An I-Structure Memory Controller (ISMC)

Computation Structures Group Memo 239
23 May 1984

Steven K. Heller

This report describes research done at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Funding for the Laboratory is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-75-C-0661. The author was supported by the International Business Machines Corporation as a part of MIT's VI-A program. This report appeared as the author's Masters Thesis at the Massachusetts Institute of Technology in May, 1983.

An I-Structure Memory Controller (ISMC)

by

Steven K. Heller

Submitted to the Department of
Electrical Engineering and Computer Science
on May 13, 1983 in partial fulfillment of the requirements
for the Degrees of Bachelor of Science and Master of Science
in Computer Science and Engineering

ABSTRACT

I-structure storage is a new type of random access storage for multi-processor systems. By employing extra "status bits," data slots can be tagged as "data-present" or "data-absent," allowing reads which precede writes to be remembered until the data arrives. In order to implement an I-structure Memory (ISM) in hardware, a sophisticated controller is required. An I-structure Memory Controller (ISMC) for the Tagged Token Data Flow Machine is developed using the IDL (Interactive Design Language) design system. The design of the controller and a discussion of the IDL System, as well as suggestions for future controllers, will be presented in this paper.

Keywords: I-structure, memory controller, dataflow, multi-processor, architecture, token, design language.

MIT Thesis Supervisor:

Arvind, Associate Professor of Computer Science

IBM Thesis Supervisor:

Dr. Harold Fleisher, IBM Fellow

ACKNOWLEDGMENTS

I would like to thank several individuals for their assistance: Professor Arvind, for his encouragement, advice, and patience; Dr. Fleisher, Ray Phoenix, and Leon Maissel, for their all around support of my project; Professor Mort Tavel (of the Institute), for his advice; and Eva Wu, for her expert profreading.

I would also like to thank IBM Poughkeepsie, and IBM Cambridge Scientific Center for their support.

TABLE OF CONTENTS

1.0 Introduction 7
1.1 Background Information on I-Structures and IDL 8
1.1.1 Introduction to I-Structures 8
1.1.1.1 Software Motivation for I-Structures 8
1.1.1.2 Kinds of I-Structures 11
1.1.1.3 A Peek at the Implementation of I-Structures . . . 12
1.1.2 Introduction the IDL Design System 13
1.1.2.1 High Level Overview of the IDL Design System . . . 13
1.1.2.2 The IDL Language 15
2.0 I-Structure Memory Operations 17
Status Transition Diagram 18
2.1 Organization of the I-Structure Memory Controller 19
2.1.1 Physical Memory 19
2.1.1.1 The I-Store Section 19
2.1.1.2 The Free-List Section 20
2.2 Normal Operations 21
2.2.1 Store and Fetch Operations 21
2.2.2 Allocate and Clear Operations 30
2.2.3 Reset Operation 31
2.2.4 Allocate-Free-Space Operation 31
2.3 Service Operations 33
2.3.1 Read and Write Absolute Operations 33
2.3.2 Initialize Operation 33
2.3.3 Load Error Destination Operation 35
2.3.4 Get Free Size Operation 35
2.4 Error Handling in the I-Structure Memory Controller . . . 35
2.4.1 Types of ISMC errors 36
2.4.1.1 Status Errors 36
2.4.1.2 Data Type Errors 37
2.4.1.3 Free List Errors 37
2.4.2 Error Recovery 37
3.0 Implementation 40
3.1 Approach to Implementation 40
3.2 Run time flexibility 41
3.2.1 Size of the Free-List 41
3.2.2 Speed of the Memory 42
3.3 Handshaking Protocols 43
3.4 Registers 43
3.5 Hardware Required 44
3.6 Suggestions For Future ISMCs 45
3.6.1 Realizable Model 45
3.6.2 Size of the I-Structure Virtual Address Space 46
3.6.3 Dynamic Refresh 46
3.6.4 End-of-Block Status 47
3.6.5 Internally Delayed Tokens 47
3.6.6 Free-List-Low Warning 47
3.6.7 Status Overloading 48
4.0 Critique of IDL 49
4.1 Problems With Using a High Level Language 49
4.1.1 IDL at Too High a Level 49
4.1.1.1 The Semantics of IDL Assignment 49
4.1.1.2 Hidden Control Signals 53

4.1.1.3 The Counting Constructs 54
4.1.2 IDL at Too Low a Level 58
4.1.2.1 Dot Notation Imitates Nested Conditionals 58
4.1.2.2 Dot Notation Represents Decision Graphs 59
4.1.2.3 Extending Dot Notation 60
4.1.2.4 Cleaning up Dot Notation 61
4.2 The Enumeration Assignment Problem 63
4.3 The Don't Care Input Assignment Problem 64
4.4 A PASCAL-like Language Models a Hardware Design 67
References 69
Appendicies 71
A.0 High Level Code 71
A.1 *store-t-fix Operation 79
A.2 *store-t-var Operation 83
A.3 *store-u-fix Operation 91
A.4 *fetch-type-stored Operation 95
A.5 *fetch-notype-stored 99
A.6 *allocate Operation 102
A.7 *clear Operation 103
A.8 *reset Operation 106
A.9 *read-absolute Operation 107
A.10 *write-absolute Operation 108
A.11 *initialize Operation 109
A.12 *load-error-destination Operation 110
A.13 *allocate-free-space Operation 111
A.14 *get-free-size Operation 113
B.0 IDL Code 114
B.1 *store Operations 123
B.2 *fetch Operations 129
B.3 *initialize Operation 133
B.4 *allocate Operation 134
B.5 *clear Operation 134
B.6 *reset Operation 136
B.7 *allocate-free-space Operation 138
B.8 *read-absolute Operation 138
B.9 *write-absolute Operation 139
B.10 *load-error-destination Operation 139

1.0 INTRODUCTION

The Data Flow Machine, [Arvind 82] and [Arvind and Iannucci 81], employs several novel ideas in support of its unconventional architecture. One of these ideas, the Incremental Structure (I-structure) [Arvind and Thomas 81], is a high level memory abstraction that requires a sophisticated control mechanism for direct hardware implementation.

A micro-code implementation of an I-structure memory controller was developed by Brian Williams [Williams 81]. Because the number of memory related instructions for a Processing Element (PE) of the Data Flow Machine had been increased, and existing instructions had been modified, a need existed for a new design of the I-structure memory controller.

In addition to a hardware design language, the IDL System [Maissel and Ostapko 82] provides a simulation, verification and documentation system for VLSI design. IDL contains several high level constructs: IF-THEN-ELSE, for example, makes it particularly well suited for the development of control mechanisms. Furthermore, the output of the IDL system can be implemented directly in hardware.

This project comprises all stages of the design of an I-structure memory controller in the IDL design system. It includes detailed design specifications, IDL code, suggestions for an improved I-structure memory controller, and discussion of the IDL Design system.

1.1 BACKGROUND INFORMATION ON I-STRUCTURES AND IDL

To provide some background in the areas of I-structures and IDL, a representative picture is described here from which the reader can extrapolate details. Additional information about I-structures and the Dataflow Machine can be found in [Arvind 82], [Arvind and Iannucci 81], and [Arvind and Thomas 81]; and about IDL, in [Maissel and Ostapko 82]. Any reader already familiar with I-structure memory or IDL may wish to skip the rest of this chapter.

1.1.1 INTRODUCTION TO I-STRUCTURES

A data structure known as I-structures have been proposed by Arvind and Thomas to efficiently manipulate arrays in functional languages. I-structure storage, or simply I-store, and associated operations, form an implementation model for I-structures. I-store is an integral part of the Tagged Token Dataflow Machine being developed by the MIT Laboratory for Computer Science by the Functional Languages and Architectures group. This machine will be an embodiment of the U-interpreter developed by Arvind and Gostelow at the University of California, Irvine. After we have described I-store, it will become clear that storage based on these ideas will be useful for any multiprocessor machine.

1.1.1.1 Software Motivation for I-Structures

Functional and applicative languages (in particular the dataflow language, Id, that will execute on our machine) are free from side effects. That is, the result of an operation depends solely on its inputs (as opposed to its input and the state of the machine).

In Id, variables are used to name partial results (as opposed to memory locations) and follow the single assignment restriction: a variable can only be assigned once. However, we can still have loops. Consider the following code fragment:

```
BEGIN-Loop
          .
          .
          .
      NEW x <- f(x,<other variables>)
          .
          .
          .
      END-Loop
```

The x on the left hand side of the assignment statement refers to a different "copy" of x than does the x on the right. In functional languages, all structures (e.g. arrays) are treated as if they were values. Thus, the APPEND array operation is used to modify one element of an array. APPEND (x,i,v) conceptually generates a completely new copy of array x, which differs from x only on selector i. Several methods to implement structures have been suggested that reduce copying without affecting the meaning of the program [Dennis 73] and [Ackerman 77]. However, none of these methods work well for one level structures such as arrays.

While copying may be tolerable for scalars, we can not afford to make unnecessary copies of large arrays in many applications. Consider the following code fragment:

```
FOR i = 1 TO 10 BY 1 DO
          .
          .
          .
    x[i] <- f(x[i-1",<other variables>)
          .
          .
          .
    END-FOR
```

In the above code, we assign each position (i.e. selector) of x only once. Hence, it is not necessary to make a new copy of x each time we execute the body of the loop. If the compiler can detect this condition for a structure, it can avoid a lot of unnecessary copying.

I-structures can be informally defined as follows: a piece of code is an I-structure producer if it assigns the array position associated with any given array selector at most once. When this condition is met, unnecessary copying which may otherwise be implied when assigning into an array can be avoided. Furthermore, the same condition is sufficient to allow the computation to proceed in parallel on the various elements of the structure.

Thus an I-structure can be viewed as an array of slots, where slots can be filled in any order, and each slot is filled at most once. The array name can be treated as the address of the first slot (i.e. a descriptor), not unlike the name of an array in FORTRAN. The Semantics of I-structures however, permit passing the name (description) of an array to other parts of the code even before all the slots have been filled. A read request (i.e. x[i]) to a slot is processed whenever the slot is filled. If the slot is never filled, then the read request never gets an answer.

The Id compiler tries to detect if an array is being generated as an I-structure, and if it can make such a determination, the code generated avoids unnecessary copying. The compiler also uses I-structures to pass parameters to procedures and to receive results from functions.

1.1.1.2 Kinds of I-Structures

When compiling a high level language, two types of structures arise: structures whose elements are all of the same type, and structures whose elements are all of different types. If all of the elements are of the same type, it would be wasteful to store the type along with each element, but if all of the elements are of different types, type information must be associated with each element.

There are two kinds of I-structures: uniform and mixed. Associated with each I-structure is an I-structure descriptor (ISD). The ISD provides information about the length (number of elements), width (size of an element), and other characteristics (uniform or mixed, starting address, etc.) of the structure.

Each element of a uniform structure has the same type and hence length. Since the type can be easily stored in the ISD, type information does not need to be stored with each entry. Uniform I-structures are stored in a format called u-fix (untyped fixed-length); they are written into using a *store-u-fix operation and read from using a *fetch-notype-stored operation. Since no type information is stored with each entry, a type must be provided when reading a u-fix.

In a mixed I-structure, each element can be a different type and length. We have developed two formats for storing mixed I-structures. The t-fix (typed fixed-length) format allocates to each element as much storage as the largest element requires. The t-var (typed variable-length) format allocates a minimum amount of space to each element, and if a value doesn't fit, an internal pointer points to a larger slot where the value has actually been stored. This use of invisible pointers was inspired by LISP machines. The type is stored along with each element in the t-fix and t-var formats. The *store-t-fix and *store-t-var operations are used to write into t-fix and t-var formats respectively, and both formats can be read from using the *fetch-type-stored operation.

1.1.1.3 A Peek at the Implementation of I-Structures

The implementation of an I-structure Memory (ISM) is fundamentally different from that of a conventional memory in two ways. Locations are tagged as data-present or data-not-present; and if data is not present and a read occurs, the read is "deferred" until the data arrives. The information required in order to monitor the "status" of a location is stored in a hidden status field.

Deferred reads must be allowed in the Dataflow machine because the timing of read and write requests may not have any inherent order. A reader and writer are synchronized at the I-structure memory through their data dependency.

The ISMC will assist memory managers by providing additional status information and accommodating variable length data. While the presence bit and the deferred bit are the only status bits required for a minimal implementation of I-structures, additional status information is used to assure consistency against possible compiler bugs and physical hardware boundaries. The variable length data facility abates the problem of storing amorphous arrays (using the t-var format). We can think of the t-fix and t-var formats as hardware assisted constructs.

All status information other than the presence bit and the deferred bit could be migrated to runtime software support. If we wish to carry the current analysis to an extreme, presence and deferred information could be maintained through the use of semaphores, but the runtime support would start to get out of hand.

Thus a datum stored in I-structure storage can be thought of as a node in a dataflow graph with one arc leading in, and several arcs leading out. The write request is the incoming token, and the read requests combine with the written value to form the outgoing tokens.

1.1.2 INTRODUCTION THE IDL DESIGN SYSTEM

IDL, developed by Maissel and Ostapko, is the hardware design environment in which the ISMC is implemented. This section provides a high level overview of the IDL system, and a brief description of the base language.

1.1.2.1 High Level Overview of the IDL Design System

The IDL design system allows a designer to explore, design, simulate, verify, document, and modify a VLSI design. The system includes an interactive user interface, a file manager, a language (IDL, a register transfer level language), a compiler, an assembler, several simulators, minimization programs, and several other logic manipulation tools. The system outputs both a canonical form of the logic designed and a self documenting specification of the design. The IDL system is particularly well suited for designing Finite State Machines (FSMs).

The specification in IDL (the language) identifies two subsections of a design: the logic box, and the external world. The executable portion of the code describes the logic box and its effect on the external world. The non-executable portion of the code describes the structure of the external world. Executable statements indicate state transitions, as well as the register, bus, memory, black box (functional description), and other actions that should occur in a given state. These actions are emulated by the simulators in order to provide a complete simulation environment. The declarative statements indicate the sizes of registers, busses, memories, and other external objects; organize inputs, outputs, and feedbacks into more easily usable groups; indicate connections between the controller and external objects; and allow the user to specify strings for textual substitution.

The compiler makes certain syntax and consistency checks as it translates code into an intermediate form. This intermediate form, called regularized IDL, can be interpreted by the "High Level Simulator." Regularized IDL can be further assembled into two-level logic. This canonical sum of products form can be manipulated in a number of ways (including minimization a la MINI [Hong, Cain, and Ostapko 74]), and can be simulated by the "Fast Simulator."

The sum of products result comes in the form of PLA (Programmable Logic Array) personality and can be translated to other forms for conversion into random logic or for various manipulations. Synthesis of a PLA from IDL output is trivial, and paths to other embodiments of the logic can take advantage of the minimized two level form.

1.1.2.2 The IDL Language

An IDL designer specifies both the details of the logic being designed and its interaction with other objects (registers, busses, etc.).

The first part of an IDL program consists of declarations which indicate the structure of various objects. The following IDL fragment demonstrates two declarative IDL statements.

```
     DIM MAR 16/ MDR 32                  (1)
     FIX CHAIN <- CNTL[4]                (2)
```

(1) declares registers MAR and MDR to be 16 and 32 bits respectively. (2) indicates that input CHAIN is obtained from the value last latched into the fourth bit of register CNTL.

IDL GROUPs, which are similar to RECORDs in programming languages, allow the designer to refer to several objects at once. IDL STRINGs allow the designer to create mnemonics for less easily understood constructs. The mnemonic may represent a hairy condition of the inputs, a value to compare against inputs, or any other textual entity the designer wishes to abstract. STRINGs are resolved by textual substitution before the rest of the program is processed.

The main part of an IDL program is the executable code. An executable statement has the following syntax:

```
     label: IF <condition> THEN <actions> [ELSE <actions>]
```

A label can be thought of as a precondition. <condition> may be any boolean function of the inputs. <actions> involve changing state and controlling the external world, and thus include setting outputs, feedbacks, and control lines. The following fragment transfers the contents of REG1 to REG2 (sets a control line to make a transfer) if IN1 is true and if IN2=IN3.

```
     IF (IN1 AND (IN2=IN3)) THEN REG1<- REG2
```

IDL also provides a mechanism (called dot notation) for building decision graphs. A regular label is the root of a decision graph, and dotted labels are the descendants. To activate any label, the ->label construct can be used.

The novice IDL user should be careful in dealing with the left arrow or assignment symbol. In IDL, left arrow is used for many purposes:

```
Declarations:
     GROUP declarations:                    GRP foobar <- foo, bar
     external connections to the PLA:       FIX input <- one-bit-register
     string declarations:                   STRING boom <- x=1 AND y=0

Executable Code:
     register transfers:                    regl <- reg2
     setting outputs:                       control <- 1
```

IDL syntax clearly distinguishes between the above cases, but none the less the left arrow is heavily overloaded.

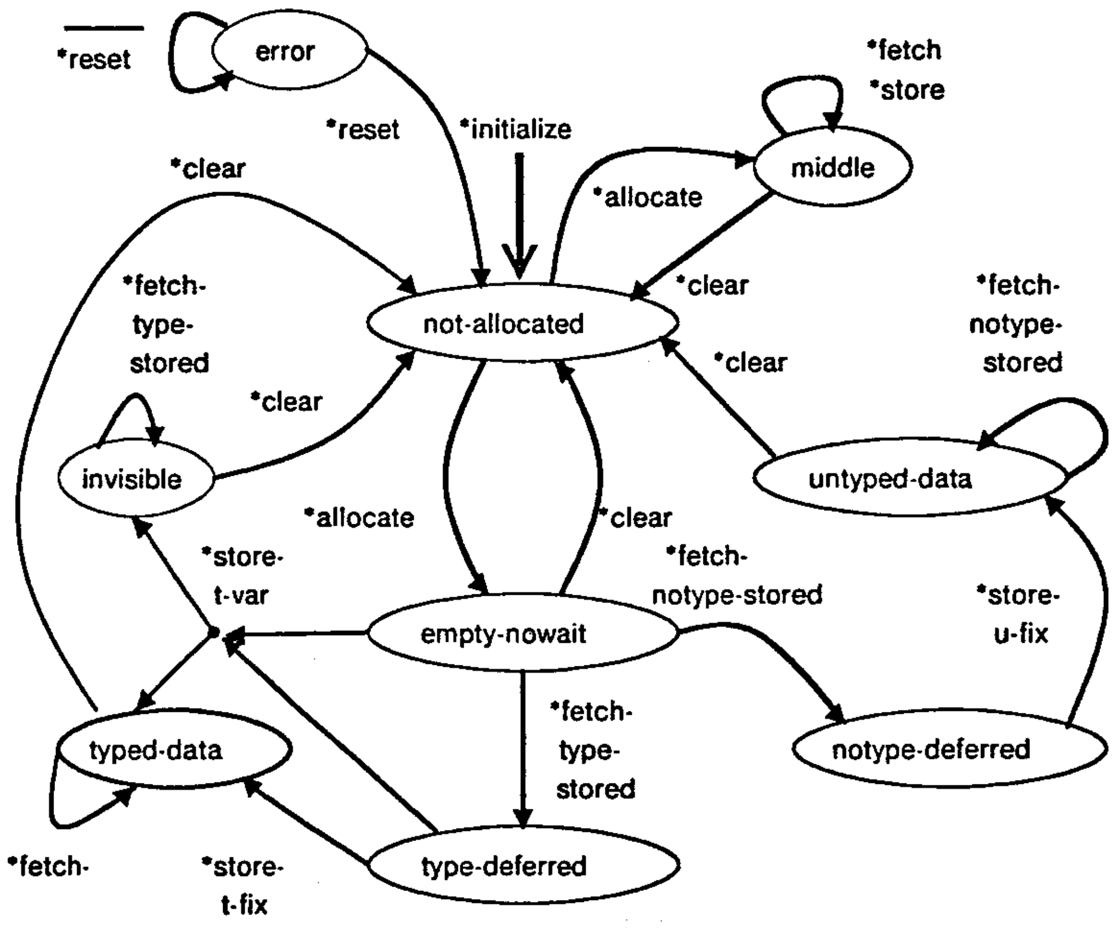
2.0 I-STRUCTURE MEMORY OPERATIONS

After I-structure memory is initialized (using the *initialize instruction), normal operation proceeds. Regions of memory are allocated using the *allocate operation, read from and written into using the *fetch and *store operations, and deallocated using the *clear operation.

As various operations touch a word, the ISM keeps track of the status of the word by setting status bits which are invisible to the user. A Status Transition Diagram can be found in Figure 1 (next page).

ISM operations fall into two categories: normal operations and service operations. This chapter describes hardware organization of the ISMC, the error-free cases of the operations, and the error recovery technique.

Status Transitions in the I-Store Section



All omitted arrows are branches to error

Figure 1

2.1 ORGANIZATION OF THE I-STRUCTURE MEMORY CONTROLLER

The ISMC is built using 288K bytes of Random Access Memory (RAM), registers, a Finite State Machine (FSM), and some random logic. This section describes the organization of these components.

2.1.1 PHYSICAL MEMORY

The ISM is divided between two physical memories: a 256K byte data memory, and a 32K byte status memory. The data memory has 16 address bits and has a 32 bit wide word. The status memory also has 16 address bits, but has a 4 bit wide word. Together they store 64K words of 32 bits each, with 4 bits of hidden status associated with each word.

The 64K words are divided into two sections: the i-store and the free list. The location at which the i-store ends and the free list begins is system programmable and can be changed whenever the system is initialized.

2.1.1.1 The I-Store Section

Each word (32 bits of data and 4 bits of status) in the i-store section is initialized with a status of not-allocated. When a block is allocated, the first word is tagged, i.e. the status field is changed to, empty-nowait, and the other words are tagged middle. When data arrives, the first word of the block is marked. If the data is stored in place, the data is tagged typed-data or untyped-data. The invisible (pointer) status is used if the data is stored indirectly on the free list. If fetches arrive before the data, the first cell in a block is tagged type-deferred or notype-deferred corresponding to the *fetch-type-stored and *fetch-notype-stored instructions.

If an error occurs at any location in the i-store, the high order bit of a status field is set to one. Information is thereby preserved about the structure of a data element that must be untangled when an error occurs. The three lower order bits of the status field are used to enumerate the values of the status field.

The details of the i-element structures can be found in the high level code declarations. A status transition diagram is given in Figure 1.

2.1.1.2 The Free-List Section

When the free list section is initialized, a linked list of double words (free-links) is created. The status field of the second word is used to indicate a cell at the end of the list, and the second half of the second word is used to point to the next free-link.

When a fetch is deferred, a free-link is taken from the free list and stores a destination as a deferred-link. The last two bytes are used to point to other destinations.

If a large datum is stored in an amorphous array (using *store-t-var), a free-link is converted into a data-cell. If the datum is large enough to require two free-cells, one is converted into a data-link, and the other is converted into a data-cell.

The details of the free list structures can be found in the high level code declarations (Appendix A).

2.2 NORMAL OPERATIONS

All of these operations require the ISMC to check the status of a word before proceeding; *stores, *fetches, *allocates, *clear, and *reset are included in this category.

2.2.1 STORE AND FETCH OPERATIONS

The *store-t-fix, *store-t-var, and *store-u-fix operations are used to store data of types t-fix, t-var, and u-fix respectively; the *fetch-type-stored operation retrieves data of type t-fix or t-var (data stored along with its type); and the *fetch-notype-stored operation retrieves data of type u-fix (data stored without its type).

A *store-t-fix operation creates the following data structure when storing a n-byte datum and its type at location i:

```
               STATUS                 VALUE
              ┌-----------------+------+------+------+------┐
i-1  ->       |                 |      |      |      |      |
              +-----------------+------+------+------+------+
i    ->       | typed-data      | type | <data begins>  ______
              +-----------------+------+------+---       /
              | middle (if k>1) |            ___________/
              +-----------------+-----     /     ------+------+
             _____________________________/     <data ends> |
              +-----------------+------+------+------+------+
i+k  ->       |                 |      |      |      |      |
              └-----------------+------+------+------+------┘
```

$$k = \left\lceil \frac{n+1}{4} \right\rceil \qquad (\lceil 3.2 \rceil = 4;\ \lceil 4.0 \rceil = 4)$$

n refers to the size of the data which follows the type field.
If n=5, 6 bytes are required to store the information.
The 0th byte is the type, and the 1st - 15th bytes are data.

When storing a n-byte datum and type at location i, the *store-t-var operation uses only one word at location i. If more space is required, a pointer is stored, and the overflow is placed in cells from the free list.

The following data structure is created if data of length 3 or less is stored (along with its type) using the *store-t-var operation.

```
               STATUS                 VALUE
           ┌-----------------+------+------+------+------┐
i-1  ->    |                 |      |      |      |      |
           +-----------------+------+------+------+------+
i    ->    | typed-data      | type |         data       |
           +-----------------+------+------+------+------+
i+1  ->    |                 |      |      |      |      |
           └-----------------+------+------+-------------┘
```

The following data structure is created if data of length greater or equal to 4 and less than or equal to 9 is stored (along with its type) using the *store-t-var operation.

```
                  STATUS              VALUE
                 ┌────────────────────┬───────┬───────┬───────┬───────┐
i-1  ->          │                    │       │       │       │       │
                 ├────────────────────┼───────┼───────┼───────┼───────┤
i    ->          │ invisible          │ type  │ data  │    pointer    │──┐
                 ├────────────────────┼───────┼───────┼───────┼───────┤  │
i+1  ->          │                    │       │       │       │       │  │
                 └────────────────────┴───────┴───────┴───────┴───────┘  │
                    ┌──────────────────────────────────────────────────────┘
                    │      STATUS       VALUE
                    │   ┌────────────┬───────┬───────┬───────┬───────┐
free cell:          └──>│  . . .     │   2nd through 5th bytes       │
                        ├────────────┼───────┼───────┼───────┼───────┤
                        │  . . .     │   6th through 9th bytes       │
                        └────────────┴───────┴───────┴───────┴───────┘

           . . . = this space not used in this operation
```

The following data structure is created if data of length greater than or equal to 10 and less than or equal to 15 is stored (along with its type) using the *store-t-var operation. The data-length part of the type field distinguishes the following data structure (with two linked cells) from the previous one (with one linked cell). A more elegant implementation would have used the status bits too.

```
               STATUS              VALUE
              ┌----------------+------+------+------+------┐
i-1 ->        |                |      |      |      |      |
              +----------------+------+------+------+------+
i   ->        | invisible      | type | data |   pointer   |--┐
              +----------------+------+------+------+------+  |
i+1 ->        |                |      |      |      |      |  |
              └----------------+------+------+------+------┘  |
                 ┌--------------------------------------------┘
                 |   STATUS     VALUE
                 |   ┌----------+------+------+------+------┐
free cell:       └-->|  . . .   |   2nd through 5th bytes   |
                     +----------+------+------+------+------+
                     |  . . .   | 6th and 7th |   pointer   |--┐
                     └----------+------+------+------+------┘  |
                 ┌---------------------------------------------┘
                 |   ┌----------+------+------+------+------┐
free cell:       └-->|  . . .   |  8th through 11th bytes   |
                     +----------+------+------+------+------+
                     |  . . .   |  12th through 15th bytes  |
                     └----------+------+------+------+------┘
```

The *store-u-fix operation operates in a similar manner as the *store-t-fix operation. When storing a n-byte datum at location i, a *store-u-fix operation creates the following data structure:

```
               STATUS              VALUE
              ┌----------------+------+------+------+------┐
i-1 ->        |                |      |      |      |      |
              +----------------+------+------+------+------+
i   ->        | untyped-data   | <data begins>             ______
              +----------------+------+------+----        /
              | middle (if k>1) |              ___________/
              +----------------+-----    /    ------+------+
          ______________________________/     <data ends> |
              +----------------+------+------+------+------+
i+k   ->      |                |      |      |      |      |
              └----------------+------+------+------+------┘

           ┌     ┐
           |  n  |
      k =  | --- |
           |  4  |
```

Before the *fetch operations can be explained, the concept of a destination must be understood. In our model, a destination (48 bits) is associated with each *fetch instruction, and indicates to which computation the fetch is to be sent. The destination is used by the token switching network to route tokens to their "destinations." Other multiprocessor systems that wish to use an I-store might simply combine a processor number with a locally unique computation identifier to form a destination. A computation identifier is needed since processors should expect to send many requests for data before the first response is recieved.

When fetching data which has already been stored, the data structure is not modified. However, when fetching data which has not yet been stored, the destination must be remembered. The destination can be

CONSed into (added to the head of) an existing destination list, or the current destination will form a new list. Each destination is stored in a cell taken from the free list.

If a *fetch-type-stored is encountered for location i, no data has been stored, and no fetches have been previously deferred at this location, a new destination list is created:

```
             STATUS               VALUE
            ┌────────────────+──────+──────+──────+──────┐
i-1 ->      │                │      │      │      │      │
            +────────────────+──────+──────+──────+──────+
i   ->      │ untyped-pointer│    . . .    │  pointer    │──┐
            +────────────────+──────+──────+──────+──────+  │
i+1 ->      │                │      │      │      │      │  │
            └────────────────+──────+──────+──────+──────┘  │
               ┌────────────────────────────────────────────┘
               │    STATUS      VALUE
               │   ┌──────────+──────+──────+──────+──────┐
free cell:     └──>│  . . .   │  <the destination goes    │
                   +──────────+             ┌──────+──────+
                   │   last   │   here>     │    . . .    │
                   └──────────+──────+──────+──────+──────┘
```

The untyped-pointer status implies deferred destinations.

If a *fetch-type-stored is encountered for location i, no data has been stored, and fetches have been previously deferred at this location, the destination is CONSed into the existing destination list:

```
             STATUS              VALUE
          ┌----------------+------+------+------+------┐
i-1 ->    |                |      |      |      |      |
          +----------------+------+------+------+------+
i   ->    | untyped-pointer |   . . .     |  pointer    |--┐
          +----------------+------+------+------+------+  |
i+1 ->    |                |      |      |      |      |  |
          └----------------+------+------+------+------┘  |
             ┌--------------------------------------------┘
             |    STATUS     VALUE
             |   ┌----------+------+------+------+------┐
free cell:   └-->|  . . .   |  <new destination goes    |
                 +----------+             ┌------+------+
                 | not-last |   here>     |  pointer    |--┐
                 └----------+------+------+------+------┘  |
             ┌---------------------------------------------┘
             |
             └--><old destination list>
```

Note that destinations are stored LIFO style,
in this operation as well as in *fetch-notype-stored.

If a *fetch-notype-stored is encountered for location i, no data has been stored, and no fetches have been previously deferred at this location, a new destination list is created (similar to the *fetch-type-stored):

```
              STATUS              VALUE
            ┌-----------------+------+------+------+------┐
i-1 ->      |                 |      |      |      |      |
            +-----------------+------+------+------+------+
i   ->      | typed-pointer   | type |      | pointer     |--┐
            +-----------------+------+------+------+------+  |
i+1 ->      |                 |      |      |      |      |  |
            └-----------------+------+------+------+------┘  |
               ┌---------------------------------------------┘
               |    STATUS     VALUE
               |   ┌----------+------+------+------+------┐
free cell:     └-->|  . . .   | <the destination goes     |
                   +----------+             ┌------+------+
                   |   last   |  here>      |   . . .     |
                   └----------+------+------+------+------┘
```

The typed-pointer status also implies deferred destinations.

If a *fetch-type-stored is encountered for location i, no data has been stored, and fetches have been previously deferred at this location, the destination is CONSed into the existing destination list:

```
              STATUS                  VALUE
            ┌─────────────────+──────+──────+──────+──────┐
i-1 ->      |                 |      |      |      |      |
            +─────────────────+──────+──────+──────+──────+
i   ->      | typed-pointer   | type |. . . |   pointer   |──┐
            +─────────────────+──────+──────+──────+──────+  |
i+1 ->      |                 |      |      |      |      |  |
            └─────────────────+──────+──────+──────+──────┘  |
               ┌─────────────────────────────────────────────┘
               |    STATUS     VALUE
               |   ┌──────────+──────+──────+──────+──────┐
free cell:     └──>|  . . .   | <new destination goes     |
                   +──────────+             ┌──────+──────+
                   | not-last |   here>     |   pointer   |──┐
                   └──────────+──────+──────+──────+──────┘  |
               ┌─────────────────────────────────────────────┘
               |
               └──><old destination list>
```

Again note that destinations are stored LIFO style.

2.2.2 ALLOCATE AND CLEAR OPERATIONS

The *allocate operation is used to prepare an array of locations for stores and fetches. The *clear operation is used to recover a sequence of locations, and return them to the unallocated state.

The *allocate operation creates the following data structure when allocating b blocks each of size w words starting with location i.

```
             STATUS              VALUE
            +----------------+------+------+------+------+
i-1  ->     |                |      |      |      |      |
            +----------------+------+------+------+------+
i    ->     | empty-nowait   | <block 1 begins>    ______
            +----------------+------+------+---   /
            | middle (if w>1) |         __________/
            +----------------+-----   /   ------+------+
           _________________________/   block 1 ends> |
            +----------------+------+------+------+------+
i+w  ->     | empty-nowait   | <block 2 begins>    ______
            +----------------+------+------+---   /
            | middle (if w>1) |         __________/
            +----------------+-----   /   ------+------+
           _________________________/   block 2 ends> |
            +----------------+------+------+------+------+
i+2w ->     | empty-nowait   | <block 3 begins>          |
            +----------------+------+------+------+------+
                             .
                             .
                             .
i+          +----------------+------+------+------+------+
(b-1)w ->   | empty-nowait   | <block b begins>    ______
            +----------------+------+------+---   /
            | middle (if w>1) |         __________/
            +----------------+-----   /   ------+------+
           _________________________/   block b ends> |
            +----------------+------+------+------+------+
i+bw ->     |                |                           |
            +----------------+------+------+------+------+
```

The *clear operation creates the following data structure when clearing b words starting with location i.

```
             STATUS              VALUE
            ┌───────────────────┬──────┬──────┬──────┬──────┐
  i-1 ->    |                   |      |      |      |      |
            +-------------------+------+------+------+------+
  i   ->    | not-allocated     |            . . .          |
            +-------------------+------+------+------+------+
            | not-allocated     |            . . .          |
            +-------------------+------+------+------+------+
                                .
                                .
                                .
            +-------------------+------+------+------+------+
  i+b-1 ->  | not-allocated     |            . . .          |
            +-------------------+------+------+------+------+
  i+b ->    |                   |      |      |      |      |
            └───────────────────+------+------+------+──────┘
```

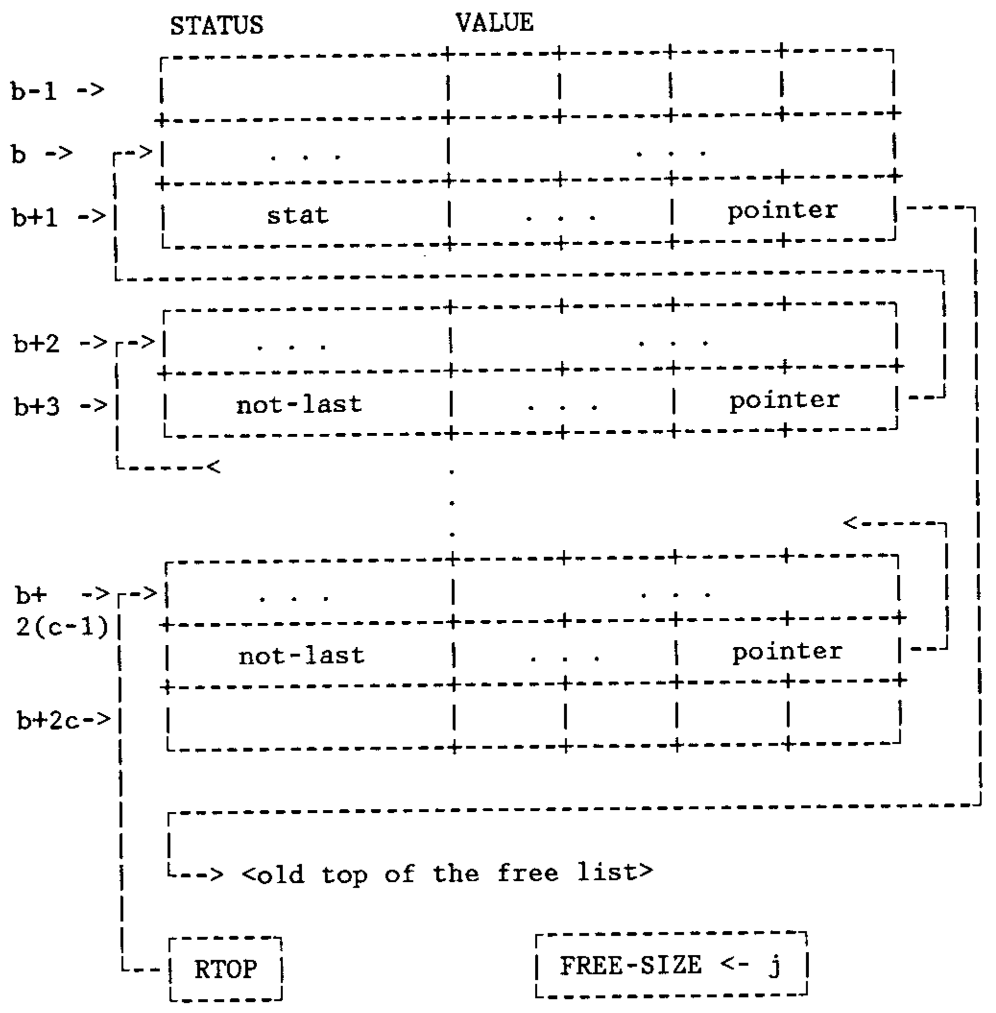
2.2.3 RESET OPERATION

The *reset operation is used to recover a word from an error state. The word in error is returned to its previous status. If no error is indicated in the status field, the error bit is set, and an error message is sent.

2.2.4 ALLOCATE-FREE-SPACE OPERATION

This operation increases the size of the free list by allocating a space in the i-store section for free cells. This space can be returned only by reinitializing the entire ISM.

The *allocate-free-space operation creates the following data structure given base b and number of cells c.



```
j = <old free-size>+c       stat = IF <list was empty>
                                   THEN last
                                   ELSE not-last
```

2.3 SERVICE OPERATIONS

With the exception of the *initialize operation, service operations are not normally used by the Data Flow machine during error free operation.

2.3.1 READ AND WRITE ABSOLUTE OPERATIONS

These operations fetch and store a word directly. Both the status and the value fields are manipulated. No type checking is performed. These absolute operations are not intended for use during normal, error free operation; they are to be used for debugging the base system and for error recovery (by managers).

2.3.2 INITIALIZE OPERATION

The *initialize operation reinitializes the ISM. Two parameters are provided: the boundary between the data section and the free list section (stored in register BOUND); and the size at which the free list should spawn a warning message. The data section is initialized as unallocated, and the free list is linked. The contents of the status and value fields of all words are ignored. The *initialize operation clears the entire memory.

The *initialize operation creates the following data structure given boundary b and warning size w.

```
              STATUS                VALUE
            ┌─────────────────+──────+──────+──────+──────┐
0 ->        | not-allocated   |           . . .           |
            +─────────────────+──────+──────+──────+──────+
1 ->        | not-allocated   |           . . .           |
            +─────────────────+──────+──────+──────+──────+
                              .
                              .
                              .
            +─────────────────+──────+──────+──────+──────+
b-1 ->      | not-allocated   |           . . .           |
            +─────────────────+──────+──────+──────+──────+
b ->     ┌->|                 |           . . .           |
         |  +─────────────────+──────+──────+──────+──────+
b+1 ->   |  |      last       |           . . .           |
         |  └─────────────────+──────+──────+──────+──────┘
         └────────────────────────────────────────────────────┐
            ┌─────────────────+──────+──────+──────+──────┐    |
b+2 -> ┌->  |                 |           . . .           |    |
       |    +─────────────────+──────+──────+──────+──────+    |
b+3 -> |    |    not-last     |    . . .     |  pointer   |───┘
       |    └─────────────────+──────+──────+──────+──────┘
       └─────<                .
                              .
                              .                      <─────┐
            ┌─────────────────+──────+──────+──────+──────┐    |
k-2 -> ┌->  |                 |           . . .           |    |
       |    +─────────────────+──────+──────+──────+──────+    |
k-1 -> |    |    not-last     |    . . .     |  pointer   |───┘
       |    └─────────────────+──────+──────+──────+──────┘
       |
       |    ┌──────┐          ┌─────────────────┐
       └─   | RTOP |          | FREE-SIZE <- j  |
            └──────┘          └─────────────────┘
```

$$k = 2^{16} \qquad j = \frac{k-b}{2} - w$$

2.3.3 LOAD ERROR DESTINATION OPERATION

A six byte register, called the error destination register contains the address of the manager to which all errors are sent. The *load-error-destination operation loads the contents of the error destination register.

2.3.4 GET FREE SIZE OPERATION

This operation sends the contents of the FREE-SIZE register to the specified destination. The true size of the free list can be determined given the value stored in FREE-SIZE and a constant (provided the last time the ISM was initialized).

2.4 ERROR HANDLING IN THE I-STRUCTURE MEMORY CONTROLLER

Since the ISMC operates at a fairly high level, it is possible to detect certain errors. For example, any attempt to write into a location that is already occupied will be caught. Several error handling mechanisms were considered for the ISMC.

In our analysis, we assumed the existence of an error manager which can be thought of as a piece of runtime software that deals with errors and inconsistencies as they occur.

When an error is detected in the ISMC, several pieces of information must be preserved. All of this information will be found in the token being processed, or in the location being addressed.

2.4.1 TYPES OF ISMC ERRORS

There are three types of errors that can occur in the ISMC: status errors, data type errors, and free-list errors.

2.4.1.1 Status Errors

A status error indicates that the operation being performed is inconsistent with the status of the addressed word. Classified within this category are BOUNDS errors, and status inconsistency errors.

A BOUNDS error indicates that the address provided is an illegal one. The ISM has 16 bit addresses, but not all addresses are legal arguments to operations. In particular, an address pointing to a location which is part of a free cell (or was part of a free cell and is now part of a deferred destination list or the tail of a long piece of data) is an illegal parameter; these locations must be accessed only through internal indirect references. Also, locations tagged as the middle of an element are illegal parameters to operations.

All other status errors are status inconsistency errors. Overwrite errors, inconsistent deferred fetch requests, and many others fall into this category. Most of the possible ISMC errors are of this type; see the status transition diagram for details (Figure 1).

2.4.1.2 Data Type Errors

A LENGTH error indicates that data is being forced into a slot which is too small to contain it. A DATA-TOO-LARGE error indicates that someone is trying to read more data from a location than it contains. Another kind of data type error is called DEFERRED-TYPE-COLLISION. When a fetch arrives with a type (*fetch-notype-stored) and discovers that there is already a deferred destination list, and the imbedded type is different from the one arriving on the fetch, there is a type inconsistency.

2.4.1.3 Free List Errors

A FREE-LIST-EMPTY error indicates that the free list is out of space. Also included in this category is the FREE-LIST-LOW warning message. While this is not actually an error condition, it indicates to a manager that a potential problem situation is arising.

The free list errors will be useful tools in the emulation of the first dataflow machine. However, it is not clear how much space should be allocated to the free list or what should be done upon reaching free list space problems. When a free list error occurs, the ISM can be left in a consistent state, and operations can continue.

2.4.2 ERROR RECOVERY

When an error is detected in the ISMC, several pieces of information are sent to a manager: all important information from the incoming token, and additional information depending on the brand of error. In addition, an error flag is set at the head of the implicated element (if there is one), except for the *allocate, *clear, and *allocate-free-space operations which set the error bit at the location where the error was actually encountered.

Three bits are reserved for status information under normal (error free) operation. An additional status bit is reserved to indicate that an error has occurred at a given location. When an error occurs, the error bit is set and the other three bits are left untouched. Hence the ISMC can detect the previous status of a word directly during error recovery.

When a free list error occurs, the partially completed operation is undone, and the active token is returned to the communication network. The process of undoing an operation may give rise to the returning of a free cell to the free list. Hence we will be sending a free-list-overflow error when the free list is not actually empty. If we don't complain to someone and delay the token a bit, a livelock situation (discussed below) may arise.

A better solution is to send all important information both about the error and the token involved. Then, a manager can take any action that it sees fit.

The following livelock should be considered. Certain errors may cause a token to be ejected (presumably untouched) back into the communication network. If the free list is empty and we need to defer a destination, for example, we might not want to mark the location being addressed. If the ejected token returns before the error is handled, (a possibility if the communication is "intelligent") another error token is generated, and the process repeats. To deal with this situation, several possibilities arise. We could mark the tokens so that the communication network sends them on a slower path. We could store tokens locally. If the free list is not full, we could put them on the free list. We could redirect the tokens to a manager. Embedding all important information in the error message being sent to a manager is the most promising compromise.

Block commands (allocate and clear) involve several groups of words in memory. If an error occurs while executing a block command, follow the scheme as outlined above, tell a manager how much of the block command has been completed, and cease processing the block command.

3.0 IMPLEMENTATION

3.1 APPROACH TO IMPLEMENTATION

In designing the ISMC, several stages of the design were developed in succession. An informal paper design describing the function and structure of the ISMC was the first step. Then a formal high level design in the form of non-executable ALGOL-like code was drawn up (Appendix A). Detailed descriptions of the algorithms were specified, and all of the data structures were then presented. The final stage of the design was the IDL code. The IDL description can be implemented in hardware, and both algorithmic and bookkeeping details are present. Since this code is executable, it can be simulated and verified.

The evolution of the ISM is a continuous process. By the time this paper had been completed, the design had become outdated. The goal of this project was to present a consistent model of the ISMC as it existed around Fall 1982.

For several reasons, the IDL code for the ISMC was not simulated. The functionality of the design was being verified by PASCAL programs derived from the High Level Code in Appendix A. Also, the ISMC design severly stressed the IDL system. Simulation using the current implmentation of IDL would have been quite difficult. Since the goal of this project was to develop an ISMC design and to study the IDL design system, and since the ISMC design had already become obsolete, IDL simulation became a less important part of the project.

3.2 RUN TIME FLEXIBILITY

Because some of the details of the design are not known at design time, the prototype may be the vehicle most appropriate for determining the values of these parameters. Thus, in order to allow some flexibility in the design of the I-structure memory controller, several design constraints were left as parameters.

3.2.1 SIZE OF THE FREE-LIST

It is unclear how memory should be divided between the i-store and free list sections. If the free list becomes empty, deferred fetches cannot be accommodated, and large elements of amorphous arrays cannot be stored. It is not even clear whether a given split will be adequate across differing applications, or across the lifetime of a single program. Hence in order to allow flexibility in the prototype engine, several techniques have been employed which allow dynamic monitoring, error checking, and recovery.

When the memory is initialized, the size of the free list section is provided as a parameter. Since a manager does not have time to wait for the adjoining section to clear in order to enlarge the free list, the boundary between the i-store and free list sections cannot be easily changed on the fly. If free space becomes scarce, the *allocate-free-space operation can convert i-store space into space for the free list. Only after a manager flushes all structures from this ISM can the space be returned by re-initialization.

The size of the free list is tracked by an up-down counter in the ISMC. The value maintained in this counter is offset from the size of the free list by an amount specified at the last initialization. This technique allows a free-list-low warning to be sent to a memory manager at a predetermined list size. In addition, the value of the counter can be monitored dynamically by any manager. The actual end of the free list is marked in the status bits of the last free-link of the list.

3.2.2 SPEED OF THE MEMORY

In an attempt to provide flexibility for the implementation and a possibility for upgrading the design by plugging in a faster FSM or RAM, the relative speeds of the FSM (presumably a PLA) and the slave RAM have been made a programmable option. By setting dip switches, the memory may operate in less then one, two, three, or four PLA cycles without impacting the design.

3.3 HANDSHAKING PROTOCOLS

Three handshaking ports have been set up to allow the ISMC to communicate with the outside world: one at the source of tokens, one at the receiver of newly formed tokens, and one for the hard interrupt line. Handshaking is accomplished through the use Set Reset (SR) latches; the sender sets the SR latch, and the receiver resets it. In order to avoid clumsiness in the IDL design, these SR latches have been modeled as simple one bit registers, each with two sources of information. By convention, the sender only sets a register when it contains a zero, and the receiver only resets a register when it contains a one.

3.4 REGISTERS

The registers can be separated into four categories: input registers, transaction registers, output registers, and system registers.

The input registers are used to receive tokens, two bytes at a time. Once a token has been accepted, it can be transferred all at once to transaction registers. A one-bit register is used to synchronize the supplier of data with the input section, and a feedback is used to synchronize the input section with the transaction section.

The transaction registers are used to interact with physical memory. This category includes registers to hold the active token, a Memory Address Register (MAR, 16 bits), and a Memory Data Register (MDR, 36 bits, 4 for status and 32 for value).

The output registers are used to output tokens one at a time. Once a token is ready to be exported, it can be transferred all at once from the transaction registers to the output registers. A feedback is used to synchronize the output section with the transaction section. I have assumed that a token buffer exists between the ISMC and the communication network.

The system registers hold the value of certain ISM variables. BOUND (16 bits), which points to the first word in the free list section, is loaded each time the ISM is initialized. RTOP (16 bits) points to the top of the free list. ERRDEST (48 bits) contains the destination to which error messages are sent. This register is loaded explicitly with a *load-error-destination operation. FREE-SIZE (16 bits) is related to the size of the free list. When the ISM is initialized, FREE-SIZE is set to the size of the free list minus a constant (a parameter of initialization). When FREE-SIZE reaches zero, a warning is sent to the error destination. The end of the free list is marked in the status field of the last free cell. FREE-LIST-STATUS (1 bit) indicates whether or not the free list is empty. PE-NUMBER (10 bits) contains the enveloping PE's number. IEXTRA (16 bits) is an extra register for pointer manipulation.

3.5 HARDWARE REQUIRED

The ISMC design required about 1500 lines of IDL code. This design stressed the IDL system to the limit, even when IDL operated in a four megabyte virtual region (under APL). We were able to compile and assemble the ISMC to PLA personality (two level logic). The following statistics were gathered:

Controller:

- Product terms: 887
- Inputs (no feedbacks): 151
- Outputs (no feedbacks): 316
- Feedbacks: 91

Registers: 718 bits

Comparators: 40 bits

3.6 SUGGESTIONS FOR FUTURE ISMCS

The ISMC presented is a consistent model. Since this project included implementing the ISMC, at some point the design had to be frozen and completed. The following considerations were not included at freezing.

3.6.1 REALIZABLE MODEL

The controller presented in this thesis, while realizable, is not a realistic design. A realistic design, for example, would have one or more busses. Without bussing, the number of control lines and data wires is too large. Some binary counters were included in the controller that (as it turns out) are better placed as external objects. Enough has been learned from this design, however, that the next generation can claim feasibility.

3.6.2 SIZE OF THE I-STRUCTURE VIRTUAL ADDRESS SPACE

At a late point in this design, we decided that each ISM could have up to 24 bits of address space. Anywhere between 16 and 24 bits could be implemented. The number of free cells, however, was not to exceed 32K so that the present data structures would not need to be modified.

The ISMC presented here assumes a 16 bit address space. Several conceptually simple modifications would accommodate this change. The only delicate matter is dealing with two different length pointers, a full length pointer (16-24 bits, depending on the implementation) for externally generated references, and a short pointer (16 bits) for internally indirected references.

3.6.3 DYNAMIC REFRESH

This ISMC does not support the refresh cycle required for a dynamic memory. With the current design, only static memories are supported. Support for dynamic memories may become important at some time in the future.

3.6.4 END-OF-BLOCK STATUS

A new status code indicating the end of an allocated block can be assigned, allowing us to avoid reading an additional word in the case of a size mismatch. Overhead that did not impact error free operation was considered secondary in this design.

3.6.5 INTERNALLY DELAYED TOKENS

An additional register could be added to the system registers that pointed to a list of internally delayed tokens. It may be desirable to temporarily swallow some tokens that are causing errors. If this is done, a more complicated structure than a list might be desirable (for example, a tree).

3.6.6 FREE-LIST-LOW WARNING

A warning is sent to a memory manager every time the free list size reaches zero (the free list may or may not be empty). If the size oscillates around zero, many warning messages might be sent; perhaps only one is needed.

3.6.7 STATUS OVERLOADING

The status bits identify the structure of the associated word. If the i-store and free list sections were physically disjoint, the address would provide additional status information implicitly. This allows us to overload the assignments of the status bits.

The two sections are not physically disjoint. Once an *allocate-free-space operation has occurred, there is no way to use address information to imply structure. The current design fully utilizes the encodings for the i-store section. We can minimize the effects of overloading by assigning all list status codes in such a way that they overlap with error codes of the i-store section.

In the next generation, all possible status possibilities should be enumerated differently, either by adding a status bit, or by modifying the error handling mechanism. In the current model, one bit indicates if an error has occurred, allowing us to preserve information about the previous structure. If we relax this model and assign a single code to "ERROR," we will have plenty of room for the other assignments.

4.0 CRITIQUE OF IDL

4.1 PROBLEMS WITH USING A HIGH LEVEL LANGUAGE

When using IDL, two types of problems related to language arose: IDL is at too high a level to notice certain details, and IDL is at too low a level to conveniently express certain constructs.

4.1.1 IDL AT TOO HIGH A LEVEL

4.1.1.1 The Semantics of IDL Assignment

The following code fragment involving two inputs and a latched feedback is translated into PLA personality:

```
Code Fragment 1               PLA 1

IF f                             ┌-<--<-┐
  THEN f <- x                 x y f     f
  ELSE f <- y                 -----------
                              0 - 1  ||  R     (P0)
                              1 - 1  ||  S     (P1)
                              - 0 0  ||  R     (P2)
                              - 1 0  ||  S     (P3)
```

And the following similar code fragment is translated into PLA personality:

```
Code Fragment 2                PLA 2

IF f AND (NOT x)                   ┌-<--<-┐
  THEN f <- 0                  x y f      f
                               ------------
IF (NOT f) AND y               0 - 1  ||  R     (P4)
  THEN f <- 1                  - 1 0  ||  S     (P5)
```

In any programming language, if f, x, and y were booleans, code fragments 1 and 2 would have identical semantics. In IDL, these fragments are only equivalent when the fedback term is latched with certain latches, ones that hold their values between cycles unless otherwise instructed. In addition, the fragments are taken out of context. If the context contains statements with overlapping conditions, they may interfere in such a way as to make the fragments distinguishable.

In code fragment 1, product terms (P1) and (P2) assert that f is set (or reset) to its current value. If f is fedback through a SET DOMINANT latch, a common and typical latch for feedbacks, and if there is no interference from other terms, as discussed below, (P1) and (P2) are not needed. When (P1) and (P2) are removed from PLA 1, PLA 2 results.

Interference caused by overlapping product terms is demonstrated below:

```
Code Fragment 3             PLA 3

IF f AND g                      ┌--<----<--┐
  THEN f <- x               x y f g h       f
                            ----------------
IF f AND h                  0 - 1 1 -   ||  R     (P6)
  THEN f <- y               1 - 1 1 -   ||  S     (P7)
                            - 0 1 - 1   ||  R     (P8)
                            - 1 1 - 1   ||  S     (P9)
```

Assume that f is fedback through a a SET DOMINANT latch. A SET DOMINANT latch holds its value unless otherwise instructed, and the SET command dominates over the RESET command. If f, g, and h are all true, there is a parallel assignment to f. This type of parallel assignment is resolved in a PLA by the "natural" pairwise ORing of x and y; f receives the value (x OR y). While product terms (P7) and (P9) both have the property that they set f to one based on the precondition that it is already set to one, we cannot eliminate (P7) and (P9). If the condition (f AND g AND h AND (x $\neq$ y)) arises and we only test terms (P6) and (P8), f will be set to zero. The terms which were previously redundant are now needed to explicitly override the RESET terms.

The confusion arises primarily because we are looking at IDL as if it were a conventional programming language such as ALGOL. In ALGOL, variables (analogous to feedbacks) maintain their value unless explicitly assigned. Hence, the ALGOL analogies of code fragments 1 and 2 are equivalent. In addition, ALGOL has no notion of parallel action. The confusion demonstrated by code fragment 3 would simply not arise in ALGOL.

IDL uses assignment to directly describe PLA personality (setting bits in the OR plane in the column corresponding to the output recieving a value) rather than the abstract notion of value transferal. While IDL assignment is frequently equivalent to value transferal, this is not always the case, as demonstrated above. The direct analogy in programming involves expressing single ideas partly in machine language. Consider the following line of PASCAL code:

```
    x := <contents of machine register number 8>
```

Although this would be a severe violation of modern programming ideology, it might not be such anathema in a hardware design language such as IDL. In fact, this mixed level notation may be necessary in order to generate an efficient implementation.

In order to avoid the confusion caused by IDL assignment, we could abandon the mixed level notation in favor of a purely abstract language. We still would be able to specify the hardware and types of latches, but the algorithms for automatically generating an efficient implementation are not obvious. It is easy to generate an implementation in hardware that uses whichever latch we desire, but cleverly using a given latch is not an easy process to automate.

As long as the IDL user realizes the difference between IDL assignment and ALGOL-like assignment, he can avoid much difficulty. Detecting possible problems due to overlapping assignments is not difficult, and the automatic reporting of such potential errors may be a useful addition to the IDL system.

4.1.1.2 Hidden Control Signals

The IDL system allows the designer to indicate register transfers in "assignment-like" statements, generates control signals for these transfers, and simulates the transfers automatically. This high level abstraction does not permit the user to talk about the control signal directly.

At a particular place in the IDL code (let's sat point x), the MDR already contained the data present in a particular register; call it R1. In other parts of the code, similar conditions caused MDR to be assigned R1. At point x, it does not matter if the control signal "MDR<-R1" rises; in fact, we would like to indicate this don't care situation in the IDL code. All we can do is indicate that the transfer occurs, or it doesn't. All system generated control signals have this weakness.

If we say "MDR<-R1", a one will be placed in the OR plane in the column corresponding to the "MAR<-R1" control signal. If we don't say "MDR<-R1", a zero is implicitly placed in the OR plane.

The problem occurs because we cannot talk about control signals directly. We would like to be able to express the idea that the control signal "R1<-R2" (transfer the contents of R2 to R1) is a valid output and can be manipulated. The following action should be expressible:

```
"R1<-R2" <- <don't care>
```

4.1.1.3 The Counting Constructs

IDL will automatically generate PLA personality to perform some arithmetic functions such as incrementing and decrementing a set of feedbacks which are collectively treated as an integer.

Part of the ISMC design includes "switch programmable" inputs which specify the relative cycle times of the PLA and the physical memory. The architect is thus allowed to specify the number of PLA cycles the controller should wait between memory accesses.

There are basically two approaches in accommodating this flexibility: load a constant initial value (such as zero) and count (up or down) until the count matches the value of a set of inputs, or load a value from inputs and count (up or down) while testing for a match with a predetermined constant value. These two approaches will be referred to as the set-and-compare paradigm and the load-and-test paradigm respectively. The following code fragments demonstrate the approaches.

Code Fragment 4

Load-and-Test Paradigm:

```
L1: IF <memory access>
       THEN count <- memtime
            -> L2

L2: IF count = 0
       THEN <continue>
       ELSE count <- count-1
            % remain in this state
```

Code Fragment 5

Set-and-Compare Paradigm:

```
L1: IF <memory access>
       THEN count <- 0
            -> L2

L2: IF count = memtime
       THEN <continue>
       ELSE count <- count+1
            % remain in this state
```

In my first design of the variable memory speed feature, I first transferred the incoming switch settings to feedbacks, and then counted down to zero (load-and-test). While the load-and-test paradigm would be the obvious choice in micro-programming (due to the jump-on-zero instruction), the set-and-compare paradigm turned out to be superior in a this design. The considerations which effect the choice of paradigms are discussed below. Unfortunately the code segments for the two prototypes are similar, and it is not at all obvious at the high level which one is desirable.

The cost of incrementing is the same as the cost of decrementing, and either one could be applied to both paradigms. Although testing for zero is unit cost, the cost for comparing two inputs for equality and the cost for transferring a value from inputs to feedbacks depend on the details of the PLA implementation.

The cost of testing two n-bit quantities for equality is different from the cost of testing for inequality. If we have a PLA with one bit decoded inputs, testing for equality takes two raised to the power n product terms (n is the number of bits in the counter), and testing for inequality takes two times n product terms. We have a PLA with two bit decoded inputs: testing for equality takes one product term, and testing for inequality takes n product terms. We may or may not be able to trade off the true and complement tests.

The cost of transferring a n-bit value from inputs to feedbacks also depends on the PLA. If we use sophisticated latches, it takes two times n product terms. (A trick allows us to load SR latches from inputs using only n+1 terms.) If we use simple polarity-hold latches (D type), the cost of the transferal drops to n product terms. Simple latches, however, make the counting process more expensive. If the condition <memory access> is complex, counting becomes even more expensive.

Since, the discussion has only addressed the microscopic picture, the most serious issue has not yet been addressed. We shall assume for the remainder of this discussion that we have two bit decoded inputs, and that the cost of a test for equality is therefore one product term. Every time we count up or down in some given state we require several lines of PLA personality (three lines for unlatched (D type) feedbacks, and two for many complex latches). If the condition associated with the state is complex, we must multiply by the number of conjuncts. We need approximately 5 product term for EACH memory access. There are more than 150 reads and writes in the ISMC code. We cannot afford to devote so many product terms to counting.

Fortunately, there is another technique similar to the set-and-compare, force-and-compare, which allows us to solve the counting problem much more efficiently. We can set up an independent process in the PLA, a counter which runs continuously, with the provision that its value may be overridden at any point. The new model involves forcing the count to 3 (binary 11) at memory access, and counting until the count matches a set of inputs. Setting the counter to all "ones" is possible because of the way a PLA operates. By performing a parallel assignment of all "ones" to the count variable, we can override any other value.

The following code fragments demonstrate this approach:

Code Fragment 6

Force-and-Compare Paradigm:

```
Universally: count <- count+1

L1: IF <memory access>
      THEN count <- '111...1'
           -> L2

L2: IF count = memtime
      THEN <continue>
      ELSE % remain in this state
```

This solution demonstrates the set-and-compare paradigm. One product term is required to force the count, and another to compare the count with inputs, giving a total of two product terms for each memory access.

We have been able to decouple the details of counting from the business of memory access. Unfortunately, we are relying on a PLA trick which allows us to override any assignment with the all ones value. Hence, a solution analogous to load-and-test is not possible.

Several interesting points stem from this analysis.

1. There is no difference between incrementing and decrementing. In fact the details of counting become fairly unimportant if we can arrange for the code to occur only once.

2. The solution chosen was not drawn from microprogramming. We have employed a technique which has no analog in programming.

3. All three paradigms are applicable in different domains, and the designer must know about the details of the implementation in order to make an intelligent choice. Unfortunately, the difference between the techniques are not obvious in the IDL code. Also, automatic translation of one paradigm to another does not appear easy. If an efficient implementation is to be obtained, it must be designed that way.

4. In our analysis we assumed that the counting was done inside the PLA. All three techniques become similar if we count and compare/test external to the PLA. This new solution is as efficient as force-and-compare, and it does not depend on two bit decoding or parallel assignment. What it does is remove from the PLA a chunk of logic (arithmetic) which we can implement efficiently externally.

4.1.2 IDL AT TOO LOW A LEVEL

4.1.2.1 Dot Notation Imitates Nested Conditionals

The IDL language has made an important leap in the direction of structured language. The best example is IDL's use of the IF-THEN-ELSE construct. Unfortunately, IF-THEN-ELSE was not carried far enough to include nested decision trees. IDL does have a dot-notation construct however, which can be used to imitate nested IF-THEN-ELSE:

Code Fragment 7

Dot-Notation

```
L1.: IF c1
       THEN -> L1.2
       ELSE -> L1.3
L1.2: IF c2
       THEN a1
       ELSE a2
L1.3: IF c3
       THEN a3
       ELSE a4
```

Code Fragment 8

Nested IF-THEN-ELSE

```
L1: IF c1
      THEN IF c2
             THEN a1
             ELSE a2
      ELSE IF c3
             THEN a3
             ELSE a4
```

The above dot notation is only slightly more clumsy than the IF-THEN-ELSE version. A more complex decision structure, however, aggravates the situation. The reader may wish to compare the High Level code presented in Appendix A with the IDL code presented in Appendix B. For example, compare the first few statements associated with the *store operations in the two versions of the code. Although the comparison is not quite fair, it is indicative of the difficulty. Dot notation has the full power of the IF-THE-ELSE construct, but it loses the advantage gained by positional association.

4.1.2.2 Dot Notation Represents Decision Graphs

Dot notation does have the interesting property that it is more general than nested IF-THEN-ELSE. While nested IF-THEN-ELSE is used to embody a decision process representable by a tree, dot notation can be used to embody any decision process representable by a Directed Acyclic Graph (DAG, the class of trees are a proper subset of the class of DAGs). Although DAGs can be "converted" into trees by replicating decision nodes, this conversion tends to clutter the code. Consider the following code fragments and corresponding graphs:

```
Code Fragment 9                   Code Fragment 10

Dot-Notation                      Nested IF-THEN-ELSE

L1.: IF c1                        L1: IF c1
       THEN -> L1.2                     THEN IF c2
       ELSE -> L1.3                            THEN S4
L1.2: IF c2                                    ELSE a1
       THEN -> L1.4                     ELSE IF c3
       ELSE a1                                 THEN S4
L1.3: IF c3                                    ELSE a2
       THEN -> L1.4
       ELSE a2
L1.4: S4
```

```
    DAG (dot notation)          TREE (nested IF-THEN-ELSE)

    c1 -> c3 -> a2              c1 -------> c3 -> a2
     V     V                     V           V
    c2 -> S4                    c2 -> S4    S4
     V                           V
    a1                          a1
```

While the nested IF-THEN-ELSE may appear more elegant, it repeats a statement. The statement itself may be a graph of decisions with many conditions and actions. Unfortunately, a linear-textual language cannot represent DAGs while taking advantage of the positional adjacency of nesting.

4.1.2.3 Extending Dot Notation

The generality of dot notation can be carried a step farther. Under the current syntax, a dotted label consists of a root name and an intermediate node name. In the above example, L1 is the root name and 2, 3, and 4 are the intermediate node names. The tie of intermediate nodes to any particular root can be lifted by naming intermediate nodes independently. The following code is a relaxed version of Code Fragment 9.

Code Fragment 11

```
L1: IF c1
       THEN -> .2
       ELSE -> .3
 .2: IF c2
       THEN -> .4
       ELSE a1
 .3: IF c3
       THEN -> .4
       ELSE a2
 .4: S4
```

While this change may not appear dramatic, one can imagine situations (frequent in the IDL code for the ISMC) in which several roots could lead to a common intermediate node. Unfortunately, the syntax duplicates the entire decision each time. This idea has no parallel in structured programming.

4.1.2.4 Cleaning up Dot Notation

There is an apparent inconsistency in IDL's view of dot notation. The research report on IDL [Maissel and Ostapko 82] describes the use of dot notation on pages 23 through 26. Code fragments 12 and 13 (below) are viewed as equivalent.

Code Fragment 12

```
L1.: IF c1
       THEN x <- 1 / ->L1.2
L1.2: IF c2
       THEN y <- 1
```

Code Fragment 13

```
L1: IF c1 / c2
      THEN x <- 1 / y <- 1
```

And code fragments 14 and 15 (below) are viewed as equivalent.

Code Fragment 14

```
L1: IF c1
       THEN x <- 1

L1: IF c1 / c2
       THEN y <- 1
```

Code Fragment 15

```
L1.:  IF c1
        THEN x <- 1

L1.:  IF c1
        THEN -> L1.2

L1.2: IF c2
        THEN y <- 1
```

But code fragments 12 and 14 are not viewed as equivalent.

In order to have some action take place at an intermediate node in a tree, the action must be isolated. Similarly, unisolated actions may or may not occur, even when the conditions up to this point in the decision structure have been met. Additional conditions may impact actions which seem to occur higher in the tree. The historical reasons for this apparent inconsistency are no longer applicable.

There is a strong analogy between dot notation and nested IF-THEN-ELSE, and GOTOs and conditionals. Since the IF-THEN-ELSE construct is not supported by the same complement of structured constructs as the programming version is, one can make a very strong case for dot notation. I am confident that a more usable system can be developed (without sacrificing power) by moving toward structure and away from dot notation.

4.2 THE ENUMERATION ASSIGNMENT PROBLEM

The problem of state assignment (discussed extensively in the literature, [Marcus 75], [Ward 68], [Miller 65], and many others) is well known and difficult.

A similar problem arose in my study of the ISMC. In the design of the i-store section, eight distinct structures arose for the data word: not-allocated, empty-nowait, typed-data, untyped-data, middle, invisible, type-deferred, and notype-deferred. The actual assignment of binary values to the various enumerations of the status bits have no intrinsic significance. The assignment will, however, have a significant impact on the logic required to operate on these bits.

This Enumeration Assignment Problem is common to the designer of digital systems. The designer must collect all decisions that will be asked of these bits and make an assignment that optimizes the decision. In practice, it is not particularly difficult to make some reasonably efficient assignment. In a design language such as IDL, however, we would like to defer the assignment as much as possible, and allow the assignment to change (automatically perhaps) as we modify the decisions made.

In the design, we wish to operate conceptually in a multiple valued logic. All decisions represent two valued predicates in the larger logic. When the time comes to implement our design in [boolean] hardware, we wish to use the list of predicates to make an assignment that yields the simplest realization of these predicates.

The problem is slightly more complex than presented so far. Various variables may be related to each other in such a way that the best enumeration assignment cannot be found independently.

4.3 THE DON'T CARE INPUT ASSIGNMENT PROBLEM

When certain combinations of inputs arise, we may not care which branch of a decision to follow. There is no way of indicating in IDL that a don't care should be associated with a decision rather than an output. Even if we express this fact using some construct, the minimization problem generated by this kind of an input don't care is different from the problem of minimizing several functions of several variables.

Consider the following Karnaugh Map, which should be considered a boolean function on two variables, and the code fragment that uses it.

```
Karnaugh Map 1            Code Fragment 16
  a
b   | 0 | 1 |             IF (Map 1) (a b)
  --|---|---|               THEN x <- 1 / y <- 0
  0 | 0 | ? |               ELSE x <- 0 / y <- 1
  --|---|---|
  1 | 1 | 1 |
  --|---|---|
```

The don't care at the ab=10 position means that both the THEN and the ELSE branch provide an acceptable assignment of values to x and y. If we expand the statement into maps for x and y, propagating the don't care, the following Karnaugh maps result:

```
     Karnaugh Map 2           Karnaugh Map 3
           x                         y
       a                         a
     b    | 0 | 1 |            b    | 0 | 1 |
        --|---|---|               --|---|---|
        0 | 0 | ? |               0 | 1 | ? |
        --|---|---|               --|---|---|
        1 | 1 | 1 |               1 | 0 | 0 |
        --|---|---|               --|---|---|
```

The don't cares at the ab=10 positions have been decoupled. There are four ways of assigning the two don't cares, but only two of those assignments preserve the semantics of the decision structure. If the don't cares at ab=10 on Map 1 and Map 2 are both assigned the same value (either 1 or 0), we encounter a peculiar situation. When ab=10, the values of outputs x and y will be equal to each other. These values are inconsistent with both the THEN and ELSE branches of code fragment 16.

If we want to move the input don't cares to the output side, we must also carry along some constraints on their assignments. In the example above we must assert that the assignments of the don't cares for x and y must be opposite values.

This problem would never have arisen if we could not express don't cares of the input variety. In fact, code fragment 16 cannot be expressed using only output don't cares. Another syntax for expressing input don't cares is the IF-EITHER-OR statement:

```
IF <condition>
  EITHER <actions>
  OR     <actions>
```

We can compress logic to two levels and preserve the semantics of our decision structures as follows. Uniquely label all input don't cares.

When we multiply out the multiple level logic, the don't cares are linked by their labels (some will be in inverted form). Whenever we assign a don't care, all instances of the don't care must be assigned the same value. This is by no means a minimization algorithm. This problem is clearly a superset of "classical minimization."

Even though we don't know how to solve the input don't care assignment problem precisely, we can try applying a few heuristics. Boolean maps can be minimized locally. We might also consider the sizes of the THEN and ELSE branches if we are working on a decision tree. Consider the following Karnaugh maps.

Karnaugh Map 4

C \ AB	00	01	11	10
0	0	0	?	1
1	?	0	1	1

we assign so that:
ON SET = A
OFF SET = not A

Karnaugh Map 5

FG \ DE	00	01	11	10
00	0	0	?	?
01	0	?	1	?
11	?	1	1	?
10	?	?	?	?

In the case of Karnaugh map 4, the best assignment seems to be assigning the don't cares in such a way that the ON SET = A, and the OFF SET = not A.

The most promising assignment for Karnaugh Map 2 is assigning the 0101 spot to one boolean value, and assigning the rest of the spots to the other value. If 0101 is assigned 0, then the OFF SET requires the single term (NOT F)(NOT D) and the ON SET requires the terms F and D. If 0101 is assigned 1, then the ON SET requires the single term EG and the

ON SET Requires the terms (NOT E) and (NOT G). Depending on the complexity of the THEN and ELSE branches, one of the two choices can be made. If the THEN branch is "heavier," we assign the spot to 1. If the ELSE branch is "heavier," we assign the spot to 0. In any case, assigning all zeros or all ones is probably non-optimal.

4.4 A PASCAL-LIKE LANGUAGE MODELS A HARDWARE DESIGN

During the course of this project, I wrote a version of the ISMC in PASCAL-like pseudo code, and subsequently in IDL. Also, our group undertook an effort to translate my pseudo code into PASCAL for use in a simulator for our dataflow machine. In this section, I will discuss several points where the designs were dissimilar.

The IDL code tended to be non-reentrant. When copying code to a similar section of the design, extreme caution had to be used in changing all references to state (labels). The PASCAL did not suffer from this problem, and it is not inherent in the hardware.

PASCAL data structures could not be used to directly model the hardware, that is, without loosing the advantages of having the data structure in the first place. When a VARIANT-RECORD is used in PASCAL, there is implied storage (to discriminate the cases). The very bits which are beyond our control are the most critical ones. Those are the bits by which decisions are to be made. We must have close control over them or know that the compiler will do a particularly good job in dealing with them.

The PASCAL code does not have any sense of time (other than sequences). This issue is of particular concern in IDL as it must be in any hardware design system. As a direct result of this problem there was no way to express the idea of a hard reset in PASCAL.

Many important details (from the hardware point of view) were glossed over in PASCAL. One of these issues involve passing tokens to the system, and transferring values within it. When we say x:=y in software, no one asks how the value travels. The details of value transferal cannot be ignored in hardware.

References

[Ackerman 77]

Ackerman, W. B. A Structure Memory for Data Flow Computers. Technical Report TR-186, Laboratory For Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, August 1977.

[Arvind 82]

Arvind and Members of the Functional Languages and Architecture Group. The Tagged Token Dataflow Architecture. Technical Report (unpublished), Massachusetts Institute of Technology, Cambridge, Massachusetts, August 1982.

[Arvind and Iannucci 81]

Arvind and Iannucci R. A. Instruction Set Definition for a Tagged Token Data Flow Machine. Memo 212, Computation Structures Group, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, December 1981 (updated May 1982.)

[Arvind and Thomas 81]

Arvind and Thomas, R. E. I-Structures: An Efficient Data Structure for Functional Languages. Memo 178, Computation Structures Group, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, October 1981.

[Dennis 73]

Dennis, J. B. First Version of a Data Flow Procedure Language. Computation Structures Group Memo 93, Laboratory For Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, November 1973. (Revised as MAC Technical Memorandum 61, May 1975.)

[Hong, Cain, and Ostapko 74]

Hong, S. J., Cain R. G., and Ostapko D. L. MINI: A Heuristic Approach for Logic Minimization, IBM Journal of Research and Development, Volume 18, pp 443-458, 1974.

[Maissel and Ostapko 82]

Maissel, L. I. and Ostapko, D. L. Interactive Design Language: A Unified Approach to Hardware Simulation, Synthesis and Documentation. ACM IEEE 19th Design Automation Conference Proceedings, (June 1982), 193-201.

[Marcus 75]

Marcus, Mitchell P. Switching Circuits for Engineers, Third Edition, Prentice-Hall Electrical Engineering Series, 1975, pp 162-184.

[Miller 65]

Miller, Raymond E. Switching Theory, Volume II: Sequential Circuits and Machines, John Wiley & Sons, Inc., 1965, pp 99-140.

[Williams 81]

Williams, B. C. Implementation of an I-Structure Memory Controller. 6.823 Paper (unpublished), Massachusetts Institute of Technology, Cambridge, Massachusetts, December 1981.

[Wood 68]

Wood, Jr., Paul E. Switching Theory Lincoln Laboratory Publications, 1968, pp 150-175.

A.0 HIGH LEVEL CODE

```
% Memory Operations
%
%   Normal Operations:
%
%      *store-t-fix (i: address, t: data-type, data: <0..119>)
%           store one, two, three, or four words
%           depending on t
%           store type information in first word
%
%      *store-t-var (i: address, t: data-type, data: <0..119>)
%           store one word
%           use invisible pointer if necessary
%           store type information in first word
%
%      *store-u-fix (i: address, data: <0..119>)
%           store one, two, three, or four words
%           depending on type
%           don't store type information
%
%      *fetch-type-stored (i: address, dest: destination)
%           follows *store-t-var
%                or *store-t-fix
%           fetch as indicated by stored type
%
%      *fetch-notype-stored (i: address, t: data-type,
%                            dest: destination)
%           follows *store-u-fix
%           fetch as indicated by t
%
%      *allocate (base: address, blocks <0..7>, element-size: <0..3>)
%           allocates blocks element-sized words
%           starting at address base
%
%      *clear (base: address, block-size <0..11>)
%           clears blocksize words
%           starting at address base
%
%      *reset (i: address)
%           used to recover from the error state.
%           if the ith word has an error status, the status
%           is changed to it's previous value.
%           if the ith word doesn't has an error status, the status
%           is changed to error.
%
%      *allocate-free-space (base: address, cells: <0..15>)
%           adds "cells" cells to the free list from the
%           contiguous unallocated block beginning at base
%   Service Operations:
%
%      *read-absolute (i: address, dest: destination)
%           read the status and data memories directly
%           perform no type checking
%
%      *write-absolute (i: address, data: <0..35>)
%           write the status and data memories directly
%           perform no type checking
%
%      *initialize (boundary: address, warning: <0..15>)
%           set the boundary between the data section and
%           the free list section to boundary.  initialize
%           the data section as unallocated, and set up a
%           free list in the free list section.  ignore the
%           status or value of any word.  this operation
%           clears the entire memory.
%
%      *load-error-destination (dest: destination)
%           sets the error destination to dest
%
%      *get-free-size (dest: destination)
%           sends the size of the free list to dest
%           (actually, the quantity sent is offset from
%           the true size by the amount specified when
%           the memory was last initialized)
% Internal I-Structure Controller side-effects (operations):
%
%    read (i: address)
%         returns (word)
%
%         reads the internal status and data memories
%         and returns the contents at address i.
%
%
%    write (i: address, status: <0..3>, data: <0..31> )
%
%         stores status in the status memory at location i.
%         stores data in the data memory at location i.
%
%
%    output-token (dest: destination, data: <0..??> )
%
%         forms a token consisting of a destination and typed data
%         the data may be of length 0 to 16 bytes (including type).
%
%
%    resubmit-token ()
%
%         forms a token consisting of the one currently being operated
%         on and dumps it back into the communication network.
%
%
%    signal (error:     error,
%            address:   address,
%            status:    status)
%
%         indicates that an exceptional condition has
%         occurred. each operation signals a subset of
%         the possible errors. see the source code
%         for details concerning a particular operation.
%         error-destination and pe-number are not arguments
%         since they are always found in the same place.
% Organization of Memory

i-structure-memory = RECORD
  [i-store:   i-store,
   free-list: free-cells]


   i-store = array [i-element]

   free-cells = array [free-cell]
```

```
i-element = MULTITYPE TAGGED ON status.type
 [not-allocated:   <0..31>,           % not allocated
  empty-nowait:    <0..31>,           % empty, no defers
  typed-data:      typed-value,       % typed data present
  untyped-data:    untyped-value,     % untyped data present
  middle:          untyped-value,     % middle of an element
  invisible:       indirect-pointer,  % invisible pointer
  type-deferred:   untyped-defer,     % *fetch-type-stored deferred
  notype-deferred: typed-defer)]      % *fetch-notype-stored deferred

  status = RECORD
   [error: (occurred => 1, none => 0),
    type:
     (not-allocated => 000,   % not allocated
      empty-nowait  => 001,   % empty, no defers
      typed-data    => 010,   % typed data present
      untyped-data  => 011,   % untyped data present
      middle        => 100,   % middle of an element
      invisible     => 101,   % invisible pointer
      untyped-defer => 110,   % *fetch-type-stored deferred
      typed-defer   => 111)]  % *fetch-notype-stored deferred

     % If the following enumerations of status do not
     % differ in more than one bit, a more minimal
     % controller is likely to emerge.  Other such pairs
     % probably exist, but they are not obvious at this point.
          % typed-data and untyped-data
          % type-deferred and notype-deferred

  typed-value = RECORD
    [type:  data-type,  % data type
     value: <8..31>]    % data value

  untyped-value = <0..31>  % data value

  indirect-pointer = RECORD
    [type:    data-type,  % type associated with object pointed to
     value:   <0..7>,     % first byte of data
     pointer:  address]   % pointer to a data-cell or data-link

  untyped-defer = RECORD
     [unused:  <0..15>,
      pointer:  address]   % pointer to a data-cell

  typed-defer = RECORD
    [type:    data-type,  % type associated with object pointed to
     unused:  <0..7>,
     pointer:  address]   % pointer to a data-cell
```

```
free-cell = MULTITYPE [free-link,       % link of the free list
                       deferred-link,   % link in a deferred read list
                       data-cell,       % data
                       data-link]       % data and link to more data

  free-link = RECORD  % link of a free list
    [status0:  <0..3>,       % not used
     status1:  list-status,  % indicates list position
     not-used: <0..47>,      % not used
     pointer:  address]      % pointer to next cell

    list-status =
       (not-last => ---0,  % not the last free-cell in a list
        last     => ---1)  % last free-cell in a list
             % not-last and last are defined loosely so
             % that a coder can play all sorts of tricks
             % if he already knows that the status field
             % satisfies the given constraints

  deferred-link = RECORD  % link in a deferred read list
    [status0:     <0..3>,       % not-used
     status1:     list-status,  % indicates list position
     destination: destination,  % destination of deferred read
     pointer:     address]      % pointer to next link
     % list position is indicated in the same word as pointer
     % information as we need both to cdr down the list

  data-cell = RECORD   % 8 bytes of data
    [status: <0..7>,   % status field is not used
     value:  <0..63>]  % data value

  data-link = RECORD  % 6 bytes of data and a link to more data
    [status: <0..7>,    % status field is not used
     value:   <0..47>,  % data value
     pointer: address]  % pointer to more data
  address = <0..15>   % address pointer

  destination = <0..47>   % destination

  data-type = RECORD [length: <0..3>, class: <4..7>]

  blank = (unused-space => 33 HEX)  % distinguished pattern 00110011

  mini-blank = (unused-space => 3 HEX)  % distinguished pattern 0011

  blank-pointer = (unused-space => 3333 HEX)
                 % distinguished pattern 0011001100110011

  data-element = <0..31>

  word = RECORD [status: status,
                 value:  data-element]

error =
  (store-t-fix-error         => 0 HEX,
   store-t-var-error         => 1 HEX,
   store-u-fix-error         => 2 HEX,
   fetch-type-stored-error   => 3 HEX,
   fetch-notype-stored-error => 4 HEX,
   allocate-error            => 5 HEX,
   clear-error               => 6 HEX,
   reset-error               => 7 HEX,
   allocate-free-space-error => 8 HEX,
   data-too-large            => 9 HEX,
   free-list-overflow        => A HEX,
   free-list-low             => B HEX,
   length-error              => C HEX,
   deferred-type-collision   => D HEX)

   % errors 0-8 should be enumerated in the same way as opcodes
   % are assigned to the corresponding operations.
   % the savings does not show up in this document;
   % it appears at a lower level.
```

System Variables

```
rtop : address  % points to the top of the free list

bound : address  % points to first word in the free-list section
                 % except those sections of the free list which
                 % came about from *allocate-free-space.

error-destination : destination  % destination to send all errors

free-size : <0..15>  % (size of the free list) - (warning size)
                     % when free-size reaches zero, a warning
                     % is sent to the error destination.
                     % if the free list is actually empty when
                     % free size hits zero, an error is sent.
                     % a warning may or may not be sent.

free-list-status: (cells-remain => 0,
                   list-empty   => 1)
% it may be most efficient to assure that free-list-status
% agrees with the last bit of list-status

pe-number : <0..9>  % this PE's number (hard-wired somewhere)

rtemp0 : address  % temporary register for address manipulation

rtemp1 : address  % temporary register for address manipulation
```

A.1 *STORE-T-FIX OPERATION

```
*store-t-fix (i: address, t: data-type, data: <0..119>)
    % data-too-large
    % store-t-fix-error

  % store one, two, three, or four words
  % depending on t
  % store type information

  r0: i-element <- read (i)

  IF r0.status.error = none
    THEN TAGCASE ON r0.status.type

      = empty-nowait   % not present, no reads deferred
        % store the data
        write (i,              % write the first word
               present,        % status
               t,              % save type information
               data[0..23] )   % first 3 bytes of data
        IF t.length > 3
          THEN % check if the next word was properly allocated
            stat: status <- (read (i+1))[0..3]
            IF stat ≠ middle
              THEN signal (data-too-large  % type of error
                           i,              % address
                           empty-nowait)   % status at this word
                   write (i, [['1',empty-nowait] % new error
                              r0.value])      % status, retain value
            ELSE
              write (i+1,
                     middle,         % status
                     data[24..55] )  % 4th - 7th bytes of data
              IF t.length > 7
                THEN % check if the next word was properly allocated
                  stat: status <- (read (i+2))[0..3]
                  IF stat ≠ middle
                    THEN signal (data-too-large  % type of error
                                 i,              % address
                                 empty-nowait)   % status at this word
                         write (i, [['1',empty-nowait] % error status
                                    r0.value])         % retain value
                  ELSE
                    write (i+2,
                           middle,         % status
                           data[56..87] )  % 8th - 11th bytes
                  IF t.length > 11
                    THEN % check if next word properly allocated
                      stat: status <- (read (i+3))[0..3]
                      IF stat ≠ middle
                        THEN
                          signal (data-too-large  % err type
                                  i,              % address
                                  empty-nowait)   % status here
                          write (i, [['1',empty-nowait] % err stat
                                     r0.value])  % retain value
                        ELSE
                          write (i+3,
                                 middle,          % status
                                 data[88..119] )  % 12th - 15th
                      END-IF
                  END-IF
              END-IF
          END-IF
      END-IF
  END-IF

= type-deferred % not present, *fetch-type-stored deferred
  % send a token to each deferred destination
  % return links to the free list
  % store the data starting at location i
  rtemp0 <- r0.pointer

  UNTIL stat: list-status = last
    % fetch the next destination
    r1: data-element          <- (read (rtemp0))[4..35]
    (stat, r2: data-element) <-  read (1+rtemp0)
    % send the data to it's destination
    output-token (r1,          % 4 bytes of destination
                  r2[0..15],   % 5th & 6th bytes of destination
                  t,           % type carried on value
                  data)

    % return this link to the free list
    % the following test only needs to be made
    % on the first time through the UNTIL
    IF free-list-status = cells-remain
      THEN write (1+rtemp0,
                  not-last,        % status
                  [blank, blank,   % this field left blank
                   rtop] )         % pointer to old top
      ELSE write (1+rtemp0,
                  last,               % status
                  [blank, blank,      % this field left blank
                   blank-pointer] )   % no pointer needed
           free-list-status <- cells-remain
      ENDIF
    rtop <- rtemp0
    free-size <- free-size + 1
    rtemp0 <- r2 [16..31]
    END-UNTIL
```

```
% now store the data
write (i,              % write the first word
       present,        % status
       t,              % save type information
       data[0..23] )   % first 3 bytes of data
IF t.length > 3
  THEN % check if the next word was properly allocated
       stat: status <- (read (i+1))[0..3]
       IF stat ≠ middle
         THEN signal (data-too-large  % type of error
                      i,              % address
                      type-deferred)  % status at this word
              write (i, [['1',type-deferred] % new error
                         r0.value])      % status, retain value
         ELSE
           write (i+1,
                  middle,         % status
                  data[24..55] )  % 4th - 7th bytes of data
         END-IF
  END-IF
IF t.length > 7
  THEN % check if the next word was properly allocated
       stat: status <- (read (i+2))[0..3]
       IF stat ≠ middle
         THEN signal (data-too-large  % type of error
                      i,              % address
                      type-deferred)  % status at this word
              write (i, [['1',type-deferred] % new error
                         r0.value])      % status, retain value
         ELSE
           write (i+2,
                  middle,         % status
                  data[56..87] )  % 8th - 11th bytes of data
         END-IF
  END-IF
IF t.length > 11
  THEN % check if the next word was properly allocated
       stat: status <- (read (i+3))[0..3]
       IF stat ≠ middle
         THEN signal (data-too-large  % type of error
                      i,              % address
                      type-deferred)  % status at this word
              write (i, [['1',type-deferred] % new error
                         r0.value])      % status, retain value
         ELSE
           write (i+3,
                  middle,          % status
                  data[88..119] )  % 12th - 15th bytes of data
         END-IF
  END-IF
```

```
    OTHERWISE  % signal an error
      signal (store-t-fix-error,  % type of error
              i,                  % address
              r0.status)          % status at this word
      write (i, [['1',r0.status.type]  % new error status
                 r0.value])            % retain value
    END-TAGCASE

  ELSE  % an error has already occurred at this location
    signal (store-t-fix-error,  % type of error
            i,                  % address
            r0.status)          % status at this word

  END-IF
END *store-t-fix
```

A.2 *STORE-T-VAR OPERATION

```
*store-t-var (i: address, t: data-type, data: <0..119>)
  % error signals:
    % free-list-overflow
    % free-list-low
    % store-t-var-error

  % store one word
  % use invisible pointer if necessary
  % store type information in word

  r0: i-element <- read (i)

  IF r0.status.error = none
    THEN TAGCASE ON r0.status.type
```

```
= empty-nowait   % not present, no reads deferred
  % store the data
  IF t.length ≤ 3
    THEN
      write (i,   % store the data in place
             present,       % status
             t,             % type
             data[0..23] )  % data
    ELSE
      % put the data in the free list
      IF free-list-status = list-empty
        THEN
          signal (free-list-overflow,  % type of error
                  blank-pointer,       % this information
                  mini-blank)          % is not necessary
          resubmit-token ()
        ELSE
          write (i,                   % point to data-cell
                 invisible-pointer  % status
                 [t,            % save type information
                  data[0..7],  % first byte of data
                  rtop] )      % invisible pointer
          % store next 4 bytes of data in free list
          write ( rtop,             % store data in free list
                  mini-blank,     % status not used
                  data[8..39] )  % 4 bytes of data
          % save a pointer to the rest of the list
          (stat: list-status, rtemp0)
            <- (read (1+rtop)) [0..3,20..35]
          IF stat0 = last
            THEN free-list-status <- list-empty
            ENDIF
          free-size <- free-size - 1
          IF free-size = 0
            THEN signal (free-list-low,  % type of error
                         blank-pointer,  % these fields
                         mini-blank)     % not used
            END-IF
          IF t.length ≤ 9
            THEN  % the data will fit with only one free cell
              write (1+rtop,          % store data in second half
                     mini-blank,      % status not used
                     data[40..71] )  % 6th - 9th bytes
              rtop <- rtemp0  % cdr down the list
          % IF t.length ≤ 9 continued
          ELSE  % another free cell is required
            IF stat = last  % the free list is empty
              THEN  % the list hasn't been cdr'd so it doesn't
                    % need restoration; simply erase all trace
                    % of the token at location i
                write (i,                  % restore location i
                       empty-nowait,       % status
                       [blank, blank,      % data field
                        blank, blank] )    % blanked out
                free-size <- free-size + 1
                signal (free-list-overflow,  % type of error
                        blank-pointer,       % this information
                        mini-blank)          % is not necessary
                resubmit-token ()
              ELSE  % store the rest of the data
                % store 2 more bytes and a pointer
                write (1+rtop,          % store the data
                       mini-blank,      % status not used
                       data[40..55],    % 6th - 7th bytes
                       rtemp0 )         % 6th - 7th bytes
                % store the next four bytes in second free cell
                write (rtemp0,           % store the data
                       mini-blank,       % status not used
                       data[56..87] )    % 8th - 11th bytes
                % get the new top of the free list
                (stat0: list-status, rtop)
                  <- (read (1+rtemp0))[0..3,20..35]
                IF stat0 = last
                  THEN free-list-status <- list-empty
                  ENDIF
                free-size <- free-size - 1
                IF free-size = 0
                  THEN signal (free-list-low, % error type
                               blank-pointer, % these fields
                               mini-blank)    % not used
                  END-IF
                IF t.length ≤ 11
                  THEN  % the data is already written
                  ELSE  % we must write the last word
                    write (1+rtemp0,          % store the data
                           mini-blank,        % status not used
                           data[88..119] )    % 12th - 15th bytes
                  ENDIF
              ENDIF
          ENDIF
      END-IF
  END-IF
```

```
= type-deferred % not present, *fetch-type-stored deferred
  % send a token to each deferred destination
  % return links to the free list
  % store the data using a free cell if necessary
  rtemp0 <- r0.pointer

  UNTIL stat: list-status = last
    % fetch the next destination
    r1: data-element        <- (read (rtemp0)) [4..35]
    (stat, r2: data-element) <-  read (1+rtemp0)
    % send the data to it's destination
    output-token (r1,          % 4 bytes of destination
                  r2[0..15],   % 5th & 6th bytes of destination
                  t,           % type carried on value
                  data)

    % return this link to the free list
    % the following test only needs to be made
    % on the first time through the UNTIL
    IF free-list-status = cells-remain
      THEN write (1+rtemp0,
                  not-last,        % status
                  [blank, blank,   % this field left blank
                   rtop] )         % pointer to old top
      ELSE write (1+rtemp0,
                  last,               % status
                  [blank, blank,      % this field left blank
                   blank-pointer] )   % no pointer needed
           free-list-status <- cells-remain
      ENDIF
    rtop <- rtemp0
    free-size <- free-size + 1
    rtemp0 <- r2 [16..31]
    END-UNTIL
```

```
% now store the data
% if the data is ≤ 3 bytes, store directly
IF t.length ≤ 3
  THEN write (i,   % store the data in place
              present,      % status
              t,            % type
              data[0..23] ) % data
  ELSE  % we need at least 1 free-cell (must exist)
    % put a byte in place and point
    % to the rest of the data
    write (i,                   % point to the top free-cell
           invisible-pointer  % status
           [t,                  % save type information
            data[0..7],         % first byte of data
            rtop] )             % invisible pointer
    % now store 4 bytes in free cell
    write (rtop,             % store in the free-cell
           mini-blank,       % status
           data [8..39] )    % 2nd - 5th bytes
    % obtain cdr information
    (stat: list-status, rtemp0)
      <- read (1+rtop) [0..3,20..35]
    IF stat = last
      THEN free-list-status <- list-empty
      ENDIF
    IF t.length ≤ 5
      THEN  % the data is written, pop the free list
        rtop <- rtemp0
      ELSEIF t.length ≤ 9
        THEN  % the data fits in the second word
          write ( 1+rtop,         % store the 1st word
                  mini-blank,     % status not used
                  data[40..71] ) % 6th - 9th bytes of data
          % pop the free list
          rtop <- rtemp0
          free-list-size <- free-list-size - 1
          IF free-size = 0
            THEN signal (free-list-low,  % type of error
                         blank-pointer,  % these fields
                         mini-blank)     % not used
            END-IF
        END-ELSE-IF
```

```
ELSE  % we need another free cell
  IF free-list-status = cells-remain
    THEN  % store a couple of bytes and a pointer
      write (1+rtop,           % store in the free-cell
             mini-blank,       % status
             data [40..55],    % 6th - 7th bytes
             rtemp0 )          % pointer to rest of data
      % store 4 more bytes in next free cell
      write (rtemp0,           % store in the free-cell
             mini-blank,       % status
             data [56..87] )   % 8th - 11th bytes
      % obtain cdr information
      (stat: list-status, rtemp0)
        <- read (1+rtop) [0..3,20..35]
      IF stat = last
        THEN free-list-status <- list-empty
        ENDIF
      IF t.length ≤ 11
        THEN  % the data is written, pop the free list
          rtop <- rtemp0
        ELSE  % write the final word
          write (1+rtemp0,          % store in free-cell
                 mini-blank,        % status
                 data [88..119] )   % 12th - 15th bytes
          % pop the free list
          rtop <- rtemp0
        END-IF
      free-size <- free-size - 1
      IF free-size = 0
        THEN signal (free-list-low,  % type of error
                     blank-pointer,  % these fields
                     mini-blank)     % not used
        END-IF
```

```
            ELSE  % we must undo the operation
              % get the pointer to the free cell we must replace
              rtemp0 <- read (i) [20..35]
              % restore location i
              write (i,                   % restore location i
                     empty-waiting,       % status
                     [blank, blank,       % data field
                      blank, blank] )     % left blank
              % cons the free cell back into the free list
              write (1+rtemp0,          % top cell of free list
                     last,              % last cell in list
                     [blank, blank,     % this slot left blank
                      rtop] )           % pointer to old top
              rtop <- rtemp0
              free-list-status <- cells-remain
              signal (free-list-overflow,  % type of error
                      blank-pointer,       % this information
                      mini-blank)          % is not necessary
              resubmit-token ()
            END-IF
        END-IF
    END-IF
    OTHERWISE  % signal an error
      signal (store-t-var-error,  % type of error
              i,                  % address
              r0.status)          % status at this word
      write (i, [['1',r0.status.type]  % new error status
                 r0.value])            % retain value
    END-TAGCASE

  ELSE  % an error has already occurred at this location
    signal (store-t-var-error,  % type of error
            i,                  % address
            r0.status)          % status at this word
  END-IF

END *store-t-var
```

A.3 *STORE-U-FIX OPERATION

```
*store-u-fix (i: address, t: data-type, data: <0..119>)
  % error signals:
    % data-too-large
    % store-u-fix-error

  % store one, two, three, or four words
  % depending on type
  % don't store type information

  r0: i-element <- read (i)

  IF r0.status.error = none
    THEN TAGCASE ON r0.status.type

      = empty-nowait   % not present, no reads deferred
        % store the data
        write (i,            % write the first word
               present,      % status
               data[0..31] ) % first 4 bytes of data
        IF t.length > 4
          THEN % check if the next word was properly allocated
            stat: status <- (read (i+1))[0..3]
            IF stat ≠ middle
              THEN signal (data-too-large  % type of error
                           i,              % address
                           empty-nowait)   % status at this word
                   write (i, [['1',empty-nowait] % new error status
                                r0.value])        % retain value
              ELSE
                write (i+1,
                       middle,          % status
                       data[32..63] )   % 5th - 8th bytes of data
                IF t.length > 8
                  THEN % check if the next word was properly allocated
                    stat: status <- (read (i+2))[0..3]
                    IF stat ≠ middle
                      THEN
                        signal (data-too-large  % type of error
                                i,              % address
                                empty-nowait)   % status at this word
                        write (i, [['1',empty-nowiat] % new error status
                                     r0.value])        % retain value
                      ELSE
                        write (i+2,
                               middle,         % status
                               data[64..95] )  % 9th - 12th bytes
                    IF t.length > 12
                      THEN % check if next word properly allocated
                        stat: status <- (read (i+3))[0..3]
                        IF stat ≠ middle
                          THEN
                            signal (data-too-large  % err type
                                    i,              % address
                                    empty-nowait)   % status here
                            write (i, [['1',empty-nowiat] % err stat
                                       r0.value])     % retain value
                          ELSE
                            write (i+3,
                            middle,           % status
                            [data[96..119],   % 13th - 15th bytes
                             blank] )
                          END-IF
                      END-IF
                  END-IF
              END-IF
          END-IF
      END-IF

  = notype-deferred % not present, *fetch-notype-stored deferred
    % send a token to each deferred destination
    % return links to the free list
    % store the data starting at location i
    rtemp0 <- r0.pointer

    UNTIL stat: list-status = last
      % fetch the next destination
      r1: data-element         <- (read (rtemp0))[4..35]
      (stat, r2: data-element) <-  read (1+rtemp0)
      % send the data to it's destination
      output-token (r1,          % 4 bytes of destination
                    r2[0..15],   % 5th & 6th bytes of destination
                    r0.type,     % type carried on fetch
                    data)

      % return this link to the free list
      % the following test only needs to be made
      % on the first time through the UNTIL
      IF free-list-status = cells-remain
        THEN write (1+rtemp0,
                    not-last,        % status
                    [blank, blank,   % this field left blank
                     top] )          % pointer to old top
        ELSE write (1+rtemp0,
                    last,               % status
                    [blank, blank,      % this field left blank
                     blank-pointer] )   % no pointer needed
             free-list-status <- cells-remain
        ENDIF
      rtop <- rtemp0
      free-size <- free-size + 1
      rtemp0 <- r2 [16..31]
      END-UNTIL
```

```
% now store the data
write (i,              % write the first word
       present,        % status
       data[0..31] )   % first 4 bytes of data
IF t.length > 4
  THEN % check if the next word was properly allocated
    stat: status <- (read (i+1))[0..3]
    IF stat ≠ middle
      THEN signal (data-too-large    % type of error
                   i,                % address
                   notype-deferred)  % status at this word
           write (i, [['1',notype-deferred] % error status
                      r0.value])            % retain value
      ELSE
        write (i+1,
               middle,         % status
               data[32..63] )  % 5th - 8th bytes of data
        IF t.length > 8
          THEN % check if the next word was properly allocated
            stat: status <- (read (i+2))[0..3]
            IF stat ≠ middle
              THEN
                signal (data-too-large    % type of error
                        i,                % address
                        notype-deferred)  % status here
                write (i, [['1',notype-deferred] % error status
                           r0.value])            % retain value
              ELSE
                write (i+2,
                       middle,         % status
                       data[64..95] )  % 9th - 12th bytes
                IF t.length > 12
                  THEN % check if next word properly allocated
                    stat: status <- (read (i+3))[0..3]
                    IF stat ≠ middle
                      THEN
                        signal (data-too-large    % err type
                                i,                % address
                                notype-deferred)  % status here
                        write (i, [['1',notype-deferred]
                                   r0.value])
                        % set error status and retain value
                      ELSE
                        write (i+3,
                        middle,           % status
                        [data[96..119],   % 13th - 15th bytes
                         blank] )
                      END-IF
                  END-IF
              END-IF
          END-IF
      END-IF
  END-IF
```

```
      OTHERWISE  % signal an error
        signal (store-u-fix-error,  % type of error
                i,                  % address
                r0.status)          % status at this word
        write (i, [['1',r0.status.type]  % new error status
                   r0.value])            % retain value
      END-TAGCASE

    ELSE  % an error has already occurred at this location
      signal (store-u-fix-error,  % type of error
              i,                  % address
              r0.status)          % status at this word
    END-IF

  END *store-u-fix
```

A.4 *FETCH-TYPE-STORED OPERATION

```
*fetch-type-stored (i: address, dest: destination)
  % error signals:
    % length-error
    % free-list-overflow
    % free-list-low
    % fetch-type-stored-error

  % follows *store-t-var
  %      or *store-t-fix
  % fetch as indicated by stored type

  r0: i-element <- read (i)

  IF r0.status.error = none
    THEN TAGCASE ON r0.status.type

      = empty-nowait   % not present, no reads deferred
        % establish a deferred read list:
        IF free-list-status = list-empty
          THEN
            signal (free-list-overflow,  % type of error
                    blank-pointer,       % this information
                    mini-blank)          % is not necessary
            resubmit-token ()
          ELSE  % defer the fetch
            write (i,                   % point to the new link
                   untyped-pointer,     % status
                   [blank,              % type is not saved
                    blank,              % left blank
                    rtop] )             % pointer
            % put destination in a link
            write (rtop,             % put first part of destination
                   mini-blank,       % status
                   dest[0..31] )     % 4 bytes of destination
            % save the pointer to this deferred link
            rtemp0 <- rtop  % this value actually remains in the MAR
            (stat: status, rtop)  % pop the free list
              <- (read(rtop+1))[0..3,20..35]
            write (1+rtemp0,            % put second part of destination
                   last,                % status field not used
                   [dest[32..47],       % 5th & 6th bytes of destination
                    blank-pointer] )    % old deferred list pointer
            IF stat0 = last
              THEN free-list-status <- list-empty
              ENDIF
            free-size <- free-size - 1
            IF free-size = 0
              THEN signal (free-list-low,   % type of error
                           blank-pointer,   % address
                           mini-blank)      % status at this word
              END-IF
          END-IF
```

```
= typed-data   % typed data has been written
  IF r0.type.length ≤ 3
    THEN output-token (dest, r0.type, r0.value)
    ELSE  % read another word
      r1: word <- read (i+1)
      IF r1.status ≠ middle
        THEN signal (length-error,  % type of error
                     i,             % address
                     typed-data)    % status at this word
             write (i, [['1',typed-data] % new error status
                        r0.value])       % retain value
        ELSE  % check if data is formed
          IF r0.type.length ≤ 7
            THEN output-token (dest, t, r0.type,
                               r0.value, r1.value)
                 %  r0 includes type
            ELSE  % read another word
              r2: word <- read (i+2)
              IF r2.status ≠ middle
                THEN signal (length-error,  % type of error
                             i,             % address
                             typed-data)    % status at this word
                     write (i, [['1',typed-data]  % error status
                                r0.value])        % retain value
                ELSE  % check if data is formed
                  IF r0.type.length ≤ 11
                    THEN output-token (dest, r0.type, r0.value,
                                       r1.value, r2.value)
                    ELSE  % the data requires four words to store
                      r3: word <- read (i+3)
                      IF r3.status ≠ middle
                        THEN signal
                               (length-error,% type of error
                                i,           % address
                                typed-data)  % status at this word
                             write (i, [['1',typed-data]
                                        r0.value])
                             % new error status, retain value
                        ELSE output-token (dest, r0.type,
                                           r0.value, r1.value,
                                           r3.value)
                        END-IF
                    END-IF
                END-IF
            END-IF
        END-IF
    END-IF
```

```
= invisible   % present, invisible pointer
  % fetch data from the free list
  r1 : untyped-value <- (read (r0.pointer))[4..35]
  IF r0.type.length ≤ 5
    THEN output-token (dest, r0.type, r0.value, r1)
    ELSE  % read another word
      r2 : data-element <- (read (1+r0.pointer))[4..35]
      % check if the indirect store required only one free cell
      IF r0.type.length ≤ 9
        THEN output-token (dest, r0.type, r0.value,
                              r1, r2)
        ELSE  % get data from next cell
          r3: data-element <- (read (r2[16..31]))[4..35]
          IF r0.type.length ≤ 11
            THEN output-token (dest, r0.type, r0.value,
                                  r1, r2[0..15], r3)
            ELSE  % fetch the last word
              r4: data-element <- (read (1+r2[16..31]))[4..35]
              output-token (dest, r0.type, r0.value,
                               r1, r2[0..15], r3, r4)
            END-IF
        END-IF
    END-IF
```

```
   = type-deferred % not present, *fetch-type-stored deferred
     % add to the deferred read list
     IF free-list-status = list-empty
       THEN signal (free-list-overflow,  % type of error
                    blank-pointer,       % address
                    mini-blank)          % status at this word
            resubmit-token ()
       ELSE  % point to the new link
         write (i,                 % point to the new link
                type-deferred,     % status
                [blank,            % first byte left blank
                 blank,            % second byte left blank
                 rtop] )           % pointer
         % put destination in a link
         write (rtop,             % put first part of destination
                mini-blank        % status
                dest[0..31] )     % 4 bytes of destination
         % save the pointer to this deferred link
         rtemp0 <- rtop  % this value actually remains in the MAR
         (stat: status, rtop)  % pop the free list
           <- (read(rtop+1))[0..3,20..35]
         write (1+rtemp0,          % put second part of destination
                not-last,          % status field not used
                [dest[32..47],     % 5th & 6th bytes of destination
                 r0.pointer] )     % old deferred list pointer
         IF stat0 = last
           THEN free-list-status <- list-empty
           ENDIF
         free-size <- free-size - 1
         IF free-size = 0
           THEN signal (free-list-low,  % type of error
                        blank-pointer,  % address
                        mini-blank)     % status at this word
           END-IF
       ENDIF

   OTHERWISE  % signal an error
     signal (fetch-type-stored-error,  % type of error
             i,                        % address
             r0.status)                % status at this word
     write (i, [['1',r0.status.type]  % new error status
                r0.value])            % retain value
   END-TAGCASE

 ELSE  % an error has already occurred at this location
   signal (fetch-type-stored-error,  % type of error
           i,                        % address
           r0.status)                % status at this word
 END-IF

END *fetch-type-stored
```

A.5 *FETCH-NOTYPE-STORED

```
*fetch-notype-stored (i: address, t: data-type, dest: destination)
  % error signals:
    % deferred-type-collision
    % length-error
    % free-list-overflow
    % free-list-low
    % fetch-notype-stored-error

  % follows *store-u-fix .
  % fetch as indicated by t

  r0: i-element <- read (i)

  IF r0.status.error = none
    THEN TAGCASE ON r0.status.type

      = empty-nowait   % not present, no reads deferred
        % establish a deferred read list:
        IF free-list-status = list-empty
          THEN
            signal (free-list-overflow,  % type of error
                    blank-pointer,       % this information
                    mini-blank)          % is not necessary
            resubmit-token ()
          ELSE  % defer the fetch
            write (i,                 % point to the new link
                   typed-pointer,     % status
                   [t,                % type is saved
                    blank,            % left blank
                    rtop] )           % pointer
            % put destination in a link
            write (rtop,            % put first part of destination
                   mini-blank,      % status
                   dest[0..31] )    % 4 bytes of destination
            % save the pointer to this deferred link
            rtemp0 <- rtop  % this value actually remains in the MAR
            (stat: status, rtop)  % pop the free list
              <- (read(rtop+1))[0..3,20..35]
            write (1+rtemp0,             % put second part of destination
                   last,                 % status field not used
                   [dest[32..47],        % 5th & 6th bytes of destination
                    blank-pointer] )     % no old pointer exists
            IF stat0 = last
              THEN free-list-status <- list-empty
              ENDIF
            free-size <- free-size - 1
            IF free-size = 0
              THEN signal (free-list-low,  % type of error
                           blank-pointer,  % address
                           mini-blank)     % status not needed
              END-IF
          END-IF
```

```
= untyped-data   % untyped data has been written
  IF t.type.length ≤ 4
    THEN output-token (dest, t, r0.value)
    ELSE  % read another word
      r1: word <- read (i+1)
      IF r1.status ≠ middle
        THEN signal (length-error,  % type of error
                     i,             % address
                     untyped-data)  % status at this word
             write (i, [['1',untyped-data]  % new error status
                        r0.value])          % retain value
        ELSE  % check if data is formed
          IF r0.type.length ≤ 8
            THEN output-token (dest, t, r0.value, r1.value)
            ELSE  % read another word
              r2: word <- read (i+2)
              IF r2.status ≠ middle
                THEN signal (length-error,  % type of error
                             i,             % address
                             untyped-data)  % status at this word
                     write (i, [['1',untyped-data]  % err status
                                r0.value])          % & retain
                ELSE  % check if data is formed
                  IF r0.type.length ≤ 12
                    THEN output-token (dest, t, r0.value,
                                       r1.value, r2.value)
                    ELSE  % the data requires four words to store
                      r3: word <- read (i+3)
                      IF r3.status ≠ middle
                        THEN signal
                               (length-error,  % type of error
                                i,             % address
                                untyped-data)  % status at word
                             write (i, [['1',untyped-data],
                                        r0.value])
                             % error stauts, retain value
                        ELSE output-token (dest, t, r0.value,
                                           r1.value, r2.value,
                                           r3.value[0..23])
                      END-IF
                  END-IF
              END-IF
          END-IF
      END-IF
  END-IF
```

```
    = notype-deferred % not present, *fetch-notype-stored deferred
      % add to the deferred read list
      % first check if the previously deferred fetches
      % carried the same type information
      IF r0.type ≠ t
        THEN signal (deferred-type-collision) % type of error
                     i,                       % address
                     notype-deferred)         % status at this word
             write (i, [['1',notype-deferred]  % new error status
                        r0.value])             % retain value
        ELSE  % point to the new link
          write (i,                 % point to the new link
                 notype-deferred,   % status
                 [t,                % type saved here
                  blank,            % second byte left blank
                  rtop] )           % pointer
          % put destination in a link
          write (rtop,           % put first part of destination
                 mini-blank,     % status
                 dest[0..31] )   % 4 bytes of destination
          % save the pointer to this deferred link
          rtemp0 <- rtop  % this value actually remains in the MAR
          (stat: status, rtop)  % pop the free list
            <- (read(rtop+1))[0..3,20..35]
          write (1+rtemp0,        % put second part of destination
                 not-last,        % status field not used
                 [dest[32..47],   % 5th & 6th bytes of destination
                  r0.pointer] )   % old deferred list pointer
          IF stat0 = last
            THEN free-list-status <- list-empty
            ENDIF
          free-size <- free-size - 1
          IF free-size = 0
            THEN signal (free-list-low,  % type of error
                         blank-pointer,  % these fields
                         mini-blank)     % not used
            END-IF
        ENDIF

    OTHERWISE  % signal an error
      signal (fetch-notype-stored-error,  % type of error
              i,                          % address
              r0.status)                  % status at this word
      write (i, [['1',r0.status.type]  % new error status
                 r0.value])            % retain value
    END-TAGCASE

  ELSE  % an error has already occurred at this location
    signal (fetch-notype-stored-error,  % type of error
            i,                          % address
            r0.status)                  % status at this word
  END-IF

END *fetch-notype-stored
```

A.6 *ALLOCATE OPERATION

```
*allocate (base: address, blocks <0..7>, element-size: <0..1>)
  % error signals:
    % allocate-error

  % allocates blocks element-sized words
  % starting at address base.

  FOR i: INTEGER <- 0 TO (blocks - 1) BY 1 DO
      r0: i-element <- read (base+i(element-size))
      IF r0.status.error = none
        THEN TAGCASE ON r0.status.type
          = not-allocated  % not allocated
            write (base+i,        % set status bits to allocated
                   empty-nowait,  % status
                   [blank,blank,  % data field blanked out
                    blank,blank]
          OTHERWISE  % signal an error
            signal (allocate-error,         % type of error
                    base+i,                 % address
                    r0.status)              % status at this word
            write (i, [['1',r0.status.type]  % new error status
                       r0.value])            % retain value
          END-TAGCASE
        ELSE  % an error has already occurred at this location
            signal (allocate-error,         % type of error
                    base+i,                 % address
                    r0.status)              % status at this word
        END-IF


    FOR j: INTEGER <- 1 TO (element-size - 1) BY 1 DO
      r0: i-element <- read (base+i(element-size)+j)
      IF r0.status.error = none
        THEN TAGCASE ON r0.status.type
          = not-allocated  % not allocated
            write (base+j,        % set status bits to middle
                   middle,        % status
                   [blank,blank,  % data field blanked out
                    blank,blank] )
          OTHERWISE  % signal an error
            signal (allocate-error,         % type of error
                    base+j,                 % address
                    r0.status)              % status at this word
            write (base+j, [['1',r0.status.type]  % new error status
                            r0.value])            % retain value
          END-TAGCASE
        ELSE  % an error has already occurred at this location
          signal (allocate-error,  % type of error
                  base+j,          % address
                  r0.status)       % status at this word
        END-IF
      END-FOR
    END-FOR
  END *allocate
```

A.7 *CLEAR OPERATION

```
*clear (base: address, blocksize: <0..9>)
  % error signals:
    % clear-error

  % clears blocksize words
  % starting at address base.

  FOR j: INTEGER <- 0 TO (block-size - 1) BY 1 DO
    r0: i-element <- read.(base + j)
    IF r0.status.error = none
      THEN TAGCASE ON r0.status.type

        = empty-nowait  % not present, no reads deferred
          write (base+j,         % set status bits to allocated
                 not-allocated,  % status
                 [blank, blank,  % data field blanked out
                  blank, blank] )

        = typed-data   % typed data is present
          write (base+j,         % set status bits to allocated
                 not-allocated,  % status
                 [blank, blank,  % data field blanked out
                  blank, blank] )

        = untyped-data   % untyped data is present
          write (base+j,         % set status bits to allocated
                 not-allocated,  % status
                 [blank, blank,  % data field blanked out
                  blank, blank] )

        = middle  % middle of an element
          write (base+j,         % set status bits to allocated
                 not-allocated,  % status
                 [blank, blank,  % data field blanked out
                  blank, blank] )
```

```
= invisible  % present, invisible pointer
  % return cells to the free list RPLACA style
  rtemp0 <- rtop
  rtop <- r0.pointer  % set the new top
  IF r0.type.length ≤ 9  % determine size of chain
    THEN  % one cell in chain
      IF free-list-status = cells-remain
        THEN write (1+rtop,           % REPLACA !!!
                    not-last,         % status
                    [blank, blank,    % this field left blank
                     rtemp0] )        % pointer to old top
        ELSE write (1+rtop,              % REPLACA
                    last,                % status
                    [blank, blank,       % this field left blank
                     blank-pointer] )    % no pointer needed
             free-list-status <- cells-remain
        ENDIF
    ELSE  % two cells in chain
      % get pointer to next cell
      rtemp1 <- (read (1+rtop)) [20..35]
      % write list status in first cell
      write (1+rtop,
             not-last,      % status
             [blank,blank,  % this field left blank
              rtemp1] )     % pointer to next cell
      % write list status in second cell
      IF free-list-status = cells-remain
        THEN write (1+rtemp1,         % REPLACA !!!
                    not-last,         % status
                    [blank, blank,    % this field left blank
                     rtemp0] )        % pointer to old top
        ELSE write (1+rtemp1,            % REPLACA !!!
                    last,                % status
                    [blank, blank,       % this field left blank
                     blank-pointer] )    % no pointer needed
             free-list-status <- cells-remain
        ENDIF
      free-list-status <- cells-remain
  % now reset the status
  write (base+j,  % set status bits to allocated
         clear,          % status
         [blank, blank,  % data field blanked out
          blank, blank] )

OTHERWISE  % signal an error
  signal (allocate-error,         % type of error
          base+j,                 % address
          r0.status)              % status at this word
  write (base+j, [['1',r0.status.type]  % new error status
                  r0.value])            % retain value
END-TAGCASE
```

```
    ELSE  % an error has already occurred at this location
      signal (allocate-error,  % type of error
              base+j,          % address
              r0.status)       % status at this word
    END-IF
  END-FOR
END *clear
```

A.8 *RESET OPERATION

```
*reset (i: address)
  % error signals:
    % reset-error

  % used to recover from the error state.
  % if the ith word has an error status, the status
  % is changed to it's previous value.
  % if the ith word doesn't has an error status, the status
  % is changed to error.

  r0: i-element <- read (i)
  IF r0.status.error = occurred
    THEN  % an error has occurred at this location
      write (base, [['0',r0.status.type]  % reset error bit
                    r0.value])            % retain value
    ELSE  % we are trying to reset when no error has occurred
      signal (reset-error,            % type of error
              i,                      % address
              r0.status)              % status at this word
      write (base, [['1',r0.status.type]  % set error bit
                    r0.value])            % retain value
    END-IF

  END *reset
```

A.9 *READ-ABSOLUTE OPERATION

```
*read-absolute (i: address, dest: destination)

  % read the status and data memories directly
  % perform no checking

  output-token (dest, read (i))
  END *read-absolute
```

A.10 *WRITE-ABSOLUTE OPERATION

```
*write-absolute (i: address, data: <0..35>)

  % write the status and data memories directly
  % perform no checking

   write (i,  % write the data directly
          data[0..3],   % status
          data[4..35],  % data
  END *write-absolute
```

A.11 *INITIALIZE OPERATION

```
*initialize (boundary: address, warning: <0..15>)

  % set the boundary between the data section and
  % the free list section to boundary.  initialize
  % the data section to unallocated, and set up a
  % free list in the free list section.  Ignore the
  % status or value at any word.  this operation
  % clears the entire memory.

  % set the boundary register
  bound <- boundary

  % initialize the data section
  FOR i: address <- 0 TO (bound-1) BY 1 DO
    write (i,                    % initialize each word
           empty-waiting,        % status
           [blank, blank,        % data field left blank
            blank, blank] )
    END-FOR

  % initialize the free list
  % set up the tail of the list
  rtemp0 <- bound
  i: address <- bound + 1
  write (i,                     % initialize each word
         last,                  % status for end of list
         [blank, blank,         % this field not used
          blank-pointer] )      % no pointer here

  % set up the middle elements of the list
  UNTIL i = 0 DO
    rtemp1 <- rtemp0  % save the pointer to the last cell
    rtemp0 <- i       % save the pointer to this cell
    % note that only one extra is really necessary
    % if we allow the MDR to hold information
    i <- i + 1
    write (i,                   % initialize each word
           not-last,            % status
           [blank, blank,       % this field not used
            rtemp1] )           % pointer to next free-cell in list
    i <- i + 1  % point to next word
    END-UNTIL

  % store the top of the list
  rtop <- rtemp0

  END *initialize
```

A.12 *LOAD-ERROR-DESTINATION OPERATION

```
*load-error-destination (dest: destination)

  % sets the error destination to dest

  error-destination <- dest
  END *load-error-destination
```

A.13 *ALLOCATE-FREE-SPACE OPERATION

```
*allocate-free-space (base: address, cells: <0..15>)
  % adds "cells" cells to the free list from the
  % contiguous unallocated block beginning at base

  r0: i-element <- read (base)
  IF r0.status.error = none
    THEN TAGCASE ON r0.status.type
      = not-allocated  % not allocated (already clear)
        % check next cell
        r1: i-element <- read (base+1)
        IF r1.status.error = none
           THEN TAGCASE ON r1.status.type
             = not-allocated  % not allocated (already clear)
               % add to the free list
               IF free-list-status = cells-remain
                 THEN write (base+1,
                             not-last,        % status
                             [blank, blank,   % this field left blank
                              rtop] )         % pointer to old top
                 ELSE write (base+1,
                             last,               % status
                             [blank, blank,      % this field left blank
                              blank-pointer] )   % no pointer needed
                        free-list-status <- cells-remain
                 ENDIF
               rtop <- base
             OTHERWISE  % signal an error
               signal (allocate-free-space-error,  % type of error
                       base+1,                     % address
                       r0.status)                  % status at this word
               write (base, [['1',r0.status.type]  % new error status
                             r0.value])            % retain value
             END-TAGCASE
           ELSE  % an error has already occurred at this location
             signal (allocate-free-space-error,  % type of error
                     base+1,                     % address
                     r0.status                   % status at this word
           END-IF
      OTHERWISE  % signal an error
        signal (allocate-free-space-error,  % type of error
                base,                       % address
                r0.status)                  % status at this word
        write (base, [['1',r0.status.type]  % new error status
                      r0.value])            % retain value
      END-TAGCASE
    ELSE  % an error has already occurred at this location
        signal (allocate-free-space-error,  % type of error
                base,                       % address
                r0.status                   % status at this word
    END-IF
```

```
% set up middle elements for the rest of the list
i: address <- base + 2
IF i < base + cells
  THEN  % there are more cells to add to the list
    rtemp0 <- base
    WHILE (i ≤ base + cells) DO
      r0: i-element <- read (i)
      IF r0.status.error = none
        THEN TAGCASE ON r0.status.type
          = not-allocated  % not allocated (already clear)
            i <- i + 1
            r1: i-element <- read (i)
            IF r1.status.error = none
              THEN TAGCASE ON r1.status.type
                = not-allocated  % not allocated (already clear)
                  rtemp1 <- rtemp0  % save a pointer to the last cell
                  rtemp0 <- i       % save the pointer to this cell
                  % note that only one extra is realy necessary
                  % if we allow the MDR to hold information
                  write (i,                % initialize each word
                         middle,           % status
                         [blank, blank,    % this field not used
                          rtemp1] )        % pointer to next free-cell
                  i <- i + 1  % point to next word
                OTHERWISE  % signal an error
                  signal (allocate-free-space-error,  % type of error
                          base+i,                     % address
                          r0.status)                  % status
                  write (base+i, [['1',r0.status.type]  % error status
                                  r0.value])            % retain value
                END-TAGCASE
              ELSE  % an error has already occurred at this location
                signal (allocate-free-space-error,  % type of error
                        base+i,                     % address
                        r0.status                   % status
              END-IF
          OTHERWISE  % signal an error
            signal (allocate-free-space-error,  % type of error
                    base+i,                     % address
                    r0.status)                  % status at this word
            write (base+i, [['1',r0.status.type]  % new error status
                            r0.value])            % retain value
          END-TAGCASE

        ELSE  % an error has already occurred at this location
          signal (allocate-free-space-error,  % type of error
                  base+i,                     % address
                  r0.status                   % status at this word
        END-IF
      END-WHILE
  END-IF
END *allocate-free-space
```

A.14 *GET-FREE-SIZE OPERATION

```
*get-free-size (dest: destination)

  % sends the size of the free list to dest
  % (actually, the quantity sent is offset from
  % the true size by the amount specified when
  % the memory was last initialized)

  output-token (dest, free-size)
  END *write-absolute
```

B.0 IDL CODE

```
% I-STRUCTURE MEMORY CONTROLLER - IDL CODE
% 04/08/83 11:50:26 % SKH

% PRIMARY INPUTS:
%   OPERATIONS:
GRP OPCODE <-OPCODE<3 2 1 0>
%    NORMAL OPERATIONS
%      <*STORE-T-FIX          <-0 HEX>
%      <*STORE-T-VAR          <-1 HEX>
%      <*STORE-U-FIX          <-2 HEX>
%      <*FETCH-TYPE-STORED    <-4 HEX>
%      <*FETCH-NOTYPE-STORED <-5 HEX>
%      <*ALLOCATE             <-6 HEX>
%      <*CLEAR                <-7 HEX>
%      <*RESET                <-3 HEX>
%      <*ALLOCATE-FREE-SPACE <-8 HEX>
%    SERVICE OPERATIONS
%      <*READ-ABSOLUTE            <-9 HEX>
%      <*WRITE-ABSOLUTE           <-A HEX>
%      <*INITIALIZE               <-B HEX>
%      <*LOAD-ERROR-DESTINATION <-C HEX>
%      <*GET-FREE-SIZE            <-D HEX>
%      *GET-FREE-SIZE IS NOT IMPLEMENTED
%
%    LENGTH: <3..0>
        GRP LENGTH <-LENGTH<3 2 1 0>   LENGTH FROM INTERNAL MEMORY
%
%    LENGTH-IN: <3..0>
        GRP LENGTHIN <-LENGTHIN<3 2 1 0> LENGTH FROM INCOMING TOKEN
%
%    STATUS-IN: STATUS-TYPE
        GRP STIN <-STIN<3 2 1 0>   STATUS FROM INTERNAL MEMORY
%
%    STATUS-TYPE:   CODES FOR STATUS STORED INTERNALLY
%      I-ELEMENT STATUS
%        <NOT-ALLOCATED    <- 0 HEX>
%        <EMPTY-NOWAIT     <- 1 HEX>
%        <TYPED-DATA       <- 2 HEX>
%        <UNTYPED-DATA     <- 3 HEX>
%        <MIDDLE           <- 4 HEX>
%        <INVISIBLE        <- 5 HEX>
%        <TYPE-DEFERRED    <- 6 HEX>  *FETCH-TYPE-STORED DEFERRED
%        <NOTYPE-DEFERRED <- 7 HEX>  *FETCH-NOTYPE-STORED DEFERRED
%        <ERROR-NOT-ALLOCATED    <- 8 HEX>
%        <ERROR-EMPTY-NOWAIT     <- 9 HEX>
%        <ERROR-TYPED-DATA       <- A HEX>
%        <ERROR-UNTYPED-DATA     <- B HEX>
%        <ERROR-MIDDLE           <- C HEX>
%        <ERROR-INVISIBLE        <- D HEX>
%        <ERROR-TYPE-DEFERRED    <- E HEX>
%        <ERROR-NOTYPE-DEFERRED <- F HEX>
%        <ERROR        <-1 - - ->    AN ERROR HAS OCCURRED HERE
%        <NO-ERROR     <-0 - - ->    NO ERROR HAS OCCURRED HERE
%        <SET-TO-NO-ERROR <-0>          SET ERROR BIT OFF
%        <SET-TO-ERROR    <-1>          SET ERROR BIT ON
%       LIST-STATUS
%        <NOT-LASTI <-- - - 1>
%        <LASTI     <-- - - 0>
%        <NOT-LASTO <-? ? ? 1>
%        <LASTO     <-? ? ? 0>
%
%    I-STRUCTURE HANDSHAKING INPUTS:  TOKIN, LASTIN, TOKOUT, INT
%      <INPUT-SECTION-REQUEST      <-TOKINI=1>
%      <LAST-TOKEN-PART-IN         <-LASTINI=1>
%      <IS-OUTPUT-SECTION-FREE     <-TOKOUTI=0>
%      <INTERRUPT-REQUEST          <-INTI=1>
%      <NO-INTERRUPT-REQUEST       <-INTI=0>
%
%    MEMORY-TIME-CONSTANT:  MEM1,MEM0
%      (ONE =>   11,  NUMBER OF PLA CYCLES THE CONTROLLER MUST
%       TWO =>   00,  WAIT BETWEEN INTERNAL MEMORY ACCESSES
%       THREE => 01,  ONE => A MEMORY OPERATION CAN
%       FOUR =>  10)  BE ISSUED EVERY CYCLE.
%
%    ALLOCATE-BLOCK-SIZE:  ABIN1,ABIN0
GRP ABSIZE<-ABSIZE<1 0>
%      (ONE =>   00,  THE NUMBER OF CELLS IN AN ENRTY DURING
%       TWO =>   01,  THE *ALLOCATE OPERATION.
%       THREE => 10,  ONE => EVERY WORD IS A HEADER.
%       FOUR =>  11)  THREE => EVERY THIRD WORD IS A HEADER.
%
%    RESET-I-STRUCTURE: <0>  PORESET   POWER-ON RESET
%      <STOP <-PORESET = 1>
%      <OK   <-PORESET = 0>
%
%   EXTERNAL HARDWARE TESTS
%     <MAR-EQUALS <-EQUALQI=1>
%     <MAR = FBR <-<MAR-EQUALS>>
%     <WRAP <-<MAR-EQUALS>>
%     <ALLOCATION-COMPLETE <-<MAR-EQUALS>>
%     <ALLOCATE-FREE-SPACE-COMPLETE <-<MAR-EQUALS>>
%       DEPENDING ON THE CONTENTS OF COMPARE (16 BIT REGISTER),
%       THIS TEST DETERMINES
%         IF WE ARE IN THE FIRST PART OF THE INITIALIZE OPERATION
%           => IF MAR HAS REACHED THE FREE-LIST SECTION
%         IF WE ARE IN THE SECOND PART OF THE INITIALIZE OPERATION
%           => IF MAR HAS WRAPPED AROUND THE MEMORY
%         IF WE ARE IN THE ALLOCATE OPERATION
%           => IF MAR HAS REACHED THE LAST WORD TO ALLOCATE
%     CHECK IF TYPE CHECKS FOR WRITE:  TYPEQI
%       <TYPE-CHECKS <-TYPEQI=1>
%

% PRIMARY OUTPUTS:
%    INTERNAL-MEMORY-OPERATION:
%      CS,WEI - CHIP SELECT, WRITE ENABLE INVERT
%      <WRITEO <-CS<-1/WEI<-0/<SET-MEMORY-COUNT>>
%      <READO <-CS<-1/WEI<-1/<SET-MEMORY-COUNT>>
```

```
%
%    INC-MAR:   INCREMENT INTERNAL MAR
%      <INC-MAR <-MAR <-INC MAR>
%
%    FREE-CHECK:   TEST FOR FREE LIST BOUNDARY
%      <CHECK-FREE          <-COMPARE<-BOUND>      FREE LIST BOUNDARY
%      <CHECK-WRAP          <-COMPARE<-CLEARREG>   WRAP AROUND
%      <CHECK-END-ALLOCATE <-COMPARE<-MDRD2T3>    END OF ALLOCATE
%
%    STATUS-OUT: STATUS-TYPE
       GRP STOUT <-STOUT<3 2 1 0>   STATUS TO INTERNAL MEMORY
%
%    I-STRUCTURE HAND SHAKING OUTPUTS: TOKIN, TOKOUT, INT
%         <INPUT-REQUEST-ACKNOWLEDGE <-TOKIN<-ZERO>
%         <OUTPUT-SECTION-REQUEST    <-TOKOUT<-ONE>
%         <INTERRUPT-ACKNOWLEDGE     <-INT<-ZERO>
%
%
%    ERROR-SIGNALS:   ERROR<3..0>
GRP ERRCODE<-ERROR<3 2 1 0>
%      <STORE-T-FIX-ERROR           <-<*STORE-T-FIX>>
%      <STORE-T-VAR-ERROR           <-<*STORE-T-VAR>>
%      <STORE-U-FIX-ERROR           <-<*STORE-U-FIX>>
%      <FETCH-TYPE-STORED-ERROR     <-<*FETCH-TYPE-STORED>>
%      <FETCH-NOTYPE-STORED-ERROR <-<*FETCH-NOTYPE-STORED>>
%      <ALLOCATE-ERROR              <-<*ALLOCATE>>
%      <CLEAR-ERROR                 <-<*CLEAR>>
%      <RESET-ERROR                 <-<*RESET>>
%      <ALLOCATE-FREE-SPACE-ERROR <-<*ALLOCATE-FREE-SPACE>>
%      <DATA-TOO-LARGE              <-9 HEX>
%      <FREE-LIST-OVERFLOW          <-A HEX>
%      <FREE-LIST-LOW               <-B HEX>
%      <LENGTH-ERROR                <-C HEX>
%      <DEFERRED-TYPE-COLLISION     <-D HEX>
%
%      ERRORS 0-8 USE THE SAME CODES AS DO THE CORRESPONDING
%      INSTRUCTIONS.
%
GRP OPERR<-OPREG,ERRORS
%<OPREG-ERRORS<-OPERR>

% FEEDBACKS:
%
%    MEMORY-TIME:   MEMTIME1,MEMTIME0
GRP MEMTIME <-MEMTIME<1 0>
UNLATCH MEMTIME
%      (ONE   =>  11,   CURRENT VALUE REPRESENTS THE NUMBER OF PLA
%       TWO =>    00,   CYCLES THE CONTROLLER HAS ALREADY WAITED
%       THREE => 01,    SINCE ISSUING THE LAST MEMORY REQUEST.
%       FOUR =>   10)   ONE => A REQUEST TOOK PLACE LAST CYCLE.
%
%    MEMORY-TIME:   MEMTIME1,MEMTIME0
%      <INCREMENT-MEMORY-COUNT <-*MEMTIME<-INCR2 MEMTIME>
%      <SET-MEMORY-COUNT <-MEMTIME <-1 1>
%      <MEM-FINI <-MEMTIME1=MEM1 / MEMTIME0=MEM0>
%
%    ALLOCATE-BLOCK-COUNT:  ABCNT
```

```
GRP ABCNT<-ABCNT<1 0>
%      (ZERO =>  00,   CURRENT VALUE REPRESENTS THE NUMBER OF WORDS
%       ONE =>   01,   THE CONTROLLER MUST WRITE 'MIDDLE' BEFORE
%       TWO =>   10,   WRITING THE NEXT HEADER.
%       THREE => 11)   ZERO => ELEMENT HEADER CAN BE WRITTEN NOW.
%
%    ALLOCATION-TIME:   ABCNT1,ABCNT0
%      <COUNT-ALLOCATION <-*ABCNT<-DECR2 ABCNT>
%      <SET-ALLOCATION-COUNT <-ABCNT <-ABSIZE>
%      <HEADER-TIME <-ABCNT = 0 0>
%
%      LIST-EMPTY => NO MORE CELLS ARE AVAILABLE IN THE FREE LIST.
%           ANY OPERATION REQUIRING SPACE ON THE FREE LIST WILL
%           CAUSE A FREE-LIST-OVERFLOW ERROR.
%      CELLS-REMAIN => AT LEAST 1 MORE CELL IS IN THE FREE LIST.
%      <IS-LIST-EMPTY <-LASTCELL=1>
%      <LIST-IS-EMPTY <-LASTCELL<-1>
%      <DO-CELLS-REMAIN <-LASTCELL=0>
%      <CELLS-DO-REMAIN <-LASTCELL<-0>
%

% ASSOCIATED REGISTERS:
% <I0T9   <- 0  1  2  3  4  5  6  7  8  9>
% <I10T19<-10 11 12 13 14 15 16 17 18 19>
% INPUT SECTION
DIM IREG0  8 / IREG1  8 / IREG2  8 / IREG3  8
DIM IREG4  8 / IREG5  8 / IREG6  8 / IREG7  8
DIM IREG8  8 / IREG9  8 / IREG10 8 / IREG11 8
DIM IREG12 8 / IREG13 8 / IREG14 8 / IREG15 8
DIM IREG16 8 / IREG17 8 / IREG18 8 / IREG19 8
GRP IREG0T1    <-IREG<0 1>
GRP IREG1T2    <-IREG<1 2>
GRP IREG2T3    <-IREG<2 3>
GRP IREG4T5    <-IREG<4 5>
GRP IREG6T7    <-IREG<6 7>
GRP IREG7T8    <-IREG<7 8>
GRP IREG8T9    <-IREG<8 9>
GRP IREG10T11 <-IREG<10 11>
GRP IREG12T13 <-IREG<12 13>
GRP IREG14T15 <-IREG<14 15>
GRP IREG16T17 <-IREG<16 17>
GRP IREG18T19 <-IREG<18 19>
% <I0T19<-<I0T9> <I10T19>>
GRP IREG     <-IREG<<I0T19>>
GRP IREG2T7  <-IREG<2 3 4 5 6 7>
GRP IREG4T19 <-IREG<4 5 6 7 8 9 <I10T19>>
FIX CHAIN<-IREG0[0]

% OUTPUT SECTION
DIM OREG0  8 / OREG1  8 / OREG2  8 / OREG3  8
DIM OREG4  8 / OREG5  8 / OREG6  8 / OREG7  8
DIM OREG8  8 / OREG9  8 / OREG10 8 / OREG11 8
DIM OREG12 8 / OREG13 8 / OREG14 8 / OREG15 8
DIM OREG16 8 / OREG17 8 / OREG18 8 / OREG19 8
DIM OREG20 8 / OREG21 8
%<I0T21<-<I0T19> 20 21>
GRP OREG     <-OREG<<I0T21>>
GRP OREG0T3   <-OREG<0 1 2 3>
GRP OREG0T5   <-OREG<0 1 2 3 4 5>
GRP OREG4T5   <-OREG<4 5>
GRP OREG6T7   <-OREG<6 7>
GRP OREG6T9   <-OREG<6 7 8 9>
GRP OREG6T21 <-OREG<6 7 8 9 <I10T19> 20 21>
GRP OREG7T21 <-OREG<7 8 9 <I10T19> 20 21>

% CONTROLLER SECTION
% I-STRUCTURE DESTINATION REGISTER
DIM DEST0 8/DEST1 8/DEST2 8
DIM DEST3 8/DEST4 8/DEST5 8
GRP DEST0T1 <-DEST<0 1>
GRP DEST2T3 <-DEST<2 3>
GRP DEST2T5 <-DEST<2 5>
GRP DEST4T5 <-DEST<4 5>
GRP DEST0T3 <-DEST<0 1 2 3>
GRP DEST <-DEST<0 1 2 3 4 5>
DIM TYPE 8
DIM ABREG 2
FIX ABSIZE<-ABREG
DIM COMPARE 16   AN ADDRESS TO BE COMPARED WITH MAR
% I-STRUCTURE OPERATION REGISTER
DIM OPREG 4
FIX OPCODE<-OPREG
% I-STRUCTURE ADDRESS REGISTER
DIM ADDR 16
% DATA ORGANIZATION
% BYTE 0 IS TYPE INFORMATION: LENGTH AND CLASS
DIM DATA0  8/DATA1  8/DATA2  8/DATA3  8
DIM DATA4  8/DATA5  8/DATA6  8/DATA7  8
DIM DATA8  8/DATA9  8/DATA10 8/DATA11 8
DIM DATA12 8/DATA13 8/DATA14 8/DATA15 8
% <I0T15 <-<I0T9> 10 11 12 13 14 15>
GRP DATA <-DATA<<I0T15>>   DATA TO BE WRITTEN
% <I1T15 <-1 2 3 4 5 6 7 8 9 10 11 12 13 14 15>
GRP DAT0T1    <-DATA<0 1>          BYTES 0  TO 1
GRP DAT0T3    <-DATA<0 1 2 3>      BYTES 0  TO 3
GRP DAT0T5    <-DATA<0 1 2 3 4 5>  BYTES 0  TO 5
GRP DAT1T4    <-DATA<1 2 3 4>      BYTES 1  TO 4
GRP DAT1T15   <-DATA<<I1T15>>      BYTES 1  TO 15
GRP DAT2T3    <-DATA<2 3>          BYTES 2  TO 3
GRP DAT2T5    <-DATA<2 3 4 5>      BYTES 2  TO 5
GRP DAT2T7    <-DATA<2 3 4 5 6 7>  BYTES 2  TO 7
GRP DAT4T7    <-DATA<4 5 6 7>      BYTES 4  TO 7
GRP DAT5T8    <-DATA<5 6 7 8>      BYTES 5  TO 8
GRP DAT6T7    <-DATA<6 7>          BYTES 6  TO 7
GRP DAT7T8    <-DATA<7 8>          BYTES 7  TO 8
GRP DAT8T9    <-DATA<8 9>          BYTES 8  TO 9
GRP DAT8T11   <-DATA<8 9 10 11>    BYTES 8  TO 11
GRP DAT9T12   <-DATA<9 10 11 12>   BYTES 9  TO 12
GRP DAT12T15 <-DATA<12 13 14 15>   BYTES 12 TO 15
GRP DAT13T15 <-DATA<13 14 15>      BYTES 13 TO 15
FIX LENGTHIN <-DATA0[0 1 2 3]
%
DIM MAR 16   MEMORY ADDRESS REGISTER
%    MDR   MEMORY DATA REGISTER
%       STATUS: 4    MDRSTAT
%       DATA:   32   MDRDATA
DIM MDRSTAT 4
DIM MDRD0A 4 / MDRD0B 4
GRP MDRDATA0 <-MDRD0<A B>
DIM MDRDATA1 8/MDRDATA2 8/MDRDATA3 8
GRP MDRDATA <-MDRDATA0,MDRDATA1,MDRDATA2,MDRDATA3
GRP MDRD0T1  <-MDRDATA0,MDRDATA1
GRP MDRD0T2  <-MDRDATA0,MDRDATA1,MDRDATA2
GRP MDRD2T3  <-MDRDATA2,MDRDATA3
FIX LENGTH <-MDRD0A
%
GRP MEMARG<-MAR,MDRSTAT,MDRDATA
FIX STIN<-MDRSTAT
FIX MDRSTAT<-STOUT
DIM EXTRA 16     EXTRA INTERNAL ADDRESS REGISTER
DIM BOUND 16     BOUNDARY BETWEEN DATA AND FREE LIST
DIM RTOP 16      CURRENT TOP OF THE FREE LIST
DIM MEMCTL 2     INTERNAL MEMORY CONTROL REGISTER
FIX MEMCTL[0]<-CS/MEMCTL[1]<-WEI
DIM ERRORS 4
FIX ERRORS <-ERRCODE
DIM ERRDEST 48    DESTINATION FOR MEMORY ERRORS
DIM CLEARREG 16   CONSTANT, CONTAINS: '0000'X
DIM TYPEQ 1    OUTPUT OF CHKTYPE, BBOX TO CHECK TYPE
FIX TYPEQI<-TYPEQ
DIM EQUALQ 1   OUTPUT OF CHKBOUND, CHECKS FREE LIST BOUNDRY
FIX EQUALQI<-EQUALQ
% HANDSHAKING REGISTERS
DIM TOKIN 1/TOKOUT 1/INT 1/LASTIN 1
FIX TOKINI<-TOKIN
FIX TOKOUTI<-TOKOUT
FIX INTI<-INT
FIX LASTINI<-LASTIN


% IDL CODE
% SUBCYCLE STRUCTURE:
%     PLA |  3 SUB     |  SUB   |     SUB
%         |  CYCLES    | CYCLE  |    CYCLE
%     ----|---|---|---|-------|------------
%         | REGISTER   | BLACK  | SECONDARY
%         | TRANSFERS  | BOXES  | BLACK BOXES
%         |            |        | (ZERO TIME DELAYS)
%


% MEMORY CYCLE:
Univ: <INCREMENT-MEMORY-COUNT> ||| MDRDATA<-MEMCTL MEMORY MEMARG

% BLACK BOX TESTS:
% (ZERO TIME DELAYS)
% (SIMULATED AS LAST SUBCYCLE ACTIONS)
Univ: |||| TYPEQ<-MDRDATA0 EQBOX DATA0   CHECK TYPE EQUALITY
Univ: |||| EQUALQ<-MAR EQBOX COMPARE     VARIOUS MAR COMPARISONS
```

```
% HARD RESET SECTION
%
Univ: IF <STOP> THEN->Univ/->INITIAL
INITIAL: IF <OK> THEN->READY/->INPUT/->OUTPUT

% INPUT SECTION
% TOKENS COME IN THROUGH A 16 BIT DATA PATH
% INPUT FORMAT:
%   STORE OPERATIONS:
%     INFO: [CHAIN:  (NO-DEST => 0-------),
%            OPCODE: (*STORE-T-FIX       => *STORE-T-FIX,
%                     *STORE-T-VAR       => *STORE-T-VAR,
%                     *STORE-U-FIX       => *STORE-U-FIX),
%            UNUSED: <4..7>],
%     ADDRESS: <0..15>,
%     DATA: [TYPE: [LENGTH: <0..3>, CLASS: <0..3>],
%            DATA: <0..119>]
%
%   FETCH OPERATIONS:
%     INFO: [CHAIN:  (DEST => 1-------),
%            OPCODE: (*FETCH-TYPE-STORED
%                       => *FETCH-TYPE-STORED,
%                     *FETCH-NOTYPE-STORED
%                       => *FETCH-NOTYPE-STORED),
%     DESTINATION: <0..47>,
%     ADDRESS: <0..15>]
%
%   ALLOCATE OPERATION:
%     INFO: [CHAIN:  (NO-DEST => 0-------),
%            OPCODE: (*ALLOCATE => *ALLOCATE)
%            UNUSED: <4..7>],
%     START: <0..15>,
%     STOP: <0..15>,
%     BLOCK-SIZE: [SIZE: (ONE   => 00, TWO  => 01,
%                         THREE => 10, FOUR => 11),
%                  UNUSED: <2..7>]]
%
%   CLEAR OPERATION:
%     INFO: [CHAIN:  (NO-DEST => 0-------),
%            OPCODE: (*CLEAR => *CLEAR)
%            UNUSED: <4..7>],
%     START: <0..15>,
%     STOP: <0..15>]
%
%   RESET OPERATION:
%     INFO: [CHAIN:  (NO-DEST => 0-------),
%            OPCODE: (*RESET => *RESET)
%            UNUSED: <4..7>],
%     ADDRESS: <0..15>]
%
%   ALLOCATE-FREE-SPACE OPERATION:
%     INFO: [CHAIN:  (NO-DEST => 0-------),
%            OPCODE: (*ALLOCATE-FREE-SPACE
%                       => *ALLOCATE-FREE-SPACE)
%            UNUSED: <4..7>],
%     START: <0..15>,
%     STOP: <0..15>]
```

```
%
%   READ-ABSOLUTE OPERATION:
%     INFO: [CHAIN:   (DEST => 1-------),
%            OPCODE: (*READ-ABSOLUTE => *READ-ABSOLUTE),
%            UNUSED: <4..7>],
%     DESTINATION: <0..47>,
%     ADDRESS: <0..15>]
%
%   WRITE-ABSOLUTE OPERATION:
%     INFO: [CHAIN:   (NO-DEST => 0-------),
%            OPCODE: (*WRITE-ABSOLUTE => *WRITE-ABSOLUTE),
%            UNUSED: <4..7>],
%     ADDRESS: <0..15>,
%     DATA: [VALUE: <0..31>, STATUS: <0..3>]]
%
%   INITIALIZE OPERATION:
%     INFO: [CHAIN:   (NODEST => 0-------),
%            OPCODE: (*INITIALIZE => *INITIALIZE),
%            UNUSED: <4..7>],
%     BOUNDARY: <0..15>]
%
%   LOAD-ERROR-DESTINATION OPERATION:
%     INFO: [CHAIN:   (DEST => 1-------),
%            OPCODE: (*LOAD-ERROR-DESTINATION
%                         => *LOAD-ERROR-DESTINATION),
%            UNUSED: <4..7>],
%     ERROR-DESTINATION: <0..47>]
%
%   GET-FREE-SIZE OPERATION:
%     INFO: [CHAIN:   (DEST => 1-------),
%            OPCODE: (*GET-FREE-SIZE => *GET-FREE-SIZE),
%            UNUSED: <4..7>],
%     DESTINATION: <0..47>]


INPUT: IF <OK>/<INPUT-SECTION-REQUEST>/INQ=0
, THEN <INPUT-REQUEST-ACKNOWLEDGE>
,   | IREGOT1<-TOKENQ TOKIN /->INPUTO
%...............................................
INPUTO: IF <OK>/<INPUT-SECTION-REQUEST> THEN->INPUTO.DONEQ
INPUTO.DONEQ: <INPUT-REQUEST-ACKNOWLEDGE>
, / IREG2T3<-TOKENQ TOKIN
INPUTO.DONEQ: IF <LAST-TOKEN-PART-IN>
, THEN INQ<-1 /->INPUT / ELSE->INPUT1
%...............................................
INPUT1: IF <OK>/<INPUT-SECTION-REQUEST> THEN->INPUT1.DONEQ
INPUT1.DONEQ: <INPUT-REQUEST-ACKNOWLEDGE>
, / IREG4T5<-TOKENQ TOKIN
INPUT1.DONEQ: IF <LAST-TOKEN-PART-IN>
, THEN INQ<-1 /->INPUT / ELSE->INPUT2
%...............................................
INPUT2: IF <OK>/<INPUT-SECTION-REQUEST> THEN->INPUT2.DONEQ
INPUT2.DONEQ: <INPUT-REQUEST-ACKNOWLEDGE>
, / IREG6T7<-TOKENQ TOKIN
INPUT2.DONEQ: IF <LAST-TOKEN-PART-IN>
, THEN INQ<-1 /->INPUT / ELSE->INPUT3
%...............................................
```

```
INPUT3: IF <OK>/<INPUT-SECTION-REQUEST> THEN->INPUT3.DONEQ
INPUT3.DONEQ: <INPUT-REQUEST-ACKNOWLEDGE>
, / IREG8T9<-TOKENQ TOKIN
INPUT3.DONEQ: IF <LAST-TOKEN-PART-IN>
, THEN INQ<-1 /->INPUT / ELSE->INPUT4
%...........................................
INPUT4: IF <OK>/<INPUT-SECTION-REQUEST> THEN->INPUT4.DONEQ
INPUT4.DONEQ: <INPUT-REQUEST-ACKNOWLEDGE>
, / IREG10T11<-TOKENQ TOKIN
INPUT4.DONEQ: IF <LAST-TOKEN-PART-IN>
, THEN INQ<-1 /->INPUT / ELSE->INPUT5
%...........................................
INPUT5: IF <OK>/<INPUT-SECTION-REQUEST> THEN->INPUT5.DONEQ
INPUT5.DONEQ: <INPUT-REQUEST-ACKNOWLEDGE>
, / IREG12T13<-TOKENQ TOKIN
INPUT5.DONEQ: IF <LAST-TOKEN-PART-IN>
, THEN INQ<-1 /->INPUT / ELSE->INPUT6
%...........................................
INPUT6: IF <OK>/<INPUT-SECTION-REQUEST> THEN->INPUT6.DONEQ
INPUT6.DONEQ: <INPUT-REQUEST-ACKNOWLEDGE>
, / IREG14T15<-TOKENQ TOKIN
INPUT6.DONEQ: IF <LAST-TOKEN-PART-IN>
, THEN INQ<-1 /->INPUT / ELSE->INPUT7
%...........................................
INPUT7: IF <OK>/<INPUT-SECTION-REQUEST> THEN->INPUT7.DONEQ
INPUT7.DONEQ: <INPUT-REQUEST-ACKNOWLEDGE>
, / IREG16T17<-TOKENQ TOKIN
INPUT7.DONEQ: IF <LAST-TOKEN-PART-IN>
, THEN INQ<-1 /->INPUT / ELSE->INPUT8
%...........................................
INPUT8: IF <OK>/<INPUT-SECTION-REQUEST> THEN->INPUT8.DONEQ
INPUT8.DONEQ: <INPUT-REQUEST-ACKNOWLEDGE>
, / IREG18T19<-TOKENQ TOKIN / INQ<-1  /->INPUT

% OUTPUT SECTION
OUTPUT.: IF <OK> THEN TOKOUT<-TOKOUT OUTPUTQ OREG

% INTERRUPT REQUESTS ARE SERVICED BETWEEN TOKENS
READY: IF <OK>/<INTERRUPT-REQUEST>
, THEN <INTERRUPT-ACKNOWLEDGE>/->INTRUPT
INTRUPT: IF <OK>/<INTERRUPT-REQUEST>
, THEN <INTERRUPT-ACKNOWLEDGE>/->READY

% ACCEPT TOKENS FOR PROCESSING
READY.: IF <OK>/INQ=1/<NO-INTERRUPT-REQUEST>
, THEN->READY.FORMAT  ACCEPT NEXT TOKEN - FIRST CHECK FORMAT
%<GRAB1<-IF CHAIN>
%<GRAB2<-THEN DEST<-IREG2T7 / ADDR<-IREG8T9 / DAT0T1<-IREG10T11>
%<GRABEM<- / INQ<-0 / OPREG<-IREG1[0 1 2 3] /->OPQ>
%<GRAB3<-ELSE ADDR<-IREG2T3 / DATA<-IREG4T19>
%<GRAB-TOKEN-IF-READY<-<GRAB1> <GRAB2> <GRABEM> <GRAB3> <GRABEM>>
READY.FORMAT: <GRAB-TOKEN-IF-READY>


OPQ: IF <OK> THEN->OPQ.OK
% NORMAL OPERATIONS:
OPQ.OK: IF *NORMALOP OPCODE  NORMAL OPERATION DETECTED
```

```
, THEN MAR<-ADDR / <READO>   READ AT INDICATED LOCATION
OPQ.OK: IF OPCODE = <*STORE-T-FIX>          THEN->STORE
OPQ.OK: IF OPCODE = <*STORE-T-VAR>          THEN->STORE
OPQ.OK: IF OPCODE = <*STORE-U-FIX>          THEN->STORE
OPQ.OK: IF OPCODE = <*FETCH-TYPE-STORED>    THEN->FETCH
OPQ.OK: IF OPCODE = <*FETCH-NOTYPE-STORED>  THEN->FETCH
OPQ.OK: IF OPCODE = <*ALLOCATE>             THEN->OPQ.ALLOCATE
OPQ.OK: IF OPCODE = <*CLEAR>                THEN->OPQ.CLEAR
OPQ.OK: IF OPCODE = <*RESET>                THEN->RESET
OPQ.OK: IF OPCODE = <*ALLOCATE-FREE-SPACE>  THEN->OPQ.ALLOCFR
% SERVICE OPERATIONS:
OPQ.OK: IF OPCODE = <*READ-ABSOLUTE>           THEN->OPQ.READAB
OPQ.OK: IF OPCODE = <*WRITE-ABSOLUTE>          THEN->OPQ.WRITEAB
OPQ.OK: IF OPCODE = <*INITIALIZE>              THEN->OPQ.INITIAL
OPQ.OK: IF OPCODE = <*LOAD-ERROR-DESTINATION>  THEN->OPQ.LOADED
%OPQ.OK: IF OPCODE = <*GET-FREE-SIZE>          THEN->OPQ.GETFREE

OPQ.ALLOCATE: COMPARE<-DATOT1 / ABREG<-DATA3[0 1] /->ALLOCATE
OPQ.CLEAR:    COMPARE<-DATOT1 /->CLEAR
OPQ.ALLOCFR:  COMPARE<-DATOT1 /->ALLOCFR
```

B.1 *STORE OPERATIONS

```
% STORE OPERATION
% CHECK THE STATUS OF THE LOCATION AND TAKE APPROPRIATE ACTION
STORE.: IF <OK>/<MEM-FINI> THEN->STORE.ST

STORE.ST: IF STIN = <NOT-ALLOCATED> THEN->STORE.ERROR
STORE.ST: IF STIN = <TYPED-DATA>    THEN->STORE.ERROR
STORE.ST: IF STIN = <UNTYPED-DATA>  THEN->STORE.ERROR
STORE.ST: IF STIN = <MIDDLE>        THEN->STORE.ERROR
STORE.ST: IF STIN = <INVISIBLE>     THEN->STORE.ERROR
STORE.ST: IF STIN = <ERROR>         THEN->STORE.ERROR
STORE.ST: IF OPCODE = <*STORE-T-FIX> THEN->STORE.TF
STORE.ST: IF OPCODE = <*STORE-T-VAR> THEN->STORE.TV
STORE.ST: IF OPCODE = <*STORE-U-FIX> THEN->STORE.UF
%..............................................
STORE.TF: IF STIN = <EMPTY-NOWAIT>    THEN->STORE.ENTF
STORE.TF: IF STIN = <TYPE-DEFERRED>   THEN->STORE.ERROR
STORE.TF: IF STIN = <NOTYPE-DEFERRED> THEN->STORE.DEFTF
%..............................................
STORE.TV: IF STIN = <EMPTY-NOWAIT>    THEN->STORE.ENTV
STORE.TV: IF STIN = <TYPE-DEFERRED>   THEN->STORE.ERROR
STORE.TV: IF STIN = <NOTYPE-DEFERRED> THEN->STORE.DEFTV
%..............................................
STORE.UF: IF STIN = <EMPTY-NOWAIT>    THEN->STORE.ENUF
STORE.UF: IF STIN = <TYPE-DEFERRED>   THEN->STORE.DEFUF
STORE.UF: IF STIN = <NOTYPE-DEFERRED> THEN->STORE.ERROR
%..............................................
STORE.ERROR:  UNEXPECTED STATUS FOUND
,             SET ERROR FLAGS AND PROCESS THE ERROR
, ERRCODE<-OPCODE/->ERROR
```

```
STORE.ENTF:  EMPTY, NOONE WAITING, TYPED DATA, FIXED LENGTH
,    STORE ONE, TWO, THREE, OR FOUR WORDS DEPENDING ON THE
,    TYPE PROVIDED.  STORE TYPE INFORMATION IN THE FIRST WORD
, <WRITEO>/STOUT<-<TYPED-DATA>   WRITE BYTES 0-2 TO MEMORY
,   /MDRDATA<-DAT0T3/->STENTF0   WITH TYPE INFORMATION
%..............................................
STENTF0.: IF <OK>/<MEM-FINI> THEN->STENTF0.CHKSIZ
STENTF0.CHKSIZ: IF LENGTHIN = 0 0 - -
,  THE DATA IS 3 BYTES OR LESS IN LENGTH
, THEN->STENTF0.FINISH  WE ARE FINISHED - PROCEED
, ELSE   WE MUST CONTINUE TO STORE DATA - FIRST CHECK STATUS
,   <READO>/<INC-MAR>/->STENTF1   READ THE NEXT LOCATION (WAIT)
%..............................................
STENTF1.: IF <OK>/<MEM-FINI> THEN->STENTF1.CHKSTAT
STENTF1.CHKSTAT: IF STIN ≠ <MIDDLE>     CHECK STATUS OF 2ND WORD
, THEN ERRCODE<-<DATA-TOO-LARGE>/->ERROR   SIGNAL AN ERROR
, ELSE <WRITEO>/STOUT<-<MIDDLE>  WRITE BYTES 3-6 TO MEMORY
,   /MDRDATA<-DAT4T7/->STENTF2     WAIT FOR MEMORY TO RESPOND
%..............................................
STENTF2.: IF <OK>/<MEM-FINI> THEN->STENTF2.CHKSIZ
STENTF2.CHKSIZ: IF LENGTHIN = 1 - - -
,  THE DATA IS > 7 BYTES OR LESS IN LENGTH
, THEN->STENTF2.FINISH  WE ARE FINISHED - PROCEED
, ELSE   WE MUST CONTINUE TO STORE DATA - FIRST CHECK STATUS
,   <READO>/<INC-MAR>/->STENTF3   READ THE NEXT LOCATION (WAIT)
%..............................................
STENTF3.: IF <OK>/<MEM-FINI> THEN->STENTF3.CHKSTAT
STENTF3.CHKSTAT: IF STIN ≠ <MIDDLE>     CHECK STATUS OF 3ND WORD
, THEN ERRCODE<-<DATA-TOO-LARGE>/->ERROR   SIGNAL AN ERROR
, ELSE <WRITEO>/STOUT<-<MIDDLE>  WRITE BYTES 7-10 TO MEMORY
,   /MDRDATA<-DAT8T11/->STENTF4    WAIT FOR MEMORY TO RESPOND
%..............................................
STENTF4.: IF <OK>/<MEM-FINI> THEN->STENTF4.CHKSIZ
STENTF4.CHKSIZ: IF LENGTHIN = 1 1 - -
,  THE DATA IS > 11 BYTES OR LESS IN LENGTH
, THEN->STENTF4.FINISH  WE ARE FINISHED - PROCEED
, ELSE   WE MUST CONTINUE TO STORE DATA - FIRST CHECK STATUS
,   <READO>/<INC-MAR>/->STENTF5   READ THE NEXT LOCATION (WAIT)
%..............................................
STENTF5.: IF <OK>/<MEM-FINI> THEN->STENTF5.CHKSTAT
STENTF5.CHKSTAT: IF STIN ≠ <MIDDLE>     CHECK STATUS OF 3ND WORD
, THEN ERRCODE<-<DATA-TOO-LARGE>/->ERROR   SIGNAL AN ERROR
, ELSE <WRITEO>/STOUT<-<MIDDLE>  WRITE BYTES 11-14 TO MEMORY
,   /MDRDATA<-DAT12T15/->STENTF6  WAIT FOR MEMORY TO RESPOND
%..............................................
STENTF6.: IF <OK>/<MEM-FINI> THEN->STENTF6.FINISH
%..............................................
% FINISH STORE EMPTY NO-WAITING T FIX
STENTF0.FINISH: IF INQ = 1
, THEN->STENTF0.FORMAT  ELSE->READY
STENTF0.FORMAT: <GRAB-TOKEN-IF-READY>
STENTF2.FINISH: IF INQ = 1
, THEN->STENTF2.FORMAT  ELSE->READY
STENTF2.FORMAT: <GRAB-TOKEN-IF-READY>
STENTF4.FINISH: IF INQ = 1
, THEN->STENTF4.FORMAT  ELSE->READY
```

```
STENTF4.FORMAT: <GRAB-TOKEN-IF-READY>
STENTF6.FINISH: IF INQ = 1
, THEN->STENTF6.FORMAT  ELSE->READY
STENTF6.FORMAT: <GRAB-TOKEN-IF-READY>

STORE.ENTV:   EMPTY, NOONE WAITING, TYPED DATA, VARIABLE LENGTH
,    STORE ONE WORD OF DATA MEMORY USING THE FREE LIST
,    IF NECESSARY TO STORE UP TO A TOTAL OF 16 BYTES
,    STORE TYPE INFORMATION IN THE DATA MEMORY WORD
, IF LENGTHIN = 0 0 - -   LENGTH ≤ 3
, THEN <WRITEO>/STOUT<-<TYPED-DATA>   WRITE BYTES 0-3 TO MEMORY
,   /MDRDATA<-DAT0T3/->STENTV0         WITH TYPE INFORMATION
, ELSE            STORE THE DATA USING SPACE FROM
,  ->STORE.FREE   THE FREE LIST - IF SPACE IS AVAILABLE
%...............................................
STENTV0.: IF <OK>/<MEM-FINI> THEN->STENTV0.FINISH
%...............................................
STORE.FREE: IF <IS-LIST-EMPTY>   THERE IS NO ROOM ON THE FREE LIST
, THEN ERRCODE<-<FREE-LIST-OVERFLOW> /->ERROR
, ELSE   WRITE THE TYPE, ONE BYTE OF DATA,
,    AND THE POINTER TO THE REST OF THE DATA
,   <WRITEO>/STOUT<-<INVISIBLE>     WRITE TYPE INFO, 1 BYTE OF DATA,
,   /MDRD0T1<-DAT0T1/MDRD2T3<-RTOP   AND A POINTER TO MORE DATA
,   /->STENTV1                      WAIT FOR MEMORY TO RESPOND
%...............................................
STENTV1.: IF <OK>/<MEM-FINI> THEN->STENTV1.STORE
STENTV1.STORE:               STORE BYTES 2-5 IN FREE LIST
, <WRITEO>/MAR<-RTOP            WRITE FOUR BYTES TO MEMORY
, /MDRDATA<-DAT2T5/->STENTV2    WAIT FOR MEMORY TO RESPOND
%...............................................
STENTV2.: IF <OK>/<MEM-FINI> THEN->STENTV2.READ
STENTV2.READ:       SAVE THE NEW TOP OF THE FREE-LIST
, <INC-MAR>/<READO>   READ FROM SECOND HALF OF FREE CELL
, /->STENTV3           WAIT FOR MEMORY TO RESPOND
%...............................................
STENTV3.: IF <OK>/<MEM-FINI> THEN->STENTV3.READ
STENTV3.READ:   STORE THE NEW TOP OF THE FREE-LIST
, RTOP <-MDRD2T3   IN RTOP; DECIDE IF MORE WRITING
, /->STENTV3.MORE   IS NECESSARY (STORE DATA)
%...............................................
STENTV3.MORE: IF *L6T15 LENGTHIN   CHECK SIZE OF DATA
, THEN->STENTV3.WRITE   WRITE MORE DATA TO MEMORY AND WAIT
, ELSE->STENTV3.FINISH
%...............................................
STENTV3.WRITE: IF *L6T9 LENGTHIN   FIT IN ONE MORE WORD?
, THEN MDRDATA<-DAT5T8              WRITE FOUR BYTES TO MEMORY
,   /<WRITEO>/->STENTV99            WAIT FOR MEMORY TO RESPOND
,   /->STENTV3.LASTQ         CHECK IF THIS WAS THE LAST FREE CELL
, ELSE   CHECK IF THERE IS ANOTHER FREE CELL
,    IF SO, STORE TWO MORE BYTES OF DATA AND A POINTER
,->STENTV3.SPACEQ
STENTV3.LASTQ:  IF STIN = <LASTI> THEN <LIST-IS-EMPTY>
STENTV3.SPACEQ: IF STIN = <LASTI>
, THEN->STENTV3.ERROR ELSE->STENTV3.SPACE
STENTV3.ERROR: MAR<-ADDR / ERRCODE<-<FREE-LIST-OVERFLOW> /->ERROR
STENTV3.SPACE: WRITE TWO BYTES AND A POINTER TO MEMORY
, MDRD0T1<-DAT6T7       THE POINTER IS ALREADY IN PLACE
, /<WRITEO>/->STENTV4   WAIT FOR MEMORY TO RESPOND
% THE ABOVE STATEMENT COULD INCLUDE A TRANSFER OF THE
% TOP POINTER.  WHAT WE REALLY WANT IS AN OUTPUT
% DON'T CARE ON THE SYSTEM GENERATED CONTROL SIGNAL.
%...................................................
STENTV4.:                 WRITE FOUR MORE BYTES OF DATA
,   <WRITEO>/MAR<-RTOP    WRITE FOUR BYTES INTO THE
,   /MDRDATA<-DAT8T11     TOP HALF OF A DATA CELL
,   /->STENTV5            WAIT FOR MEMORY TO RESPOND
%...................................................
STENTV5.: IF <OK>/<MEM-FINI> THEN->STENTV5.READ
STENTV5.READ:         SAVE THE NEW TOP OF THE FREE-LIST
, <INC-MAR>/<READO>   READ FROM SECOND HALF OF FREE CELL
, /->STENTV6           WAIT FOR MEMORY TO RESPOND
%...................................................
STENTV6.: IF <OK>/<MEM-FINI> THEN->STENTV6.READ
STENTV6.READ:   STORE THE NEW TOP OF THE FREE-LIST
, RTOP <-MDRD2T3   IN RTOP; DECIDE IF MORE WRITING
, /->STENTV6.MORE   IS NECESSARY (STORE DATA)
, /->STENTV6.LASTQ     CHECK IF THIS WAS THE LAST FREE CELL
STENTV6.LASTQ:  IF STIN = <LASTI> THEN <LIST-IS-EMPTY>
%...................................................
STENTV6.MORE: IF *L10T11 LENGTHIN   CHECK SIZE OF DATA
, THEN->STENTV6.WRITE                WRITE FOUR BYTES TO MEMORY
, ELSE->STENTV6.FINISH               WAIT FOR MEMORY TO RESPOND
%...................................................
STENTV6.WRITE:         STORE BYTES 12-15 IN FREE LIST
, MDRDATA<-DAT12T15      WRITE FOUR BYTES TO MEMORY
, /<WRITEO>/->STENTV99   WAIT FOR MEMORY TO RESPOND
%...................................................
STENTV99.: IF <OK>/<MEM-FINI> THEN->STENTV99.FINISH
%...................................................
STENTVO.FINISH: IF INQ = 1
, THEN->STENTVO.FORMAT  ELSE->READY
STENTVO.FORMAT: <GRAB-TOKEN-IF-READY>
STENTV3.FINISH: IF INQ = 1
, THEN->STENTV3.FORMAT  ELSE->READY
STENTV3.FORMAT: <GRAB-TOKEN-IF-READY>
STENTV6.FINISH: IF INQ = 1
, THEN->STENTV6.FORMAT  ELSE->READY
STENTV6.FORMAT: <GRAB-TOKEN-IF-READY>
STENTV99.FINISH: IF INQ = 1
, THEN->STENTV99.FORMAT  ELSE->READY
STENTV99.FORMAT: <GRAB-TOKEN-IF-READY>

STORE.ENUF:  EMPTY, NOONE WAITING, UNTYPED DATA, FIXED LENGTH
,    STORE ONE, TWO, THREE, OF FOUR WORDS OF DATA
,    STORE EMPTY NO-WAITING U FIX
, STOUT<-<UNTYPED-DATA>       WRITE BYTES 1-4 TO MEMORY
, /MDRDATA<-DAT1T4/<WRITEO>   (BYTE 0 IS TYPE INFORMATION)
, /->STENUFO                  WAIT FOR MEMORY TO RESPOND
%...................................................
STENUFO.: IF <OK>/<MEM-FINI> THEN->STENUFO.MORE
STENUFO.MORE: IF *LOT4 LENGTHIN   LENGTH ≤ 4
, THEN->STENUFO.FINISH   WE ARE DONE; FINISH
, ELSE  CHECK IF THE NEXT CELL IS A MIDDLE CELL
,    READ NEXT WORD TO DETERMINE STATUS
,   <INC-MAR>/<READO>/->STENTF1
%............................................
STENUF1.: IF <OK>/<MEM-FINI> THEN->STENUF1.MIDDLEQ
STENUF1.MIDDLEQ: IF STIN = <MIDDLE>
, THEN ERRCODE<-<DATA-TOO-LARGE> / MAR<-ADDR /->ERROR
, ELSE <WRITEO>/MDRDATA<-DAT5T8   WRITE BYTES 5-8 TO MEMORY
,   /->STENTF2                    WITH TYPE INFORMATION
%............................................
STENUF2.: IF <OK>/<MEM-FINI> THEN->STENUF2.MORE
STENUF2.MORE: IF *L5T8 LENGTHIN   5 ≤ LENGTH ≤ 8
, THEN->STENUF2.FINISH   WE ARE DONE; FINISH
, ELSE  CHECK IF THE NEXT CELL IS A MIDDLE CELL
,    READ NEXT WORD TO DETERMINE STATUS
,   <INC-MAR>/<READO>/->STENTF3
%............................................
STENUF3.: IF <OK>/<MEM-FINI> THEN->STENUF3.MIDDLEQ
STENUF3.MIDDLEQ: IF STIN = <MIDDLE>
, THEN ERRCODE<-<DATA-TOO-LARGE> / MAR<-ADDR /->ERROR
, ELSE <WRITEO>/STOUT<-<MIDDLE>   WRITE BYTES 9-12 TO MEMORY
,   /MDRDATA<-DAT9T12/->STENTF4    WITH TYPE INFORMATION
%............................................
STENUF4.: IF <OK>/<MEM-FINI> THEN->STENUF4.MORE
STENUF4.MORE: IF *L9T12 LENGTHIN   9 ≤ LENGTH ≤ 12
, THEN->STENUF4.FINISH   WE ARE DONE; FINISH
, ELSE  CHECK IF THE NEXT CELL IS A MIDDLE CELL
,    READ NEXT WORD TO DETERMINE STATUS
,   <INC-MAR>/<READO>/->STENTF5
%............................................
STENUF5.: IF <OK>/<MEM-FINI> THEN->STENUF5.MIDDLEQ
STENUF5.MIDDLEQ: IF STIN = <MIDDLE>
, THEN ERRCODE<-<DATA-TOO-LARGE> / MAR<-ADDR /->ERROR
, ELSE <WRITEO>/MDRD0T2<-DAT13T15   WRITE BYTES 13-15 TO MEMORY
,   /->STENTF6
%............................................
STENUF6.: IF <OK>/<MEM-FINI> THEN->STENUF6.FINISH
%............................................
STENUF0.FINISH: IF INQ = 1
, THEN->STENUF0.FORMAT  ELSE->READY
STENUF0.FORMAT: <GRAB-TOKEN-IF-READY>
STENUF2.FINISH: IF INQ = 1
, THEN->STENUF2.FORMAT  ELSE->READY
STENUF2.FORMAT: <GRAB-TOKEN-IF-READY>
STENUF4.FINISH: IF INQ = 1
, THEN->STENUF4.FORMAT  ELSE->READY
STENUF4.FORMAT: <GRAB-TOKEN-IF-READY>
STENUF6.FINISH: IF INQ = 1
, THEN->STENUF6.FORMAT  ELSE->READY
STENUF6.FORMAT: <GRAB-TOKEN-IF-READY>

% STORE WITH DEFERRED DESTINATIONS
,  SEND A TOKEN TO EACH DEFERRED DESTINATION
,  RETURN LINKS TO THE FREE LIST
,  STORE THE DATA
STORE.DEFTF:  TYPE <-DATAO    /->STORE.DEFER
STORE.DEFTV:  TYPE <-DATAO    /->STORE.DEFER
STORE.DEFUF:  TYPE <-MDRDATAO /->STORE.DEFER
STORE.DEFER: <READO>/MAR<-MDRD2T3  GET THE FIRST DESTINATION
```

```
, / EXTRA<-MDRD2T3 /->STDEF0       SAVE A POINTER TO THE FIRST DEST
%.............................................
STDEF0.: IF <OK>/<MEM-FINI> THEN->STDEF0.OK
STDEF0.OK: IF <IS-OUTPUT-SECTION-FREE>
, THEN OREG0T3<-MDRDATA   TRANSFER DEST<0..3> AND DATA
,   / OREG6<-TYPE / OREG7T21 <-DAT1T15
,   / <READ0> / <INC-MAR>   GET THE REST OF THE FIRST DESTINATION
,   /->STDEF1
%.............................................
STDEF1.: IF <OK>/<MEM-FINI> THEN->STDEF1.OK
STDEF1.OK: OREG4T5<-MDRD0T1  HAND OVER THE REST OF THE DESTINATION
, / <OUTPUT-SECTION-REQUEST>    SEND THIS TOKEN OFF
, /->STDEF1.MOREQ               CHECK FOR MORE DESTINATIONS
%.............................................
STDEF1.MOREQ: IF STIN = <NOT-LASTI>
, THEN->STDEF1.NEXT   READ THE NEXT DESTINATION
, ELSE->STDEF1.CONS   THERE ARE NO MORE DESTINATIONS
,    REPLACE THE FREE CELLS, AND STORE THE DATA
%.............................................
STDEF1.NEXT: <READ0>/MAR<-MDRD2T3   GET THE NEXT DESTINATION
, /->STDEF3
%.............................................
STDEF3.: IF <OK>/<MEM-FINI> THEN->STDEF3.OK
STDEF3.OK: IF <IS-OUTPUT-SECTION-FREE>
, THEN OREG0T3<-MDRDATA   TRANSFER DEST<0..3> AND DATA
,   / <READ0> / <INC-MAR>   GET THE REST OF THE FIRST DESTINATION
,   /->STDEF1
%.............................................
STDEF1.CONS:   FIRST APPEND THE FREE LIST ONTO THIS CELL
,   THEN CHANGE THE HEAD POINTER
,  <WRITE0> / MDRD2T3<-RTOP /->STDEF2 /->STDEF1.STAT
STDEF1.STAT: IF <IS-LIST-EMPTY>
,  THEN <CELLS-DO-REMAIN>
,  ELSE STOUT<-<NOT-LASTO>
%.............................................
STDEF2.: IF <OK>/<MEM-FINI> THEN->STDEF2.OK
STDEF2.OK: RTOP<-EXTRA /->STDEF2.OPQ   NOW WRITE THE DATA
STDEF2.OPQ:  IF OPCODE = <*STORE-T-FIX> THEN->STDEF2.ENTF
STDEF2.OPQ:  IF OPCODE = <*STORE-T-VAR> THEN->STDEF2.ENTV
STDEF2.OPQ:  IF OPCODE = <*STORE-U-FIX> THEN->STDEF2.ENUF
%.............................................
STDEF2.ENTF:   EMPTY, NOONE WAITING, TYPED DATA, FIXED LENGTH
,    STORE ONE, TWO, THREE, OR FOUR WORDS DEPENDING ON THE
,    TYPE PROVIDED.  STORE TYPE INFORMATION IN THE FIRST WORD
, <WRITE0>/STOUT<-<TYPED-DATA>   WRITE BYTES 0-2 TO MEMORY
,   /MDRDATA<-DAT0T3/->STENTF0    WITH TYPE INFORMATION
,   /MAR<-ADDR
%.............................................
STDEF2.ENTV:   EMPTY, NOONE WAITING, TYPED DATA, VARIABLE LENGTH
,    STORE ONE WORD OF DATA MEMORY USING THE FREE LIST
,    IF NECESSARY TO STORE UP TO A TOTAL OF 16 BYTES
,    STORE TYPE INFORMATION IN THE DATA MEMORY WORD
, IF LENGTHIN = 0 0 - -   LENGTH ≤ 3
, THEN <WRITE0>/STOUT<-<TYPED-DATA>   WRITE BYTES 0-3 TO MEMORY
,   /MDRDATA<-DAT0T3/->STENTV0        WITH TYPE INFORMATION
, ELSE           STORE THE DATA USING SPACE FROM
,  ->STDEF2.FREE   THE FREE LIST - IF SPACE IS AVAILABLE
```

```
%...................................................
STDEF2.FREE: IF <IS-LIST-EMPTY>  THERE IS NO ROOM ON THE FREE LIST
, THEN ERRCODE<-<FREE-LIST-OVERFLOW> /->ERROR
, ELSE  WRITE THE TYPE, ONE BYTE OF DATA,
,    AND THE POINTER TO THE REST OF THE DATA
,   <WRITEO>/STOUT<-<INVISIBLE>     WRITE TYPE INFO, 1 BYTE OF DATA,
,   /MDRD0T1<-DAT0T1/MDRD2T3<-RTOP   AND A POINTER TO MORE DATA
,   /->STENTV1                      WAIT FOR MEMORY TO RESPOND
%...................................................
STDEF2.ENUF:  EMPTY, NOONE WAITING, UNTYPED DATA, FIXED LENGTH
,    STORE ONE, TWO, THREE, OF FOUR WORDS OF DATA
,    STORE EMPTY NO-WAITING U FIX
, STOUT<-<UNTYPED-DATA>        WRITE BYTES 1-4 TO MEMORY
, /MDRDATA<-DAT1T4/<WRITEO>    (BYTE 0 IS TYPE INFORMATION)
, /->STENUFO                   WAIT FOR MEMORY TO RESPOND
```

B.2 *FETCH OPERATIONS

```
%****************************************************
% FETCH OPERATION
% CHECK THE STATUS OF THE LOCATION AND TAKE APPROPRIATE ACTION
FETCH.: IF <OK>/<MEM-FINI> THEN->FETCH.ST
%...................................................
FETCH.ST: IF STIN = <NOT-ALLOCATED> THEN->FETCH.ERROR
FETCH.ST: IF STIN = <MIDDLE>        THEN->FETCH.ERROR
FETCH.ST: IF OPCODE = <*FETCH-TYPE-STORED>   THEN->FETCH.TS
FETCH.ST: IF OPCODE = <*FETCH-NOTYPE-STORED> THEN->FETCH.NTS
%...................................................
FETCH.TS: IF STIN = <EMPTY-NOWAIT>    THEN->FETCH.ENTS
FETCH.TS: IF STIN = <TYPED-DATA>      THEN->FETCH.TYPED
FETCH.TS: IF STIN = <UNTYPED-DATA>    THEN->FETCH.ERROR
FETCH.TS: IF STIN = <INVISIBLE>       THEN->FETCH.INVIS
FETCH.TS: IF STIN = <TYPE-DEFERRED>   THEN->FETCH.TDEF
FETCH.TS: IF STIN = <NOTYPE-DEFERRED> THEN->FETCH.ERROR
%...................................................
FETCH.NTS: IF STIN = <EMPTY-NOWAIT>    THEN->FETCH.ENNTS
FETCH.NTS: IF STIN = <TYPED-DATA>      THEN->FETCH.ERROR
FETCH.NTS: IF STIN = <UNTYPED-DATA>    THEN->FETCH.UNTYPED
FETCH.NTS: IF STIN = <INVISIBLE>       THEN->FETCH.ERROR
FETCH.NTS: IF STIN = <TYPE-DEFERRED>   THEN->FETCH.ERROR
FETCH.NTS: IF STIN = <NOTYPE-DEFERRED> THEN->FETCH.NTDEF
%...................................................
FETCH.ERROR:  UNEXPECTED STATUS FOUND
,             SET ERROR FLAGS AND PROCESS THE ERROR
, ERRCODE<-OPCODE/->ERROR
%...................................................
FETCH.ENTS: IF <IS-LIST-EMPTY>
, THEN ERRCODE<-<FREE-LIST-OVERFLOW> /->ERROR
, ELSE MDRD2T3<-RTOP      SAVE POINTER TO NEW DESTINATION
,   / <WRITEO> /->FENO
FETCH.ENNTS: MDRDATA1<-DATA0 /->FETCH.ENTS
%...................................................
FENO.: IF <OK>/<MEM-FINI> THEN->FENO.OK
FENO.OK: MAR<-RTOP / MDRDATA<-DESTOT3   STORE THE DESTINATION
, / <WRITEO> /->FEN1                   FIND NEW RTOP
%...............................................
FEN1.: IF <OK>/<MEM-FINI> THEN->FEN1.OK
FEN1.OK: <INC-MAR> / <READO> /->FEN2   GET NEW RTOP
%...............................................
FEN2.: IF <OK>/<MEM-FINI> THEN->FEN2.OK
FEN2.OK: RTOP<-MDRD2T3 / <WRITEO>   STORE REST OF DEST
, / MDRDOT1<-DEST4T5 /->FINISH
FEN2.OK: IF STIN = <LASTI> THEN <LIST-IS-EMPTY>
%...............................................
FETCH.TYPED: IF *LOT3 LENGTH   DATA ALL IN FIRST WORD
, THEN->FETCH.SHIP   SEND THE DATA
, ELSE DATOT3<-MDRDATA / <INC-MAR>   GET MORE DATA
,   / <READO> /->FTYO
%...............................................
FTYO.: IF <OK>/<MEM-FINI> THEN->FTYO.OK
FTYO.OK: IF STIN ≠ <MIDDLE>
, THEN ERRCODE<-<LENGTH-ERROR> /->ERROR
, ELSE DAT4T7<-MDRDATA /->FTYO.LENGTH
FTYO.LENGTH: IF LENGTHIN = 0 1 - -   THE DATA IS IN TWO WORDS
, THEN->FTYO.SHIP   SEND THE DATA
, ELSE <INC-MAR> / <READO> /->FTY1   GET MORE DATA
%...............................................
FTY1.: IF <OK>/<MEM-FINI> THEN->FTY1.OK
FTY1.OK: IF STIN ≠ <MIDDLE>
, THEN ERRCODE<-<LENGTH-ERROR> /->ERROR
, ELSE DAT8T11<-MDRDATA /->FTY1.LENGTH
FTY1.LENGTH: IF LENGTHIN = 1 0 - -   THE DATA IS IN THREE WORDS
, THEN->FTY1.SHIP   SEND THE DATA
, ELSE <INC-MAR> / <READO> /->FTY2   GET MORE DATA
%...............................................
FTY2.: IF <OK>/<MEM-FINI> THEN->FTY2.OK
FTY2.OK: IF STIN ≠ <MIDDLE>
, THEN ERRCODE<-<LENGTH-ERROR> /->ERROR
, ELSE DAT12T15<-MDRDATA /->FTY2.SHIP
%...............................................
FETCH.SHIP: IF <IS-OUTPUT-SECTION-FREE>
, THEN OREGOT5<-DEST / OREG6T21<-DATA   TRANSFER DESTINATION
,   / <OUTPUT-SECTION-REQUEST>         TRANSFER DATA
,   /->FETCH.FINISH                    TRY AND GRAB NEXT TOKEN
FETCH.FINISH: IF INQ = 1
, THEN->FETCH.FORMAT  ELSE->READY
FETCH.FORMAT: <GRAB-TOKEN-IF-READY>
%...............................................
FTYO.SHIP: IF <IS-OUTPUT-SECTION-FREE>
, THEN OREGOT5<-DEST / OREG6T21<-DATA   TRANSFER DESTINATION
,   / <OUTPUT-SECTION-REQUEST>         TRANSFER DATA
,   /->FTYO.FINISH                     TRY AND GRAB NEXT TOKEN
FTYO.FINISH: IF INQ = 1
, THEN->FTYO.FORMAT  ELSE->READY
FTYO.FORMAT: <GRAB-TOKEN-IF-READY>
%...............................................
FTY1.SHIP: IF <IS-OUTPUT-SECTION-FREE>
, THEN OREGOT5<-DEST / OREG6T21<-DATA   TRANSFER DESTINATION
,   / <OUTPUT-SECTION-REQUEST>         TRANSFER DATA
,   /->FTY1.FINISH                     TRY AND GRAB NEXT TOKEN
```

```
FTY1.FINISH: IF INQ = 1
, THEN->FTY1.FORMAT  ELSE->READY
FTY1.FORMAT: <GRAB-TOKEN-IF-READY>
%...............................................
FTY2.SHIP: IF <IS-OUTPUT-SECTION-FREE>
, THEN OREG0T5<-DEST / OREG6T21<-DATA   TRANSFER DESTINATION
,   / <OUTPUT-SECTION-REQUEST>         TRANSFER DATA
,   /->FTY2.FINISH                     TRY AND GRAB NEXT TOKEN
FTY2.FINISH: IF INQ = 1
, THEN->FTY2.FORMAT  ELSE->READY
FTY2.FORMAT: <GRAB-TOKEN-IF-READY>
%...............................................
FETCH.UNTYPED: IF *L0T4 LENGTHIN   DATA IS ALL IN FIRST WORD
, THEN->FETCH.SHIP  SEND THE DATA
, ELSE DATA0 <-TYPE / DAT1T4<-MDRDATA   GET MORE DATA
,   / <INC-MAR> / <READ0> /->FUNTY0
%...............................................
FUNTY0.: IF <OK>/<MEM-FINI> THEN->FUNTY0.OK
FUNTY0.OK: IF STIN ≠ <MIDDLE>
, THEN ERRCODE<-<LENGTH-ERROR> /->ERROR
, ELSE DAT5T8<-MDRDATA /->FUNTY0.LENGTH
FUNTY0.LENGTH: IF *L5T8 LENGTHIN   THE DATA IS IN TWO WORDS
, THEN->FUNTY0.SHIP  SEND THE DATA
, ELSE <INC-MAR> / <READ0> /->FUNTY1  GET MORE DATA
%...............................................
FUNTY1.: IF <OK>/<MEM-FINI> THEN->FUNTY1.OK
FUNTY1.OK: IF STIN ≠ <MIDDLE>
, THEN ERRCODE<-<LENGTH-ERROR> /->ERROR
, ELSE DAT9T12<-MDRDATA /->FUNTY1.LENGTH
FUNTY1.LENGTH: IF *L9T12 LENGTHIN  THE DATA IS IN THREE WORDS
, THEN->FUNTY1.SHIP  SEND THE DATA
, ELSE <INC-MAR> / <READ0> /->FUNTY2  GET MORE DATA
%...............................................
FUNTY2.: IF <OK>/<MEM-FINI> THEN->FUNTY2.OK
FUNTY2.OK: IF STIN ≠ <MIDDLE>
, THEN ERRCODE<-<LENGTH-ERROR> /->ERROR
, ELSE DAT13T15<-MDRD0T2 /->FUNTY2.SHIP
%...............................................
FUNTY0.SHIP: IF <IS-OUTPUT-SECTION-FREE>
, THEN OREG0T5<-DEST / OREG6T21<-DATA   TRANSFER DESTINATION
,   / <OUTPUT-SECTION-REQUEST>         TRANSFER DATA
,   /->FUNTY0.FINISH                   TRY AND GRAB NEXT TOKEN
FUNTY0.FINISH: IF INQ = 1
, THEN->FUNTY0.FORMAT  ELSE->READY
FUNTY0.FORMAT: <GRAB-TOKEN-IF-READY>
%...............................................
FUNTY1.SHIP: IF <IS-OUTPUT-SECTION-FREE>
, THEN OREG0T5<-DEST / OREG6T21<-DATA   TRANSFER DESTINATION
,   / <OUTPUT-SECTION-REQUEST>         TRANSFER DATA
,   /->FUNTY1.FINISH                   TRY AND GRAB NEXT TOKEN
FUNTY1.FINISH: IF INQ = 1
, THEN->FUNTY1.FORMAT  ELSE->READY
FUNTY1.FORMAT: <GRAB-TOKEN-IF-READY>
%...............................................
FUNTY2.SHIP: IF <IS-OUTPUT-SECTION-FREE>
, THEN OREG0T5<-DEST / OREG6T21<-DATA   TRANSFER DESTINATION
,   / <OUTPUT-SECTION-REQUEST>         TRANSFER DATA
,   /->FUNTY2.FINISH                  TRY AND GRAB NEXT TOKEN
FUNTY2.FINISH: IF INQ = 1
, THEN->FUNTY2.FORMAT  ELSE->READY
FUNTY2.FORMAT: <GRAB-TOKEN-IF-READY>
%..............................................
FETCH.TDEF: IF <IS-LIST-EMPTY>
, THEN ERRCODE<-<FREE-LIST-OVERFLOW> /->ERROR
, ELSE EXTRA<-MDRD2T3      SAVE POINTER TO OLD DESTINATION LIST
,   / <WRITEO> /->FDEFO   WRITE POINTER TO NEW DESTINATION
,   | MDRD2T3<-RTOP
FETCH.NTDEF: IF TYPEQI   TYPE BEING DEFERRED DOES NOT MATCH
, THEN ERRCODE<-<DEFERRED-TYPE-COLLISION> /->ERROR
, ELSE->FETCH.TDEF
%..............................................
FDEFO.: IF <OK>/<MEM-FINI> THEN->FDEFO.OK
FDEFO.OK: MAR<-RTOP / MDRDATA<-DESTOT3   STORE THE DESTINATION
, / <WRITEO> /->FDEF1                     FIND NEW RTOP
%..............................................
FDEF1.: IF <OK>/<MEM-FINI> THEN->FDEF1.OK
FDEF1.OK: <INC-MAR> / <READO> /->FDEF2   GET NEW RTOP
%..............................................
FDEF2.: IF <OK>/<MEM-FINI> THEN->FDEF2.OK
FDEF2.OK: RTOP<-MDRD2T3 / <WRITEO>   STORE REST OF DEST
, / MDRDOT1<-DEST4T5 /->FINISH
, | MDRD2T3<-EXTRA                    ALSO POINT TO OLD DEST LIST
FDEF2.OK: IF STIN = <LASTI> THEN <LIST-IS-EMPTY>
%..............................................
FETCH.INVIS:  FOLLOW THE POINTER AND GET THE DATA
, DATOT1<-MDRDOT1 / MAR<-MDRD2T3  SAVE TYPE AND ONE BYTE OF DATA
,   / <READO> / EXTRA<-MDRD2T3 /->FINVO  SAVE PTR TO CELL
%..............................................
FINVO.: IF <OK>/<MEM-FINI> THEN->FINVO.OK
FINVO.OK: DAT2T5<-MDRDATA  SAVE 4 BYTES OF DATA
, /->FINVO.CHECK           CHECK FOR MORE DATA
FINVO.CHECK: IF *L2T5 LENGTHIN  WE NOW HAVE ALL THE DATA
, THEN->FINVO.FINISH  RETURN CELL AND SHIP DATA
, ELSE <INC-MAR> / <READO> /->FINV1
%..............................................
FINV1.: IF <OK>/<MEM-FINI> THEN->FINV1.OK
FINV1.OK: DAT6T7<-MDRDOT1  SAVE 2 BYTES OF DATA
, /->FINV1.CHECK           CHECK FOR MORE DATA
FINV1.CHECK: IF *L6T9 LENGTHIN  WE NOW HAVE ALL THE DATA
, THEN DAT7T8<-MDRD2T3 /->FINV1.FINISH  RETURN CELL AND SHIP DATA
, ELSE MAR<-MDRD2T3 / <READO> /->FINV2  READ MORE DATA
%..............................................
FINV2.: IF <OK>/<MEM-FINI> THEN->FINV2.OK
FINV2.OK: DAT8T11<-MDRDATA  SAVE 4 BYTES OF DATA
, /->FINV2.CHECK            CHECK FOR MORE DATA
FINV2.CHECK: IF *L8T11 LENGTHIN  WE NOW HAVE ALL THE DATA
, THEN->FINV2.FINISH  RETURN CELLS AND SHIP DATA
, ELSE <INC-MAR> / <READO> /->FINV3
%..............................................
FINV3.: IF <OK>/<MEM-FINI> THEN->FINV3.OK
FINV3.OK: DAT12T15<-MDRDATA  SAVE 4 BYTES OF DATA
, /->FINV3.FINISH  RETURN CELLS AND SHIP DATA
%..............................................
FINVO.FINISH:  RETURN A CELL TO FREE LIST AND SEND DATA
```

```
,  / <INC-MAR> / <WRITEO> /->FINVEND
,  | MDRD2T3<-RTOP
FINVO.FINISH: IF <DO-CELLS-REMAIN>
, THEN STOUT<-<NOT-LASTO>
, ELSE STOUT<-<LASTO> / <CELLS-DO-REMAIN>
%...............................................
FINV1.FINISH:  RETURN A CELL TO FREE LIST AND SEND DATA
,  / <WRITEO> /->FINVEND
,  | MDRD2T3<-RTOP
FINV1.FINISH: IF <DO-CELLS-REMAIN>
, THEN STOUT<-<NOT-LASTO>
, ELSE STOUT<-<LASTO> / <CELLS-DO-REMAIN>
%...............................................
FINV2.FINISH:  RETURN CELLS TO FREE LIST AND SEND DATA
,  / <INC-MAR> / <WRITEO> /->FINVEND
,  | MDRD2T3<-RTOP
FINV2.FINISH: IF <DO-CELLS-REMAIN>
, THEN STOUT<-<NOT-LASTO>
, ELSE STOUT<-<LASTO> / <CELLS-DO-REMAIN>
%...............................................
FINV3.FINISH:  RETURN A CELL TO FREE LIST AND SEND DATA
,  / <WRITEO> /->FINVEND
,  | MDRD2T3<-RTOP
FINV3.FINISH: IF <DO-CELLS-REMAIN>
, THEN STOUT<-<NOT-LASTO>
, ELSE STOUT<-<LASTO> / <CELLS-DO-REMAIN>
%...............................................
FINVEND.: IF <OK>/<MEM-FINI> THEN->FINVEND.SHIP
FINVEND.SHIP: RTOP<-EXTRA
FINVEND.SHIP: IF <IS-OUTPUT-SECTION-FREE>
, THEN OREGOT5<-DEST / OREG6T21<-DATA  TRANSFER DESTINATION
,   / <OUTPUT-SECTION-REQUEST>        TRANSFER DATA
,   /->FINVEND.FINISH                 TRY AND GRAB NEXT TOKEN
FINVEND.FINISH: IF INQ = 1
, THEN->FINVEND.FORMAT  ELSE->READY
FINVEND.FORMAT: <GRAB-TOKEN-IF-READY>
```

B.3 *INITIALIZE OPERATION

```
%*************************************************
OPQ.INITIAL:  INITIALIZE THE MEMORY
, MAR<-CLEARREG / STOUT<-<NOT-ALLOCATED> / ERRDEST<-DATOT5
,   / <WRITEO> / <CHECK-FREE>  /->INITTOP
%...............................................
INITTOP.: IF <OK>/<MEM-FINI> THEN->INITTOP.FREEQ
INITTOP.FREEQ: IF <MAR = FBR>  WE HAVE HIT THE I-STORE BOUNDARY
, THEN->INITTOP.FREE              SET UP THE FREE LIST
, ELSE <INC-MAR> / <WRITEO>       CONTINUE TO INITIALIZE I-STORE
%...............................................
INITTOP.FREE:  SET UP THE FREE LIST
, EXTRA<-MAR / <WRITEO> /->INITBOTO  SAVE THE POINTER TO THIS CELL
,   / <CHECK-WRAP> | <INC-MAR>
%...............................................
```

```
INITBOTO.: IF <OK>/<MEM-FINI> THEN->INITBOTO.OK
INITBOTO.OK: STOUT<-<LASTO>  WRITE STATUS FOR LAST CELL
, / <INC-MAR> / <WRITEO> /->INITBOT1
%.............................................
INITBOT1.: IF <OK>/<MEM-FINI> THEN->INITBOT1.ENDQ
INITBOT1.ENDQ: IF <WRAP>    WE HAVE HIT THE MEMORY BOUNDARY
, THEN->READY                 WE'RE FINISHED, CONTINUE
, ELSE <INC-MAR> / <WRITEO>   CONTINUE TO INITIALIZE FREE-LIST
,   /->INITBOT2
%.............................................
INITBOT2.: IF <OK>/<MEM-FINI> THEN->INITBOT2.OK
INITBOT2.OK: STOUT<-<NOT-LASTO>  WRITE STATUS FOR NOT LAST CELL
, / <WRITEO> /->INITBOT1
, / MDRD2T3<-EXTRA | EXTRA<-MAR | <INC-MAR>
```

B.4 *ALLOCATE OPERATION

```
%***************************************************
ALLOCATE.: IF <OK>/<MEM-FINI> THEN->ALLOCATE.OK
ALLOCATE.OK: ALLOCATE A BLOCK OF MEMORY
, IF STIN = <NOT-ALLOCATED>
,   THEN->ALLOCATE.BEGIN
,   ELSE ERRCODE<-<ALLOCATE-ERROR> /->ERROR
ALLOCATE.BEGIN: <SET-ALLOCATION-COUNT>  WRITE FIRST HEADER
, / STOUT<-<EMPTY-NOWAIT> / <WRITEO>
ALLOCATE.BEGIN: IF <ALLOCATION-COMPLETE>
, THEN->FINISH ELSE->ALLOCO
%.............................................
ALLOCO.: IF <OK>/<MEM-FINI> THEN->ALLOCO.OK
ALLOCO.OK: <INC-MAR> / <READO> / <COUNT-ALLOCATION> /->ALLOC1
%.............................................
ALLOC1.: IF <OK>/<MEM-FINI> THEN->ALLOC1.STATQ
ALLOC1.STATQ: IF STIN = <NOT-ALLOCATED>
,   THEN->ALLOC1.BEGIN
,   ELSE ERRCODE<-<ALLOCATE-ERROR> /->ERROR
ALLOC1.BEGIN: IF <HEADER-TIME>
, THEN STOUT<-<EMPTY-NOWAIT> / <SET-ALLOCATION-COUNT>
, ELSE STOUT<-<MIDDLE>
ALLOC1.BEGIN: <WRITEO>
ALLOC1.BEGIN: IF <ALLOCATION-COMPLETE>
, THEN->FINISH ELSE->ALLOCO
```

B.5 *CLEAR OPERATION

```
%***************************************************
% CLEAR OPERATION
CLEAR.: IF <OK>/<MEM-FINI> THEN->CLEAR.STATUSQ
CLEAR.STATUSQ: IF STIN = <NOT-ALLOCATED>   THEN->CLEAR.ERROR
CLEAR.STATUSQ: IF STIN = <EMPTY-NOWAIT>    THEN->CLEAR.CLEAR
CLEAR.STATUSQ: IF STIN = <TYPED-DATA>      THEN->CLEAR.CLEAR
CLEAR.STATUSQ: IF STIN = <UNTYPED-DATA>    THEN->CLEAR.CLEAR
CLEAR.STATUSQ: IF STIN = <MIDDLE>          THEN->CLEAR.CLEAR
CLEAR.STATUSQ: IF STIN = <INVISIBLE>       THEN->CLEAR.INVIS
CLEAR.STATUSQ: IF STIN = <TYPE-DEFERRED>   THEN->CLEAR.DEFER
CLEAR.STATUSQ: IF STIN = <NOTYPE-DEFERRED> THEN->CLEAR.DEFER
CLEAR.STATUSQ: IF STIN = <ERROR>           THEN->CLEAR.ERROR
%..................................................
CLEAR.CLEAR:  VANILLA CLEARING OF THIS WORD.
, MAR<-ADDR / STOUT<-<NOT-ALLOCATED>
,   / <WRITEO> /->CLEAR0
%..................................................
CLEAR.ERROR: ERRCODE<-<ALLOCATE-ERROR> /->ERROR
%..................................................
CLEAR.INVIS:  RETURN CELLS TO THE FREE LIST
, DATA0<-MDRDATA0 / <READO> /->CLINV0  SAVE TYPE INFO
,   / ADDR<-MAR / EXTRA<-MDRD2T3     SAVE THIS AND CELL'S ADDR
,   | MAR<-MDRD2T3 | <INC-MAR>
%..................................................
CLINV0.: IF <OK>/<MEM-FINI> THEN->CLINV0.OK
CLINV0.OK: IF *LOT9 LENGTHIN  THERE IS ONLY ONE CELL
, THEN->CLINV0.ONECELL
, ELSE MAR<-MDRD2T3 / <READO>  FOLLOW POINTER
,   | <INC-MAR> /->CLINV1
%..................................................
CLINV0.ONECELL: IF <IS-LIST-EMPTY>
, THEN STOUT<-<LASTO> / <CELLS-DO-REMAIN>
, ELSE STOUT<-<NOT-LASTO>
CLINV0.ONECELL:  CONS THIS CELL INTO FREE LIST
, MDRD2T3<-RTOP / <WRITEO> /->CLDONE
%..................................................
CLINV1.: IF <OK>/<MEM-FINI> THEN->CLINV1.OK
CLINV1.OK: MDRD2T3<-RTOP  JOIN THESE CELLS TO THE FREE LIST
, / <WRITEO> /->CLDONE
%..................................................
CLDONE.: IF <OK>/<MEM-FINI> THEN->CLDONE.OK
CLDONE.OK: RTOP<-EXTRA / MAR<-ADDR /->CLEAR0
%..................................................
CLEAR.DEFER:  RETURN CELLS TO THE FREE LIST
, <READO> /->CLDEF0                   TRACE FIRST LINK
,   / ADDR<-MAR / EXTRA<-MDRD2T3     SAVE THIS AND CELL'S ADDR
,   | MAR<-MDRD2T3 | <INC-MAR>
%..................................................
CLDEF0.: IF <OK>/<MEM-FINI> THEN->CLDEF0.OK
CLDEF0.OK: IF STIN = <LASTI>  THIS IS THE LAST CELL
, THEN MDRD2T3<-RTOP / <WRITEO>  JOIN THIS CELL TO FREE LIST
,   /->CLDONE
, ELSE MAR<-MDRD2T3 | <INC-MAR> / <READO>  FOLLOW POINTER
%..................................................
CLEAR0.: IF <OK>/<MEM-FINI> THEN->CLEAR0.OK
CLEAR0.OK: IF <MAR = FBR>  WE HAVE HIT THE END OF THE BLOCK
, THEN->CLEAR0.FINISH         FINISHED WITH CLEAR OPERATION
, ELSE <INC-MAR> / <READO>    CONTINUE TO CLEAR THIS BLOCK
CLEAR0.FINISH: IF INQ = 1
, THEN->CLEAR0.FORMAT  ELSE->READY
CLEAR0.FORMAT: <GRAB-TOKEN-IF-READY>
%..................................................
```

```
CLEAR1.: IF <OK>/<MEM-FINI> THEN->CLEAR1.STATUSQ
CLEAR1.STATUSQ: IF STIN = <NOT-ALLOCATED>   THEN->CLEAR1.ERROR
CLEAR1.STATUSQ: IF STIN = <EMPTY-NOWAIT>    THEN->CLEAR1.CLEAR
CLEAR1.STATUSQ: IF STIN = <TYPED-DATA>      THEN->CLEAR1.CLEAR
CLEAR1.STATUSQ: IF STIN = <UNTYPED-DATA>    THEN->CLEAR1.CLEAR
CLEAR1.STATUSQ: IF STIN = <MIDDLE>          THEN->CLEAR1.CLEAR
CLEAR1.STATUSQ: IF STIN = <INVISIBLE>       THEN->CLEAR1.INVIS
CLEAR1.STATUSQ: IF STIN = <TYPE-DEFERRED>   THEN->CLEAR1.DEFER
CLEAR1.STATUSQ: IF STIN = <NOTYPE-DEFERRED> THEN->CLEAR1.DEFER
CLEAR1.STATUSQ: IF STIN = <ERROR>           THEN->CLEAR1.ERROR
%..............................................
CLEAR1.CLEAR:  VANILLA CLEARING OF THIS WORD.
, MAR<-ADDR / STOUT<-<NOT-ALLOCATED>
,   / <WRITEO> /->CLEARO
%..............................................
CLEAR1.ERROR: ERRCODE<-<ALLOCATE-ERROR> /->ERROR
%..............................................
CLEAR1.INVIS:  RETURN CELLS TO THE FREE LIST
, DATAO<-MDRDATAO / <READO> /->CLINVO  SAVE TYPE INFO
,   / ADDR<-MAR / EXTRA<-MDRD2T3     SAVE THIS AND CELL'S ADDR
%..............................................
CLEAR1.DEFER:  RETURN CELLS TO THE FREE LIST
, <READO> /->CLDEFO                 TRACE FIRST LINK
,   / ADDR<-MAR / EXTRA<-MDRD2T3     SAVE THIS AND CELL'S ADDR
,   | MAR<-MDRD2T3 | <INC-MAR>
```

B.6 *RESET OPERATION

```
%**************************************************
% RESET OPERATION
RESET.: IF <OK>/<MEM-FINI> THEN->RESET.STATUSQ
RESET.STATUSQ: IF STIN = <ERROR-NOT-ALLOCATED>   THEN->RESET.RESET
RESET.STATUSQ: IF STIN = <ERROR-EMPTY-NOWAIT>    THEN->RESET.RESET
RESET.STATUSQ: IF STIN = <ERROR-TYPED-DATA>      THEN->RESET.RESET
RESET.STATUSQ: IF STIN = <ERROR-UNTYPED-DATA>    THEN->RESET.RESET
RESET.STATUSQ: IF STIN = <ERROR-MIDDLE>          THEN->RESET.RESET
RESET.STATUSQ: IF STIN = <ERROR-INVISIBLE>       THEN->RESET.INVIS
RESET.STATUSQ: IF STIN = <ERROR-TYPE-DEFERRED>   THEN->RESET.DEFER
RESET.STATUSQ: IF STIN = <ERROR-NOTYPE-DEFERRED> THEN->RESET.DEFER
RESET.STATUSQ: IF STIN = <NO-ERROR>              THEN->RESET.ERROR
%..............................................
RESET.RESET:  VANILLA RESETTING OF THIS WORD.
, MAR<-ADDR / STOUT<-<NOT-ALLOCATED>
,   / <WRITEO> /->RESETO
%..............................................
RESET.ERROR: ERRCODE<-<RESET-ERROR> /->ERROR
%..............................................
RESET.INVIS:  RETURN CELLS TO THE FREE LIST
, DATAO<-MDRDATAO / <READO> /->REINVO  SAVE TYPE INFO
,   / ADDR<-MAR / EXTRA<-MDRD2T3     SAVE THIS AND CELL'S ADDR
,   | MAR<-MDRD2T3 | <INC-MAR>
%..............................................
REINVO.: IF <OK>/<MEM-FINI> THEN->REINVO.OK
REINVO.OK: IF *LOT9 LENGTHIN  THERE IS ONLY ONE CELL
, THEN->REINVO.ONECELL
, ELSE MAR<-MDRD2T3 / <READO>  FOLLOW POINTER
,   | <INC-MAR> /->REINV1
%...............................................
REINVO.ONECELL: IF <IS-LIST-EMPTY>
, THEN STOUT<-<LASTO> / <CELLS-DO-REMAIN>
, ELSE STOUT<-<NOT-LASTO>
REINVO.ONECELL:  CONS THIS CELL INTO FREE LIST
, MDRD2T3<-RTOP / <WRITEO> /->REDONE
%...............................................
REINV1.: IF <OK>/<MEM-FINI> THEN->REINV1.OK
REINV1.OK: MDRD2T3<-RTOP  JOIN THESE CELLS TO THE FREE LIST
, / <WRITEO> /->REDONE
%...............................................
REDONE.: IF <OK>/<MEM-FINI> THEN->REDONE.OK
REDONE.OK: RTOP<-EXTRA / MAR<-ADDR /->RESETO
%...............................................
RESET.DEFER:  RETURN CELLS TO THE FREE LIST
, <READO> /->REDEFO                TRACE FIRST LINK
,   / ADDR<-MAR / EXTRA<-MDRD2T3     SAVE THIS AND CELL'S ADDR
,   | MAR<-MDRD2T3 | <INC-MAR>
%...............................................
REDEFO.: IF <OK>/<MEM-FINI> THEN->REDEFO.OK
REDEFO.OK: IF STIN = <LASTI>  THIS IS THE LAST CELL
, THEN MDRD2T3<-RTOP / <WRITEO>  JOIN THIS CELL TO FREE LIST
,   /->REDONE
, ELSE MAR<-MDRD2T3 | <INC-MAR> / <READO>  FOLLOW POINTER
%...............................................
RESETO.: IF <OK>/<MEM-FINI> THEN->RESETO.OK
RESETO.OK: IF <MAR = FBR>  WE HAVE HIT THE END OF THE BLOCK
, THEN->RESETO.FINISH        FINISHED WITH RESET OPERATION
, ELSE <INC-MAR> / <READO>   CONTINUE TO RESET THIS BLOCK
RESETO.FINISH: IF INQ = 1
, THEN->RESETO.FORMAT  ELSE->READY
RESETO.FORMAT: <GRAB-TOKEN-IF-READY>
%...............................................
RESET1.: IF <OK>/<MEM-FINI> THEN->RESET1.STATUSQ
RESET1.STATUSQ: IF STIN = <ERROR-NOT-ALLOCATED>   THEN->RESET1.RESET
RESET1.STATUSQ: IF STIN = <ERROR-EMPTY-NOWAIT>    THEN->RESET1.RESET
RESET1.STATUSQ: IF STIN = <ERROR-TYPED-DATA>      THEN->RESET1.RESET
RESET1.STATUSQ: IF STIN = <ERROR-UNTYPED-DATA>    THEN->RESET1.RESET
RESET1.STATUSQ: IF STIN = <ERROR-MIDDLE>          THEN->RESET1.RESET
RESET1.STATUSQ: IF STIN = <ERROR-INVISIBLE>       THEN->RESET1.INVIS
RESET1.STATUSQ: IF STIN = <ERROR-TYPE-DEFERRED>   THEN->RESET1.DEFER
RESET1.STATUSQ: IF STIN = <ERROR-NOTYPE-DEFERRED> THEN->RESET1.DEFER
RESET1.STATUSQ: IF STIN = <NO-ERROR>              THEN->RESET1.ERROR
%...............................................
RESET1.RESET:  VANILLA RESETING OF THIS WORD.
, MAR<-ADDR / STOUT<-<NOT-ALLOCATED>
,   / <WRITEO> /->RESETO
%...............................................
RESET1.ERROR: ERRCODE<-<ALLOCATE-ERROR> /->ERROR
%...............................................
RESET1.INVIS:  RETURN CELLS TO THE FREE LIST
, DATAO<-MDRDATAO / <READO> /->REINVO  SAVE TYPE INFO
,   / ADDR<-MAR / EXTRA<-MDRD2T3     SAVE THIS AND CELL'S ADDR
```

```
%.............................................
RESET1.DEFER:   RETURN CELLS TO THE FREE LIST
, <READO> /->REDEFO                 TRACE FIRST LINK
,   / ADDR<-MAR / EXTRA<-MDRD2T3      SAVE THIS AND CELL'S ADDR
,   | MAR<-MDRD2T3 | <INC-MAR>
```

B.7 *ALLOCATE-FREE-SPACE OPERATION

```
%*****************************************************
% ALLOCATE FREE CELLS IN THE I-STORE SECTION
ALLOCFR.: IF <OK>/<MEM-FINI> THEN->ALLOCFR.STATUSQ
ALLOCFR.STATUSQ: IF STIN ≠ <NOT-ALLOCATED>   CHECK FIRST CELL
, THEN STOUT<-<ALLOCATE-FREE-SPACE-ERROR> /->ERROR
, ELSE   READ THE SECOND CELL
,   EXTRA<-MAR | <INC-MAR> / <READO> /->AFSO
%.............................................
AFSO.: IF <OK>/<MEM-FINI> THEN->AFSO.STATUSQ   CHECK SECOND CELL
AFSO.STATUSQ: IF STIN ≠ <NOT-ALLOCATED>
, THEN STOUT<-<ALLOCATE-FREE-SPACE-ERROR> /->ERROR
, ELSE   CONS THIS CELL ONTO THE FREE LIST
,   MDRD2T3<-RTOP / <WRITEO> /->AFSO.STAT /->AFS1
AFSO.STAT: IF <IS-LIST-EMPTY>
, THEN STOUT<-<LASTO> / <CELLS-DO-REMAIN>
, ELSE STOUT<-<NOT-LASTO>
%.............................................
AFS1.: IF <OK>/<MEM-FINI> THEN->AFS1.ENDQ
AFS1.ENDQ: IF <ALLOCATE-FREE-SPACE-COMPLETE>
, THEN->READY        WE'RE FINISHED, CONTINUE
, ELSE <INC-MAR> / <READO> /->AFS2   READ ANOTHER CELL
%.............................................
AFS2.: IF <OK>/<MEM-FINI> THEN->AFS2.STAT
AFS2.STAT: IF STIN ≠ <NOT-ALLOCATED>   CHECK FIRST CELL
, THEN STOUT<-<ALLOCATE-FREE-SPACE-ERROR> /->ERROR
, ELSE EXTRA<-MAR | <INC-MAR>   READ SECOND CELL
,   /->AFSO
```

B.8 *READ-ABSOLUTE OPERATION

```
%*****************************************************
OPQ.READAB:   READ ABSOLUTE
, DEST<-DAT2T7 / ADDR<-DAT8T9
,   | MAR<-ADDR / <READO> /->READABO
%.............................................
READABO.: IF <OK>/<MEM-FINI> THEN->READABO.SHIP
READABO.SHIP: IF <IS-OUTPUT-SECTION-FREE>
, THEN OREGOT5<-DEST / OREG6T9<-MDRDATA   TRANSFER DESTINATION,
,   / OREG10[0 1 2 3]<-MDRSTAT          DATA, AND STATUS
,   /->READABO.FINISH                  TRY AND GRAB NEXT TOKEN
READABO.FINISH: IF INQ = 1
```

```
, THEN->READABO.FORMAT  ELSE->READY
READABO.FORMAT: <GRAB-TOKEN-IF-READY>
```

B.9 *WRITE-ABSOLUTE OPERATION

```
%***************************************************
OPQ.WRITEAB:   WRITE ABSOLUTE
, ADDR<-DAT2T3 / MDRDATA<-DAT0T3 / MDRSTAT<-DATA4[0 1 2 3]
,   | MAR<-ADDR / <WRITEO> /->FINISH
```

B.10 *LOAD-ERROR-DESTINATION OPERATION

```
%***************************************************
OPQ.LOADED:   LOAD ERROR DESTINATION
, ERRDEST<-DEST /->READY
%***************************************************
% ROUTINES COME HERE TO HAVE A VANILLA FINISH TO THEIR EXECUTION
FINISH.: IF <OK>/<MEM-FINI> THEN->FINISH.OK
FINISH.OK:  IF INQ=1  WE ARE FINISHED, CLEAN UP
, THEN->FINISH.FORMAT  ELSE->READY
FINISH.FORMAT: <GRAB-TOKEN-IF-READY>
%...............................................
ERROR:  ERROR STATE.  SET THE ERROR BIT TO 1 AND EXPORT A MESSAGE
,       ASSUME ERRCODE HAS BEEN SET, MAR POINTS TO THE CORRECT
,       LOCATION, AND MDR HOLDS THE CONTENTS OF THAT LOCATION.
, / <WRITEO> / STOUT3<-<SET-TO-ERROR>   SET THE ERROR BIT ON
, /->ERROR1                            WAIT FOR MEMORY TO RESPOND
ERROR1: IF <OK>/<MEM-FINI>     MEMORY FINISHED
, / <IS-OUTPUT-SECTION-FREE>
, THEN OREG0T5<-ERRDEST          SEND TO ERROR MANAGER
,   / OREG6T7<-MAR               SEND ADDRESS
,   / OREG8<-<OPREG-ERRORS>      SEND OPCODE AND ERROR CODE
,   / <OUTPUT-SECTION-REQUEST> /->READY
```