Stanford Computer Systems Laboratory
Memo CSL-TR-167

March 1980

Computer Science Department
Report No. STAN-CS-80-792

MAINSAIL IMPLEMENTATION OVERVIEW

by

Clark R. Wilcox
Mary L. Dageforde
and
Gregory A. Jirak

Research sponsored by

National Institutes of Health

COMPUTER SCIENCE DEPARTMENT
Stanford University

Stanford Department of Computer Science
Report No. STAN-CS-80-792

March 1980

Stanford Computer Systems Laboratory
Report No. CSL TR-167

MAINSAIL IMPLEMENTATION OVERVIEW

by

Clark R. Wilcox
Mary L. Dageforde
Gregory A. Jirak

DEPARTMENT OF COMPUTER SCIENCE
School of Humanities and Science
Stanford University

This work was supported by the Biotechnology Resources Program of the National Institutes of Health under grant RR-00785.

Stanford Department of Computer Science
Report No. STAN-CS-80-792

Stanford Computer Systems Laboratory
Report No. CSL TR-167

March 1980

MAINSAIL IMPLEMENTATION OVERVIEW

by

Clark R. Wilcox
Mary L. Dagcforde
Gregory A. Jirak

Abstract

The MAINSAIL programming language and the supporting implementations have been developed over the past five years as an integrated approach to a viable machine-independent system suitable for the development of large, portable programs. Particular emphasis has been placed on minimizing the effort involved in moving the system to a new machine and/or operating system. For this reason, almost all of the compiler and runtime support is written in MAINSAIL, and is utilized in each implementation without alteration. This use of a high-level language to support its own implementation has proved to be a significant advantage in terms of documentation and maintenance, without unduly affecting the execution speed. This paper gives an overview of the compiler and runtime implementation strategies, and indicates what an implementation requires for the machine-dependent and operating-system-dependent parts.

Key Words and Phrases: high-level language, portability, machine independence, compiler design, memory management, file system, MAINSAIL.

TABLE OF CONTENTS

SECTION 1 INTRODUCTION AND OVERVIEW

The MAINSAIL project has been a part of the SUMEX Computer Project at Stanford University since the spring of 1974. It began as a project to develop PDP-11 code generators for the programming language SAIL ([1]), which is implemented just for the PDP-10. It became apparent that a reasonable implementation of SAIL on the PDP-11 would require a number of changes to SAIL. Over a period of time the number of changes reached the point that the altered language begin to take on its own character, and was generalized toward the broader goal of portability over a wide range of machines. The result is the MAINSAIL programming language, which has significant differences from SAIL.

For a complete description of MAINSAIL, see the MAINSAIL Language Manual ([2]). A comparison of MAINSAIL and PASCAL ([3]) may also be of interest' although the two languages were developed for different reasons and MAINSAIL is much broader in scope. For the most part, this report will assume a familiarity with MAINSAIL as described in the language manual.

The overall goals of the MAINSAIL project have been to provide an integrated set of tools for the creation of efficient portable software on a variety of computer systems, and to promote the development and distribution of portable software. By "portable software" is meant programs that are able to be executed' without modification, on a range of computers. An overview of MAINSAIL and its approach to portability can be found in "The MAINSAIL Project: Developing Tools for Software Portability" ([4]).

This report provides an overview of the implementation of MAINSAIL. It demonstrates the integration of the system, the aspects that have been affected by portability issues, and what is involved in implementing MAINSAIL on a new machine. There are sections pertaining to the compiler design, runtime data structures, operating-system and machine-dependent interfaces, memory management by the runtime system, and the file system model.

In designing the MAINSAIL programming system, a concerted effort has been made to guarantee the characteristics of a sufficiently rich and efficient programming environment that the programmer will seldom if ever feel the need to utilize machine-dependencies other than those which are inherent to the task being performed.

MAINSAIL's approach to portability is different from the "classical" approach of other languages such as FORTRAN, COBOL, BASIC, and ALGOL. These languages were designed with the idea that a new compiler would be written for each different computer system. In writing a given compiler, the implementor attempts to adhere to a "language standard." However, it is difficult, if not impossible, to specify a language standard in such a way as to be complete and unambiguous, and to guarantee that all implementations would be compatible. In addition to differences encountered due to ambiguities in the language standard, many implementors yield to the temptation to "enhance" the language, thereby introducing machine and system dependencies which render programs non-portable. Others simply implement a subset of the language' making it impossible to run programs utilizing the unimplemented operations.

MAINSAIL, on the other hand, has a single compiler and runtime system (both of which are written in MAINSAIL) that are employed in all implementations. This is made possible by MAINSAIL's ability to compile itself into code for a variety of machines. Only the compiler's code generators, and small machine-dependent and operating-system-dependent interfaces, need be rewritten for each new implementation. These parts of MAINSAIL are at a level which has already been defined by the machine-independent parts, and do not affect the language from the user's viewpoint. Thus the "language standard" has been reduced to a "semantic standard" which is surrounded by machine-independent software. The facilities provided by the portable compiler and runfime system are an inherent part of the language. In this sense MAINSAIL is more of a portable programming system than simply a programming language which can be implemented on many machines. A single compiler and runtime system which are used at all sites appears to be the only realistic means of obtaining the goal of portability.

MAINSAIL is not oriented toward any particular application. Dynamic memory utilization makes it suitable for tasks with memory requirements which are difficult to predict prior to execution. The string capabilities (automatic handling of variable-length sequences of characters) facilitate interactive programming and word processing applications such as compilers, text editors and document preparation. These same facilities require runtime support, so that a MAINSAIL program is not a stand-alone body of code, and thus may not be appropriate for some machine-dependent system utilities which must execute with no runtime support.

The requirement that MAINSAIL provide for the development of portable software necessitates machine-independent approaches to all aspects of the language. These include a notion of data type which is not tied to a particular machine organization, the ability to handle various character sets, and the design of a file system model that is implementable at all sites.

MAINSAIL does not have user-declared types, concurrency, or interrupts. It has not been designed for program verification. What is distinctive about MAINSAIL is its design for portability' an extremely flexible implementation of modules and intermodule communication, dynamic memory support, strings, garbage collection, the handling of i/o in a machine-independent manner, and extensive compiletime facilities.

In addition to the language characteristics, the machine-independent design of MAINSAIL necessitates a model of runtime interactions that can be-supported on a broad range of computers. In particular, MAINSAIL must be able to execute in a limited address space, which means that programs must be broken into pieces (modules) that need be in memory only when executing.

The inability to characterize existing linkage editors and overlay systems in a machine-independent manner has forced the MAINSAIL runtime system to take over these functions. It thus assumes duties often considered part of the operating system. MAINSAIL contains its own notion of inter-module communication. MAINSAIL programs consist of independently compiled modules which may be executed from any address within memory, i.e., the modules are position-independent. The runtime system swaps-modules in and out of memory as needed during execution, thus providing a virtual memory facility. This allows large programs to execute in a small address space, if there is enough space to contain the data. There is no automatic mechanism for moving the data in and out of memory, i.e. there is virtual CODE space, but no virtual DATA space.

Not only is a dynamically-linked system considered necessary to support the machine-independent design' but it is also much more flexible and easier to use from the programmer's viewpoint. For example, the programmer does not have to design an overlay scheme; MAINSAIL automatically swaps modules in and out of memory dynamically as needed. With a linked system, every time a module is changed, added, or deleted, the entire program must be re-linked (and sometimes the overlay system must be redesigned). MAINSAIL's approach, on the other hand, allows for viewing a program as an open-ended collection of modules; i.e., the programmer need not specify what modules make up a program. It is not necessary to link together all modules that could potentially be called: the modules may originate from many files at execution, as contrasted to the common approach of having a single "save file" or "load module" which may contain an overlay structure.

The initial research and development of MAINSAIL under the NIH grant is now complete. MAINSAIL's use has so far been demonstrated on PDP-10 and PDP-11 computers with various operating systems. These operating systems include TENEX, TOPS-20, TOPS-lo, RSX-11, RT-11, and UNIX. Further distribution and support of MAINSAIL, and the development of new implementations' will be carried out by a private company comprised of the individuals who have worked on the MAINSAIL project.

SECTION 2 COMPILER

The MAINSAIL compiler is written entirely in MAINSAIL itself, and can be ported among machines simply by recompiling it into the proper target code. The compiler and runtimes were developed concurrently with the language, and thus features that were felt necessary for their efficient implementation were simply put into the language. For example, MAINSAIL has a comprehensive set of compiletime facilities which are invaluable in the construction of large programs.

Many compilers are designed to work in a sequential access mode, and suffer from the resulting limitations. The MAINSAIL compiler is designed with the understanding that the source files are on random-access devices, so that it need not progress through a file in a strictly linear fashion. Any number of nested input files are allowed. The same file may even be scanned several times during compilation (as contrasted with a compiler designed for input from punched card decks).

As discussed in Section 1, a MAINSAIL program is composed of separately-compiled modules. The compiler outputs position-independent code for each module it compiles, so that the runtime system may move and/or swap modules during execution when necessary.

This section g‘ives an overview of the compiler, in particular specifying the compilation approach, the design of the symbol table, a register utility module, and the intermediate code from which target code is generated.

2.1 OVERVIEW OF THE COMPILATION APPROACH

The compiler is conceptually divided into two passes. The first pass is target-independent and translates the source text into an intermediate code. The second pass generates the desired target code from the intermediate code. This pass contains a target-independent part which examines the intermediate code and calls the appropriate code generation procedures. Code generation procedures are written for each target machine. Note that the compiler is independent of the host machine (the machine on which the compilation is being done). Thus it can be considered a cross compiler, since a given code generator can be used on any machine.

```
+---------------+      +---------------+      +---------------+
|               |      |               |      |               |
|    source     |      | intermediate  |      |    target     |
|               |------>|               |------>|               |
|     text      |      |     code      |      |     code      |
|               |      |               |      |               |
+---------------+      +---------------+      +---------------+
```

2.2 FIRST PASS

The first pass of the compiler utilizes a standard recursive-descent technique to process the source text. It outputs the intermediate code in an internal format into files which are then processed by the second pass. The second pass also utilizes other information in memory-resident data structures, so that the intermediate-code files do not by themselves contain sufficient information for the second pass.

The compiler utilizes a symbol table containing symbols for all the identifiers and constants encountered in the source text, as well as all predeclared or predefined symbols. Each symbol contains information to uniquely identify the entity it represents. This information includes the symbol's identifier (value if a constant): a unique integer "tag" assigned to it: its data type; its "klass" (e.g. macro, constant, parameter); and a "hash code" computed from the identifier that is used for symbol entry and search.

A symbol is represented in two forms: as a record while its fields are being accessed; and as a string while simply residing in the symbol table. The latter-representation is referred to as a "packed symbol" since it requires less space than a symbol record.

SYMBOL TABLE

The basic approach to the symbol table is to keep heavily accessed symbols in memory, while less frequently accessed symbols overflow to files. This allows the symbol table to be much larger than would be possible if it were all in memory.

The symbol table consists of four parts, two of which are in memory: .

1) procedure cache -- symbols local to the current procedure. This cache does not overflow to a file' since the number of symbols declared within a procedure is in general small.

2) memory cache -- most recently accessed non-local symbols. This cache overflows to the file SYM.

3) SYM -- file which contains symbols which have overflowed from the memory cache.

4) base file -- contains symbols declared before the actual compilation of the current module begins, as created by the SAVEON compiler directive.

The Procedure Cache

The procedure cache is a pointer array whose elements are the heads of linked lists of symbols local to the current procedure. The ith list contains symbols that hash to i.

PROCEDURE CACHE

```
         +--------+        +--------+              +--------+
   0     |        |------->|        |------> . . . |        |
         +--------+        +--------+              +--------+

         +--------+        +--------+              +--------+
   1     |        |------->|        |------> . . . |        |
         +--------+        +--------+              +--------+
         .        .
         .        .
         .        .

         +--------+        +--------+              +--------+
  n-1    |        |------->|        |--em--> . . . |        |
         +--------+        +--------+              +--------+
```

The Memory Cache

The memory cache is a string array whose elements are packed symbols. The array is conceptually divided into fixed-size "buckets". A symbol with hash code i is put into the ith bucket.

MEMORY CACHE

```
                  +----------------+
                  |                |
   bucket 0       |                |
                  |                |
                  +----------------+
                  |                |
   bucket 1       |                |
                  |                |
                  +----------------+
                  .                .
                  .                .

                  +----------------+
                  |                |
   bucket m-1     |                |
                  |                |
                  +----------------+
```

The search for a symbol in the memory cache begins with the first element in the appropriate bucket, and searches linearly to the end of the bucket. During this search, each element is moved forward by one, leaving the first element free. When the target element is found (possibly in SYM or the base file), it is put into the freed first element. As a result, each bucket is always sorted from most recently accessed symbol to least recently accessed symbol. This both speeds up the linear search, and conveniently results in the oldest symbol being at the end of the bucket. If the bucket is full, and the target symbol is found in SYM or the base file, this last (and "oldest") symbol "falls off the end" of the memory cache and is put into the appropriate SYM bucket if it is not already there (as indicated by a bit in the packed symbol).

The SYM File

The SYM file is conceptually divided into fixed-size buckets which contain the symbols which overflow from the memory cache. When a symbol is "bumped off the end" of the memory cache for the first time, its hash code is used to compute the SYM bucket to which it belongs.

If a SYM bucket becomes full, a new one (beyond those initially allocated) is used, and the full bucket is linked to the overflow bucket. An overflow bucket can also be linked to another overflow bucket. Thus, there is in effect a linked list of buckets containing symbols that hash to each given initial bucket. In the simplest scheme each (initial) SYM bucket would correspond to a memory cache bucket. However, the implementation allows for a different number of memory cache and SYM buckets, in particular so that the number of SYM buckets is independent of memory constraints on the number of memory buckets. The number of memory and SYM buckets and their sizes are parameters which can be independently adjusted to the memory constraints.

```
                         SYM

                +---------------+
                |               |
   bucket  0    |               |
                |               |
                +---------------+
                .               .
                                .

                +---------------+
                |               |
   bucket  n-l  |               |
                |               |
                +===============+
   0            |               |  -- First bucket available to be
   v            |               |        an overflow bucket
   e            |               |
   r            +---------------+
   f   B        .               .
   1   u        .               .
   o   c
   w   k        +---------------+
       e        |               |
       t        |               |
       s        |               |
                +---------------+
```

SYM buckets are used for two other purposes:

1) To contain a linked list of buckets containing all the string constants. The string-constant symbols are also put in regular buckets, but having all the string constants grouped together facilitates their output to the descriptor section during the second pass (see 3.2).

2) To contain a linked list of buckets with information about classes and modules not needed frequently enough to be allocated in memory.

The Base File

The base file is in exactly the same format as the SYM file: initial buckets followed by overflow buckets.

There is an initial symbol table file for MAINSAIL consisting of all predeclared and predefined identifiers. At the beginning of each compilation, this file is opened as the base file for the symbol table.

MAINSAIL's SAVEON and RESTOREFROM compiler directives allow the symbol table to be preserved and then restored in later compilations to avoid recompiling declarations and definitions that a number of different modules need. A SAVEON to save the "current state" of the symbol table will first dump the memory cache to the SYM file and then combine the base and SYM files into a single file which becomes the base file for all compilations that restore it.

Symbol Table Lookup

To look up a symbol, the compiler first searches the appropriate list in the procedure cache, as determined by a hash code. If it does not find the symbol there, it then looks in the appropriate bucket in the memory cache, using a different hash code (since there are a different number of buckets). If it is not there, but the memory cache bucket is full (signifying that it might have overflowed to SYM), then the appropriate SYM bucket is examined, once again using a different hash code. Finally, the base file is searched if the symbol still has not been found. When found, the newly-looked-up symbol is copied to the beginning of the appropriate memory-cache bucket to increase the probability of finding it in memory rather than on a file, and the oldest symbol in that bucket is copied to the SYM file if necessary.

2.3 INTERMEDIATE CODE

The intermediate code recasts the source program into a form which facilitates code generation. It employs no special knowledge about the target machine.

The intermediate code consists of a linear sequence of operators followed by zero or more operands, where "operand" is used as a general term to refer to "regular" operands, labels, and other items that need to be passed to the second pass (such as the initialization values specified for an array by an INIT statement). An operator is specified in the intermediate code as an integer code. A regular operand is represented as the packed symbol for the operand. A label is an integer. An "operand cache" is utilized to decrease the number of packed symbols actually put into the intermediate file.

The following table summarizes the intermediate code. Note that the operators are generic in the sense that the operands determine the exact data type. For example, "plus" applies to integer, long integer, real and long real.

OPERATOR	OPERANDS	COMMENTS
be	lbl,opr1,opr2	IF oprl = opr2 THEN branch to lbl
blt	"	IF oprl < opr2 THEN branch to lbl
bgt	"	IF oprl > opr2 THEN branch to lbl
tst	"	IF oprl TST opr2 THEN branch to lbl
tsta	"	IF oprl TSTA opr2 THEN branch to lbl
branch	lbl	branch to lbl
ntsta	lbl,opr1,opr2	IF oprl NTSTA opr2 THEN branch to lbl
ntst	"	IF oprl NTST opr2 THEN branch to lbl
ble	"	IF oprl LEQ opr2 THEN branch to lbl
bge	"	IF oprl GEQ opr2 THEN branch to lbl
bne	"	IF oprl NEQ opr2 THEN branch to lbl
assign	oprl ,opr2	opr2 := oprl
min	opr1,opr2,opr3	opr3 := oprl MIN opr2
max	"	opr3 := oprl MAX opr2
plus	"	opr3 := oprl + opr2
minus	"	opr3 := oprl - opr2
ior	"	opr3 := oprl IOR opr2
xor	"	opr3 := oprl XOR opr2
clr	"	opr3 := oprl CLR opr2
msk	"	opr3 := oprl MSK opr2
times	"	opr3 := oprl * opr2
div	"	opr3 := oprl DIV opr2
mod	"	opr3 := oprl MOD opr2
shl	"	opr3 := oprl SHL opr2
shr	"	opr3 := oprl SHR opr2

neg	opr1,opr2	opr2 := - opr1
call	opr1,opr2 opr3,...,oprN	call procedure oprl, with arguments opr3,...,oprN. If typed, put the result in opr2.
return	none	RETURN
returnResult	oprl	RETURN(opr1)
putResult	oprl	save oprl as a "temporary" result
getResult	oprl	oprl := last value saved by putResultOp

makeValue	lbl,opr1	generate code with the effect:
		        oprl := TRUE; branch 1;
		  lbl: oprl := FALSE;
		  1:

case	lbl,opr1,opr2, op r3	lbl = label for the default case,
		oprl = symbol for the index variable,
		opr2 = min selector value (constant),
		opr3 = max selector value (constant),

caseLabel	lbl	generate case-statement branch table entry to label lbl
caseErr	oprl	generate code for case index out of bounds, with case index value oprl
init	lbl,opr1,opr2, initialization values	array initialization. lbl is used to refer to start of value encodings in descriptor section; oprl is symbol for the array initialization procedure; opr2 is symbol for the array.
consttyWrite	oprl	ttyWrite(opr1) where oprl is a string constant
consfWrite	opr1,opr2	write(opr1,opr2) where oprl is a file pointer and opr2 is a string constant
enter	opr1,c,lbl opr2,...,oprn	entry to procedure oprl. c = 1 <=> the proc is CODED. lbl is minimum label used in the proc. opr2,...,oprn are the parameter symbols.
exit	none	end of current procedure
label	lbl	place label lbl
adr	opr1,opr2,opr3	create address temp (see below)
temp	oprl ,opr2	create value temp (see below)
keep	oprl	oprl is a value temp. Copy it to memory if necessary.

unKeep	oprl	Undoes some internal bookkeeping caused by keep operator.
rlse	oprl	oprl no longer needed: bookkeeping only.
dscr	**pLocs,sLocs, oLocs**	tells number of pointer, string, and other local variables of procedure being entered.
code	**s**	Put the string constant s into the output file.
encodeCon	type of con, con id, the defined con	**encodeConOP** and **encodeVarOP** are for outputting symbolic definitions of selected variables and defined constants so they may be referenced by the text of a Code statement.
encodeVar	id to be encoded	see **encodeCon**
check	none	Henceforth emit code to check certain conditions at **runtime** that cannot be determined at compiletime (e.g., that array subscripts are in bound, and that pointers used are not null).
noCheck	none	Henceforth don't emit checking code.
modEnd	none	end of module
error	none	error condition: second pass should never encounter this.

Address Temps

In order to represent a variable of the form p.f (pointer.field, used to reference a field of a record), a.f (address.field, used to reference memory), m.f (module.field, used to reference interface procedures and variables), or **a[i], a[i,j]** or **a[i,j,k]** (array elements) as a single operand in the intermediate code, an "address temp" is created and used as the operand.

An-address temp is simply a symbol that links together the appropriate symbols for the components of a structured variable. It is not itself allocated a memory location. For example, for a variable of the form **p.f,** the address temp points to the symbol for p, which in turn points to the symbol for f:

```
adrTemp             p               f
+--------+     +--------+     +--------+
|        |---->|        |---->|        |
|        |     |        |     |        |
+--------+     +--------+     +--------+
```

For a variable of the form **m.f,** the address temp points to the symbol for f, which points to the symbol for m:

```
adrTemp             f               m
+--------+     +--------+     +--------+
|        |---->|        |---->|        |
|        |     |        |     |        |
+--------+     +--------+     +--------+
```

For a subscripted variable **(ary[i]** or **ary[i** ,j] or **ary[i,j,k]),** the address temp points to the symbol for ary which points to a linked list of the subscript symbols. For example, for ary[i,j]:

```
adrTemp           ary               i                j
+--------+     +--------+     +--------+     +--------+
|        |---->|        |---->|        |---->|        |
|        |     |        |     |        |     |        |
+--------+     +--------+     +--------+     +--------+
```

An example of the use of an address temp is as follows. The assign operator has two operands, one for the source and one for the destination. For example, the MAINSAIL statement

```
i := j
```

would be translated by the first pass into the intermediate code

```
assign j i
```

(letting i stand for the symbol for i, and j for the symbol for j). Now consider the MAINSAIL statement

```
p.f := j
```

The second operand for assign needs to stand for the combination of two symbols, one for p and one for f. This is accomplished by first forming an address temp for the field variable **p.f,** and then using the address temp as the operand to assign:

```
adr x p f
assign j x
```

(where x refers to the address temp formed).

Value Temps

Whenever an expression occurs for which there is no explicit destination (e.g. as an argument to a procedure) a "value temp" temporarily stores the value of the expression. Thus a value temp is simply a symbol used for temporary storage of a value. It has the status of a variable declared local to a procedure, and is allocated a memory location in the stack frame. Consider for example the MAINSAIL statement

```
        i := a * b + c
```

First, the code to compute the value of the expression a * b is output. The intermediate code will have the mul operator, which has three operands: two sources and a destination. A value temp t will be used as the temporary destination:

```
        mul a b t
```

and then to represent the value of a * b as a source in the subsequent operation

```
        plus t c i
```

which has an explicit destination, i. It should be emphasized that even though the intermediate code treats value temps as if they were normal local variables, code generators will be aware that an operand is a value temp, and will in general attempt to keep the value temp in a register rather than utilize the memory location allocated for it. For example, a typical code sequence for the above intermediate code might be:

```
        LOAD    r,a     ;mul a b t
        MUL     r,b
        ADD     r,c     ;plus t c i
        STORE   r,i
```

2.4 SECOND PASS

The second pass generates target code from the intermediate code. It consists of a target-independent part which reads the intermediate code and calls the various code generation procedures, which must be rewritten for each target machine.

The compiler thus employs target-dependent modules to generate the code rather than some kind of table which would attempt to encode the information needed to generate code. This provides the code generators all the facilities of MAINSAIL and avoids preconceived notions of how the target machine is structured. There has been no attempt to capture some form of description of the target machine which could then be used as the basis of an "automatic" code generator. It is felt that the effort involved in writing such a machine description is at least as difficult as writing the code generator itself, and certainly not as flexible.

Though not mandated by the target-independent part of the compiler, most code generators produce assembly language (rather than immediately executable code) for several reasons. For one thing, the generated code can be easily examined' which is useful for debugging the code generators. Also, in certain rare cases it is useful to be able to edit the generated assembly code before assembling it, which would not be possible if binary code were generated. Another reason is that it avoids having to make packing decisions that would be difficult to do in a machine-independent way. For example, when compiling on a 16-bit machine to output code for a 36-bit machine, it is not obvious how the 36-bit word sequences should be output. Also, floating point constants can be output as text rather than in an internal form which would be difficult to compute on a host machine different from the target machine. Finally, the Code Statement would be difficult to implement if binary code were being generated, whereas with the current approach, the string constant is simply put into the output file.

Register Utility Module

There is an additional target-independent module' reguty, available during the second pass for managing registers. It implements a target-independent notion of register. Any code generator may use reguty if the target machine has registers which conform to the concepts implemented by this module. If a machine has no registers, or has registers with characteristics which are not captured by reguty, then the services provided by reguty are simply not used.

Reguty employs a "register model" whose underlying implementation is irrelevant to the code generators. Under this model, operands can be . associated with registers, and any register can belong to one or more of up to 16 code-generator-defined classes, so that the code generators can ask for a free register belonging to a specific class.

There is also the notion of "double register", which is a pair of adjacent registers that can be used either individually or as a unit.

To use reguty, the code generators first call a reguty procedure to initialize each register they will need in the register model. A call to this procedure specifies the name to be used for the register and the classes it belongs to.

Once the registers have been initialized, the code generators can call other reguty procedures to: 1) ask for a free register of a particular class; 2) associate an operand with a register; 3) "clear" a register (break the association between it and an operand); 4) reserve a register; 5) ask if a specific operand is in a register; 6) "dump" the contents of registers to memory.

Reguty uses a conservative strategy to keep track of register contents:

1) Dump all registers at branches -- when any of the branching operators are encountered' any register whose contents are not also in memory is stored into memory.

2) Clear all registers at labels -- rather than keep track of register contents at all branches to a label, and then take the "intersection" of the contents, reguty simply assumes the worst: all registers are cleared.

3) Dump all registers at procedure calls -- registers are not preserved over procedure calls. At procedure calls, any register whose contents are not also in memory is stored into memory. Upon return, nothing is assumed about the register contents (except a result value may be associated with a particular register) -- all the registers are cleared.

The result is that linear sequences of code remember the contents of registers, but the contents are forgotten at points of control transfer. Each implementation uses selected registers for dedicated purposes such as stack pointers-. Such registers are not under the control of reguty, and hence are not affected by branches.

2.5 COMPILER DEVELOPMENT

The MAINSAIL compiler was initially bootstrapped via SAIL: by means of macros and conditional compilation, the compiler was originally able to be compiled by the SAIL compiler. The compiler, running as a SAIL program, was then used to compile the compiler and the runtime system into MAINSAIL code. That code was then assembled, resulting in a MAINSAIL compiler and runtime system that were free of SAIL. MAINSAIL is now completely independent of SAIL, i.e. SAIL no longer plays a role as a bootstrap mechanism.

The compiler has undergone a number of changes and improvements over the past few years. It is a large program whose development demonstrates the fact that care must be taken in designing large portable programs if they are to be able to be run in small (32K word) address spaces. The compiler in fact was unable to run in such address spaces until a number of data reductions and efficiency improvements were made. The MAINSAIL runtime system swaps modules out of memory when necessary, but it does not swap the data used by a program, and the compiler's data consumed most of the available memory.

Analysis of the compiler data structures showed that some were larger than necessary, others were not being disposed (freed) when they could have been, and others could be totally eliminated or reduced in size through design improvements.

Where possible, data is now maintained on a file rather than in memory. For example, to save space required by the text of string constants, the compiler error messages have been placed on a file. Calls to error message procedures specify the location on the file of the appropriate error message.

As a result of the data reductions and efficiency improvements, the compiler is now able to run on machines with 32K-word address spaces. A reduction in compilation time was also accomplished as described by the following.

On a machine with a large address space the compiler is most efficient if it consists of a few modules, since that reduces the number of intermodule calls. But on a machine with a small address space, there is not enough room for large modules; module swapping is necessary, and compilation time is roughly proportional to the number of swaps. Various compiler configurations (ways in which the compiler procedures could be combined into modules) were examined to determine "optimal" configurations (ones with the least amount of swapping) for various address spaces.

This was done by writing a MAINSAIL program to simulate compilation on machines with-various memory sizes. The simulation was driven by exact data, obtained from traces of all procedure calls made during a sample compilation. A format was devised for easily specifying potential compiler configurations, and the simulator tested their efficiency.

The simulation results led to the current use of two compiler configurations: a "big" configuration to be run on machines with large address spaces, and a "small" configuration to be run on those with small address spaces. MAINSAIL's modular structure and the use of standard MAINSAIL compiler directives and language features permits automatic reconfiguration. As predicted by the simulation, use of the "optimal" small configuration significantly increased the compilation speed on a computer with a small address space, the PDP-11. Use of the "big" configuration, with just a few large modules, also improved the compiler speed on computers with large address spaces.

Despite the increase in compilation speed thus obtained, compilation on machines with 32K address spaces is still considered too slow for such machines to be heavily used as development machines. The address space is simply too small, especially considering the fact that up to one-fourth of it is often occupied by the machine's operating system.

It should be noted that the unsuitability of limited-address-space machines for development does not preclude the efficient execution on such machines of MAINSAIL programs smaller than the compiler. If a larger machine is available for cross-compilation, programs may be developed for use on small machines. This approach is used extensively at SUMEX, where program development takes place on a large time-shared computer facility, and then the debugged programs are sent via hardwire lines to smaller machines for production use.

SECTION 3 RUNTIME DATA STRUCTURES

The runtime representations of modules, arrays, classes and strings are described in this section, as are the details of procedure call, entry' exit and return. Details of memory management by the runtime system are given in the next section, Section 4.

3.1 CHUNKS

"Chunk" is a general name for the data structure used for the underlying representation of records, arrays, data sections (repositories for the data of modules, described below), and "descriptors". A descriptor exists for every class, module, and array. The use of each type of descriptor is specified below in sections 3.2, 3.3, and 3.4.

Every chunk is composed of a header and a body. The header is used by the runtime system for a number of purposes, including garbage collection (described in Section 4.9). The body contains the information portion of the chunk' such as the elements of an array or the fields of a record. The following diagram gives the format of every chunk:

```
                  CHUNK

+--------------------------------+
| garbLink                       |        ADDRESS
|------------------------------- |
| dscrLink                       |        POINTER
|=============================== |
|                                |
|                                |
| chunk body                     |
|                                |
|                                |
+--------------------------------+
```

The garbLink and the dscrLink compose the chunk header. The garbLink is used only by the garbage collector, while the dscrLink discriminates between the two underlying classes of chunks: those that are self-descriptive (descriptors and free chunks - 4.6)' and those that are not (ones which have separate descriptors: classes, modules and arrays). Chunks are described further in Section 4.6. They are introduced here simply to familiarize the reader with the structure that will appear in drawings in this section illustrating theimplementation of various data structures.

3.2 MODULE IMPLEMENTATION

MAINSAIL programs consist of independently compiled modules that may be executed from any address within memory, i.e. the modules are position-independent. A module is represented at runtime as a "control section" (the result of a compilation), one or more "data sections" which are created dynamically during execution, and a "module descriptor."

Control Section

A control section consists of an "entry vector", the code for the module's procedures, and a "descriptor section" containing information used to support the module's execution:

```
                 CONTROL SECTION

       +--------------------------------+
       | offset to descriptor section   |
       |------------------------------  |
       |                                |
       | entry vector                   |
       |                                |
       |--------------------------------|
       |  code for procedure 1          |
       |--------------------------------|
       |  code for procedure 2          |
       |--------------------------------|
       |                                |


       |--------------------------------|
       |  code for procedure n          |
       |------------------------------  |
       | descriptor                     |
       | section                        |
       +--------------------------------+
```

The start of the control section contains the displacement to the descriptor section, so that given the address of the control section, the start of the descriptor section can be determined.

The ENTRY VECTOR provides the interface-procedure-calling mechanism. The compiler maps each interface procedure of the module (i.e., each procedure that is to be accessible from other modules) into an index into the entry vector. The entry vector is simply a "jump" table with each entry referencing the associated procedure.

The DESCRIPTOR SECTION contains information which supports the module during execution:

```
                DESCRIPTOR SECTION

        +--------------------------------+
        | module bits                    |     BITS
        |--------------------------------|
        | module name                    |     LONG BITS
        |--------------------------------|
        | version number                 |     BITS
        |--------------------------------|
        | number of pointers and strings |     BITS
        |--------------------------------|
        | data section size              |     INTEGER
        |--------------------------------|
        | offset to module name table    |     INTEGER
        |--------------------------------|
        | offset to string constants     |     INTEGER
        |--------------------------------|
        | offset to module pointers      |     INTEGER
        |--------------------------------|
        | offset to class pointers       |     INTEGER
        |--------------------------------|
        | class                          |     each an INTEGER
        I table                          |
        |--------------------------------|
        | class information              |
        |                                |     one for each class
        |                                |
        |--------------------------------|
        | module name                    |     each a LONG BITS
        I table                          |
        |--------------------------------|
        | string constants               |     each:
        |                                |      n,char1,...charn
        |--------------------------------|
        | array initialization           |     depends on type of
        | constants                      |     array
        +--------------------------------+
```

The module bits provide information about the module to the runtime system. For example, a bit is reserved to indicate whether or not the module has a FINAL procedure.

Each module is identified by its name, which must be unique and less than or equal to six characters long. The name is encoded in the descriptor section as a long bits.

The version number tells the runtime system version under which the control section is expected to run. When the runtime system allocates a module, it compares the current runtime version number with that in the descriptor section, and complains if they are different, thereby notifying the user that the module must be recompiled for a more up-to-date version. Version numbers are updated only when major changes that affect the execution of modules are made to the MAINSAIL system.

The number of pointers and strings tells the number of non-interface pointers (including array variables' module pointers, and class pointers) and strings that will be in a data section for the module. This is needed for garbage collection.

The data section size gives the size of data sections for the module.

The offset to the module name table gives the offset from the base of the descriptor section to the module name table.

The offset to string constants gives the offset from the base of the descriptor section to the string constant information.

The offset to the module pointers gives the offset from the base of a data section for the module to its module pointers (as described further in the next section). There is a pointer in the data section of a module m for each module referenced by m. The ordering of these module pointers and the entries in the module name table is identical so that a reference to a module can be represented by a single integer. The compiler generates this integer as a displacement from the base of the data section to the module pointer for the module. The runtime system uses this integer-directly to reference the module pointer, and when necessary it suitably modifies it to reference the appropriate entry in the module name table.

The offset to the class pointers gives the offset from the base of a . data section for the module to its class pointers (see next section). There is a class pointer in the module's data section for each class c for which a "new(c)" appears in the module. new(c) is the generic form of a call to the system procedure newRecord(which creates a new record for class c and returns a pointer to it.

The class table is an integer table with an entry corresponding to each class pointer. The value of each entry is the offset from the base of the descriptor section to the class information for the class. The ordering of the class pointers in the data section and the entries in the class table is identical. The compiler generates the offset to a class pointer as the argument to newRecord calls. This offset is used to locate the desired class pointer, and if Zero it is then used to locate the information necessary to build a new class descriptor.

The class information for each class is used to create the class descriptor for that class (see 3.4).

The module name table has an entry for each module referenced by the module, as described above. The value of the entry is the long bits encoding of the module name.

All of the string constants used in a module are put into the "string constant" part of the descriptor section. It contains the text of each string constant,. preceded by an encoding of its length. The code generators use the displacement from the base of the control section to the string constant to refer to a string constant. The string constant copier (3.5) can then reference the text of the constant.

Finally, all array initialization values specified in the module are encoded in the last part of the descriptor-section. Whenever the compiler encounters an **INIT** statement (which specifies array initialization values)' it generates a call to a procedure that will dynamically initialize the array. This procedure is passed the offset from the base of the control section to the desired initialization values.

Data Section

A module's DATA SECTION contains the interface and own data associated with the module. It consists of an INTERFACE RECORD followed by an OWN AREA:

```
                    DATA SECTION

          +-------------------------------+
          | garbLink                      |        ADDRESS
          |-------------------------------|
          |                               |
          | dscrLink                      |        POINTER(modDscr)
          |===============================|
          |                               |
```

The **dscrLink** points to the module's descriptor. A module descriptor describes the module's data section and control section. Module descriptors are discussed in the next section.

The interface record contains the data fields which can be accessed by other modules, as well as throughout the module with which the data section is associated. Thus the interface record is the basis of intermodule data sharing.

The own area contains data which can be accessed by procedures within the module, but cannot be explicitly accessed by other modules.

In addition to own variables explicitly declared within the module, the own area also contains a contiguous block of MODULE POINTERS which reference other data sections (see "Intermodule Communication", below), and a contiguous block of CLASS POINTERS which reference class descriptors (3.4).

```
            OWN AREA OF A DATA SECTION

        +------------------------------+ --+
        | module pointers .            |   |
        |------------------------------|   | contiguous block of
        | class pointers               |   | pointers
        |------------------------------|   |
        | other own pointers           |   |
        |------------------------------| --+
        | own string variables         |   | contiguous block of
        |------------------------------| --+ string descriptors
        | other own variables          |
        +------------------------------+
```

It is sometimes useful to have "multiple instances" of a module, which is implemented as a single control section and a single module descriptor, but multiple data sections. All data sections associated with the same module are identical in format, as described by the descriptor section. Whenever a request for a multiple instance is executed, the MAINSAIL runtime system creates a new data section for the module and returns a pointer to the newly created data section. The program must subsequently explicitly use that pointer to access the data fields.

Each module may have a default data section, called the "bound data section". In most cases this is the only data section that exists for the module. A default data section is a convenience; a program may refer to fields in it simply by name (if their names are unique), or as moduleName.fieldName, rather than via a pointer variable and the field name.

There are two ways in which a data section can become the bound data section for a module: 1) The first time an interface procedure of an unallocated module is called, the module is automatically allocated (see below) by the runtime system, and the data section then created becomes the bound data section; 2) When the system procedure "bind" explicitly allocates a module, the new data section becomes the bound data section.

One of the module-descriptor fields points to the bound data section for the module. This provides the mechanism for accessing the bound data section fields when they are referenced without a pointer variable.

A-pointer to a data section provides a "handle" on a module since it identifies the data section and, indirectly, the control section. The identification of the control section occurs via the data section's dscrLink, which points to the module descriptor, which in turn specifies information such as the location (in memory or on a file) of the control section.

Module Descriptor

A module descriptor is allocated for each allocated module. It associates a module name with a control section and (bound) data section, gives information about the memory and file location of the control section, and specifies enough information about the data sections to perform garbage collection without having the control section resident in memory. It also contains a pointer to the bound data section and a link for the "modDscrList", which links together all module descriptors.

MODULE DESCRIPTOR

garbLink	ADDRESS
NULLPOINTER	POINTER
chunkPtr	POINTER
chunkBits	BITS
ctrlSec	ADDRESS
oldCtrlSec	ADDRESS
handle	mdFileHandle
startPage	INTEGER
numPages	INTEGER
age	INTEGER
modLink	POINTER(modDscr)
itfClassDscr	POINTER(cl assDscr)
modBits	BITS
modName	LONG BITS
numPtrsStrs	BITS

chunkPtr points to the'bound data section.

chunkBits indicates that the chunk is a module descriptor and gives the total size of a data section for the module.

ctrlSec is the address of the control section, if it is in memory; otherwise it is NULLPOINTER. Whenever a control section is swapped out of memory, oldCtrlSec is first set to ctrlSec. Later, when it is swapped back in, a search is made through the return address stack to update all pending return addresses to the module. oldCtrlSec serves two purposes during this search: it, along with the size of the control section as contained in the module descriptor, allows the identification of which return addresses refer to the old location of the control section; and the displacement(oldCtrlSec,ctrlSec) is the amount by which each return address needs to be displaced.

handle is the machine-dependent "handle" (5.5) identifying the file that contains the control section. If handle is Zero, then the module has been brought into memory from an individual file' and has not yet been copied out to the swap file. Once it is copied to the swap file, the handle will be set to that of the swap file. The handle of modules obtained from a library (4.5) references the library file.

startPage gives the starting page number of the control section in the file which contains it (4.5). For example, a control section may start on the 10th page of a library file.

numPages gives the size of the control section in pages.

age is used by the memory management procedures to perform its "throw away oldest" swapping algorithm (4.4) and gets incremented once for each call or return to the control section.

modLink is used to link together on the modDscrList all module descriptors.

itfClassDscr points to a class descriptor which describes the interface record. This class descriptor is built from information in the "class information" part of the module's descriptor section. It is Zero if the module has no interface record.

modBits are used to indicate certain special conditions about the control section such as that it has a FINAL procedure, or that the control section must be permanently resident.

modName is the long bits encoding of the module name.

numPtrsStrs encodes the number of non-interface pointers and strings in a data section for the module (needed by the garbage collector).

Many of the fields of a module descriptor are copied from the module's descriptor section when the module is first allocated.

Module allocation

A module is allocated as the result of one of the following:

1) Automatically the first time one of its interface procedures is called.

2) When the system procedure "bind" is called to explicitly allocate the module's bound data section.

3) When the system procedure "new" is called to create a new instance of the module (i.e. a new data section).

The allocation of a module involves the following:

1) Bring the control section into memory, if necessary.
2) Allocate and initialize the module descriptor, if necessary.
3) Allocate and clear the data section.
4) Call the initial procedure, if there is one.

A module may be disposed, in which case the space occupied by its control section (if in memory), data sections, and module descriptor is reclaimed. Individual data sections of a module may also be disposed without disposing the entire module.

A module may also be unbound, which executes the final procedure (if any), disposes of the bound data section, and breaks the linkage of all modules which have established linkage to m. This involves visiting all data sections and clearing each "module pointer" which references the bound data section.

Intermodule Communication

MAINSAIL contains its own notion of inter-module communication: there is no reliance on a machine-dependent linkage system. Modules communicate via INTERFACE FIELDS, which are the variables and procedures of each module that the programmer declares to be accessible from other modules.

In the data section for each module m there is reserved one "module pointer" for each module that m refers to. When a data section for m is created, each of these module pointers is Zero until linkage is established between m and the corresponding module. When linkage is established, the module pointer is pointed at the module's data section.

Module m may at any time call an interface PROCEDURE in another module n. However, m can access an interface VARIABLE in n only if linkage has already been established from m to n (i.e., only if the module pointer in m's data section reserved for n points to n's data section). The MAINSAIL language design actually allows for automatic linkage to all interface variables, just as it does for interface procedures. But the implementation overhead for interface data access, in both execution efficiency and amount of generated code, could significantly degrade program execution, and hence the current implementations do not provide automatic linkage for data access.

Linkage can be established between m and n in one of three ways:

1) LINK-BY-PROGRAM-STRUCTURE

Whenever a module m is allocated, MAINSAIL automatically establishes linkage between it and all other modules that it references which are currently allocated.

It goes through the module name table in m's descriptor section, getting the names of each module that is referenced by m. It compares each name with the module names in the modDscrList module descriptors. If a match is found, then the module with that name (say n) has already been allocated, and the appropriate module pointer in m's data section is set to point to n's bound data section.

2) LINK-BY-CALL

Whenever m makes an intermodule call to n, MAINSAIL automatically establishes linkage from m to n if it hasn't already been established. That is, a call from m to one of n's interface procedures automatically initializes m's pointer to n (if it is not already initialized). This is explained below under “Interface Procedure Access".

3) LINK-BY-BIND

If m explicitly "binds" n by calling the system procedure bind with n as an argument, then linkage is established from m to n. That is, link-by-bind provides a means of explicitly establishing linkage in case neither of the previous "automatic" methods is sufficient.

The list of module descriptors is searched to see if n has already been allocated. If not, then MAINSAIL allocates it. In any case, the pointer in m's data section reserved for n is then set to point to n's bound data section.

Interface Procedure Access

When an intermodule call is made from module m to a procedure in module n, a machine-dependent routine referred to as the "intermodule caller" is given control. It is passed the offset in m's data section to the module pointer reserved for n (this offset is assigned by the compiler) and the offset in the entry vector of n's code section for the procedure to be called.

If the module pointer is non-Zero, then linkage has already been established.

If the module pointer is Zero, then the intermodule caller automatically allocates n (if it has not already been allocated), and sets the pointer in m's data section reserved for n to point to n's data section.

The procedure to be called can then be accessed indirectly via the pointer to n's data section. The data section contains a pointer to n's module descriptor, and the module descriptor specifies the location of the control section (which may need to be brought into memory). The given offset in the entry vector of the control section is then used to give control to the appropriate procedure.

Interface Variable Access

When m accesses an interface variable in module n, the generated code examines the module pointer to n's data section from m's data section. If it is Zero, then linkage has not yet been established from m to n, and MAINSAIL will report a runtime error (the design allows automatic linkage to be provided at this point, as for procedures, but it has not been implemented for code-generation efficiency reasons). Otherwise, the value of the interface variable is obtained by offsetting the appropriate amount from the base of n's data section. Thus access to an interface field, m.f, is essentially identical to access to a record field, p.f, where p is an own pointer.

3.3 ARRAY IMPLEMENTATION

Arrays are represented by a two-chunk structure: a descriptor chunk and an element chunk. The descriptor chunk has the following form:

```
                ARRAY DESCRIPTOR

      +--------------------------------+
      | garbLink                       |        ADDRESS
      |------------------------------  |
      | NULLPOINTER                    |
      |==============================  |
      | chunkPtr                       |        POINTER
      |--------------------------------|
      | chunkBits                      |        BITS
      |------------------------------  |
      | trueOrigin                     |        POINTER
      |------------------------------  |
      | arrayName                      |        STRING
      |------------------------------  |
      | arraySize                      |        INTEGER
      |------------------------------  |
      | arrayType                      |        INTEGER
      |------------------------------  |
      | dimension                      |        INTEGER
      |------------------------------  |
      |                                |
      | bounds descriptors             |
      |                                |
      +--------------------------------+
```

chunkPtr tells the virtual origin of the array (the address of the possibly non-existent element with all subscripts equal to 0). This value is used in the calculation of the address of subscripted variables.

chunkBits identifies the chunk as an array descriptor and specifies the size of the descriptor.

The trueOrigin points to the element chunk: arrayName gives the name of the array: arraySize gives the size of the element chunk; arrayType tells the data type of the array's elements; and dimension gives the number of dimensions.

Bounds descriptors give the information necessary for performing bounds checking and for element access. Each bounds descriptor gives the lower bound, the upper bound, and the multiplier used in the calculation of array element locations.

Row major storage is utilized. The address of the element with indices xl,..., xn (where n is the dimension of the array) is

```
    displace(virtualOrigin,m1*x1 + . . . + mn*xn)
```

where displace(a,b) displaces address a by b address units, and mi is the multiplier for the ith dimension. mi is equal to

```
    w * s[i + 1] * . . . * s[n]          (i < n),
    w                                    (i = n)
```

where w is the number of address units per array element, and s[i] is the number of elements in the ith dimension.

The element chunk contains the array elements. Its size is determined by the total number and type of the array's elements. The form of an element chunk is as follows:

```
              ARRAY ELEMENT CHUNK

    +------------------------------+
    | garbLink                     |        ADDRESS
    |------------------------------|
    | dscrLink                     |        POINTER
    |------------------------------|
    |                              |
    | elements                     |
    |                              |
    +------------------------------+
```

The dscrLink points to the array's descriptor chunk.

An array variable is simply a pointer to the array descriptor. Arrays may be assigned, passed as parameters, and compared. In each case, only the pointer to the array descriptor takes part in the operation (rather than the elements of the array). Thus a single array descriptor may be pointed to by many array variables.

In contrast with some languages, array allocation is not tied to syntactic structure. Arrays are allocated dynamically by explicit calls to the MAINSAIL array-allocation procedure.

When an array is allocated, its descriptor and element chunks are created and the array variable specified to the allocation procedure is pointed at the array descriptor.

Arrays may also be explicitly deallocated when they are no longer needed. When an array is deallocated, its descriptor and element chunks are put on free chunk lists (see 4.6) specifiying that their storage is available for reuse.

3.4 CLASS AND RECORD IMPLEMENTATION

Classes provide a description of a records' fields. At compiletime they provide the syntactic mechanism for defining and referring to the fields of records. At runtime, the class descriptor provides information necessary for record allocation and garbage collection. All collectible objects (string descriptors and pointers) within records of the class must be accessable to the collection process and the class descriptor provides the necessary information.

```
                 CLASS DESCRIPTOR

     +--------------------------------+
     | garbLink                       |        ADDRESS
     +--------------------------------+
     | NULLPOINTER                    |
     +================================+
     | chunkPtr                       |        POINTER (classDscr) .
     +--------------------------------+
     | chunkBits                      |        BITS
     +--------------------------------+
     |                                |
     | field descriptors              |
     |                                |
     +--------------------------------+
```

The MAINSAIL runtime system maintains a list of all class descriptors. The chunkPtr field of descriptors is used to link them together.

chunkBits indicates that the chunk is a class descriptor and also gives the size of chunks for records of this class. The field descriptors tell the data types of all the fields of records of the class, in order.

Each record is an instance of a particular class. A record chunk contains the values of the fields of the record:

```
                   RECORD

     +-------------------------------+
     | garbLink                      |        ADDRESS
     |-------------------------------|
     | dscrLink                      |        POINTER
     |-------------------------------|
     |                               |
     | fields                        |
     |                               |
     +-------------------------------+
```

The **dscrLink** points to the record's class descriptor. It is used by the garbage collector to access the class descriptor to determine the characteristics of the record.

Records are dynamically allocated by explicit calls to **newRecord,** the record-allocation procedure. A class descriptor is created whenever a **newRecord** is performed on a class that does not yet have a class descriptor. Class descriptor and record allocation are implemented as follows.

As introduced in 3.2' there is a pointer allocated in the data section of a module for each class c for which new(c) occurs in the module. These "class pointers" are reserved for pointing to the appropriate class descriptors. They are each initially NULLPOINTER.

The compiler generates the offset to the appropriate class pointer as the argument to **newRecord.** Whenever a **newRecord** is done, the record allocation procedure examines the class pointer at the given offset. If it is NULLPOINTER, then the allocator searches the list of class descriptors to see whether or not one already exists for the class. If not, it creates one by using the same offset to access the class table in the module's descriptor section to obtain the information necessary to create the descriptor. This new descriptor is then added to the list of class descriptors. Once the class descriptor has either been found or created, the record allocator then initializes the class pointer to point to the descriptor.

In any case, a new record of the given class is then allocated using the information in the class descriptor.

3.5 STRING IMPLEMENTATION

Certain memory pages, collectively referred to as "string space", are used to store the characters of strings. String space is described in 4.0.

A string variable is implemented as a "string descriptor" that gives the currentlength (number of characters) of the string and the location in string space of its starting character. The assignment of one string to another only involves assigning the string descriptor' not copying the characters. Thus the same sequence of characters may be referenced by many string descriptors.

The text of all the string constants used in a module is allocated in the descriptor section. If the MAINSAIL system were static, then the characters in the descriptor section could be directly referenced by string variables. But since control sections may be dynamically swapped and moved in memory, the descriptor section is a moving target, and hence is not appropriate for string-variable reference.

In general, whenever a string constant is used, its characters are copied from the descriptor section into string space, and a string descriptor is created for the copied text.

MAINSAIL avoids copying string constants in two specific situations, for efficiency: when a string constant is written to a file or to a terminal, the text is copied directly from the descriptor section to the file or terminal. Additional optimizations could be added, for example a comparison involving a string constant need not copy the text into string space.

3.6 PROCEDURE CALL, ENTRY, EXIT, RETURN

Due to the nested behavior of procedure calls, stacks are natural data structures to be used for the allocation and deallocation of procedure parameters' return addresses, and local variables.

Conceptually' a MAINSAIL implementation maintains two stacks: one for local variables (including procedure parameters), and one for return addresses. In reality, there may be from one to three stacks for local variables, depending on the particular code generation strategy adopted. The reason for more than one local-variable stack is to facilitate the processing of pointer and string variables by the garbage collector. That is' there could be three stacks (one for pointers' one for strings' and one for all other data types), two stacks (one for "collectible" items -- pointers and strings -- and the other for all other items), or a single stack containing all data types. In each case, a machine-dependent strategy is used to identify accessible pointers and strings during garbage collection.

For each stack, there is a pointer to the "top" of the stack (the location of the last element pushed onto the stack). All elements on the local-variable stack(s) are referenced as displacements from the current top of the stack. The displacements are assigned by the compiler. Since procedures cannot be statically nested, and since all parameters are passed by value' only the local variables for the current procedure can be accessed on the stack. i.e., there is no "up-level" addressing.

The RETURN STACK contains a sequence of return addresses, pushed upon nested procedure invocations. In addition, every time an intermodule call is made, a pointer to the base of the caller's data section may be pushed onto this stack (this is a machine-dependent decision).

Control sections may sometimes be moved in memory. Whenever they are, the return addresses must be correspondingly adjusted. The information required to properly adjust the addresses can be determined via the data sections; each data section points to the corresponding module descriptor, which specifies the former and current locations of the module's control section. From this information, the correct new return addresses can be calculated.

The method by which the runtime system distinguishes between return addresses and pointers to data sections saved on the return stack is machine-dependent.

The LOCAL VARIABLE STACK (or stacks) is a sequence of "procedure frames", each consisting of the parameters and locals for a specific procedure. When a procedure is invoked, its parameters are pushed, and then space is allocated for its local variables. All parameters of the same type (USES, PRODUCES, or MODIFIES) are allocated together. The exact order in which they are allocated (e.g. uses-modifies-produces or modifies-uses-produces) depends on the implementation.

SAMPLE LOCAL VARIABLE STACK

```
                        .                       .
                        |                       |     |
                        |  will be              |     |
                        |  local variables      |     |    callee's procedure
                        |-----------------------|     |--  frame being built  .
  current   ----->      |  produces  parameters |     |    during procedure call
  top of                |-----------------------|     |
  stack                 |  uses parameters      |     |
                        |-----------------------|     |
                        |  modifies  parameters |     |
                        |=======================|  ---+
                        |                       |     |
                        |  local variables      |     |
                        |                       |     |
                        |-----------------------|     |
                        |  produces  parameters |     |--  caller's
                        |-----------------------|     |    procedure frame
                        |  uses parameters      |     |
                        |-----------------------|     |
                        |  modifies  parameters |     |
                        |=======================|  ---+
                        |                       |

                           older procedure
                           frames
                        |                       |
                        +-----------------------+
```

Local Procedure Invocation

Local procedure invocation is described below. For the sake of discussion, an implementation with a modifies-uses-produces order of parameter pushing is described. Note that this description is target-independent, and hence the details of its implementation will vary among implementations. Generally speaking, MAINSAIL is quite efficient with regard to procedure entry and exit, with often just one or two instructions upon entry and exit to allocate and deallocate the procedure frame.

A local procedure call involves the following steps:

1) Push any modifies parameters onto the proper stack(s).
2) Push any uses parameters onto the proper stack(s).
3) Push the return address onto the return stack (often part of the next step).
4) Transfer to the target procedure.

Procedure entry involves the following steps:

1) Allocate space on the stack(s) for any produces parameters.
2) Allocate space on the stack(s) for any local variables.

Procedure return involves the following steps:

1) Adjust the stack(s) to "deallocate" the locals.
2) If there are any uses parameters, but no produces parameters, then adjust the stack(s) to "deallocate" the uses parameters.
3) Pop the return address off the return stack, and return to the calling procedure.

Upon return:

1) Pop any produces parameters.
2) Adjust the stack(s) to deallocate the uses parameters (if not already done).
3) Pop any modifies parameters.

Produces parameters are assigned locations adjacent to the incoming parameters, as shown in the above diagram. If the procedure is typed, then the result is treated as if it were an additional produces parameter of the proper type.

On most conventional machines utilizing registers, the "rightmost" (as written in the argument list) produces or modifies parameter (if any exist) is put in a designated register prior to the return from the procedure. If the procedure is typed, then the result is taken as the "rightmost" produces parameter. Thus if a procedure is returning any values to the point of call, one of them will be in a register. The net effect is that step 2 of the return sequence almost never occurs, since the called procedure is able to adjust the stack(s) to remove all uses parameters.

Intermodule Procedure Invocation

An intermodule procedure call involves a context switch from the module containing the call to the module being called. Again, for the sake of discussion, an implementation with a modifies-uses-produces order of parameter pushing is described.

An intermodule procedure call from module m to module n involves the following steps:

1) Push any modifies parameters.
2) Push any uses parameters.
3) Push the return address onto the return stack and transfer to the intermodule caller, with the target procedure index (in the entry vector) and the target module pointer p (if the call has the form p.f) or offset to the module pointer m (if the call has the form m.f) as parameters. The intermodule caller carries out the remaining steps.
4) Push db (the base of m's data section) onto the return stack.
5) Let p represent the module pointer allocated in m's data section for referring to n's data section. If p is NULLPOINTER, then allocate n (if it has not already been allocated) and initialize p to point to n's data section.
6) Bring n's control section into memory if necessary.
7) Update n's age counter.
8) Set db to p.
9) Set cb (base of m's control section) to base of n's control section.
10) Transfer to the target procedure via the entry vector.

Procedure entry is the same as for a local procedure call.

Procedure return involves the following steps:

1) Pop the return address off the return stack.
2) Bring m's control section into memory if necessary.
3) Update m's age counter.
4) Pop the saved data base off the return stack and into db.
5) Set cb to the base of m's control section.
6) The remaining steps are the same as for a local procedure call.

SECTION 4 MEMORY MANAGEMENT

The MAINSAIL runtime system handles all aspects of memory initialization, organization, and maintenance during the execution of a MAINSAIL program. These runtime functions are described in this section. These functions are at a "lower level" than those described in the previous section.

4.1 VIEW OF MEMORY

MAINSAIL assumes that it can use a single contiguous portion of memory which it organizes into pages of fixed size corresponding to operating-system dependent pages:

```
pageNumber
                  +----------------+
     0            |                |
                  +----------------+
     1            |                |
                  +----------------+
     2            |                |
                  +----------------+
     3            |                |
                  +----------------+
                  |                |


                  |                |
                  +----------------+
     n            |                |
                  +----------------+
```

Each page is one of five types:

Free pages	pages which are not currently being used.
Control pages	control sections.
Static pages	storage which cannot be moved once allocated. A typical use is for file buffers (5.3).
String pages	string characters,
Chunk pages	chunks: records, arrays, data sections, descriptors, and free chunks

MAINSAIL maintains a "page map" containing one element for every page of memory. Each page map element contains an integer code telling the type of the corresponding page. For every group of adjacent memory pages that comprise a unit (e.g. for a control section), the page map element for the first page contains its integer code, and the continuation pages are specified by having their corresponding map elements contain the negative of this code.

Chunk, static, and string pages are collectively referred to as "data pages". Control pages are allocated at one end of memory. Data pages are allocated at the opposite end of memory. Thus control and data pages grow towards each other, with free pages in between (free pages will in general also be located among the data and control pages):

```
+-----------------------+
|                       |
|       data  pages     |
|                       |
+-----------------------+
|                       |
|                       |
.                       .
.       free  pages     .
.                       .
|                       |
|                       |
+-----------------------+
|                       |
|     control pages     |
|                       |
+-----------------------+
```

No control page may be allocated below the highest data page, and correspondingly, no data page may be allocated above the lowest control page. What happens when there isn't enough room at the appropriate end of memory for pages that need to be allocated is described in section 4.4.

4.2 KERNEL

The "kernel" is a basic runtime module which is always resident. It is the only module which is "linked" by the host system linkage editor. This is necessary since it is the module which initially gains control from the operating system, and thus cannot itself be brought into memory by MAINSAIL.

Since the kernel is permanently resident, an attempt has been made to keep it as small as possible. The kernel contains only those functions which cannot exist in a swapped control section. For example, page maintenance is the most primitive form of memory management' and the necessary algorithms are contained in the kernel. The intermodule calling mechanism and data section initialization also reside in the kernel.'

Both the memory management procedures and the data section initialization require i/o support. The kernel must contain the procedures to open, close, read and write host system files containing control sections. It must also be capable of opening the "swap file" (4.5).

4.3 EXECUTION INITIALIZATION

The mechanism which obtains control from the host system and initiates the execution of MAINSAIL is termed the "runtime initializer".

The kernel and the runtime initializer are first brought into memory in an implementation-dependent manner. This process also specifies desired values for "configuration parameters" that must be initialized for each implementation. These parameters include:

1) initMaxPageAdr -- the address of the memory page that is to initially be the highest page utilized.

2) pageLimitAdr -- the address of the last word of the absolute highest memory page that could be utilized, should the initial memory allocation be insufficient.

3) initStrPages -- the number of string space pages to allocate initially.

4) strPageInc -- the (minimum) number of string space pages to add whenever the current allocation is insufficient.

5) swapFileName -- the name of the file to be created for storing swapped-out control sections (4.5). It is created the first time a module is swapped out, which rarely occurs on machines with sufficient address space.

6) chunkPageInc -- the number of new chunk page allocation requests that can be made before a garbage collection is to be triggered (4.6).

7) sizes for machine-dependent stacks (3.6).

Once the kernel and the initializer are brought into memory, the initializer is executed. It must perform the following functions:

1) Get the initial memory allocation from the operating system, asking for enough pages to have the initial maximum page address be initMaxPageAdr.

2) Allocate any stacks.

3) Allocate the kernel's module descriptor and data section.

4) Allocate the pageMap and the chunkTbl (containing lists of free chunks, described in 4.6) on static pages.

5) Bind the garbage collector module, so that its data section is already allocated should a garbage collection occur.

6) Open **cmdFile** and **logFile** to the terminal.

7) Invoke the top-level kernel procedure, which identifies MAINSAIL to the user, and initiates the execution of the module subsequently specified by the user. Note: a means is provided whereby the **runtime** initializer can contain the name of the first user module to be given control.

Since the initializer code is only executed once, it is desirable to reclaim the memory it uses for other purposes. This effect can be obtained by having it initially brought into memory at a location which will be specified in the page map as being free, and thus which will subsequently be used for other purposes.

String space is automatically allocated upon demand.

4.4 PAGE ALLOCATION

As mentioned in **4.1**, a page map is maintained to keep track of the current use or availability of each page. When new pages of a given type need to be allocated, this map is used to select free pages, and then the corresponding elements of the map are set to the code of the appropriate page type.

This section describes the allocation of memory pages. Allocation and deallocation of components of chunk and static pages are described in sections 4.6 and 4.7. String space is described in 4.8.

The allocation of control and data pages are handled separately.

1) Control Page Allocation

Whenever a control section of size n needs to be brought into memory, n contiguous free pages above the current highest data page must be found for it. There are three possibilities:

a) If there exist n contiguous free pages, then they are used for the control section.

b) Otherwise if there exists at least n total free pages (again, above the highest data page), then control pages are moved in memory to make n contiguous pages. The compaction algorithm determines and'utilizes the compaction which minimizes the number of pages which must be moved.

c) If **there are** not n total free pages above the highest data page, then an attempt is made to swap out control sections until there are n total free pages.

The swap out algorithm searches the **modDscrList** (see 3.2) to find the oldest (according to the module descriptor age field) swappable module with a memory-resident control section. If one exists, then its control section is swapped out of memory **(4.5)**, and the pages it had occupied are marked as free. The module swapper continues swapping out the oldest control sections until either n total free pages are thereby obtained, or there are no more modules that can be swapped. In this latter case, the operating-system-dependent procedure **mdMorePages** (6.2) is called to try to obtain enough more memory pages from the operating system to have a total of n free pages. If **mdMorePages** fails, then an "Insufficient memory" message is given, and execution terminates.

If at least n pages become free as a result of swapping, then control pages are compacted if necessary to obtain n contiguous pages.

MAINSAIL will continue to execute as long as there is room for the data, the kernel, and the largest control section being utilized, though it will spend most of its time swapping modules into memory if memory space is too scarce.

2) Data Page Allocation

Whenever n data pages need to be allocated, a search of the **pageMap** is made for n contiguous free pages below the current lowest control page. There are two possibilities:

a) If there exist n contiguous free pages, then they are used.

b) Otherwise the lower bound of control space must be moved up enough to leave n contiguous free data pages. Note: there is currently no data space compaction, though the design allows for it.

If there are at least n total free pages above the current highest data **page**, then as many control pages as are necessary are moved up in memory (compacted) until there are n free pages between the current highest data page and the new lowest control page.

If there are not n total free pages above the current highest data page, then an attempt is made to swap out control sections to obtain enough free pages. If the swap out algorithm, as described above, fails to obtain enough pages, then **mdMorePages** is called. If it fails, then execution terminates. Otherwise, control space is compacted as described above.

4.5 MODULE SWAPPING

Control sections may reside singly in files, or may be grouped together into a "library" file. Control sections residing alone in a file are handled differently from those residing in library files.

Whenever the space for a memory-resident control section needs to be reclaimed as a result of the situations described in 4.4, a control section residing in a library file is not physically swapped out. Rather, the memory pages it occupies are marked as free, and the module descriptor fields are adjusted to specify that the module is no longer in memory. Subsequent requests for the control section will again obtain it from the library file, which stays open throughout execution until it is explicitly closed by the program.

The runtime system opens a file containing a single control section only once during a MAINSAIL execution. The first time that such a control section (one residing in a non-library file) needs to be swapped out, a file called the "swap file" is created. The name for this file is specified as a configuration parameter (4.3). The control section is written to the swap file, the module descriptor fields are adjusted to indicate the control section resides in the swap file, and subsequent requests for the control section will obtain it from this file. All control sections that need to be swapped and that reside in non-library files are similarly written to the swap file. Thus, later references to such a control section can simply read it out of the swap file instead of reopening the control section's original file. In a sense, the swap file is like a dynamically created library file. The swap file may be created on a special high speed device, if available (e.g. bulk memory), so that swap time will be minimized.

If a library file is closed, any of its allocated modules whose control sections are not in memory are automatically brought into memory, and disassociated with the library. The next time such a module is swapped out, it will be copied to the swap file, as described above. If all active modules have been obtained from still-open libraries, then no module is ever physically copied from memory. This is the case with all the MAINSAIL system modules (which are contained in a single library) and all the compiler modules (which are contained in one library of the target-independent modules and one for each code generator). Thus the user is encouraged to put the modules of a multi-module program into a library, to avoid use of the swap file.

4.6 CHUNK ALLOCATION AND DISPOSAL

As described in Section 3.1, records, arrays, data sections, and descriptors are represented by chunks. Whenever any of those data structures is disposed (freed) by a program (or by the garbage collector), its chunk can be reused by later chunk requests.

In order to indicate that the chunk is no longer being used, it is formatted into a "free chunk", and then placed on a list of free chunks. The following diagram gives the general format of a free chunk:

```
              FREE  CHUNK

    +-----------------------+
    | garbLink              |        ADDRESS
    |-----------------------|
    | NULLPOINTER           |        POINTER
    |■■■■■■■■■■■■■■■■■■■■■■■|
    | chunkPtr              |        POINTER
    |-----------------------|
    | chunkBits = 0         |        BITS
    |-----------------------|
    | free chunk size       |        INTEGER
    |-----------------------|
    |                       |
    |                       |
    | free space            |
    |                       |
    |                       |
    +-----------------------+
```

As for all chunks, the first two fields are the chunk header. The first is used by the garbage collector. The second is NULLPOINTER to indicate that the chunk is self-descriptive. The **chunkPtr** field links the chunk onto the free-chunk list, and **chunkBits** = 0 specifies that the chunk is a free chunk.

The next field tells the size of the free chunk body (i.e./excluding. the chunk header). It is in a field by itself (rather than combined with the **chunkBits** code' as for other self-descriptive chunks) since free chunks might be large (e.g.' from a large disposed array) and hence the size might not fit in the information field of **chunkBits**, which is guaranteed to be only 12 bits.

The rest of the chunk is cleared before the chunk is added to the free list.

The "free list" is actually a group of linked lists. For each size chunk (up to a maximum called **maxSmallChunk**), there is a list of free chunks of that size. One additional list is maintained (the "big free chunk list"), for chunks larger than the maximum. The addresses of the first chunks of all the free lists are maintained by the **runtime** system in what is referred to as the **"chunkTbl"**.

The list of all free chunks of size 5, for example, could be pictured as follows (where the box on the left represents the **chunkTbl** element referencing the first chunk on the list of chunks of size 5):

```
                   +-------------+          +-------------+
                   | NULLADDRESS |          | NULLADDRESS |
                   |-------------|          |-------------|
                   | NULLPOINTER |          | NULLPOINTER |
   +----+          |=============|          |=============|
   | --|--->|          ---|----------->|          ---|----------> ...
   +----+          |-------------|          |-------------|
                   | 0           |          | 0           |
                   |-------------|          |-------------|
                   | 5           |          | 5           |
                   |-------------|          |-------------|
                   | 0           |          | 0           |
                   |-------------|          |-------------|
                   | 0           |          | 0           |
                   +-------------+          +-------------+
```

This free list structure allows for the rapid recycling of disposed chunks. Whenever a new chunk needs to be allocated, there may already be one of the appropriate size in one of the free lists. If there is an available free chunk of the exact size, then that chunk is taken off its free list and used. Otherwise,if there is a large enough free chunk on the "big free chunk list" (first fit algorithm), then it is split into two chunks: whatever is not required for the new chunk is put back on the free list -- on the appropriate "small free list" if it's small enough and the list is not empty, and on the "big free chunk list" otherwise. (Note: it is insisted that the small list not be empty to avoid putting chunks onto small free lists that do not correspond to record sizes being used by the program).

If no free chunk is found by the above method, then the minimum number of needed contiguous pages are allocated as chunk pages (note: a garbage collection may first occur at this point, as described below). A chunk of the required size is allocated from these pages, and the remainder of the (last) page is placed on the appropriate free list.

A configuration parameter called **chunkPageInc** (4.3) is used to trigger a garbage collection every **chunkPageInc'd** time that new chunk pages are allocated. The collection will compact string space and free all chunks that are no longer referenced' as described in 4.9. In this case, the chunk allocation algorithm is re-entered to see if a chunk of the appropriate size has been freed by the collection.

4.7 STATIC ALLOCATION AND DEALLOCATION

Static pages are used for storage which cannot be moved once allocated. A program can request an entire static page, or a specified portion of a static page' referred to as "scratch space", though such requests are relatively rare, especially from user modules.,

The MAINSAIL **runtime** system maintains a list of currently unused portions of static pages. Each unused portion of a static page is referred to as a "scratch block". The first storage unit of a scratch block specifies the address of the next scratch block on the list, and the second storage unit specifies the size (number of storage units) of the scratch block.

When scratch space of size n is requested by a program, the list of scratch blocks is first searched for one of size n or greater. If there is such a scratch block, then it is taken off the list. n storage units of the scratch block are then allocated for the scratch space, and whatever portion remains (if any) is put on the list of scratch blocks by the appropriate setting of its first two storage units.

If no scratch block is large enough to satisfy the scratch space request, then a new static page is allocated. n storage units of the new page are allocated for the scratch space' and whatever portion of the page is left over (if any) is put on the list of scratch blocks.

When a program no longer needs some scratch space, it can explicitly dispose it. When scratch space is disposed, it is put on the list of scratch blocks. If this results in a page containing nothing but scratch blocks, then this page is disposed (marked in the page map as free).

4.8 STRING SPACE

"String space" is a collective term referring to the pages of memory used for storage of the characters of strings.

String space consists of one or more "string blocks". The first time string space is needed, an initial string block consisting of **initStrPages** (4.3) pages is allocated. Additional blocks, of size at least **strPageInc**, are dynamically allocated as needed.

To facilitate garbage collection, as described in 4.9, each block is conceptually divided into fixed-size sections called "parcels", and each parcel is divided into "pieces", each of which consists of a specific number of characters. Typically' a parcel consists of n pieces where n is the number of bits per word. For example, an implementation on a machine with 32-bit words would likely choose a parcel to be 32 words' and a piece to be one word, consisting of 4 characters.

```
     STRING BLOCK

  +---------------+ -----+
  |     piece     |      |
  |---------------|      |
  |     piece     |      |
  |---------------|      |
  .               .      |--- parcel
  .               .      |
                         |
  |---------------|      |
  |     piece     |      |
  |===============|------+
  |     piece     |      |
  |---------------|      |
  |     piece     |      |
  |---------------|      |
  .               .      |--- parcel
  .               .      |
                         |
  |---------------|      |
  |     piece     |      |
  |===============|------+
  .               .
  .               .
  .               .
  .               .
  |===============|------+
  |     piece     |      |
  |---------------|      |
  |     piece     |      |
  |---------------|      |
  .               .      |--- parcel
                         |
                         |
  |---------------|      |
  |     piece     |      |
  +---------------+------+
```

Associated with each block is a record specifying information about the block, including the charadr of its first character, the size of the block, the number of free character positions remaining in the block, and the first free character position in the block. A list of the records for all currently-existing string blocks is maintained.

One of the blocks is always considered to be the "current" one. When a string of n characters is to be added to string space, the characters are added starting at the first free position in the current block, if there is enough room for them in that block. If there is not enough room in the current block, then the list of string blocks (or, more precisely, the list of records associated with the blocks) is examined in an attempt to find a string block with at least n free characters. If such a block is found, the block becomes the current block, and the characters are put into it.

If there are no string blocks with enough free character positions, then a garbage collection is done, as described in 4.9. Collection disposes of characters that are no longer referenced by any string descriptors, and compacts string space so that all the characters that are still being referenced are pushed down to one end of string space.

If the garbage collection does not free a significant number of characters (in particular, if it does not free at least n characters), new data pages are allocated to be used as a new string block. The number of pages allocated is the maximum of strPageInc and the number of pages required to contain n characters.

4.9 GARBAGE COLLECTION

A number of languages do not provide garbage collection. MAINSAIL does primarily because of the generality of the string data type. Explicit string disposal would not only be inconvenient from the programmer's viewpoint, but also would raise implementation questions as to how freed characters could be marked as available and used for subsequent strings.

In addition to string collection, MAINSAIL's garbage collector collects pointers, or more precisely, data structures (chunks) referenced by pointers. It is not necessary that they be collected, since explicit disposal is provided, and thus could be required in order to reclaim their space. There are basically two reasons that MAINSAIL does chunk collection. One is that, by guaranteeing collection when necessary, MAINSAIL frees the programmer from being required to know precisely when each record, array and data section is no longer referenced, and thus may be freed. In reality, most programs should be able to determine this information (for example, the MAINSAIL compiler and runtime system explicitly dispose of all the data structures they use).

The second reason for having chunk collection is that string collection requires examination of all used chunks anyway (see below), so there is not much overhead in also marking the used chunks and freeing unmarked ones.

. Garbage collection involves the following steps:

1) Mark all accessible characters in string space, and all accessible chunks. A chunk is accessible if any accessible pointer references it, and string character is accessible if any accessible string descriptor references it.

2) Compact string space.

3) Adjust all string descriptors.

4) Find all unaccessible chunks and rebuild the free chunk lists.

The additional conceivable steps of compacting chunk space and adjusting all pointers are not currently implemented.

1) The marking of chunks and of characters in string space

The reason that string collection requires examination of all chunks currently being used is outlined by the following. In order to do string collection, all string descriptors must be found. To find all string descriptors, all data structures (chunks) that could contain string descriptors must first be found.

A recursive process following all pointers on the return and appropriate local variable stack (3.6) is all that is required to examine all used chunks. That is, for each pointer on each stack, the chunk pointed at by the pointer is marked as being used, and the characters referenced by all the string descriptors in the chunk are also marked as described in (2) below. The pointers in each such chunk are followed to mark the chunks they reference and the string descriptors within those chunks, and so on.

The "marking" of a chunk is done via the **garbLink** field, which is the initial field of all chunks, as introduced in 3.1. For unmarked chunks, this field is Zero. To mark a chunk, the **garbLink** field is set to the address of the previous chunk that was marked (or to an initial non-Zero "dummy" value for the first chunk marked).

This method of marking thus performs two functions: marking all used chunks, and linking them all together to facilitate step 3, below.

The marking process also processes each string descriptor on the appropriate local variable stack.

2) The marking and compaction of string space

When the collector finds a string descriptor, as described above, the characters it refers to must be marked as being used. After all string descriptors have been found, string space is "compacted". That is, all the used characters are pushed down through string space as much as they can be (replacing unused characters), so that up to a certain point of string space, only used characters exist. Then all the remaining space in string space is free to be reused. Finally, all string descriptors are updated to refer to the adjusted character positions.

The marking of used characters is described by the following, and the updating of string descriptors is described in (3) below.

Corresponding to each character in string space there could be a single bit. To mark the characters referred to by a string descriptor, the corresponding bits could be turned on. But this would require a considerable amount of extra space (one bit for every character). String garbage collection does not have to be exact. That is, the purpose of collection is to free some memory, but it is acceptable to leave some unused characters in string space. Thus, the implementation maintains one bit for every piece (4.8). which in general consists of several characters. Suppose each piece consists of m characters. If any of those m characters is used, the corresponding bit gets turned on. When string space is compacted, groups of m characters whose bit is turned on get moved as a unit, since it is impossible to discriminate which of the m characters are actually used.

After the compaction of string space (described further in (3) below), all string blocks that have become empty are freed (their pages are marked in the page map as free). The "current" string block is set to be the one with the most free characters.

3) Adjusting all string descriptors

String descriptors cannot simply be updated as characters are moved. In order to do that, for each character moved, either all string descriptors which refer to it would have to be searched for, or a large data structure would have to be built up during the marking of string descriptors, to specify all the descriptors that refer to each character.

Thus, it is desirable to first compact string space, and then search memory to find and update all string descriptors in one pass. What is required is a specification of where each character has been moved.

Corresponding to each piece (i.e. to each of the bits described in (2) above), there could be an address specifying the new location of the first character in the piece. Then a string's new starting charadr could be calculated as the character address of the corresponding piece plus the number of characters in the piece that precede the first character of the string.

This data structure (consisting of one address for every piece) would consume an undesirable amount of space. So a reduced data structure has been implemented. It has an address corresponding to every parcel, instead of every piece. Thus the charadr of the first character of a string can be calculated as the new address of the corresponding parcel PLUS $pieceSize (6.1) times the number of active pieces (ones that have been marked) which precede the string within the parcel PLUS the number of characters in the piece that precede the first character of the string. This requires the ability to quickly determine the number of l-bits in a masked bits variable, where the l-bits correspond to the piece marks. The code to determine this count has been left machine-dependent since some machines have single instructions to count l-bits.

To update all string descriptors, the list of all used chunks that was built as described in (1) above is examined. All string descriptors in each chunk on the list are updated as just described. After each chunk's string descriptors have been updated, the chunk's garbLink is set to Zero so it is properly initialized for the next garbage collection.

4) Rebuilding the free lists

The lists of free chunks (4.6) are rebuilt as follows. All the individual chunks contained in all chunk pages are examined by making a linear scan of chunk space. Each chunk that has been marked is simply skipped over. Unmarked (free) chunks are added to the appropriate free list. However, adjacent free chunks are first combined if the resulting chunk size would be greater than **maxSmallChunk.** Any complete pages obtained in this manner are freed (they are marked in the page map as being free), and the remaining free chunks are placed on the appropriate list.

MAINSAIL employs a simple yet powerful view of a file system whose operations are guaranteed to be implemented for every MAINSAIL implementation. This section describes the file model employed, and then outlines how that model is implemented.

5.1 FILE MODEL

A file may be viewed as a sequence of cells numbered 0, 1, 2, ... If the file is a text file, then the cells are characters. Otherwise, the file is a data file, and the cells are storage units. A storage unit is the basic measure for the amount of memory required by the various data types -- e.g. it may represent a "byte" or a "word", though these concepts are not used by MAINSAIL.

For each file, there is a file pointer that is always positioned at one of the cells, referred to as the "current cell".

```
+-------+-------+-------+-------+-------+-------+-------+---
|       |       |       |       |       |       |       |
+-------+-------+-------+-------+-------+-------+-------+---
    0       1       2       3       4       5       6       ...
                    t
        file position pointer
```

When a file is created, the cell contents are initially undefined. When a file is opened, the current cell is number 0, i.e., the file pointer is positioned at cell 0.

The following primitive operations are defined on a file f:

v := read(f)	set v to the value in the current cell, then position the file pointer one cell to the right.
write(f,v)	write v into the current cell, then position the file pointer one cell to the right.
n := getPos(f)	set n to the number of the current cell.
setPos(f,n)	position the file pointer to cell number n. n must be greater than or equal to 0.
relPos(f,n)	position the file pointer n cells to the right (if n > 0) or left (if n < 0).

Each file has an end-of-file position (the last position in the file to which a value has been written). eof (end-of-file) is the condition: getpos(f) greater than or equal to end-of-file position.

5.2 IMPLEMENTATION MODEL

A simple approach to implementing the file model is to write a **machine-**dependent procedure for each of the primitive operations defined above. In fact, this was done in early implementations of MAINSAIL. But many operating systems do not have instructions corresponding to those primitives, so that algorithms had to be devised for maintaining file buffers. In writing the machine-dependent procedures for various implementations, it became clear that the algorithms needed to implement the procedures on different machines were conceptually quite similar; the ideas could be abstracted out from the machine-dependent code.

For these reasons, it was decided to add more structure to the file model, thereby increasing the amount of file-handling code that could be made machine-independent, and reducing the machine-dependent amount.

This expanded model adds the concept of a "window" into the file. Conceptually' a portion of the file is always visible through a "window". In order to reference a cell in the file, the window must first be associated with a portion (machine-dependent file "block") of the file that includes that cell.

```
                                              FILE

                                       +------- -------- +
                                       |                 |
                                       |                 |
           WINDOW                      |                 |
                                       |                 |
       +--------------+ - - - - - - -  |                 |
       |              |                |                 |
       |              |                |                 |
       |              |                |                 |
       |              |                |                 |
       +--------------+ - - - - - - -  |                 |
                                       |                 |
                                       |                 |
                                       |                 |
                                       |                 |
                                       |                 |
                                       |                 |
                                                         .
                                                         .
```

The window (or "buffer") is a machine-independent concept, and the procedures corresponding to the primitive operations can be written in MAINSAIL to actually deal with the window rather than the file directly. The only machine-dependent code now required is that to open and close a file (which would be necessary under any model), and that to handle the "movement" of the window, or its association with various parts of the file. This code resides in a "device module" (5.4).

5.3 RUNTIME REPRESENTATION OF FILES

For each file that is open at any given time, MAINSAIL maintains a "file buffer" which implements the window concept' and a "file record" providing information needed to access the file via the buffer. The buffer's location and size are known to the machine-dependent "window association" procedure, $devNewBuf, as are the fields of the file record.

References to a file are actually done via the file buffer rather than directly to the file itself. At all times, the buffer is associated with a portion of the file.

The steps that $devNewBuf must use to associate the buffer with a new portion of the file are:

1) If the buffer is "dirty" (if anything has been written into it since its last association), then first write the buffer to the corresponding file locations. For some operating systems, the window concept is literally the case, so that the buffer can in fact be visualized as part of the actual file, and no explicit write operation is needed.

2) If the new portion of the file has anything written in it, then read ("map" on some operating systems) from that portion of the file to the buffer.

3) Update the file record fields **bufPos**, **filledSize**, and **remSize** (see 5.5).

Device modules, which contain the machine-dependent file-handling procedures (including $devNewBuf), are discussed below. Next, the file record and the relation between its fields and the file and file buffer are described. Then the implementation of the primitive operations via the file buffer is explained. Finally, different file system characteristics and their effect on MAINSAIL's file implementation are discussed.

5.4 DEVICE MODULES

Each file is considered to exist on some "device." The device is specified in a machine-dependent manner as part of the file name (a string), when the file is opened.

MAINSAIL's notion of device is very general: in fact, a device is just a abstract concept which is implemented by a "device module", which is a module which performs the following functions:

1) Open a file, establishing a link between it and the file record and file buffer allocated for it.

2) Close a file, breaking the link.

3) Associate a portion of a file with the buffer.

These are the only functions required of a device module. However, it may contain any number of additional procedures that might prove useful (for example, advanced graphics handling procedures). The predefined class which describes the three required procedure fields serves as a prefix class to a device module declaration which adds additional interface fields.

At least two device modules must be written for each operating system that is to support a MAINSAIL implementation. These are referred to as the "disk" and "terminal" device modules.

The disk device module provides random, file structured i/o. The physical device involved is usually a disk, hence the module name. The disk device module is generally the default device module if a program does not explicitly specify one when opening a file.

The generality of the implementation model allows a terminal to be treated as a sequential text input or output file. The terminal device module provides communication with the user's terminal.

Since modules are dynamically brought into memory from a variety of files, device modules can be added as desired, i.e. there is no need to modify the runtime system just to add a new device module. Thus the user can write special-purpose device modules, each of which are automatically used whenever a file is opened and specified as being on the given device. It is only necessary to choose a unique name for the device which will be used in file names. For example, some file systems use the syntax DEV:NAME.EXT for file names, where DEV is the device name, NAME is the primary name of the file, and EXT is an extension to identify a particular form of the file. In this case, the machine-dependent open procedure uses DEV as the name of the device module.

The concept of device is so flexible that a machine-independent device module, MEM, has been written which simply maintains a file in memory (on static pages), thereby providing "memory-resident" files. Open simply allocates static pages for the entire file (buffer), close disposes these static pages, and the newBuf procedure has essentially nothing to do. To utilize such a file, the user need only specify a name such as MEM:XXX.YYY (assuming the syntax above), with the remainder of the program not even aware that the file is in memory. The compiler uses such memory-resident files for the symbol table file SYM (2.2) on machines for which there is sufficient memory.

5.5 FILE RECORD

The file record is an instance of one of two classes, depending on whether the file is a text or data file.

The initial fields of both the **textFile** and **dataFile** classes are the same (and in fact are fields of a prefix class called "file"):

INITIAL FILE RECORD FIELDS

handle	$mdFileHandle
device	POINTER(deviceCls)
name	STRING
statusBits	BITS
link	POINTER(file)
bufSize	INTEGER
filledSize	INTEGER
remSize	INTEGER
bufPos	LONG INTEGER
eofPos	LONG INTEGER

A host-dependent data type termed a **fileHandle** identifies a particular file to the operating system, thus allowing it to be accessed. The data type declaration of the **fileHandle** appears in the operating-system-dependent interface (6.2) as a macro definition of **"$mdFileHandle"** to the appropriate data type. Whenever a module declares something to be of type $mdFileHandle, the compiler substitutes the appropriate data type name. The operating-system-dependent file opening procedure sets the "handle" field of a file record to the particular **fileHandle** value to be used to refer to the given file.

The "device" field of a file's record points to the data section of the device module which manages access to the file. The name of the file is stored as a string.

The **statusBits** give current status information for the file, such as whether it's a text or data file, and the type of access (sequential or random' input or output or both) for which it has been opened.

The link field links together all file records.

bufSize tells the amount of storage allocated for the file buffer. **filledSize** is the effective size of the buffer, i.e. the amount of the buffer which currently contains data.

remSize tells the amount of the "filled" part of the buffer beyond the current cell.

bufPos tells the position in the file that corresponds to the first cell in the buffer, and **eofPos** specifies the end-of-file position.

In addition to the above-mentioned fields they have in common, each of the two file classes has two more fields. For a data file, they tell the address of the first cell in the buffer **(firstData)**, and the address of the cell in the buffer that immediately follows the most recently referenced cell **(nextData)**. For a text file, the fields tell the charadr (character address) of the first character in the buffer **(firstText)**, and the charadr of the character in the buffer that immediately follows the most recently referenced character **(nextText)**.

The relation between some of the fields of a file record and the buffer for the file described by the record may be pictured by the following diagram. The diagram pictures a text file record, but would be similar for a data file record.

```
. TEXT FILE RECORD                   FILE BUFFER
  ----------------                   -----------

  +---------------+
  |    handle     |                 +---------------+            -+- - -+
  +---------------+          +----->|               |   -----     |     |
  |    device     |          |      |               |             |     |
  +---------------+          |      |               |             |     |
  |     name      |          |      |               |             |     |
  +---------------+          |      |               |             |     |
  |  statusBits   |          |      |               |             |f    |
  +---------------+          |      |               |             | i   |
  | (file) link   |          |      |               |             | l   | b
  +---------------+          |      |               |             | l   | u
  |    bufSize    |          |  +-->|               |--+          | e   | f
  +---------------+          |  |   |               |  |          | d   | S
  |  filledSize   |          |  |   |               |  |          | S   | i
  +---------------+          |  |   |               |  |- remSize | i   | z
  |   remSize     |          |  |   |               |  |          | z   | e
  +---------------+          |  |   |               |  |          | e   |
  |    bufPos     |          |  |   |               |  |          |     |
  +---------------+          |  |   |---------------|--+ - - - - -+     |
  |    eofPos     |          |  |   |               |                   |
  +---------------+          |  |   |               |                   |
  | firstText     |----------+  |   |               |                   |
  +---------------+             |   +---------------+  - - - - - - - - -+
  |   nextText    |-------------+
  +---------------+
```

5.6 FILE RECORD INITIALIZATION

Before a file can be referenced by a program, it must be opened. The file opening procedure is passed the name of the file and some bits specifying the type of file and what kind of access is to be allowed to it. Some of the possible bits and their interpretation in view of the file model are:

text	the cells are characters
data	the cells are storage units
input	reading a cell is allowed
output	writing a cell is allowed
random	**setPos** and **relPos** are allowed

When a file is opened, a buffer of memory for the window, and a new file record are allocated for it. The file record is added to the list of file records, and the fields of the new record are initialized:

handle	the handle returned by the operating system
device	points to the data section of the appropriate device module
name	name of the file
statusBits	set according to the bits passed to the open procedure, telling what kind of file it is (text or data; opened for sequential or random access; input and/or output allowed)
link	links the file record onto the list of all file records
bufSize	size of the buffer allocated
filledSize	0
remSize	0
bufPos	0
eofPos	position of last "cell" in the file (returned by open procedure); 0 if creating file
firstText(data)	charadr(address) of start of file buffer
nextText(data)	charadr(address) of start of file buffer

5.7 IMPLEMENTATION OF FILE OPERATIONS

The implementation of the primitive file operations defined above in 5.1 can now be described.

1) v := read(f)

To read a cell, the value of **remSize** is first checked. If **remSize** is 0, then $devNewBuf (5.3) is called.

The cell is read out of the buffer position referenced by **nextData** (or **nextText**). Then a **relPos** by 1 is done.

2) **write(f,v)**

To write a cell, the value of **remSize** is first checked. If it is 0, then $devNewBuf is called.

The cell is written into the buffer position referenced by **nextData** (**nextText**), and the "dirty" bit is turned on in **statusBits**. Then a **relPos** by 1 is done.

If **remSize** is then 0, $devNewBuf is called. An example of the need for this is for terminal output. It is desirable to write characters to the terminal as soon as they are written into the buffer by a program. This is done by having the **bufSize** and **filledSize** be 1, thereby causing **remSize** to be 0 after a character is written to the buffer.

3) n := getPos(f)

The "current position" in a file is the one corresponding to the file buffer position referenced by **nextData** (or **nextText**). This file position can be exactly calculated as

```
        bufPos + (filledSize - remSize)
```

4) setPos(f,n)

If n is not in the portion of the file currently associated with the file buffer, then $devNewBuf is called to make such an association.

NextData (**nextText**) is then set to refer to the buffer position corresponding to n, and **remSize** is set to the number of cells between **nextData** (**nextText**) and **filledSize**.

5) **relPos(f,n)**

relPos(f,n) is equivalent to setPos(f,getPos(f) + n).

5.8 FILE SYSTEM DIFFERENCES

Some file systems require that the size of a file (number of cells, or more usually number of "blocks") be specified when the file is created, and the file cannot grow beyond this size. This conflicts with the file model notion of a file as an infinite sequence of cells. To account for the file size, an optional parameter is added to the file opening procedure that allows the programmer to specify the initial size of a file. If it is unspecified' but a size specification is required by the file system, then an implementation-dependent default value is used.

Many file systems organize files as fixed-size blocks, and remember only the number of blocks in a file, so that the exact end-of-file position . (that is, the last position in the file that had a value written to it) cannot be determined. The MAINSAIL system procedure **"eof(f)"** returns TRUE if file f's position corresponding to **nextData (nextText)** is AT OR BEYOND the file model's concept of end-of-file.

Some file systems have separate file organizations for sequential and random files. MAINSAIL will use the random organization on those devices which can support random access (even if random access was not specified when the file was created), and thus may not be able to take advantage of the most efficient sequential access method. However, this means that a file that is created as a sequential file may later be opened as a random file (and vice versa, since sequential is a special case of random). If the file organizations are identical, MAINSAIL may choose to optimize sequential access (if possible, e.g., by double buffering) for the duration of a particular open if the open indicates that the file is being opened only for sequential access.

SECTION 6 MACHINE AND OPERATING-SYSTEM INTERFACES

In order to bring up a new MAINSAIL implementation, the following must be written:

1) code generators .
2) a machine-dependent interface
3) an operating-system-dependent interface

Code generation was described above in Section 2.4.

This section describes machine-dependent and operating-system-dependent interfaces. The machine-dependent and operating-system-dependent interfaces each consist of a small number of well-defined procedures and constant definitions. These procedures and constants are known to the machine-independent parts of MAINSAIL; it is simply their implementation that is machine or operating-system dependent.

6.1 MACHINE-DEPENDENT INTERFACE

The macros known to the machine-independent parts of MAINSAIL that must be defined by the machine-dependent interface are:

(Note: a "cell" occupies one address unit.)

$charsPerCell	number of characters per cell
$bitsPerBits	number of bits in data type BITS
$bOnes	bits constant of all l-bits
$lbOnes	long bits constant of all l-bits
$maxChar	maximum character code
$charsPerBits	maximum number of characters needed to print a BITS
$charsPerLongBits	maximum number of characters needed to print a LONG BITS
$charsPerInteger	maximum number of characters needed to print an INTEGER
$charsPerLongInteger	maximum number of characters needed to print a LONG INTEGER
$charsPerReal	maximum number of characters needed to print a REAL
$charsPerLongReal	maximum number of characters needed to print a LONG REAL
$mdPieceType	Piece data type. INTEGER or LONG INTEGER See 4.8
$pieceSize	number of storage units (basic memory units) required by a piece
$charsPerPiece	number of characters that can be stored in a piece
$piecesPerParcel	number of pieces in a parcel (4.8)
$parcelSize	$piecesPerParcel * $pieceSize

$charsPerPage	number of characters per page
$valPageMap(i)	a macro which returns the value of the ith element of the pageMap (see 4.1)
$setPageMap(i,val)	a macro which sets the value of the ith element of the pageMap to val
$valChunkTbl(chunkSize)	a macro which returns the address of the first chunk on the chunkTbl list (see 4.6) for chunks of size chunkSize
$setChunkTbl(chunkSize,val)	a macro which sets val as the address of the first chunk on the chunkTbl list for chunks of size chunkSize

The procedures that must be written for the machine-dependent interface are described below:

```
1)  PROCEDURE idRead   (POINTER(dataFile)  f;
                        PRODUCES REPEATABLE INTEGER v)
    PROCEDURE bdRead   (POINTER(dataFile) f; PRODUCES REPEATABLE BITS v)
    PROCEDURE lidRead  (POINTER(dataFile) f;
                        PRODUCES REPEATABLE LONG INTEGER v)
    PROCEDURE lbdRead  (POINTER(dataFile) f;
                        PRODUCES  REPEATABLE  LONG  BITS v)
    PROCEDURE rdRead   (POINTER(dataFile)  f; PRODUCES REPEATABLE REAL v)
    PROCEDURE lrdRead  (POINTER(dataFile) f;
                        PRODUCES REPEATABLE LONG REAL v)
```

These procedures each read a value of v's data type into v from the file buffer associated with the file referenced by f (5.3).

They and the corresponding data writing procedures, described below, are included in the machine-dependent interface because MAINSAIL makes no assumptions about how data is packed or aligned in file buffers. In general these procedures are very simple, and many of them are identical to one another since the only distinction is the size of the data item.

```
2)  PROCEDURE idWrite  (POINTER(dataFile) f;  REPEATABLE INTEGER v)
    PROCEDURE bdWrite  (POINTER(dataFile) f;  REPEATABLE  BITS v)
    PROCEDURE lidWrite (POINTER(dataFile) f; REPEATABLE LONG INTEGER v)
    PROCEDURE lbdWrite (POINTER(dataFile) f; REPEATABLE LONG BITS v)
    PROCEDURE rdWrite  (POINTER(dataFile) f; REPEATABLE REAL v)
    PROCEDURE lrdWrite (POINTER(dataFile) f; REPEATABLE LONG REAL v)
```

These procedures each write the value v to the file buffer associated with the data file referenced by f. See 1) above.

3) LONG BITS PROCEDURE $packId (STRING s)

$packId encodes the string s into a long bits. s must be an identifier not exceeding six characters.

$packId is usually used to encode module names in LONG BITS, e.g. in the module name table of descriptor sections, and in the **modName** field of module descriptors (3.2).

$packId could be made machine-independent, since it is straightforward to devise an algorithm which packs up to six alphabetic characters into thirty two bits (the number guaranteed for long bits). The reason it is machine-dependent is that many machines have natural methods of packing six-character symbols which are supported by system software such as debuggers.

4) STRING PROCEDURE $unpackId (LONG BITS **modName)**

$unpackId decodes the module name encoded (by $packId) in **modName.**

The machine-dependence of $packId necessitates the machine-dependence of **$unpackId** as well.

5) INTEGER PROCEDURE $pageNumber (ADDRESS adr)

$pageNumber returns the number of the page containing the address adr.

Conceptually, $pageNumber could be computed in MAINSAIL as

```
    displacement(pageBaseAdr,adr) DIV $pageSize
```

(where $pageSize is defined by the operating-system interface). However, the MAINSAIL displacement procedure returns the integer displacement between its arguments, whereas the displacement between the base address of the first page and adr might be larger than can be represented by an integer. For this reason, $pageNumber is currently a machine-dependent procedure.

6) ADDRESS PROCEDURE $pageAddress (INTEGER n)

. $pageAddress returns the starting address of the nth page.

Conceptually, $pageAddress could be computed in MAINSAIL as

```
    displace(pageBaseAdr,n * $pageSize)
```

However, the second argument to the displace procedure must be an integer, and the calculation n * $pageSize might result in a value too large to be represented by an integer. For this reason, $pageAddress is currently a machine-dependent procedure.

```
7) PROCEDURE $mdFixRtnAdrs (POINTER($modDscr) targetMod;
                            ADDRESS          loAdr,hiAdr;
                            INTEGER          dspl)
```

$mdFixRtnAdrs adjusts the addresses on the return stack after control sections have been moved, as described' in 3.6.

$mdFixRtnAdrs is machine-dependent because the machine-independent parts of MAINSAIL make no assumptions about the format or location of the return stack or return addresses.

8) PROCEDURE $mdInterfaceCall (POINTER dataSec; INTEGER procNumber)

$mdInterfaceCall is used to call initial and final procedures of modules. dataSec is a pointer to the module's data section, and procNumber specifies which procedure in the module is to be called.

$mdInterfaceCall is machine-dependent because there is no machine-independent way (i.e. no MAINSAIL language construct) to reference an offset in an entry vector (such as procNumber) and call the associated procedure.

9) PROCEDURE mdAdjustString (MODIFIES STRING s; ADDRESS newPieceAdr)

mdAdjustString adjusts the string descriptor for s to specify the new location of s following a garbage collection. The relative character of s within the new piece is the same as within the old piece, and can be determined from s.

mdAdjustString is machine-dependent because the format of a string descriptor is machine-dependent.

10) PROCEDURE mdCollect

mdCollect is called by a machine-independent garbage collection procedure to locate any pointers or strings which would not otherwise be found by the garbage collector (e.g. those on the stacks, whose organization is machine dependent).

11) INTEGER PROCEDURE oneBits (BITS b)

oneBits returns the number of bits in b that are turned on.

oneBits could be written in MAINSAIL, but some machines have efficient bit counting instructions. Since oneBits plays a crucial role in string garbage collection, efficiency was deemed more important than machine-independence.

12) BOOLEAN PROCEDURE **nextCall** (MODIFIES ADDRESS **dataSec;**
PRODUCES INTEGER offset)

nextCall is used by the machine-independent **errMsg** procedure to step through the return stack, listingthe previous procedure call or the sequence of calls (most recent first) that led up to the call to **errMsg.**

Like $mdFixRtnAdrs, **nextCall** is machine-dependent because the **machine-**independent parts of MAINSAIL make no assumptions about the format or location of the return stack.

A **dataSec** argument of NULLADDRESS signifies to **nextCall** that it should initialize a top-of-stack pointer and start processing from there. Otherwise, **dataSec** is the address of the previous data section found by **nextCal**1.

nextCall examines the stack from where it last left off. If it encounters a data section address, it sets **dataSec** to this new address. **NextCall** returns the offset in the control section (for the module whose data section address is **dataSec)** to the last procedure call made.

6.2 OPERATING-SYSTEM-DEPENDENT INTERFACE

The macros known to the machine-independent parts of MAINSAIL which must be defined by the operating-system-dependent interface are:

$pageSize	the size of pages
$mdFileHandle	the data type of file handles (5.5).
$nullMdFileHandle	the Zero file handle
$ttyInputChars	maximum **#** of characters per terminal input line
eol	1 or 2 character end-of-line sequence
eop	1 character end-of-page

The procedures that must be written for the operating-system-dependent interface are described below. Note: $devOpen, $devClose, and $devNewBuf procedures described below must be written for each device module. Device modules, file buffers, and file records are described in Sections 5.3 - 5.5.

1) PROCEDURE **sttyWrite** (REPEATABLE STRING s)

sttyWrite writes the string s to the user's terminal, without using a MAINSAIL device module.

2) PROCEDURE **ttycWrite** (REPEATABLE INTEGER char)

ttycWrite writes the character char to the user's terminal, without using a MAINSAIL device module.

3) PROCEDURE $ttyUnpackId (LONG BITS **modName)**

The effect of $ttyUnpackId(b) is the same as ttyWrite($unPackId(b)). The difference is that the latter uses string space (which could ultimately lead to a garbage collection), whereas $ttyUnpackId is not allowed to affect memory, so that it can be used' in "sensitive" situations. For example, $ttyUnpackId is used by the **runtime** system to report the names of modules that are swapped (when that user option is in effect).

```
4) INTEGER PROCEDURE $mdttyRead
                        (CHARADR                          c;
                         INTEGER                          maxChars;
                         PRODUCES OPTIONAL INTEGER        brkChr)
```

$mdttyRead reads a string from the user's terminal to the area of memory starting at c. It should provide editing of the string. The accepted string termination characters are ones that are appropriate for the operating system: e.g., they usually consist of at least eol (end-of-line) and an end-of-file character.

brkChr is the character code of the terminating character. The CHARADR references an area of memory **maxChars** long. The string read in must not exceed this number of characters. If **maxChars** are read in before a string-termination character is encountered, then $mdttyRead sets **brkChr** to -1.

The value returned by $mdttyRead is the total number of characters in the string, excluding any terminating characters.

5) PROCEDURE **fastExit** (OPTIONAL STRING msg)

fastExit writes msg to the user's terminal, and then immediately exits from MAINSAIL. It is used as a panic exit, and thus provides a less orderly exit than the machine-independent "exit" procedure.

6) BOOLEAN PROCEDURE $devOpen (MODIFIES POINTER(file) f)

This procedure attempts to open a file using the information contained in the file record referred to by f. If the open is successful, then **$devOpen** allocates the file buffer, initializes further file record fields, and then returns TRUE (5.6). Otherwise, $devOpen returns FALSE. This procedure is an interface field of each device module.

```
7) PROCEDURE $devClose (POINTER(file) f; OPTIONAL BITS ctrlBits)
```

$devClose is responsible for closing the file referenced by f.

If the relevant file buffer is dirty (5.3), then $devClose first clears the unused portion of the buffer and then writes the buffer out. In any case, $devClose returns the file handle to the operating system. If the delete bit is set in ctrlBits, the file is deleted from the file system. This procedure is an interface field of each device module.

```
8) BOOLEAN PROCEDURE $devNewBuf (POINTER(file)          f;
                                 OPTIONAL BITS          ctrlBits;
                                 OPTIONAL LONG INTEGER newPos)
```

$devNewBuf is responsible for associating a file buffer with a new portion of the corresponding file, as described in Section 5.3. This procedure is an interface field of each device module.

```
9) BOOLEAN PROCEDURE $mdNewBuf (POINTER(file)          f;
                                OPTIONAL LONG INTEGER newPos)
```

Much of the processing that must be done to associate a file buffer with a new portion of the corresponding file can be handled by a machine-independent procedure called $newBuf. Some device modules' $devNewBuf procedures, for example, simply need to call $newBuf. For such device modules, an $mdNewBuf procedure must also be written for $newBuf to call to handle the operating-system-dependent i/o.

```
10) BOOLEAN PROCEDURE mdOpen (MODIFIES POINTER(file) f)
```

The device on which a file resides is specified in an operating-system-dependent manner as part of the file name. mdOpen is responsible for determining the device on which a file resides, based on the file name. It binds the appropriate device module and calls the $devOpen procedure in that module.

```
11) PROCEDURE $ctrlOpen (PRODUCES $mdFileHandle               handle:
                         STRING                               fileName;
                         BITS                                 ctrlBits;
                         PRODUCES OPTIONAL LONG INTEGER ctrlSize)
```

$ctrlOpen is a special-purpose, stand-alone procedure used to open files containing control sections. ctrlBits indicates whether the open is for input, output, or both, and whether the file should be created. If the file is not being created then ctrlSize indicates its current size.

Files containing control sections are handled by the procedures $ctrlOpen, $ctrlClose and $ctrlIn rather than by the file-handling procedures of a device module. The difference is that each such file is completely copied into memory (entire pages are read in) rather than using a file buffer as a window into the file. Also, in order to use a device module procedure to open a control section file, the file containing the device module control section would first need to be opened, but it can't open itself.

12) PROCEDURE $ctrlClose ($mdFileHandle h)

$ctrlClose closes the file associated with the handle h.

13) PROCEDURE $ctrlIn ($mdFileHandle handle;
```
                        INTEGER         filePage;
                        ADDRESS         memAdr;
                        INTEGER         numPages;
                        OPTIONAL BITS   ctrlBits)
```

$ctrlIn reads control sections from files opened with $ctrlOpen. filePage gives the number of the first page in the file, memAdr is the address of the first memory location to be read into, and numPages tells the number of pages to be read.

14) PROCEDURE $ctrlOut ($mdFileHandle handle;
```
                        INTEGER         filePage;
                        ADDRESS         memAdr;
                        INTEGER         numPages)
```

$ctrlOut writes a control section from memory into a file opened with $ctrlOpen. filePage gives the number of the first page in the file to which the control section will be written, memAdr is the address of the first memory location to be written from, and numPages tells the number of pages to be written.

15) BOOLEAN PROCEDURE mdMorePages (INTEGER numPages)

mdMorePages obtains more memory from the operating system when the currently-used memory becomes insufficient (4.4). mdMorePages must not ask for more memory if the request would exceed a configuration parameter pageLimitAdr (4.3) specifying the maximum page that may be used.

16) INITIAL PROCEDURE osdRunMainsail

osdRunMainsail is the runtime initialization procedure (4.3). It must initialize any machine-dependent runtime data structures, and then invoke the top-level machine-independent procedure, runMainsail. It is common that osdRunMainsail has nothing to do other than invoke runMainsail. osdRunMainsail resides in the kernel.

SECTION 7 SUMMARY

This paper has given an overview of the design and implementation of MAINSAIL in view of the portability goals of the MAINSAIL project. Both the machine-independent and the requisite machine-dependent aspects have been described.

The initial design and development is now complete. MAINSAIL's use has been demonstrated on PDP-10 and PDP-11 computers under various operating systems. MAINSAIL has been implemented at sites on the Arpanet, and ported via magnetic tape to other locations.

A number of sites are evaluating MAINSAIL, and it presently supports an active and growing body of users. Examples of programs written in MAINSAIL are a machine-independent tape transfer program, a three dimensional graphics package, a comprehensive system of image processing programs, and a machine-independent interprocessor communications facility.

A large computer-assisted instructional program teaching elementary programming in the BASIC language has been redesigned and translated from SAIL to MAINSAIL so that it could be run on computers other than the PDP-10. A formatting program was written concurrently with the writing of the MAINSAIL language manual: it inputs the manual as written (with encoded section names and cross-references) and outputs a ready-to-print manual complete with table of contents, index, and section headings at the top of each page. That program was also used to format this paper.

Research is currently underway to develop an efficient MAINSAIL language representation (instruction set) and a high-level language machine architecture ("MAINSAIL machine") which will directly execute MAINSAIL programs. The clean design of MAINSAIL, both internally and externally, its machine-independence, and the ability of the code generators to be retargeted, provide an ideal environment for this research.

Future MAINSAIL development, to be carried out by a private company, will include implementations for new machines, a comprehensive symbolic debugger, and a collection of portable MAINSAIL applications programs. There are plans for a display-oriented text editor and a simple operating system so that MAINSAIL can run stand-alone.

Possible new implementations to be begun in the near future include the Digital Equipment Corporation VAX-11/780, Motorola MC68000, and Data General ECLIPSE computers. The new implementations will demonstrate, test, and lend further credibility to MAINSAIL's design for portability.

REFERENCES

[1] Reiser, John F., editor, "SAIL", Stanford Artificial Intell igence Laboratory, Memo AIM-289, Stanford University, August 1976.

[2] Wilcox, C.R., Dageforde, M.L., and Jirak, G.A., "MAINSAIL Language Manual", SUMEX Computer Project, Stanford University Medical Center, Stanford, CA., July 1979.

[3] Wilcox, Clark R., "Comparison of MAINSAIL and PASCAL", SUMEX Computer Project, Stanford University Medical Center, Stanford, CA., Spring 1978.

[4] Wilcox, Clark R., "The MAINSAIL Project: Developing Tools For Software Portability", Proceedings, Computer Application in Medical Care, October 1977, pp. 76-83.