOOP:   fgep    v,[FRAME_AREA_LEN]
        swt     v,[fp+0],done
        fo1     APC.R, $+1
        mov     [FRAME_AREA_LB], v      || jump APC
FCIT:   fcit    v,[fp+0]                || jump $1.r
APC: [fp+1]:
        apc     vl,vr           ; clear location
        mov     [fp+0],v        ; increment INDEX
        fadd    v,[slit$float$one]
        mov     v,[fp+0]
        jump    loop
DONE:   stop
}
```

6.2 Heap Management

The heap manager will be invoked by procedures calls to the Id procedures that implement the heap manager.

6.3 Get Aggregate

Procedure`get_aggregate` is called with a single argument, an integer, which indicates the number of words of storage to allocate. `Get_aggregate` will allocate the object and return a pointer to the object.

6.3.1 Return Aggregate

Procedure `return_aggregate` takes a single argument, which must be a pointer to the first word of the active area[4] of the object to be deallocated. This call will deallocate the object and return an acknowledgment signal.

6.3.2 Initialization of Heap Storage

The heap manager requires a rather complex data structure to be constructed before any calls are made to `get_aggregate` or `return_aggregate`. This initialization is accomplished by the invocation of the procedure `init_heap`.

7 Initialization

This section describes the steps to be performed in setting up Monsoon so that Id programs may be run on it. This starts with the clearing of frame and heap memory and ends with the initialization of the heap data structures. We will assume that the loader is managing its own areas of memory, and that those have been initialized already. Some of these steps could be performed by loading MOC records.

7.1 Exception Handler Initialization

This step initializes the exception vectors so that when the processor executes an SVC instruction or takes an exception, the correct handler will be invoked. This step consists of loading code into the exception vectors and loading any other code that is necessary to support the processing of exceptions. This step is performed only once.

7.1.1 Initialization of Intrinsic Operation Code

This step consists of loading the code for the handling of intrinsic operations, such as `%%ISTR`, which simulates the second half of split phase I-structure operations on the processor. This step is performed only once.

[4] See Chapter 5 of [4] for more details.

7.2 Initialization of Memory

This clears all of frame and heap memory, if necessary. Frame memory must be cleared whenever a program terminates improperly. Normally, programs are self-cleaning — when a frame is returned all slots are empty — and frame memory does not have to be cleared. Heap memory must be cleared whenever it becomes full.

7.2.1 RTS Initialization

The context and structure managers are initialized by executing `initialize_contexts` and then `initialize_heap`.

7.2.2 Global Constants

The value cells for global constant identifiers must be reset so that if they are *forced* during execution they will be recomputed. This is described in Chapter 5 of [4].

7.3 Execution Manager

A single frame is reserved for use by the invocation manager. The invocation manager must read the arguments and transmit them to the procedure being called. It then must cause the processor to execute instructions, while collecting statistics, until the procedure terminates. Finally, it should return or display the result.

7.3.1 Boot Code Block

This code block catches the results from the procedure being invoked, and writes the result and termination to an absolute frame slot, so that the execution manager can read them from the host processor.

References

[1] Michael J. Beckerle and Jonathan Young. Monsoon instruction subsets. Technical Report Internal Memorandum, Motorola Cambridge Research Center and Massachusetts Institute of Technology, Cambridge, MA, June 1990.

[2] Guy L. Steele. Common Lisp. Digital Press, 1990.

[3] Kenneth R. Traub. Monasm reference manual. MCRC-TR 5, Motorola Cambridge Research Center, Cambridge, MA, April 1990.

[4] Kenneth R. Traub, Michael J. Beckerle, James E. Hicks, Gregory M. Papadopoulos, Andrew Shaw, and Jonathan Young. Monsoon software software interface specifications. Technical Report MCRC-TR-1 and CSG Memo 296, Motorola Cambridge Research Center and Massachusetts Institute of Technology, Cambridge, MA, January 1990.

[5] Jonathan Young. Context Management in the ID Run Time System. Computation Structures Group Memo 319, Massachusetts Institute of Technology, Cambridge, MA, September 1990.