VD

MAINSAIL®

Documentation User's Guide, Overview, and Master Index

MAINSAIL® Documentation User's Guide

MAINSAIL Overview

Master Index

24 March 1989

xidak™

Table of Contents

List of Examples

List of Tables

1. Documentation Overview

XIDAK, Inc., has organized the documentation on its MAINSAIL language and MAINSAIL programming environment into a series of manuals and guides. The present guide is intended both for the new user of MAINSAIL who wishes to learn how to use MAINSAIL and wants to know what to read first, and also for the experienced MAINSAIL user who needs a specific piece of information and wishes to know what manual to consult.

A summary of the MAINSAIL documentation is provided in Tables 1.6-2, 1.6-3, and 1.6-4.

1.1. Introductory Documents and Language Documentation

The documents listed in Table 1.6-2 are designed to provide information to the new user of MAINSAIL and to serve as references on the MAINSAIL language.

If you do not know anything about MAINSAIL, but are familiar with other programming languages and systems, you should consult the "MAINSAIL Overview". It provides a high-level view of the facilities available from MAINSAIL.

If you have access to a computer that runs MAINSAIL and wish to begin writing MAINSAIL code right away, you should follow the examples in the "MAINSAIL Tutorial". The first part of the "MAINSAIL Tutorial" assumes a minimum of previous programming experience, although it may be of interest to experienced programmers as well. The second part of the "MAINSAIL Tutorial" provides details on MAINSAIL's implementation and suggestions for making programs written in MAINSAIL as maintainable, efficient, and portable as possible.

The "MAINSAIL Language Manual" is a thorough but concise reference on the syntax and semantics of the MAINSAIL language. It is intended for the experienced programmer who wishes to know the precise definition of a feature of MAINSAIL. If you are just learning about MAINSAIL, you will find the "MAINSAIL Language Manual" makes more sense after you have read the "MAINSAIL Overview" or the "MAINSAIL Tutorial".

1.2. MAINSAIL Environment Documentation

The guides in Table 1.6-3 describe various programs written in MAINSAIL that constitute the MAINSAIL programming environment (often called "the MAINSAIL environment" or "the MAINSAIL system"). These are utilities that are useful in writing MAINSAIL programs; some of them (e.g., MAINEDIT, the text editor) may be used to accomplish tasks unrelated to MAINSAIL programming.

1.3. System-Specific Documentation

The guides listed in Table 1.6-4 provide information specific to the use of MAINSAIL on particular operating systems. They discuss those details of MAINSAIL that depend on the host system. Since MAINSAIL is designed for portability, these guides are small. They tell how to invoke MAINSAIL on the given operating system, and explain system-dependent features of MAINSAIL, such as MAINSAIL's interaction with the file system and MAINSAIL's handling of system-dependent exceptions.

1.4. Additional Documents Available from XIDAK

On most releases of MAINSAIL, XIDAK issues a release note that describes any changes since the previous release note.

If you are considering the purchase of a MAINSAIL system or any of its components, you should request a copy of the most recent "XIDAK Product Catalog" from XIDAK. From time to time, XIDAK also issues product data sheets and other information of interest to potential purchasers of MAINSAIL. These do not contain any technical information that is not also available in the standard documents listed in this guide.

1.5. Online Documentation

With each standard MAINSAIL system, XIDAK includes current versions of its documentation on the distribution medium. The file names under which the documents appear are listed in quotes in Tables 1.6-2, 1.6-3, and 1.6-4.

1.6. Recommended Reading Order

If you are using MAINSAIL for the first time, you should read the documents in Table 1.6-1 in the order shown.

The "MAINSAIL Overview" describes the MAINSAIL language and MAINSAIL environment to the new user. The operating-system-specific user's guide describes how to invoke MAINSAIL on your system. You need to know how to invoke MAINSAIL before starting the tutorial. The first part of the tutorial guides you step-by-step through the writing of some simple MAINSAIL programs and the use of a number of the MAINSAIL utilities.

MAINEDIT, the MAINSAIL compiler, MAINEX, CONF, MODLIB, and MAINDEBUG are among the most important components of the MAINSAIL environment. The tutorial provides an introduction to each of these, but more information is found in the appropriate user's guides.

```
1. MAINSAIL Overview (skim)

2. Operating-system-specific user's
   guide for your system (skim)

3. MAINEDIT User's Guide (if you are
   using MAINEDIT)

4. MAINSAIL Tutorial (first half)

5. MAINSAIL Compiler User's Guide, first
   two chapters

6. MAINSAIL Utilities User's Guide,
   chapters on MAINEX, CONF, and MODLIB

7. MAINDEBUG User's Guide (if you are
   using MAINDEBUG)

8. MAINSAIL Tutorial (second half)
```

Table 1.6-1. Reading Order for New MAINSAIL Programmers

The second part of the tutorial contains suggestions for the construction of efficient, portable MAINSAIL programs and for sophisticated use of the tools in the MAINSAIL environment. It is of interest to the programmer who has mastered the basics of the MAINSAIL language.

Title ("File Name")	Function
MAINSAIL Documentation User's Guide, MAINSAIL Overview, and Master Index ("mslov.doc")	Gives an overview of the other documents. Summary of the main features of the MAINSAIL language and programming environment, intended as an overview for those evaluating MAINSAIL or using it for the first time. Master index to all MAINSAIL documents.
MAINSAIL Tutorial ("mtut.doc")	Step-by-step instructions for and examples of writing MAINSAIL programs, and tips for writing good MAINSAIL code.
MAINSAIL Language Manual ("mlanm.doc")	Comprehensive reference on the MAINSAIL language.

Table 1.6-2. Introductory MAINSAIL Documents and Language Documentation

Title	Function
MAINSAIL Compiler User's Guide	Describes the MAINSAIL compiler and related utilities, including the MAINSAIL disassemblers.
MAINDEBUG User's Guide	Describes the MAINSAIL portable source-level debugger.
MAINEDIT User's Guide	Describes the MAINSAIL full-screen text editor.
MAINKERMIT User's Guide	Describes a MAINSAIL implementation of the KERMIT file transfer program
MAINPM User's Guide	Describes the MAINSAIL performance monitor, which allows the programmer to determine where the inefficiencies in a program lie
MAINSAIL Structure Blaster User's Guide	Describes the routines used for fast, simple input/output of MAINSAIL data structures
MAINSAIL STREAMS User's Guide	Describes STREAMS, a package for portable distributed applications
MAINSAIL Utilities User's Guide	Describes miscellaneous (but important) components of the MAINSAIL environment

Table 1.6-3. Documentation on the MAINSAIL Environment (Combined in the "MAINSAIL Tools User's Guides", File Name "toolu.doc")

```
The following operating-system-specific MAINSAIL user's
guides are available as of March, 1989:

Title
Aegis MAINSAIL User's Guide

VM/SP CMS MAINSAIL User's Guide

UNIX MAINSAIL User's Guide

VAX/VMS MAINSAIL User's Guide
```

Table 1.6-4. System-Dependent MAINSAIL Documentation (Combined in the "MAINSAIL System-Specific User's Guides", File Name "osu.doc")

MAINSAIL® Overview

24 March 1989

2. Introduction

The "MAINSAIL Overview" summarizes the features of the MAINSAIL language; the MAINSAIL compiler; MAINDEBUG, the MAINSAIL debugger; MAINEDIT, the MAINSAIL text editor; MAINPM, the MAINSAIL performance monitor; the MAINSAIL Structure Blaster; and MAINSAIL STREAMS. These components, together with a number of utility programs, make up the MAINSAIL system, a powerful, highly portable programming environment supported and marketed by XIDAK, Inc.

The MAINSAIL programming environment provides a complete set of tools to support the entire software development cycle. Since these tools are themselves written in MAINSAIL, they operate identically on every computing system on which MAINSAIL is supported. Programmers, once trained to use these tools, can move their program development from one computing system to another without having to learn new tools each time they move.

MAINSAIL is a sophisticated language primarily intended for large projects. Its flexible structure encourages a high-level, object-oriented programming style. Large MAINSAIL software systems are easier to write and maintain than systems written in other commercially available programming languages, both because of the variety of tools provided to support large-system development and because of the inherent clarity and power of the language itself.

The "MAINSAIL Overview" is recommended reading for new users of MAINSAIL and for potential purchasers of XIDAK products. The MAINSAIL language is described in detail in the "MAINSAIL Language Manual" and the "MAINSAIL Tutorial"; other XIDAK products have their own user's guides. Consult the "MAINSAIL Documentation User's Guide and Master Index" or a current "XIDAK Product Catalog" for more information on XIDAK's documentation products.

The features of the language and other components of the system are not described exhaustively in this document; only the most commonly used facilities have been covered.

The MAINSAIL Language

3. Introduction to the MAINSAIL Language

MAINSAIL was originally developed at Stanford University under the auspices of the SUMEX Computer Project by the founders of XIDAK, Inc. It is an ALGOL-like programming language derived from the SAIL programming language. SAIL was developed at the Artificial Intelligence Laboratory of Stanford University. MAINSAIL retains some of the most popular features of SAIL, such as variable-length strings and garbage collection, but eliminates all the machine-dependent characteristics of SAIL.

MAINSAIL was specifically designed to provide true source-level portability. The entire MAINSAIL compiler and the bulk of the runtime system are written in MAINSAIL, minimizing the effort required by XIDAK to move MAINSAIL to a new processor or operating system. MAINSAIL's unprecedented level of portability allows XIDAK to guarantee a uniform programming environment across all MAINSAIL implementations.

MAINSAIL is an extremely powerful programming language offering features not found in other popular programming languages such as C, Pascal, Modula-2, or Ada. Independently of its complete portability, MAINSAIL is an outstanding choice for most programming tasks, even those that are not intrinsically portable.

MAINSAIL provides a complete I/O interface that supports sequential and random access to files of text or data. Terminal interaction is also part of the I/O interface.

MAINSAIL provides a very powerful module facility similar to the "class" concept of SIMULA. Modules are both the unit of compilation and the unit of execution. They can be used as packaging devices or for the implementation of abstract data types. Modules can be embedded in data structures by means of pointers, and multiple copies of a module can be dynamically allocated and deallocated.

Other powerful features include dynamic arrays and records, all subject to the automatic storage reclamation strategy known as "garbage collection", which is usually available only in very-high-level languages, such as LISP.

Unlike most commercially available programming languages, MAINSAIL programs are not statically linked. A statically linked language requires all code that might be used in a particular execution to be loaded when any program written in that language begins execution. In MAINSAIL, object modules are loaded as needed; in a large system, this may result in far less code being in memory if much of the code is needed only occasionally. The code constituting a MAINSAIL "program" is therefore determined dynamically.

MAINSAIL was designed to simplify the management and maintenance of large programming projects involving many programmers. The compiler and runtime system provide extensive facilities for incremental development of components and subsequent integration of components into a complete system.

MAINSAIL is a "natural" programming language; programmers can program the way they think. Programmers need not revert to programming "tricks" or obscure coding techniques to get their jobs done. MAINSAIL's clear, clean syntax makes it easy to understand MAINSAIL programs. This dramatically reduces the resources required for both program development and program maintenance.

MAINSAIL is a broad-spectrum programming language. Its utility spans the range of applications, from systems programming, such as compilers and text editors, through scientific and technical applications, to business and financial applications. MAINSAIL runs on a diverse set of processors, from mainframes to microprocessors. Great care has been taken to provide a language definition that supports compatible implementations across such a wide range of machines. An approach that uses the same compiler, runtime system, and support software across all implementations is the key to such compatibility.

4. Basic Concepts

4.1. Character Set

MAINSAIL does not specify the exact character set of the machines on which it runs. Instead, guarantees are given that must hold for all character sets under which MAINSAIL is implemented. System procedures are provided to complement these assumptions. Predefined string constants are provided for horizontal tab, end-of-line, and end-of-page characters.

4.2. Comments

A comment starts with "#" and extends to the end of a line. Several methods are available for commenting out a large body of text; for example, a directive is provided to skip entire pages.

4.3. Compiletime Evaluation

Most expressions consisting entirely of constant operands are evaluated at compiletime. The compiler uses variable-length string representations in performing such evaluation so that any host machine limitations do not affect the precision of arithmetic results.

4.4. Low-Level Storage Manipulation

Storage is measured in "storage units" and "character units". Storage units are independent of the byte or word size of the processor on which MAINSAIL executes. Character units are always eight-bit bytes. A number of predefined identifiers specify the characteristics of the target processor, permitting low-level code to be written in a highly portable fashion.

4.5. Log and Command Files

A command file (cmdFile) and a logging file (logFile) are utilized for standard input and output. These files are normally associated with "TTY", which represents primary input and output (usually the user's terminal), but may be redirected to other files so that a program can utilize any file for what appears to the program as terminal-oriented I/O.

5. Data Types

There are eleven MAINSAIL data types. Each data type includes a "zero" value, referred to as "Zero", which is represented in memory as an all-zero bit pattern.

5.1. Boolean

True or false. Boolean Zero is false.

5.2. Integer and Long Integer

An integer is guaranteed (at least) the range -32767 through 32767 (representable in 16 bits). A long integer is guaranteed (at least) the range -2147483647 through +2147483647 (representable in 32 bits). (Long) integer Zero is 0.

5.3. Real and Long Real

A real is guaranteed a fraction of at least 6 full decimal digits, and the exponent is guaranteed to range at least from 1.0E-38 to 1.0E+38. For a long real, the fraction is guaranteed to consist of at least 11 full decimal digits, and the exponent range is at least as large as that of a real. (Long) real Zero is 0.0.

5.4. Bits and Long Bits

These data types are for representing sequences of bits. A bits consists of (at least) 16 bits and a long bits consists of (at least) 32 bits. These data types may take part in bit operations such as masking, shifting and testing. (Long) bits Zero has all bits equal to 0-bit.

5.5. String

A string represents a variable-length sequence of characters. A string variable is implemented as a descriptor that gives the current length (number of characters) of the string, and the location of the first character. MAINSAIL guarantees that a string may contain up to 32766 characters. The characters themselves usually reside in an area of memory known as "string space", where they are subject to a storage reclamation method known as "garbage collection".

A string constant is a sequence of characters enclosed in double quotes. A double quote is made part of a string constant by using two double quotes. String Zero is "", the string with no characters.

5.6. Pointer

A pointer is a data type for referencing dynamically allocated objects such as records or modules. These dynamically allocated objects are subject to garbage collection. Pointer Zero is nullPointer, which points to no object.

5.7. Address

Address is a data type for representing arbitrary memory addresses. An address can reference all data types. To access individual characters, the data type "charadr" must be used. Address Zero is nullAddress.

A classified address variable, e.g., "a" declared as "ADDRESS(c) a", can be used in field variables of the form a.f, where f is a field of the class c. This allows a class to be used as a storage template placed over memory starting at the address contained in "a".

Memory is viewed as a linear sequence of addressable cells ("storage units"). Addresses are ordered with respect to the relative position of the referenced cells. This order is used when comparing addresses.

5.8. Charadr

Charadr ("character address") is a data type for representing the location of a character. Charadr is distinct from address since some machines address words that may contain several characters. Charadr Zero is nullCharadr. Charadr provides a more primitive handling of characters than the string data type.

Charadr and address are used only by programs manipulating the contents of memory in a low-level fashion; many applications do not use address and charadr at all.

5.9. Conversion

There is no implicit data type conversion; instead, explicit system procedures are provided for data type conversions.

6. Expressions

An expression provides the means of accessing and computing values.

6.1. Constants

Constants are the predefined values for each data type, e.g., "FALSE", "37", or "3.14159". They may be represented symbolically as macro constants.

6.2. Variables

There are five kinds of variables:

- (Non-own) local variable: allocated dynamically upon procedure entry and deallocated upon procedure exit; accessible only within declaring procedure.
- Own variable: allocated dynamically upon module allocation and deallocated upon module deallocation; accessible only within module (and only within procedure if declared within procedure).
- Interface variable: allocated the same as an own variable; accessible within declaring module and from other modules.
- Subscripted variable: element of an array. All arrays are dynamically allocated.
- Field variable: field of a record, data section, or storage template. All records, data sections, and scratch memory are dynamically allocated.

6.3. Procedure Expression

A procedure expression is a procedure call used as an expression. The invoked procedure must be typed, i.e., declared as returning a value.

6.4. Substrings

"s[i TO j]" is the string consisting of characters i through j of the string s. "s[i FOR j]" is the string consisting of characters i through i+j-1 of s. The integer expressions i or j may contain the keyword "INF", which stands for the length of s.

6.5. If Expression

An If Expression selects among several possible values. It has the general form:

```
IF e1 THEN v1
EF e2 THEN v2
...
EL vn
```

where "EF" abbreviates "ELSE IF", and "EL" abbreviates "ELSE". The "EF" clauses may be omitted. All the vi must be of the same type, and this type is the type of the If Expression.

6.6. Assignment Expression

An Assignment Expression has the form "v := e", where v is a variable and e is an expression of the same data type as v. The result is the value of e. "_" may be used in place of ":=".

6.7. Universal Operations

The following operations apply to all data types:

```
NOT e          IF e THEN FALSE ELSE TRUE
v := e         assignment expression
e1 OR e2       IF e1 THEN TRUE EF e2 THEN TRUE EL FALSE
e1 AND e2      IF NOT e1 THEN FALSE EF e2 THEN TRUE EL FALSE
e1 = e2        equal
e1 NEQ e2      not equal ("<>" may be used for NEQ)
```

e2 is evaluated only if necessary for "OR" and "AND".

6.8. Comparison Operations

The following operations apply only to those data types that have an ordering, i.e., (long) integer, (long) real, string, address, and charadr:

```
e1 < e2        less than
e1 LEQ e2      less than or equal ("<=" may be used for LEQ)
e1 > e2        greater than
e1 GEQ e2      greater than or equal (">=" may be used for GEQ)
e1 MIN e2      minimum
e1 MAX e2      maximum
```

6.9. Arithmetic Operations

The following operations apply to (long) integer and (long) real, except as otherwise specified. Both arguments must be the same type except as otherwise indicated for "^":

```
- e            negative of e
e1 ^ e2        exponentiation (allowed type combinations:
                 i^i, li^i, r^i, lr^i, r^r, lr^r)
e1 + e2        sum
e1 - e2        difference
e1 * e2        product
e1 / e2        (long) real quotient
e1 DIV e2      (long) integer quotient (discard remainder)
e1 MOD e2      (long) integer modulus (remainder)
```

"**" may be used in place of "^".

6.10. Bitwise Operations

The following operations apply to (long) bits. Both arguments must be the same type, except that e2 is an integer for "SHL" and "SHR":

e1 TST e2	TRUE if any 1-bit in e2 is a 1-bit in e1
e1 NTST e2	TRUE if no 1-bit in e2 is a 1-bit in e1
e1 TSTA e2	TRUE if all 1-bits in e2 are 1-bits in e1
e1 NTSTA e2	TRUE if not all 1-bits in e2 are 1-bits in e1
e1 IOR e2	inclusive or
e1 XOR e2	exclusive or
e1 MSK e2	Clear any bits in e1 that are 0-bits in e2
e1 CLR e2	Clear any bits in e1 that are 1-bits in e2
e1 SHL e2	shift left ("<<" may be used in place of SHL)
e1 SHR e2	shift right (">>" may be used in place of SHR)
e1 ! e2	= e1 IOR e2, except ! has higher precedence

6.11. String Operations

The following operation applies to strings:

e1 & e2	concatenation: the string made up of the characters of e1 followed by the charcters of e2

6.12. Operator Precedence

Table 6.12-1 shows the precedence of the operators. Operators on the same line have equal precedence.

```
OR                     (least precedence -- least binding)
AND
NOT
=   NEQ   <   LEQ   >   GEQ   TST   NTST   TSTA   NTSTA
:=
MIN   MAX
+   - (binary)   IOR   XOR   MSK   CLR
*   /   &   DIV   MOD   SHL   SHR
!   ^
- (unary)              (most precedence -- most binding)
```

Table 6.12-1. Operator Precedence

Operators of equal precedence are associated from left to right (except for assignment). The order of evaluation of the operands is in general not specified. The precedence of ":=" was chosen so that it could be used in expressions without the need for parentheses in most common cases.

6.13. Dotted Operators

The expression "v .op e" is a short form of "v := v op e", except that v is evaluated just once. For example:

```
v[i .+ 1] .+ 5
```

adds 5 to v[j], where j = i + 1; in addition, i is incremented. Almost all operators can be "dotted" this way.

7. Statements

This chapter describes eleven of the thirteen MAINSAIL statements; the other two, the Init and Handle Statements, are described in Chapters 9 and 16, respectively.

7.1. Assignment Statement

An Assignment Statement has the form of an Assignment Expression, except that it occurs as a statement rather than as an expression.

7.2. Expression Statement

An Expression Statement is a dotted expression used as a statement; e.g., "i .- 1" is a statement that decrements i.

7.3. Procedure Statement

A Procedure Statement has the form of a Procedure Expression, except that the procedure may be untyped. If it is typed, the result is discarded.

7.4. Return Statement

A Return Statement has the form "RETURN" for an untyped procedure, and "RETURN(e)" for a typed procedure, where the expression "e" is of the same type as declared for the procedure. It causes immediate termination of the procedure's execution, returning the specified value.

7.5. Begin Statement

A Begin Statement has the form "BEGIN s1; ...; sn END", where the si are statements. It is a means of grouping a list of statements into a single statement. A name can be given to a Begin Statement by inserting a string constant after the "BEGIN", in which case the same name must be inserted after the matching "END".

7.6. If Statement

An If Statement has the general form:

```
IF e1 THEN s1
EF e2 THEN s2
...
EF en THEN sn
EL s
```

The "EF" and "EL" lines can be omitted. "EF" abbreviates "ELSE IF", and "EL" abbreviates "ELSE" (the longer forms can be used). An "ELSE" ("EL") or "EF" matches with the innermost unmatched "IF" or "EF".

7.7. Case Statement

A Case Statement is illustrated by:

```
CASE e OFB
    [c1]                   s1;
    [c2 TO c3]             s2;
    [c4][c5 TO c6]         s3;
    [ ]                    s4;
    END
```

"e" is an integer expression, the ci are integer constants, and the si are statements. "OFB" is an abbreviation for "OF BEGIN", which may be used instead. If e has the value c1, s1 is executed. If e is between c2 and c3, inclusive, s2 is executed. If e is c4 between c5 and c6, s3 is executed. Otherwise, s4 is executed. In general, any number of case selectors may be utilized for a given statement, as two are shown for s3. The catchall "[]" case may be omitted, in which case an error occurs if no statement is selected. Only the selected si is executed. si may be a Begin Statement, i.e., a list of statements to be executed.

7.8. Iterative Statement

An Iterative Statement has the general form:

```
FOR i := e1 UPTO e2    WHILE e3       DO s    UNTIL e4
(FOR-clause)           (WHILE-clause)         (UNTIL-clause)
```

where i is an integer variable, e1 and e2 are (long) integer expressions, e3 and e4 are any expressions, s is any statement, and "UPTO" may be replaced with "DOWNTO". Any of the clauses may be omitted; thus, there are eight possible forms (ignoring the distinction between "UPTO" and "DOWNTO").

"DO s" alone means repeatedly execute s until some action terminates the Iterative Statement, most likely a Done or Return Statement or an exception. "UPTO" increments i by 1, and "DOWNTO" decrements i by 1. e3 is evaluated before each iteration, and e4 after each iteration. e2 is evaulated just once, before any iterations. An "UNTIL" is matched with the innermost unmatched "DO". A name may be given to the Iterative Statement by placing a string constant after "DO", "DO BEGIN" or "DOB" (an abbreviation for "DO BEGIN"). In the latter two cases, the same name must be placed after the matching "END". Such a name may be used by the Done or Continue Statement as described below.

7.9. Done Statement

A Done Statement has the form "DONE", or:

```
DONE "c"
```

where "c" is the name of an enclosing Iterative Statement. The referenced Iterative Statement is terminated; in the absence of "c", the innermost enclosing Iterative Statement is terminated.

7.10. Continue Statement

A Continue Statement has the form of the Done Statement, except that "CONTINUE" replaces "DONE". The referenced Iterative Statement is continued (tests in the clauses are performed, and if they pass, the iterated statement is re-started) as if the iterated statement had terminated normally.

7.11. Empty Statement

The Empty Statement consists of nothing at all. It serves as a place holder in certain situations; for example, it allows superfluous semicolons between statements.

8. Declarations

Identifiers must be declared before they are referenced.

8.1. Scope of Identifiers

An identifier declared within a procedure is accessible only within that procedure. An identifier declared outside any procedure is accessible within procedures in the same module that follow the declaration, except those that redeclare it.

8.2. Simple Variable Declarations

A simple variable declaration has the form "type v1, ..., vn", where the vi are identifiers, and type is the name of a data type. In addition, the type keywords "POINTER" and "ADDRESS" may be followed by a parenthesized class name.

8.3. "OWN" Qualifier

The "OWN" qualifier specifies that a local variable is to be allocated as if it were declared in the outer block. This means that it is allocated and deallocated along with the module data, rather than upon procedure entry and exit. This allows a procedure to have a private variable that can retain information across procedure calls.

9. Arrays

9.1. Array Declarations

An array declaration has one of the forms:

```
type ARRAY(l1 TO u1, ..., lm TO um) v1, ..., vn
```

or:

```
type LONG ARRAY(l1 TO u1, ..., lm TO um) v1, ..., vn
```

For reasons of efficiency MAINSAIL supports two sizes of array. Short arrays can use only integer subscripts while long arrays can use either long integer or integer subscripts. Subscript calculations for short arrays are performed with integer arithmetic; long array subscript calculations are performed with long integer arithmetic. Since long integer arithmetic is slower than integer arithmetic on some systems, short arrays permit greater runtime efficiency, but cannot be as large as long arrays.

li and ui specify the bounds of the ith dimension. Up to three dimensions are allowed. Each li or ui is either a (long) integer constant, with li LEQ ui, or "*" to indicate that the bound is not known at the point of declaration (it must be given when the array is allocated).

The data type and/or the parenthesized bounds list may be omitted, in which case the array cannot be used for element access. It may still be used as a parameter or assigned to or compared with some other array.

The array Zero is nullArray.

9.2. Array Allocation

It is the programmer's responsibility to allocate an array before an element is accessed. MAINSAIL differs in this respect from most languages, which automatically allocate and deallocate arrays according to the context in which the array is declared.

An array is allocated by "new(a,l1,u1,...,ln,un)", where a is the array to be allocated, and the li and ui are expressions for the bounds. Bounds that can be determined from the array declaration may be omitted. An array may be allocated any number of times. Each new allocation replaces the old one; no elements are copied. All elements of a newly allocated array are initialized to Zero.

MAINSAIL's explicit array allocation permits a program to decide dynamically when to allocate arrays and how large each must be. For example, a procedure can allocate arrays of a size dependent upon input data. Many arrays can be declared, only some of which are actually allocated during a given program execution.

Arrays may be explicitly deallocated, or the programmer may allow the garbage collector to free the storage occupied by unused arrays.

9.3. Array Initialization

The Init Statement may be used to initialize an array. The array must already have been allocated. The general form of the Init Statement is:

```
INIT v (c1, ..., cn)
```

where v is the array to be initialized, and the ci are constant expressions of v's type.

9.4. Accessing an Array Element

An array element is accessed by a variable of the form "a[i]", "a[i,j]", or "a[i,j,k]", depending on the number of dimensions of the array a. i, j, and k are (long) integer expressions that must be within the bounds declared for the corresponding dimensions.

9.5. Array Variables

An array variable is implemented as a pointer to the array's elements. Array variables may be assigned, passed as parameters, and compared. In each case, only the array variable itself (i.e., the pointer to the array elements) takes part in the operation. Thus, a single array's elements may be pointed to by many array variables.

10. Classes and Records

10.1. Records

A record is a data structure that differs from an array in that its components, called fields, may be of differing data types and are accessed by name instead of by subscripts.

Records are dynamically allocated; the programmer does not declare records, and there is no way to create a record at compiletime. Just "classes", which give templates for the structure of records created at runtime, and pointers, which point to records, are declared. All records are allocated during program execution under direct control of the program.

10.2. Classes

Each record is an instance of a programmer-declared class that serves as a template describing the various fields of the record. The most common form of a class declaration is:

```
CLASS v (declarations of fields of class)
```

Class declarations may occur only in the outer declarations of a module. The fields of a class can be variables of any data type. The order in which they occur in the class declaration is the order in which they occur in memory. Thus, a class can be a template to be overlayed on an already-existing data structure, such as might be created by a procedure in some other language and passed to MAINSAIL.

Classes are also used to describe module interfaces; such classes may include procedure fields.

10.3. Record Allocation and Disposal

Any number of new records of a class may be created at runtime by calls to the system procedure "new". "p := new(c)" allocates a record that conforms to the class c and sets the pointer p to reference the record. All the fields of the newly allocated record are cleared.

The storage occupied by a record pointed to by p is freed by "dispose(p)". Records no longer pointed to by any pointer are automatically disposed by the MAINSAIL garbage collector.

Arrays and module data sections may also be disposed.

10.4. Classified Pointers

A pointer p can be declared as "POINTER(c) p" to indicate that it will point only to records of class "c". Such a pointer is a "classified pointer". The compiler ensures that classified pointers are not mistakenly used to refer to records in other classes.

10.5. Unclassified Pointers

The class in a pointer variable declaration may be omitted if the pointer is to be used for an unknown class, or for several different classes. The programmer must be especially careful when using unclassified pointers since class checking is not provided for them.

10.6. Accessing Fields of Records

A field f of a record of a class c pointed to by a pointer p, declared as "POINTER(c) p", is accessed by the field variable "p.f". "p" is the base part, and "f" the field part.

The base part may itself be a field variable. For example, "p.q.f" has the base part "p.q", where q is a pointer field of the record pointed to by p. Base parts may also be subscripted variables (e.g., "a[i].f"), procedure calls (e.g., "proc(parms).f"), or parenthesized pointer expressions (e.g., "(p := q).f").

10.7. Explicit Classes in Field Variables

The variable "p:c.f", where p is a pointer expression, c is a class and f is a field of c, provides a means of explicitly specifying the class of a pointer at the point of use as a base part.

10.8. Prefix Classes

A class can inherit its initial fields from a previously declared class, called its "prefix class". The form of a declaration for such a class is:

```
CLASS(c1) c2 (declarations for additional fields)
```

In this case c1 is the prefix class. If c1's declaration were:

```
CLASS c1 (INTEGER i,j,k)
```

then the effect on c2 would be the same as:

```
CLASS c2 (INTEGER i,j,k; declarations for additional fields)
```

Two classes are "related" if one is a prefix class of the other or if they are the same class. Pointer assignments and argument-parameter matching are allowed only between related classes.

MAINSAIL's prefix classes play a role similar to that of Pascal's "variant records", though prefix classes are a simpler concept and require no runtime overhead. Prefix classes allow classes that share some fields to have these common fields "abstracted out" into a separate class. Procedures that manipulate pointers to a prefix class cannot mistakenly access fields of prefixed classes.

11. Procedures

11.1. Procedure Declarations

The basic form of a procedure declaration is:

```
type PROCEDURE v (declarations list for parameters);
procedure body
```

"type" is present only if the procedure is to return a value; the parameter list may be omitted if there are no parameters. The procedure body is either a statement, or a list of local variable declarations followed by a list of statements, all within a "BEGIN"-"END" pair. Procedures cannot be statically nested; i.e., a procedure body cannot contain a procedure. The initial values of the local variables are not defined, except for uses and modifies parameters, as described below.

11.2. Procedure Calls

A procedure call has the form "p(e1,...,en)", where p is a procedure, and the ei are arguments. The order of evaluation of the ei is not specified. If p has no arguments, the parenthesized list may be omitted. Any procedure may be invoked recursively.

11.3. Procedure Parameters

There are three kinds of parameters, distinguished in the parameter declarations by the qualifiers "USES", "PRODUCES", and "MODIFIES". A uses parameter (the default) is initialized by the value of the argument. A produces parameter is not initialized by its argument, but instead sends its final value back to the argument (which must be a variable) when the procedure returns. A modifies parameter has the charcteristics of both uses and produces parameters, i.e., is initialized by the argument and sends its value back to the argument.

A parameter behaves like a local variable inside the body of the procedure. In particular, modifies and produces parameters are not passed as the address of the corresponding arguments, as is the case for the "reference" and "name" parameters of ALGOL or FORTRAN.

11.4. Optional Arguments

Trailing parameters may be qualified with "OPTIONAL" to indicate that their arguments may be omitted in procedure calls, in which case the compiler passes Zero values of the appropriate data type. This allows little-used parameters to be left out of most calls.

11.5. Repeatable Arguments

Trailing parameters of an untyped procedure may be qualified with "REPEATABLE" to indicate that a call may give several sets of arguments for the repeatable group of parameters. The compiler acts as if several calls had been explicitly made, with arguments before the repeated ones computed just once, and passed for each call. This allows constructs such as "write(f,a,b,c)", which is equivalent to:

```
write(f,a); write(f,b); write(f,c)
```

except that f may be evaluated just once.

11.6. Forward Procedures

A procedure must be declared before it can be called. However, if two procedures call each other, a vicious circle results since each must be declared before the other. To get around this problem, one of the procedures is first given a "forward" declaration, which is like a normal declaration except it is qualified with "FORWARD", and just the procedure header (not the body) is given. Later the procedure is declared as usual; the compiler automatically figures out that a previously declared forward procedure is now being given a body. Calls to the procedure may appear at any point after the forward declaration.

11.7. Compiletime Libraries

A related use of forward procedure declarations is to specify the name of the file that contains the complete procedure declaration. This is done by "FORWARD(f)" where f is a string constant for the name of the file. If at the end of compilation the procedure has been called, but no body has been declared for it, the compiler automatically compiles the file f, expecting to encounter the procedure's declaration.

This mechanism allows the creation of "compiletime libraries" that contain full procedure declarations for commonly used procedures and are automatically accessed by the compiler to obtain bodies of called procedures. In most cases, however, intmods are a better way of organizing a set of procedures; see Chapter 15.

11.8. Inline Procedures

Procedures or procedure calls may be marked with the keyword "INLINE", which causes the procedure or particular call to be expanded inline. This avoids the usual procedure call overhead at the expense of producing more code. It is appropriate to declare small, frequently-called procedures inline, since the procedure call overhead may represent a substantial fraction of the execution time of the procedure.

11.9. Generic Procedures

A generic procedure allows a single identifier to represent several procedures. A particular one is chosen in a procedure call as determined by the data types of the arguments. This is a convenience to the programmer in that it allows the same name to be used for related procedures.

An example of a generic procedure declaration is:

```
GENERIC PROCEDURE p "p1,p2,...,pn"
```

where the pi are procedure identifiers. Whenever "p" is used in a procedure call, the compiler acts as if "p1" had been used instead, except that if some "error" occurs (e.g., argument-parameter type mismatch), the compiler "backs up" and acts as if p2 had been used instead of p1. If another "error" occurs, the compiler proceeds to p3, and so forth, until a pi is found that causes no error (the compiler complains if no such pi is found).

The generic mechanism can be combined with repeatable arguments to provide a quite general procedure calling capability with no execution time overhead. For example, the generic system procedure "write" allows any number and any type of arguments to be written to several different kinds of files, or to a string, or to memory, all based on the generic and repeatable mechanisms.

Generic procedures (including predefined system procedures) may be extended by the user. For example, if a generic procedure is originally declared as:

```
GENERIC PROCEDURE p "a,b,c"
```

then the declaration:

```
GENERIC PROCEDURE p "x,y,z"
```

is concatenated with the original declaration, so that the effect is as if p had been declared as:

```
GENERIC PROCEDURE p "x,y,z,a,b,c"
```

12. Modules

Modules are used to divide a program into small, manageable units that are separately compiled and manipulated during execution. A program is a collection of modules, some of which are contributed by the programmer, and others by MAINSAIL's runtime system.

Many instances of a module may exist, and modules may be manipulated by code that knows only a limited amount about the module. These capabilites permit an "object-oriented" style of programming, in which each instance of a MAINSAIL module is treated as an object.

MAINSAIL's approach to modules goes well beyond that found in other algorithmic programming languages, and provides the programmer with previously unavailable capabilities, such as dynamic linking and loading, which have a profound effect on program organization and portability.

Unlike most programming languages, MAINSAIL does not utilize a "link" step prior to execution. Instead, the modules are brought into memory as needed during execution and MAINSAIL provides all the facilities for intermodule communication. The programmer need never specify what modules make up a program; a program is an open-ended collection of modules the identity of which need not be determined until execution time. The runtime selection of modules provides a degree of flexibility lacking in statically linked systems. It also frees MAINSAIL from any dependence on machine-specific linkers, with their attendant restrictions and peculiarities.

Only the currently executing module need be in memory; MAINSAIL automatically swaps modules in and out of memory during execution. It tries to keep a "working set" of modules in memory. Modules are compiled into position-independent code so that they may reside anywhere in memory, and may even be moved about in memory during execution.

```
+----------+   +----------+   +----------+        +----------+
|INTERFACE |   |INTERFACE |   |INTERFACE |        |INTERFACE |
| (PUBLIC) |   | (PUBLIC) |   | (PUBLIC) |        | (PUBLIC) |
+----------+   +----------+   +----------+        +----------+
|          |   |          |   |          |  ...   |          |
| PRIVATE  |   | PRIVATE  |   | PRIVATE  |        | PRIVATE  |
|          |   |          |   |          |        |          |
+----------+   +----------+   +----------+        +----------+

  MODULE 1       MODULE 2       MODULE 3            MODULE n
```

A module is written according to the following general layout:

```
+-------------------------+
| BEGIN "modNam"          |
| +--------------------+  |
| | outer              |  |
| | declarations 1     |  |
| +--------------------+  |
| +--------------------+  |
| | procedure 1        |  |
| +--------------------+  |
|           .             |
|           .             |
|           .             |
|                         |
| +--------------------+  |
| | outer              |  |
| | declarations n     |  |
| +--------------------+  |
| +--------------------+  |
| | procedure n        |  |
| +--------------------+  |
| END "modNam"            |
+-------------------------+
```

"modNam" gives the name of the module, which must be an identifier of six characters or fewer. Since the runtime system uses a module's name to identify it, every module in a program must have a unique name.

The "outer declarations" declare variables to be accessible throughout the remainder of the module, but not in any other modules. The outer declarations of a module m must include declarations for all modules referenced by m. In addition, m must declare itself if it has any interface fields.

12.1. Module Allocation and Communication

Modules communicate through "interface fields", which are the variables and procedures of each module that the programmer declares to be accessible from other modules. The interface fields of a module are analogous to the interface fields of a record. Data (non-procedure) interface fields reside in a data structure called a "data section". Data sections exist in "bound" and "nonbound" forms.

Nonbound data sections are allocated by means of the procedure "new". Nonbound data section interface fields are always accessed with explicit pointers; the syntax used is identical to that for records. A module may have more than one nonbound data section.

Bound data sections are allocated by means of "bind". The interface field of a bound data section may be accessed by name (with no module or pointer prefix) if the reference is unambiguous; bound data sections may also be accessed with explicit pointers. A module may not have more than one bound data section at any given time. Bound data sections allow the programmer to write code without knowing the name of the module in which an interface variable or procedure actually resides, so that a system of many modules can be reconfigured with minimal source code changes.

Data sections contain the outer and local own variables of a module as well as the interface fields, but only the interface variables are accessible from outside the module.

When a module's data section is allocated with "bind" or "new", the interface and own variables in the data section are cleared, and the initial procedure is invoked. MAINSAIL automatically allocates a bound data section the first time one of its interface procedures is called, if it has not already been allocated.

At the end of program execution, MAINSAIL automatically invokes the final procedures associated with any active modules and then closes any open files.

12.2. Module Syntax

The most common form of a module declaration is:

```
MODULE v (declarations of interface fields)
```

where v is the module's name. Interface fields may be variables and/or procedures, in any order. The declaration of an interface procedure gives only the header. It serves as a forward declaration for the procedure. The procedure body must be given within the module v, where the procedure is declared as usual.

A module's fields can also be supplied by means of a class, with a declaration such as "MODULE(c) m (additional fields)", where c is a class that specifies the first fields of m's interface and the parenthesized list of additional field declarations is optional.

Each module may contain at most one typeless and parameterless "initial procedure" that is to be called whenever a data section for the module is allocated. This procedure is qualified with the keyword "INITIAL". A module invoked from the MAINSAIL executive leads to execution of an entire program by executing the module's initial procedure. Thus, initial procedures play two roles: one is to perform any kind of initialization for a module, and the other is to lead to execution of what the user views as a program. As far as the MAINSAIL runtime system is concerned, a program is just a set of modules that are brought into memory and initialized.

Each module may contain a single typeless and parameterless "final procedure" that is automatically invoked when the module is disposed. This procedure is qualified with the keyword "FINAL".

12.3. Combining Module Declarations in One File

The outer declarations of a module m include declarations for all modules accessed by m. A module must be declared identically (up to the last interface field accessed) in all modules that access any of its interface fields.

To ensure consistent module declarations, and to save retyping and updating the declarations in every module of a program, all module declarations for a related group of modules can be put in a single file, and that file included in the compilation of all the modules by means of the "SOURCEFILE" compiler directive. Alternatively, the declarations may be stored in a compiler symbol table file (an intmod) for faster processing at compiletime. MAINSAIL's compiler directives have been chosen to provide flexibility in the construction of the source text for large programs.

12.4. Objmod Libraries

An objmod (object module) library is a file that contains object modules. A librarian program, MODLIB, is provided for maintaining objmod libraries. Any number of objmod libraries may be open for execution. MAINSAIL automatically searches all open libraries to find a module. If a module is not found in any objmod library, a file name is formed from the name of the module, and an attempt is made to obtain the object module from that file. If that file cannot be opened, MAINSAIL may ask the user during execution where the module resides.

A means is provided for indicating what file contains an object module, and for dynamically associating the "true" module name for a "dummy" module name, thereby allowing programs to be written without knowledge of the specific physical module that provides a given service.

13. Macros

A macro allows an identifier to represent either a constant or arbitrary text. Each occurrence of the macro identifier (a macro call) is replaced by the compiler with the associated constant or text specified when the macro was defined.

13.1. "DEFINE"

The form of a simple macro definition is:

```
DEFINE v1 = macroBody1, ..., vn = macroBodyN;
```

where the vi are identifiers and macro bodies are constants or "bracketed text", e.g., "[xxx]" where "xxx" is arbitrary text.

The form of a definition for a macro with parameters is:

```
DEFINE v(v1, ..., vn) = [text];
```

where the macro identifier v is followed by a parenthesized list of parameter identifiers (the vi) that may be used within the bracketed text. Subsequent occurrences of v (i.e., macro calls) are followed by a parenthesized list of arguments, much like a procedure call. Each occurrence of the identifier vi within the bracketed text (even within string constants and comments) is replaced with the corresponding argument text.

A macro definition may occur virtually anywhere in a program, even in the midst of an expression, for example. However, a macro definition cannot occur in the midst of another definition, except within bracketed text.

Macro calls may be used anywhere, even in subsequent macro definitions.

"REDEFINE" is used to change the body of a previously defined macro. For example, "DEFINE n = 0; ... REDEFINE n = n + 1" does compiletime counting.

13.2. Macro Constants

Macro constants (constants used as macro bodies) may be constants of any data type. Defining a macro identifier to be equal to a constant (or constant expression) allows a meaningful name to be used in the program text instead of a "bare" constant. If the value of the constant expression needs to be changed, then only the macro definition has to be changed instead of all occurrences of the constant expression.

13.3. Interactive Define and Redefine

A macro definition may omit the "=" and subsequent macro body, in which case the compiler prompts for the macro body. Such interactive definitions allow the programmer to make limited contributions to the source text during compilation, in lieu of having to edit the source text just prior to compilation. For example, "DEFINE debugOption;" prompts with "DEFINE DEBUGOPTION =" during compilation. The user types in a value "v", and the compilation proceeds as if it had instead encountered "DEFINE debugOption = v;".

13.4. Macro Calls

A "macro call" is the occurrence of a macro identifier at any point in a program after it has been defined. It directs the compiler to scan the body of the macro as if it appeared in place of the macro call.

If the macro was defined with parameters, a parenthesized list of macro arguments separated with commas may appear after the macro identifier. The macro arguments replace all occurrences of the corresponding parameter identifiers in the macro body.

14. Compiler Directives and Conditional Compilation

A compiler directive indicates which source text is to be compiled or conveys information to the compiler that is used while compiling the program.

A compiler directive may occur wherever a declaration or statement may occur (except that "BEGINSCAN" must be the first thing on a page), and must be terminated with a semicolon.

14.1. "MESSAGE"

"MESSAGE c" writes the string constant "c" onto a new line of logFile during compilation.

14.2. "SOURCEFILE"

"SOURCEFILE c" directs the compiler to compile the file with name "c" as if it appeared in place of the directive. Sourcefile nesting may occur to any level. The name "c" can be obtained interactively; e.g.:

```
DEFINE f; SOURCEFILE f;
```

prompts the user for the name of the source file, then uses the string constant macro f in the source file directive.

14.3. Checking, Arithmetic Checking, and Optimization

Checking directs the compiler to emit code to check certain conditions at runtime that cannot be determined at compiletime. Arithmetic checking directs the compiler to emit code to check for (long) integer overflows, if such code is necessary. Optimization directs the compiler to make efforts to produce the best code it knows how (at the expense of a longer compilation time).

Directives are provided to control checking, arithmetic checking, and optimization at the module level as well as on a per-procedure or even per-expression basis.

14.4. "IFC", "THENC", "$EFC", "ELSEC", and "ENDC"

The conditional compilation form:

```
IFC c THENC text1 $EFC c2 THENC text2 ELSEC text3 ENDC
```

where c is an expression evaluated at compiletime, causes the compiler to compile text1 and ignore text2 and text3 if c is non-Zero. If c is Zero and c2 non-Zero, then text2 is compiled and text1 and text3 ignored. If c and c2 are both Zero, text3 is compiled and text1 and text2 ignored. There may be zero, one, or many "$EFC ci THENC texti" parts, and "ELSEC text3" may be omitted. IFC's may be nested to any depth.

"$CASEC" and "$DOC" provide compiletime analogues of Case and Iterative Statements, just as "IFC" is analogous to the If Statement.

14.5. "DCL"

"DCL(<identifier>)" is true if the identifier has been declared or defined and false otherwise. "DCL" is useful in conjunction with conditional compilation.

14.6. "DSP"

"DSP(c.f)" returns the offset in storage units to field "f" of class "c".

14.7. "$TYPEOF", "$CLASSOF", "$ISCONSTANT"

"$TYPEOF(x)" returns the type code for the expression x. "$CLASSOF(x)" returns the class name of the expression x. "$ISCONSTANT(x)" returns true if and only if the expression x is constant, i.e., can be evaluated at compiletime. All three are pseudo-procedures evaluated at compiletime, and all three discard the expression x without actually evaluating it.

14.8. Scanning Directives

The scanning directives are "BEGINSCAN", "SKIPSCAN", and "DONESCAN". "SKIPSCAN" and "BEGINSCAN" may be followed by string constant names. "SKIPSCAN" specifies that source text is to be skipped up to the next appropriately named "BEGINSCAN" directive. "BEGINSCAN" is used to mark places in the source text where a "SKIPSCAN" search may terminate. "DONESCAN" causes the compiler to terminate compilation of the current file as if end-of-file had been reached.

These directives allow essentially arbitrary use of text files as repositories of fragments of source text that [illegible] "scooped up" during the compilation of many different modules. When combined with the flexible conditional compilation directives, the programmer can piece together a program in almost any manner from any number of files, in accordance with options specified interactively during compilation.

14.9. "NEEDBODY" and "NEEDANYBODIES"

The compiletime pseudo-procedures "NEEDBODY" and "NEEDANYBODIES" are used in conjunction with the "FORWARD" qualifier to determine whether a forward procedure needs a body, i.e., has been called but has not yet been given a declaration containing the procedure body. They are typically used in compiletime libraries to ensure that the compiler sees just those procedures for which bodies are needed.

The form "NEEDBODY(id)" is true if id is the name of a procedure that has been declared "FORWARD" and has appeared in a procedure call, but has not been declared with a body.

"NEEDANYBODIES(c)" is true if there are any procedure bodies in the file "c" that need to be compiled.

The form "NEEDANYBODIES" is equivalent to "NEEDANYBODIES(c)" where c is the name of the file that caused the current automatic sourcefile.

14.10. "$DIRECTIVE" Directives

"$DIRECTIVE" is a directive that takes a string argument. The string argument begins with a keyword that functions as a directive. "$DIRECTIVE" is used for a number of directives with miscellaneous functions; many of them allow compiler subcommands to be inserted into the source text of a module.

15. Intmods

Intmods allow the compiler's symbol table for a module to be preserved. The symbols in it may be used in the compilation of other modules or by MAINSAIL system programs such as MAINDEBUG and MAINPM. When the symbols are used in a compilation, an intmod may contain symbols used by many modules. Such an intmod replaces a "header" file, which would be recompiled once for each module that uses its symbols, with a module compiled just once to produce the intmod. A header file is a file that contains definitions and declarations (e.g., of classes and modules) of use to several modules.

An intmod is produced for a module in which the compiler encounters the "SAVEON" directive. The form of the "SAVEON" directive is "SAVEON c", where "c" is a string constant expression for the name of the intmod on which the symbol table is to be saved.

The "RESTOREFROM m" directive may be used to make an intmod m "visible" to another module, meaning that the symbols in it can be used in the other module (the symbols themselves are also said to be visible). Symbols that are not visible can be used only with a special qualifier of the form:

```
<intmod name>$<symbol name>
```

Directives exist to specify that only certain symbols in an intmod should be made visible (usable without the qualifier) to modules that make the intmod visible, thereby providing information hiding. For example, if only certain procedures from an intmod are to be visible, the programmer may still include supporting procedures in the intmod without making the identifiers for those procedures visible in other modules (possibly conflicting with identifiers from the other module).

Identifiers from an intmod may be outer declarations of the intmod (variables, classes, modules, macros, etc.) and procedures. The procedures are incorporated into the using module only if they are called.

16. Exceptions

An exception is an unusual or erroneous condition that occurs during a program's execution. When an exception occurs, it may cause the execution of a statement called an exception handler. A handler may, for some exceptions, repair the error and resume execution at the place where the exception occurred, or it may recover from the error by aborting the execution of one or more nested statements (including procedure invocations), or it may propagate the exception to another handler. If there is no handler for an exception, the MAINSAIL runtime system reports an error by calling the system procedure errMsg.

Exceptions are divided into two categories: those predefined by MAINSAIL and those known only to user programs. User exceptions must be explicitly caused by the user program by means of the system procedure $raise; predefined exceptions are caused by MAINSAIL.

16.1. Handle Statement

A Handle Statement associates an exception handler with a statement in the program (called the handled statement) and executes the handled statement. If an exception occurs during the execution of the handled statement, that statement's execution is interrupted and the handler is executed. If no exception occurs, the handler is ignored.

The general form of the Handle Statement is:

```
$HANDLE s1 $WITH s2
```

in which s1 and s2 are statements. The statement s1 is the handled statement and s2 is the handler.

When a Handle Statement is executed, its statement s1 is initiated. If an exception occurs during s1's execution (which may entail several levels of procedure calls), and the exception has not been handled by another Handle Statement initiated during s1's execution, then s1's execution is suspended and the exception handler s2 is executed. Otherwise, s2 is ignored.

A handler can either recover from an exception and allow the program's execution to continue or it can propagate the exception to another exception handler. In the first case, the handler is said to have handled the exception.

16.2. Handling Exceptions

There are two ways a handler can allow a program's execution to continue:

- If the exception that occurred was caused by a call to the system procedure $raise, the handler can resume execution at the place where the exception occurred by calling the system procedure $raiseReturn. This terminates the handler's execution. When s1 terminates after its execution is resumed, s2 is ignored. If the exception that occurred was not caused by a call to $raise, a runtime error occurs if $raiseReturn is called to continue from the exception.

- The handler can terminate the Handle Statement's execution in one of three ways:

    1. It can resume execution at the statement following the Handle Statement, if any, by having control fall out of the handler.

    2. If the Handle Statement is contained within an Iterative Statement, the handler can terminate the Handle Statement's execution and either repeat or terminate the execution of the Iterative Statement by means of a Continue or Done Statement.

    3. The handler can terminate the Handle Statement's execution and return from the procedure containing the Handle Statement by means of a Return Statement.

When a handler terminates the execution of its Handle Statement, the handled statement s1, the execution of which was suspended by the occurrence of the exception, is aborted, along with all other statements initiated as a result of s1's execution (and all procedures thus invoked). When a procedure is aborted in this manner, if it contains any active handled statements, MAINSAIL raises the predefined exception $abortProcedureExcpt and executes the handlers associated with the handled statements. This gives each procedure a chance to do any cleaning up that might be necessary before it is aborted.

16.3. Propagating Exceptions

If a handler is unable to handle an exception, it can propagate the exception to the next handler by calling the system procedure $raise with no arguments. If there is no next handler within the user program, the MAINSAIL runtime system reports an error by calling the system procedure errMsg.

16.4. Information about the Current Exception

A handler can obtain the name of the current exception by calling the system macro $exceptionName. Other information about the current exception is available by calling $exceptionStringArg1, $exceptionStringArg2, $exceptionPointerArg, $exceptionBits, and $exceptionCoroutine.

16.5. Nested Exceptions

Exceptions can be nested. If an exception occurs during a handler's execution, the handler's execution is suspended and a handler for the new exception is searched for and initiated, as described above. If the new handler resumes execution at the place where the new exception occurred, the previous exception is restored to being the current exception and the execution of its handler continues. If the new handler aborts the execution of its Handle Statement and that Handle Statement caused the execution of the previous handler's Handle Statement, the execution of the previous handler's Handle Statement is also aborted.

17. Coroutines

A "coroutine" is a context that preserves the state of a procedure so that its execution may be "resumed" at the preserved state by a procedure in some other coroutine. System procedures are provided to create, resume, and kill coroutines.

A coroutine may be thought of as a "thread of execution" that progresses independently of other threads of execution in an interleaved fashion. Thus, a coroutine executes for a while and then explicitly resumes some other coroutine. That new coroutine executes for a while, and then explicitly resumes another coroutine, perhaps the one that resumed it. Coroutines must explictly resume other coroutines; i.e., coroutines do not execute in parallel, and there is no automatic timer interrupt that causes resumption of coroutines. However, user or system procedures may resume coroutines in such a way as to give the illusion of parallel execution.

Example 17-1 shows a use of coroutines. generateNextNode generates a node in some data structure, and processNextNode processes the node.

The procedure processNextNode first creates a coroutine for the generator. The arguments indicate the data section and procedure name where the first resumption is to start. processNextNode then uses the returned pointer to resume the coroutine to get each node, and finally to kill it. There can be any number of procedures like processNextNode that use generateNextNode. generateNextNode sets the outer variable nextNode to point to the next node, and then resumes the coroutine in which processNextNode is running. Outer or interface variables must be used for communication among coroutines, since $resumeCoroutine does not have any parameters for intercoroutine communication.

17.1. Coroutine Implementation

A coroutine consists of a stack to hold the procedure frames and a record of the predeclared class $coroutine, which contains information about the coroutine.

A tree structure is imposed on coroutine records based on a parent-child relationship. Each coroutine record has a link $up that points to its parent coroutine record, i.e., the coroutine that created it. The link $down points to the first-created (oldest) child. $right points to the next younger sibling, and $left points to the next older sibling.

The $up, $down, $left, and $right links are "structural" links in that they depend on the order of coroutine creation. In addition, $coroutine records are maintained on a "dynamic" list by

```
POINTER(node) nextNode;
POINTER($coroutine) processCoroutine;

PROCEDURE generateNextNode;
BEGIN
...
DOB ...
    nextNode := ...;
    $resumeCoroutine(processCoroutine);
    ...
    END;
END;

PROCEDURE processNextNode;
BEGIN
POINTER($coroutine) p;
...
processCoroutine := $thisCoroutine;
p := $createCoroutine(thisDataSection,"generateNextNode");
DOB ...
    $resumeCoroutine(p);
    ... use nextNode here ...
    END;
$killCoroutine(p);
...
END;
```

Example 17-1. Generator/Processor Coroutines

means the $prev and $next links. Each time a coroutine is resumed, it is moved to the head of this list. The head of the list is pointed to by the predeclared variable $thisCoroutine, so that $thisCoroutine points to the record for the currently executing coroutine, and "$thisCoroutine.$next" points to the coroutine that most recently resumed the current one.

17.2. Coroutines and Exceptions

A parameter to $raise may be used to initiate handling of an exception in a coroutine other than the current one.

Exceptions not handled in a child coroutine are propagated to the coroutine's ancestors; first to its parent, then its grandparent, and so on, until the root coroutine "MAINSAIL" is reached. $raiseReturn erases traces of the current exception in all affected coroutines and resumes the coroutine from which the exception was raised.

18. Files

A file is an ordered series of data with a beginning position, a current position, and possibly an ending position. A file may reside on some external medium (e.g., an operating system's file structure) that is not defined by MAINSAIL or under MAINSAIL's complete control.

Some files may exist independently of the execution of a program, so that a program can create a file that can later be accessed by another program. Thus, files can provide continuity from one program execution to another.

Every file has a name, which is represented in a MAINSAIL program as a string.

MAINSAIL distinguishes between "text files" and "data files". A text file is composed of character units, a data file of storage units. MAINSAIL also distinguishes two methods of access to a file: sequential and random. The current position in a sequential file is updated to point to the next datum as each datum is read or written, in order, starting from the beginning. The current position of a random file may be explicitly changed to be anywhere within the file by means of the procedure "setPos".

18.1. File I/O

System procedures are provided for opening, closing, deleting, and renaming files, for reading from a file, and for writing to a file.

Before a file can be used by a program, it must be "opened" by a call to the system procedure "open". Arguments to the open procedure specify the file name and indicate how the file is to be accessed (sequentially or randomly, for input and/or output, etc.). A file is "closed" by a call to the system procedure "close" to indicate that it will no longer be used by the program, unless opened again.

A file is referenced in a MAINSAIL program with a pointer returned by the open procedure. The pointer belongs to one of the predeclared classes "textFile" or "dataFile".

18.2. Text Files

When a (long) integer, (long) real, or (long) bits value is written to a text file (with the system procedure "write"), an automatic conversion is made to the appropriate string representation of that value, and then the string is written to the file. MAINSAIL does not insert any additional characters such as blanks or tabs, so that the user has full control over the layout of the file.

When the non-string data types are read from a text file (with the system procedure "read"), a scan for an appropriate string representation takes place, and when found, a conversion is made to the appropriate internal representation.

A string is read from a file by using "read", which gets the next line; "fldRead", which reads a string of a specified width; or "scan", which employs a scan table to break on a specified set of characters.

18.3. Terminal I/O

"TTY" is the file name associated with the user's terminal, or with the operating-system-dependent standard input. "ttyRead" and "ttyWrite" are system procedures used for explicit communication with "TTY". ttyRead reads a line typed by the user, and ttyWrite writes a string to the terminal.

18.4. Memory Files and Data Sinks

Files maintained in memory may be accessed with any name beginning with "MEM>" (except that the ">" character may be replaced with another character on some operating systems). A data sink file may be accessed with a name beginning with "NUL>". Data written to such a file are discarded; on input, a data sink file always acts as if end-of-file had been reached.

19. Areas

Areas provide a method for precise control of memory management. Data structures (records, arrays, data sections, and string text) may be grouped into areas, and the memory management of each area controlled separately. Areas are appropriate for programs that deallocate large conglomerations of data at a time. Careless use of areas, however, can cause bugs that are very difficult to track. Programs can be written without reference to areas; the use of areas is purely an optimization.

19.1. Advantages of Areas

An area can contain a single large, complex data structure. When the data structure is no longer needed, the entire area can be disposed. The disposal of an entire area has several advantages over individually disposing each chunk (record, array, or data section) in it:

- It is faster.
- Entire pages become free, rather than individual chunks (the memory manager makes better use of free pages than of free chunks).
- It is easy to be sure that entire data structure has been freed.

An area may also be marked as not subject to garbage collection. This is useful if no garbage is being generated in the data structures in the area, since garbage collections can be shorter.

19.2. Area Facilities

Procedures are provided to allocate areas, clear them (remove all data from them), and deallocate them. Parameters are provided to most procedures that allocate chunks or string text to specify the area into which the data are to go. If the area parameters are omitted, data automatically go into the common area, $defaultArea.

19.3. Area Caveats

To avoid bugs that are difficult to track and reproduce, the programmer must be careful about the following:

- A pointer or string referencing data in a disposed area is said to be "dangling". Use of a dangling pointer or string has undefined effects.

- A program must not pass to a system procedure or macro or assign to a system variable any pointer or string referencing data in an area that is to be disposed or cleared before MAINSAIL exits.

Because these rules are not always easy to follow, the use of areas is recommended only for experienced programmers writing programs in which the benefits of areas are really needed.

20. System Procedures, Macros, and Variables

Predefined procedures, macros, and variables provide services that support the execution of MAINSAIL programs. Much of the power of MAINSAIL comes from the large number of system procedures.

20.1. System Procedures, Variables, and Macros Summary

Table 20.1-1 contains a summary of all MAINSAIL system procedures, variables, and macros.

```
open             open a file
$reOpen          open a file with new open bits
close            close a file
$closedFile      determine whether a file has been closed

$createUniqueFile
                 create file with unique name

$devModBrk       device module name break character
$devModBrkStr    string consisting of $devModBrk

$delete          delete a file
$rename          rename a file

$copyFile        copy (part of) one file to another

$truncateFile    truncate a file to given length

getPos           get file position
setPos           set file position
relPos           set relative file position
$getEofPos       get end-of-file position of byte-stream
                 file
```

Table 20.1-1. System Procedures, Macros, and Variables Summary (continued)

eof	true when positioned at or beyond end-of-file
$gotValue	determine if actually read last value; better way of checking for end-of-file
read	read values
write	write values
$storageUnitRead	read a number of data efficiently from a file
$storageUnitWrite	write a number of data efficiently to a file
$characterRead	read a number of characters efficiently from a file
$characterWrite	write a number of characters efficiently to a file
$pageRead	read a page of data from a file
$pageWrite	write a page of data to a file
cRead	read a character from file, string, or charadr
cWrite	write characters to file, string, or charadr
$clearFileCache	uncache all or part of file
$queryFileCacheParms	information about file cache
$setFileCacheParms	control file cache
$concat	concatenate strings (same as "&" operator)
$dup	perform multiple concatentations
rcRead	reverse character read (from the end of a string)
rcWrite	reverse character write (to the beginning of a string)

Table 20.1-1. System Procedures, Macros, and Variables Summary (continued)

fldRead	read a string field
fldWrite	write a string field
ttyRead	read a line from "TTY"
ttyWrite	write values to "TTY"
ttycWrite	write characters to "TTY"
$removeBoolean	parse boolean string
$removeBits	parse bits string
$removeInteger	parse integer string
$removeReal	parse real string
confirm	get yes/no confirmation
cmdMatch	match a command (command recognition)
errMsg	raise an exception and/or write a message and get a response
cmdFile	standard input file
logFile	standard output file
enterLogicalName	establish logical file name
lookUpLogicalName	find logical file name
$setSearchPath	set file searchpath
$globalLookup	look up global symbol
$globalEnter	enter global symbol
$globalRemove	remove global symbol
$registerException	register an exception name so that it can be raised in response to an errMsg prompt
$deRegisterException	undo $registerException
$newException	assign a unique exception name
$raise	raise an exception
$raiseReturn	terminate an exception handler

Table 20.1-1. System Procedures, Macros, and Variables Summary (continued)

```
$exceptionBits  return information about current exception
$exceptionName  return name of current exception
$exceptionCoroutine
                return raising coroutine of current
                exception
$exceptionPointerArg
                return pointer argument of current
                exception
$exceptionStringArg1, $exceptionStringArg2
                return a string argument of current
                exception

scanSet         set up scan bit
$scanSet        set up scan integer
scanRel         release scan bits or integers

scan            scan a file or string according to a
                scan specification

$removeLeadingBlankSpace, $removeTrailingBlankSpace
                remove blank space from string
$removeWord     remove non-blank chars from string

$formParagraph  fill and justify string
```

Table 20.1-1. System Procedures, Macros, and Variables Summary (continued)

$cvbo	convert to boolean
cvi	convert to integer
cvli	convert to long integer
cvr	convert to real
cvlr	convert to long real
cvb	convert to bits
cvlb	convert to long bits
cvs	convert to string
cvp	convert to pointer
cva	convert to address
cvc	convert to charadr
cvAry	convert to array
cvcs	convert a character code to a single-character string
cvu	convert to upper case
cvl	convert to lower case
$length	length of result of cvs
first	first character of a string
last	last character of a string
length	number of characters in a string
compare	-1, 0 or 1 as result of (optionally "caseless") comparison of two strings
equ	checks (optionally "caseless") equality of two strings
isLowerCase	true if argument is a lowercase letter ("a" through "z")
isUpperCase	true if argument is an uppercase letter ("A" through "Z")
isAlpha	true if argument is a letter ("A" through "Z" or "a" through "z")

Table 20.1-1. System Procedures, Macros, and Variables Summary (continued)

nextAlpha	alphabetically next character after argument character
prevAlpha	alphabetically previous character before argument character
isNul	true if argument is a "null" character
copy	copy a record, array, memory, or characters
clear	clear a record, array, memory, or characters
newUpperBound	adjust the upper bound of a one-dimensional array
$adrOfFirstElement	get the address of the first element of an array
new	allocate a record, array, or data section
$newRecords	allocate multiple records
dispose	dispose of a record, array, data section, or module
bind	bind a module
unBind	unbind a module
$canFindModule	whether a module can be allocated without error
$isBound	whether a module is already bound
$invokeModule	invoke a module the way MAINEX does
$useProgramInterface	true if bound because an interface procedure called
$programName	name under which MAINSAIL was invoked
$getCommandLine	get program arguments
$setCommandLine	set program arguments

Table 20.1-1. System Procedures, Macros, and Variables Summary (continued)

thisDataSection	return pointer to current data section
$moduleName	return name of module, given data section pointer
$searchCallChain	find caller from particular module
$writeCalls	show call stack of coroutine
$fieldInfo	return information about a record or data section field
$className	return name of class of a pointer
$classInfo	return names and types of record or data section fields
$dscrPtr	class descriptor for pointer
$classDscrFor	class descriptor for a given class
$isArray	true if pointer points to an array
$createClassDscr	create a new class at runtime
$createRecord	create a record given a class descriptor
openLibrary	open a module library file
closeLibrary	close a module library file
setModName	set a module name association
relModName	release a module name association
setFileName	set a module file name association
relFileName	release a module file name association
exit	orderly exit from MAINSAIL
fastExit	fast exit from MAINSAIL
$setExitCode	set exit code for operating system

Table 20.1-1. System Procedures, Macros, and Variables Summary (continued)

floor	largest (long) integer not exceeding a (long) real
ceiling	smallest (long) integer not exceeded by a (long) real
truncate	truncate a (long) real to a (long) integer
abs	absolute value of a (long) integer or (long) real
bMask	form a bits mask (sequence of 1-bits)
lbMask	form a long bits mask (sequence of 1-bits)
sin	(long) real sine
cos	(long) real cosine
tan	(long) real tangent
$cot	(long) real cotangent
aSin	(long) real arcsine
aCos	(long) real arccosine
aTan	(long) real arctangent
$atan2	(long) real two-argument arctangent
sinh	(long) real hyperbolic sine
cosh	(long) real hyperbolic cosine
tanh	(long) real hyperbolic tangent
exp	(long) real exponential
ln	(long) real natural logarithm
log	(long) real base-10 logarithm
sqrt	(long) real square root
$log2	truncated base 2 logarithm of constant
$hash	compute hash code
size	size of a class or data type
$ioSize	size of data type when written to file
$bitsPerStorageUnit	bits in a storage unit
$bitsPerChar	bits in a character unit
$typeName	name of a type, given type code

Table 20.1-1. System Procedures, Macros, and Variables Summary (continued)

displace	displace a pointer, address, or charadr
displacement, lDisplacement	distance from one address or charadr to another
eol	end-of-line string
eop	end-of-page string
tab	tab string
$nulChar	null character
$pageSize	storage units per page
$charsPerPage	character units per page
$charsPerStorageUnit	character units per storage unit
(x)Load	load a value (of type x) from an address
cLoad	load a character from a charadr
store	store a value into an address or charadr
newString	make a string from a charadr and an integer (length)
$getToTop	put a string at top of string space
$getInArea	put a string in an area's string space
newPage	get some pages
pageDispose	dispose of pages
newScratch	get some scratch space
$newScratchChars	get some scratch space measured in chars
scratchDispose	dispose of scratch space
$date	get the date
$time	get the time
$dateAndTime	get the date and time simultaneously
$setTheDate	set the date, if necessary

Table 20.1-1. System Procedures, Macros, and Variables Summary (continued)

```
$assembleDate    convert year-month-date combination into
                 standard representation
$assembleTime    convert hour-minute-second combination
                 into standard representation
$assembleDateAndTime
                 combined $assembleDate and $assembleTime
$disassembleDate
                 convert standard representation into
                 year-month-date combination
$disassembleTime
                 convert standard representation into
                 hour-minute-second combination
$disassembleDateAndTime
                 $disassembleDate and $disassembleTime

$dateToStr       convert date representation to string
$timeToStr       convert time representation to string
$dateAndTimeToStr
                 combined $dateToStr and $timeToStr
$strToDate       convert string to date representation
$strToTime       convert string to time representation
$strToDateAndTime
                 combined $strToDate and $strToTime
$removeDateAndTime
                 parse date and time string

$addToDateAndTime
                 add two dates and times
$dateAndTimeDifference
                 subtract two dates and times
$dateAndTimeCompare
                 compare two dates and times

$dateFormat      whether date is GMT, local, or difference
$timeFormat      whether time is GMT, local, or difference

$convertDateAndTime
                 convert GMT time to local or vice versa
$timeSubcommandsSet
                 whether GMT conversion info available
```

Table 20.1-1. System Procedures, Macros, and Variables Summary (continued)

$cpuTime	get system-dependent CPU time for current program
$cpuTimeResolution	number of $cpuTime units per second
$timeout	pause for specified period
$userID	return the system-dependent user ID, if available
$cpuID	return the system-dependent CPU ID, if available
$currentDirectory	name of system-dependent current working or connected directory or catalog
$homeDirectory	home directory or catalog of current user
$directory	list files in a directory
$fileInfo	return information about a file
$moduleInfo	information about objmod
$collect	perform a garbage collection
$checkConsistency	verify that MAINSAIL data structures are in order
$addMemMngModule	specify module to invoke before memory management operations
$removeMemMngModule	undo $addMemMngModule
$collectLock	used to prevent/permit garbage collections
$overheadPercentExitValue	used to prevent thrashing

Table 20.1-1. System Procedures, Macros, and Variables Summary (continued)

```
$areaOf          determine area of pointer or string
$clearArea       empty an area
$clearStrSpc     empty an area's string space
$defaultArea     default area
$disposeArea     reclaim an area
$disposeDataSecsInArea
                 dispose only data sections in area
$findArea        find area with given title
$inArea          determine if pointer or string in given
                 area
$newArea         allocate area

$createCoroutine
                 create a coroutine
$resumeCoroutine
                 continue or start execution in a coroutine
$killCoroutine   get rid of a coroutine
$killedCoroutine
                 determine whether a coroutine has been
                 killed
$moveCoroutine   move coroutine to another point in tree
$findCoroutine   return a pointer to a coroutine record,
                 given its name
$thisCoroutine   current coroutine

$majorVersion, $minorVersion
                 get MAINSAIL version number

$maxChar         maximum character code
$maxInteger      maximum integer
$maxLongInteger  maximum long integer
$minInteger      minimum integer
$minLongInteger  minimum long integer
```

Table 20.1-1. System Procedures, Macros, and Variables Summary (continued)

```
$platformNameAbbreviation, $platformNameFull,
   $platformNumber
                    identify target platform
$systemNameAbbreviation, $systemNameFull, $systemNumber
                    identify target operating system
$processorNameAbbreviation, $processorNameFull,
   $processorNumber
                    identify target processor

$attributes         attributes of target system

$charSet            character set of target operating system

$preferredRadix "natural" radix for addresses, etc.

$compileTimeValue
                    information about current compilation
$thisFileName       file name currently being compiled

$clrConfigurationBit
                    clear bit governing runtime system
$clrSystemBit       clear bit governing runtime system
$setConfigurationBit
                    set bit governing runtime system
$setSystemBit       set bit governing runtime system
$tstConfigurationBit
                    examine bit governing runtime system
$tstSystemBit       examine bit governing runtime system
```

Table 20.1-1. System Procedures, Macros, and Variables Summary (end)

21. Sample MAINSAIL Code

The code for a subset of the MAINSAIL utility HSHMOD is shown in Example 21-2. A program uses HSHMOD by creating a separate HSHMOD data section for each hash table. The header declarations for this subset of HSHMOD are shown in Example 21-1. The declarations for the full HSHMOD are stored in the MAINSAIL system source library, and may be picked up by a user program with:

```
REDEFINE $scanName = "hshHdr"; SOURCEFILE "(system library)";
```

```
# prefix class for hashed records
CLASS hashedRecord (
    STRING key;
    POINTER(hashedRecord) link;
);

# explicit class so user can classify pointers to it
CLASS hshCls (
    PROCEDURE hashInit   (OPTIONAL INTEGER tableSize);
    PROCEDURE hashEnter   (POINTER(hashedRecord) p);
    POINTER(hashedRecord)
    PROCEDURE hashLookUp (STRING key);
    POINTER(hashedRecord)
    PROCEDURE hashRemove (STRING key);
    POINTER(hashedRecord)
    PROCEDURE hashNext  (POINTER(hashedRecord) p);
);

MODULE(hshCls) hshMod;
```

Example 21-1. HSHMOD Declarations

```
BEGIN "hshMod"

# this module maintains a general-purpose hash table

REDEFINE $scanName = "hshHdr"; # Pick up interface
SOURCEFILE "(system library)"; # declarations

DEFINE
    numCharsToHash       =        4,
    defaultTableSize     =        131;

INTEGER numberOfHashLists;

POINTER(hashedRecord) ARRAY(0 TO *) hashList;

#       +--------+
#    0  |        | ---> linked list of all records
#       +--------+        whose keys hash to 0
#    1  |        | ---> linked list of all records
#       +--------+        whose keys hash to 1
#    2  |        | ---> linked list of all records
#       +--------+        whose keys hash to 2
#    3  |        | ---> linked list of all records
#       +--------+        whose keys hash to 3
#       |        |
#
#       |        |
#       +--------+

PROCEDURE hashInit (OPTIONAL INTEGER tableSize);
BEGIN
IF tableSize LEQ 0 THEN tableSize := defaultTableSize;
new(hashList,0,tableSize - 1);
numberOfHashLists := tableSize;
END;
```

Example 21-2. HSHMOD Source Text (continued)

```
INTEGER PROCEDURE hash (STRING key);
BEGIN
INTEGER h,i,j;

# s hashes to
#    (length(s) + 3 * char1 + 5 * char2 +
#       7 * char3 + 9 * char4)
#       MOD numberOfHashLists
#
# where chari represents ith character of s

h := length(key); i := h MIN numCharsToHash; j := 1;
WHILE i .- 1 GEQ 0 DO h .+ cRead(key) * (j .+ 2);
RETURN(h MOD numberOfHashLists) END;



POINTER(hashedRecord) PROCEDURE search
    (STRING key;
     PRODUCES OPTIONAL INTEGER hashValue;
     PRODUCES OPTIONAL POINTER(hashedRecord)
        beforeTarget);
BEGIN
POINTER(hashedRecord) target;

# general-purpose search procedure

IF NOT hashList THEN hashInit;  # automatic initialization
hashValue := hash(key);
beforeTarget :=  NULLPOINTER;
target := hashList[hashValue];
WHILE target AND target.key NEQ key DOB
    beforeTarget := target; target := target.link END;
RETURN(target) END;
```

Example 21-2. HSHMOD Source Text (continued)

```
PROCEDURE hashEnter (POINTER(hashedRecord) p);
BEGIN # enter p at front of its hash list
INTEGER h;
IF NOT hashList THEN hashInit;
IF p THENB
    h := hash(p.key); p.link := hashList[h];
    hashList[h] := p END
EL errMsg("hashEnter: argument is NULLPOINTER") END;



POINTER(hashedRecord) PROCEDURE hashLookUp (STRING key);
RETURN(search(key)); # return record with given key
                     # (Zero if not found)



POINTER(hashedRecord) PROCEDURE hashRemove (STRING key);
BEGIN # remove record with given key
INTEGER               h;
POINTER(hashedRecord)  target,beforeTarget;
IF target := search(key,h,beforeTarget) THEN
    IF beforeTarget THEN beforeTarget.link := target.link
    EL hashList[h] := target.link;
RETURN(target) END;



POINTER(hashedRecord) PROCEDURE hashNext
    (POINTER(hashedRecord) p);
BEGIN
OWN INTEGER h;
POINTER(hashedRecord) q;

# generate next record in hashList (successive calls
# starting with p = NULLPOINTER will generate all records,
# then NULLPOINTER)
```

Example 21-2. HSHMOD Source Text (continued)

```
IF NOT p THEN h := -1
EF q := p.link THEN RETURN(q)
EL h := hash(p.key);
DOB IF h .+ 1 GEQ numberOfHashLists THEN
        RETURN(NULLPOINTER);
    IF p := hashList[h] THEN RETURN(p) END END;

END "hshMod"
```

Example 21-2. HSHMOD Source Text (end)

The MAINSAIL Environment

22. The MAINSAIL Compiler

The MAINSAIL compiler translates MAINSAIL source text into ready-to-run object modules. The compiler has a full set of subcommands governing characteristics of the output object module. Compiler subcommands are listed in Table 22-1.

22.1. Code Generators

The MAINSAIL compiler uses a separate code generator for each hardware architecture. Each code generator module is of the same class and is dynamically selected and bound during the execution of the compiler. Since each code generator is written in MAINSAIL, any code generator can be used on any computing system. This provides for universal cross-compilation.

The platforms on which MAINSAIL runs at the time of this writing are shown in Table 22.1-1. XIDAK is constantly adding new systems to the list.

22.2. Disassemblers

Corresponding to each code generator, XIDAK supplies a disassembler capable of producing a text file that shows the original MAINSAIL source text interspersed with a mnemonic listing of the machine code generated by the compiler. This permits the user to evaluate the quality of the code emitted by the compiler and to compare the relative efficiency of different constructs.

22.3. Foreign Language Interface

MAINSAIL programmers can easily interface to code written in other programming languages through the facility called the Foreign Language Interface (FLI). The programmer supplies the FLI with a description of the foreign entry points, and the FLI automatically generates interface code that is linked with a MAINSAIL bootstrap and the foreign object module.

Subcommand	Description
ABORT	Abort this compilation
ACheck	Set default to emit code to catch arithmetic overflow, etc.
ACHECKALL	ACHECK unconditionally
ALIST	Allow disassembly
Check	Set default to emit code to catch subscript errors, etc.
CHECKALL	CHECK unconditionally
CMDLINE s	Add s to the end of the cmdLine list (nonsticky)
DEBUG	Make this module debuggable, and turn on INCREMENTAL
FLDXREF {f}	Write field cross reference {to file f (nonsticky)}
FLI s	Generate code for foreign interface s
GENcode	Generate code
GENINLINES	Generate bodies for inline procs
INCREMENTAL	Allow output to be incrementally recompiled
ININTLIB f	Input intmod is in library f
INOBJFILE f	Input objmod is in file f (nonsticky)
INOBJLIB f	Input objmod is in library f
ITFXREF {f}	Write interface cross reference {to file f (nonsticky)}
LOG	Show log info
MODTIME	Measure time spent in this module
MONITOR	Turn on PER{MOD,PROC,STMT} and {MOD,PROC}TIME
OPtimize	Set default to optimize all procs
OPtimize p	Optimize procs p = p1 p2 ... pn (nonsticky)
OPTIMIZEALL	Optimize all procs

Table 22-1. MAINSAIL Compiler Subcommands (continued)

OUTINTFILE f	Output intmod to file f (nonsticky)
OUTINTLIB f	Output intmod to library file f
OUTOBJFILE f	Output objmod to file f (nonsticky)
OUTOBJLIB f	Output objmod to library file f
PERMOD	Count total statements executed in the module
PERPROC	Count total statements executed in each proc
PERSTMT	Count times each statement is executed
PROCS	Show names of procs as they are parsed and generated
PROCTIME	Measure time spent in each proc
RECOMPILE p	Recompile procs p = p1 p2 ... pn (nonsticky)
REDEFINE x y	Do $GLOBALREDEFINE x = [y];
RESPONSE	Get user response to error messages
RPC {C}	generate code for remote procedure call {in C}
SAVEON {f}	Create intmod containing all compiler info {save on file f}
SLIST {f}	Write source listing {to file f (nonsticky)}
SUBCOMMAND s	Execute MAINEX subcommand s
TARGET s	Generate for target system s
UNBOUND	Nonbound-invocation module
# s	A comment (s is ignored)

Table 22-1. MAINSAIL Compiler Subcommands (continued)

NOACheck	Turn off ACHECK
NOACHECKALL	Turn off ACHECKALL
NOALIST	Turn off ALIST
NOCheck	Turn off CHECK
NOCHECKALL	Turn off CHECKALL
NODEBUG	Turn off DEBUG and INCREMENTAL
NOFLDXREF	Turn off FLDXREF
NOGENcode	Turn off GENCODE
NOGENINLINES	Turn off GENINLINES
NOINCREMENTAL	Turn off INCREMENTAL
NOININTLIB	Turn off ININTLIB
NOINOBJLIB	Turn off INOBJLIB
NOITFXREF	Turn off ITFXREF
NOLOG	Turn off LOG
NOMONITOR	Turn off PER{MOD,PROC,STMT} and {MOD,PROC}TIME
NOOPtimize	Turn off OPTIMIZE
NOOPtimize p	Do not optimize proc(s) p, where p = p1 ... pn
NOOPTIMIZEALL	Turn off NOOPTIMIZEALL
NOOUTINTLIB	Turn off OUTINTLIB
NOOUTOBJLIB	Turn off OUTOBJLIB
NOPROCS	Turn off PROCS
NOREDEFINE	Remove all global definitions
NOREDEFINE x	Remove global definition(s) of x, where x = x1 ... xn
NORESPONSE	Turn off RESPONSE
NORPC	Turn off RPC
NOSAVEON	Turn off SAVEON
NOSLIST	Turn off SLIST
NOUNBOUND	turn off UNBOUND

For backward compatibility:

LIBRARY f	Same as GENCODE, OUTOBJLIB f
OUTPUT {f}	Same as GENCODE {, OUTOBJFILE f}
NOLIBRARY	Same as NOOUTOBJLIB
NOOUTPUT	Same as NOGENCODE

Table 22-1. MAINSAIL Compiler Subcommands (end)

Platform Abbrev.	Processor	Platform Name
aeg	M68000	Apollo's Aegis on Motorola M68000
aix	System/370	IBM's AIX on IBM System/370
alnt	M68000	Alliant's CONCENTRIX on Motorola M68000
cms	System/370	IBM's VM/SP CMS on IBM System/370
hp20	MC68020/ MC68881	HP's HP-UX on Motorola MC68020/MC68881
hp38	80386	SCO's XENIX on HP Vectra with Intel 80386
hpux	M68000	HP's HP-UX on Motorola M68000
ip32c	Interpro 32C	Intergraph's System V UNIX on Interpro 32C
ipsc2	80386	Intel's iPSC/2 System V UNIX on Intel 80386
ix20	MC68020/ MC68881	Apollo's DOMAIN/IX on Motorola MC68020/MC68881
ixfpa	MC68020/ Weitek	Apollo's DOMAIN/IX on Motorola MC68020/Weitek FPA
ixpri	PRISM	Apollo's DOMAIN/IX on Apollo PRISM
sun2	M68000	Sun Microsystems' SunOS on Motorola M68000
sun3	MC68020/ MC68881	Sun Microsystems' SunOS on Motorola MC68020/MC68881
sun38	80386	Sun Microsystems' SunOS on Intel 80386
sun4	SPARC	Sun Microsystems' SunOS on SPARC
ultrx	VAX-11	DEC's ULTRIX-32 on VAX-11
vms	VAX-11	DEC's VAX/VMS on VAX-11
xcms	System/370 XA	IBM's VM/XA SP CMS on IBM System/370

Table 22.1-1. Computer Systems on Which MAINSAIL Is Supported

23. MAINDEBUG, the MAINSAIL Debugger

MAINDEBUG is a source-level debugger for MAINSAIL programs. The programmer interacts with the MAINSAIL source text by using a cursor to indicate the location of breakpoints.

MAINDEBUG can operate in either line-oriented or display-oriented mode. The display-oriented mode is integrated with MAINEDIT, so that single-keystroke commands move the terminal cursor over source text statements as they are executed. The display-oriented debugger keeps the source text in one or more windows and displays program output in a separate window.

When a program generates an error (e.g., an array subscript error), the debugger can be dynamically invoked to point at the offending statement. It can then be used to examine the call stack and the values of variables so that the cause of the error can be determined.

MAINDEBUG's command processor is highly flexible because it invokes the MAINSAIL compiler to process all expressions specified in commands. The debugger is also able to interpret MAINSAIL statements on the fly; this is useful, for example, to examine the effect of a procedure called with a certain set of arguments. A breakpoint can be placed at the beginning of the procedure, and then the procedure can be called by interpreting the call from the debugger. The user can then step through the procedure.

A summary of MAINDEBUG commands is shown in Table 23-1.

`A ary,l1,u1,l2,u2,l3,u3`	show array slice ary[l1 TO u1,...]
`{+}B{@{m.}i}{[cond]}{:cmds}`	set break at cursor {or mod m, iUnit i}
`{n}{.i}C`	continue {at iUnit i}, till nth break
`{n}.C`	continue at cursor, till nth break
`.D d1;...;dn`	compile defs or dcls d1;...;dn
`E {m}`	execute MAINSAIL exec {or module m}
`.F p,f1,f2,...`	V p.f1,p.f2,... (p can be unclassified)
`H e1,e2,...`	hex values of e1,e2,...
`.H p1,p2,...`	hex values of objects at p1,p2,...
`{1}I`	display {abbreviated} debug info
`.I i1,i2,...`	display info about identifiers
`{n}J`	step n times, jump into procs
`K n`	break when count = n
`M`	set to break context
`M m`	open module m (m can be a file name)
`-M m`	close m's intmod and dispose m's objmod
`.M p`	open module with data section p
`{n}N`	move to nth caller from current proc

Table 23-1. Debugging Command Summary (continued)

{n}-N	move to nth callee from current proc
{n}.N	move to where exception was raised
{n}-.N	undo .N command n times
O n	cursor to iUnit n, current module
OC s	open coroutine s
OI s	open intmod library s
OL s	open objmod library s
{-}OP s	set {clear} options s
Q	quit (exit the program)
+Q	exit MAINDEBUG
R	remove break at cursor
R@m.i	remove break at module m, iUnit i
R@@	remove all breaks
{n}S	step n times, do not enter procs
{+}T{@{m.}i}{[cond]}{:cmds}	same as B, except set temp break
V e1,e2,...	values of e1,e2,...
.V p1,p2,...	values of objects at p1,p2,...
XM a	examine memory at address a
XS s1;...;sn	execute statements s1;...;sn
<ECM>	enter MAINEDIT command mode

Table 23-1. Debugging Command Summary (end)

24. MAINEDIT, a Portable Text Editor

MAINEDIT is a portable, display-oriented text editor that supports simultaneous editing of multiple files in multiple windows. It supports different keystroke interpreters, or front ends, including emulators for the popular vi and EDT text editors. MAINEDIT uses a set of "display modules", each of the same class, to support a number of different display terminals. XIDAK can implement a display module for a new terminal easily and rapidly.

The first MAINEDIT front end was MAINED, which includes commands that take full advantage of MAINEDIT's multiple-buffer, multiple-window capabilities. The front ends that emulate other text editors lack some these commands, since they were not present in the original text editors; however, a MAINEDIT user can invoke any front end's commands from any other front end, so the MAINED buffer and window commands may be used from the other editor emulators.

Some of the other features of MAINED include:

- An "Again" command, which repeats the last command.
- Deletion, recovery, and copying of characters, words, lines, or pages.
- Execution of any MAINSAIL program within a MAINEDIT window.
- An "Undo" command, which undoes the last command.
- An "Abort" key, which aborts the current command or macro execution.
- A keyboard macro facility, which allows the user to invoke a series of commands with a single command character or macro name.

MAINEDIT display modules are currently available for the display terminals listed in Table 24-1.

Module	Terminal(s)
AM48	48-line Ann Arbor Ambassador
AM60	60-line Ann Arbor Ambassador
AT386	IBM PC/AT and compatibles
BIGSUN	Sun Microsystems workstation, arbitrary number of lines
BORRO	Apollo Computer workstation
D400	Data General DASHER
D460	Data General D410/460
D460C	Data General D410/460, 132 columns
DATAME	Datamedia 3000, Telemedia
EWY100	ELXSI-modified Wyse 100
FBORRO	Apollo Computer workstation
FRAME	Apollo Computer workstation
FFRAME	Apollo Computer workstation
HEATH	Heath (or Zenith) H-19
HP300H	large-screen Hewlett-Packard terminal
HPTERM	Hewlett-Packard terminals
LINDPY	any terminal; line-oriented
SUN	Sun Microsystems workstation
SUN3	Sun Microsystems window
SUN46	Sun Microsystems workstation, 46 lines
TELEVI	Televideo (except model 950)
TVI950	Televideo model 950
TRMCAP	any terminal for which information is available in a UNIX-style database
VIS550	Visual 550
VT100	VT100
VT102	VT102 (VT100 with insert and delete)
VT102M	VT102 (imperfect emulators)
WY43	Wyse WY-60
WY50	Wyse WY-50
WY5043	Wyse WY-50, 43-line mode
WY75	Wyse WY-75

Table 24-1. Available Display Modules

A complete summary of MAINED commands follows:

Command Mode

```
nA       do last command again, count = n, original modifier
QA       do last command again, original count and modifier
+nA      do last command again, count = n, "+" modifier
Q+A      do last command again, original count, "+" modifier
-nA      do last command again, count = n, "-" modifier
-QA      do last command again, original count, "-" modifier
("-" direction is towards beginning of file)
("+" direction is towards end of file)
.A       anchor current window
+.A      anchor at bottom
-.A      anchor at top
n.A      anchor, change size to n rows
+n.A     anchor at bottom, change size to n rows
-n.A     anchor at top, change size to n rows
..A      unanchor current window
Q..A     unanchor all windows
Q+..A    unanchor all windows at bottom of screen
Q-..A    unanchor all windows at top of screen
```

```
B        break line, remove spaces
nB       break line, indent n spaces
QB       break line, leave spaces
-B       break line, remove spaces, leave cursor
-nB      break line, indent n spaces, leave cursor
-QB      break line, leave spaces, leave cursor
.Bs      edit buffer s, use current window if not on screen
+.Bs     edit buffer s, new window at bottom if not on screen
n.Bs     same as ".Bs", except n-row window
+n.Bs    same as "+.Bs", except n-row window
-.Bs     edit buffer s, new window at top if not on screen
-n.Bs    same as "-.Bs", except n-row window
--{n}.Bs
         overlay (n-row) window at top
++{n}.Bs
         overlay (n-row) window at bottom
..{n}Bs
         edit s, making window 1/mth of screen, where m is the
         number of windows; but no window is allowed to be
         smaller than n lines
Q.Bs     change bufferName of current buffer to s
Q..Bs    change command front end to s
+Q..Bs   change command and view front ends to s
-Q..Bs   kill front end s

nC[C|W|L|P]      copy n objects at and after
QC[C|W|L|P]      copy all objects at and after
-nC[C|W|L|P]     copy n objects before
-QC[C|W|L|P]     copy all objects before
n.C              center n lines at and after
Q.C              center all lines at and after
CM               push savedMode onto mode stack

nD[C|W|L|P]      delete n objects at and after
QD[C|W|L|P]      delete all objects at and after
-nD[C|W|L|P]     delete n objects before
-QD[C|W|L|P]     delete all objects before
(".D" copies text into delete buffer, but does not delete it;
"..D" deletes text, but does not copy it into the delte buffer)

E        escape to caller, if any
QEs      execute module s (dispose-bind-unbind)
.E       show name of currently executing module
Q.E      show names of all executing modules
```

```
F       prompt to save altered buffers, then continue
+F      change autoSaveLimit (0 means no autoSave reminder)
QF      If a program invoked with "QE" is running, raise the
        exception $abortProgramExcpt; otherwise, exit from
        MAINEDIT, continuation not allowed
-QF     prompt to save altered buffers, then pause
.Fs     edit file s, use current window if not on screen
+.Fs    edit file s, new window at bottom if not on screen
n.Fs    same as ".Fs", except n-row window
+n.Fs   same as "+.Fs", except n-row window
-.Fs    edit file s, new window at top if not on screen
-n.Fs   same as "-.Fs", except n-row window
--{n}.Fs
        overlay (n-row) window at top
++{n}.Fs
        overlay (n-row) window at bottom
..{n}Fs edit s, making window 1/mth of screen, where m is the
        number of windows; but no window is allowed to be
        smaller than n lines
Q.Fs    change fileName of current buffer to s

G       go to first line of next page
-G      go to first line of previous page
p.1G    go to page p, line 1
.G      go to first line of current page
.1G     go to line 1 of current page
pG      go to first line of page p
+nG     first line of current page + n
-nG     first line of current page - n
(start with "Q" to set mark ("@" command) before going)

nH      undo previous n changes
QH      undo all changes on current line
-nH     redo next n changes
-QH     redo all changes on current line

I       enter insert mode
1IB     insert a buffer (name is asked)
nICc    insert n c's (n required)
QICc    insert c's to right margin
1IF     insert a file (name is asked)
nIL     insert n blank lines (n required)
QIL     insert blank lines to end of window
.I      insert blank line, enter insert mode
n.I     insert blank line, indent n spaces, enter insert mode
```

```
nJ      join next to current line, n separating spaces
QJ      join next to current line, leave spaces
-nJ     join current to previous line, n separating spaces
-QJ     join current to previous line, leave spaces
.J      fill current paragraph to right margin of
         window
n.J     fill n lines
+.J     fill and justify to right margin of window
-.J     fill starting at cursor column
Q.J     fill all remaining paragraphs in buffer
.mJ     fill to right margin (justify) n column m
nQ.J    fill next n paragraphs
(All modifiers may be combined; i.e., nQ+-.m.J means fill next n
paragraphs from cursor column, justifying to column m).

nK      delete (kill) n characters at and to right
QK      delete all characters at and to right
-nK     delete n characters to left
-QK     delete all characters to left
.K      prompt to kill each buffer (prompts to save)
Q.K     kill one buffer (prompts for name)
..K     kill a character without copying into delete buffer

nL[C|W|L]        make n objects lower case
QL[C|W|L]        make all objects lower case

nM[C|W|L|P]      move current object n further
QM[C|W|L|P]      move current object to end
-nM[C|W|L|P]     move current object n before
-QM[C|W|L|P]     move current object to start
.M[C|W|L]        mark the appropriate delete buffer

0N      refresh message line
nN      refresh n lines at and below in current window
-nN     refresh n lines above in current window
QN      refresh entire screen
Q.N     refresh current window

O       enter overstrike mode
nOCc    overstrike n c's (n required)
QOCc    overstrike c's to right margin
.O      set editor option
-.O     clear editor option
.O?     show option settings
```

```
P       insert page mark above

Q       emphasize the command

R[C|W|L|P]      recall and insert group of objects
nR[C|W|L|P]     recall and insert n objects
QR[C|W|L|P]     recall and insert all objects to mark
(".R" means leave in delete buffer)
RM              pop mode stack into curMode
.RM             set curMode to top of mode stack (not popped)
QRM             set curMode to savedMode

nSc     skip right to nth occurrence of character "c"
+Sc     skip over c's
-nSc    skip left to nth occurrence of character "c"
-+Sc    skip left over c's
QnSc    skip down to nth "c"-line
Q+Sc    skip down to next line not starting with c
-QnSc   skip up to nth "c"-line
Q+Sc    skip up to next line not starting with c
(a "c"-line is a line with first visible char equal to "c".
A <sp>-line is one with no visible characters.)

Ts<eol>       search right and all lines down for s ...
T<eol>        search right and all lines down for last target(s)
nTs<eol>      search right and n-1 lines down for s ...
QTr<eol>s...<eol><eol>
              search right and all lines down for r or s or ...
QnTr<eol>s...<eol><eol>
              search right and n-1 lines down for r or s or ...
(-T searches left and up)
(qualifying with "+" wraps around buffer beginning or end)
(qualifying with "QQ" makes into a line search)
("{-}.T" is an "identifier" search, i.e. the target cannot be
bordered by an alphabetic or digit)

U       same as "L", except convert to upper case

V       give character position, prompt for new one
Q..Vs   change view front end to s
```

```
W        scroll up 4/5 of a window
nW       scroll up n lines
QW       scroll up all lines
-W       scroll down 4/5 of a window
-nW      scroll down n lines
-QW      scroll down all lines
n.W      move current line to line n from top of window
-n.W     move current line to line n from bottom of window

nX       move to column n of window (x-coordinate)
-nX      move to window width - n + 1
QnX      put right margin at column n (from line origin)
-QnX     put left margin at column n (from line origin)

nY       move to row n of window (y-coordinate)
-nY      move to row n of window, count from bottom up
QY       set current window to maximum size
QnY      set number of window rows to n (n = 0 kills window)
+QY      expand window to bottom of screen
+QnY     expand window n rows
-QY      synonym for Q0Y
-QnY     contract window n rows
n.Y      move cursor to nth next window on screen
-n.Y     move cursor to nth previous window on screen
Q.Y      move cursor to bottommost window on screen
-Q.Y     move cursor to topmost window on screen

Z        same as "S", except delete skipped objects
(".Z" means do not delete, but put into delete buffer;
"..Z" means delete, but do not put into delete buffer)

n<bs>    move left n columns
Q<bs>    move to left margin

n<tab>   move cursor to nth next tab stop
-n<tab>  move cursor to nth previous tab stop
Q<tab>   set tab stops

n<lf>    move down n rows
Q<lf>    move to last row

<abort>  abort current command, enter command mode

n<eol>   move to left margin of nth next line
-n<eol>  move to left margin of nth previous line
```

```
n<sp>    move right n columns
Q<sp>    move to last column
(equivalent to ">")

'n       insert character with decimal code "n"
'Bn      insert character with binary code "n"
'Hn      insert character with hexadecimal code "n"
'On      insert character with octal code "n"

n(       move left n words
n.(      move to one past end of nth word to left
Q(       move to first visible character of line

n)       move right n words
n.)      move to one past end of (n-1)st word to right
Q)       move to after end of line

,        invoke named macro

/x.../   define x to be ..., where x is a macro name
(... is carried out as it is typed in)

n<       move left n columns
Q<       move to first column

=        show line info
Q=       show buffer info
+Q=      show buffer info with front end info

n>       move right n columns
Q>       move to last column

.@       set mark to current location
@        go to marked location
Q@       set mark, go to previously marked location

n\       move down n rows
Q\       move to bottom row of window

n^       move up n rows
Q^       move to top row of window

n<del>   move left n columns
Q<del>   move to left margin
```

```
n.+v<eol>       An := An + v
n.-v<eol>       An := An - v
-n.-v<eol>      An := v - An
n.*v<eol>       An := An * v
n./v<eol>       An := An / v
-n./v<eol>      An := v / An
n.^v<eol>       An := An ^ v  (raise An to the power v)
-n.^v<eol>      An := v ^ An  (raise v to the power in An)
n._v<eol>       An := v       (and set An's format to v's)
n.=             Display value of accumulator n
Q.=             Display value of all active accumulators
```

Overstrike Mode

```
<bs>, <del>     move left 1 column, except end of previous line
                 if at left margin
<tab>           overstrike spaces to next tab stop
<lf>            move down 1 row
<eol>           move to left margin of next line
```

Insert Mode

```
<bs>     move left 1 column
<tab>    insert spaces to next tab stop
<lf>     move down 1 row
<eol>    break line, move cursor to start of new line
<del>    delete character to left, except join current line to
          previous line if at left margin (like "-QJ" command)
```

25. MAINPM, the MAINSAIL Performance Monitor

MAINPM is a performance monitor for MAINSAIL programs. Once a MAINSAIL program is functionally correct, MAINPM can be used to isolate performance problems. MAINPM lets the programmer examine the time used by a program in different degrees of granularity, based on module, procedure, or statement. Program execution can also be sampled with periodic interrupts. Consumption of string and chunk (array, record, and data section) space can be monitored. MAINPM can produce both deep and shallow information as well as a source listing annotated with statement counts.

Sample MAINPM output is shown in Examples 25-1 and 25-2.

```
NAME                  SHALLOW TIME          DEEP TIME        STMT COUNT
(mod or mod.proc)    (seconds)    (%)    (seconds)    (%)
-----------------    ---------    ---    ---------    ---    ----------
NUMBER.FIBONACCI        45.057  99.64       45.057  99.64        485570
NUMBER.F                  .003    .01         .003    .01            22
NUMBER.INITPROC           .002    .00       45.169  99.89             9
NUMBER.IFACTORIAL         .000    .00         .000    .00            12


Total execution time: 45.221 seconds
```

Example 25-1. Timing and Statement Counts Table

```
                    STATEMENT COUNTS
                    ----------------

                  SOURCE FILE: number
                  -------------------

       BEGIN "number"

       LONG INTEGER PROCEDURE iFactorial (INTEGER n);
       BEGIN
       LONG INTEGER total;
       INTEGER i;
     1 total := 1L;
     1 FOR i := 2 UPTO n DO
     9                     total .* cvli(i);
     1 RETURN(total)
       END;

       LONG INTEGER PROCEDURE f (INTEGER n);
       # Return n factorial.
    11 IF n = 0 THEN
     1               RETURN(1L)
    10 ELSE RETURN(cvli(n) * f(n - 1));

       LONG INTEGER PROCEDURE fibonacci (INTEGER n);
242785 IF n LEQ 1 THEN
121393                 RETURN(cvli(n))
121392 ELSE RETURN(fibonacci(n - 2) + fibonacci(n - 1));

       INITIAL PROCEDURE;
       BEGIN
     1 ttyWrite("10 factorial computed recursively is ",
           f(10),eol);
     1 ttyWrite("10 factorial computed iteratively is ",
           iFactorial(10),eol);
     1 ttyWrite("The 25th Fibonacci number is ",
           fibonacci(25),eol)
       END;

       END "number"
```

Example 25-2. Source Text with Statement Counts

26. The MAINSAIL Structure Blaster

The MAINSAIL Structure Blaster allows an arbitrary MAINSAIL data structure to be written to or read from a file with a single procedure call. The file I/O is performed as efficiently as the underlying operating system permits.

The Structure Blaster may be used to "checkpoint" a data structure at a given point in a program's execution, or may function as a data base primitive. In addition, since a facility is provided to translate a structure from one machine's format to another, a data structure built on one machine may be shipped to another (presumably faster) machine for processing by another MAINSAIL program, then shipped back to the originating machine.

The headers of some of the Structure Blaster procedures are shown in Table 26-1. $structureWrite writes a structure to a file; $structureRead reads a structure from a file; $structureDispose frees up the memory occupied by a structure; and $structureCopy makes a copy of a structure. In each case, the parameter "root" is a pointer to the (arbitrary) structure on which the operation is to be performed.

Facilities are also provided to write a human-readable form of a structure to a text file (or to allow a user to enter a structure as text and have it "compiled" by the Structure Blaster).

```
LONG INTEGER
PROCEDURE   $structureWrite
                        (POINTER(dataFile) f;
                         POINTER root;
                         OPTIONAL LONG INTEGER
                             startPageOrPos;
                         MODIFIES OPTIONAL
                             POINTER($strucInfo)
                             strucInfo;
                         OPTIONAL BITS ctrlBits);

POINTER
PROCEDURE   $structureRead
                        (POINTER(dataFile) f;
                         OPTIONAL LONG INTEGER
                             startPageOrPos,
                             numPagesOrSize;
                         PRODUCES OPTIONAL LONG INTEGER
                             actualNumPagesOrSize;
                         OPTIONAL BITS ctrlBits;
                         OPTIONAL POINTER($area) area);

PROCEDURE   $structureDispose
                        (MODIFIES POINTER root;
                         OPTIONAL BITS ctrlBits);

POINTER
PROCEDURE   $structureCopy
                        (POINTER root;
                         OPTIONAL BITS ctrlBits;
                         OPTIONAL POINTER($area) area);
```

Table 26-1. Selected Structure Blaster Procedure Headers

27. MAINSAIL STREAMS

STREAMS is a collection of facilities for distributed applications, process and device control, and enhancements to the functionality of MAINSAIL coroutines. At present, STREAMS is still under development, so not all facilities are implemented on all systems where they could be, and some interface changes may still be made.

The main high-level function provided by STREAMS is the RPC (remote procedure call) package. RPC allows interprocess communication to look like calls to an ordinary MAINSAIL module. It requires the programmer to write an "RPC server", a module that provides a set of functions (each implemented as an interface procedure), and compile it with a special compiler subcommand to produce two modules that are both compiled with the regular compiler. The two modules transmit and receive the server interface procedure arguments between processes; one runs in the server process, and the other in the process (the "RPC client") that invokes the server functions.

The STREAMS package includes the Scheduler, which allows one coroutine to run while another blocks. Each coroutine looks like an ordinary sequential MAINSAIL application; coroutines are rescheduled automatically when the perform I/O operations. In conjunction with RPC, the Scheduler allows a MAINSAIL application to be distributed among a number of processes to achieve coarse-grain parallel processing; this allows programmers to take advantage of multi-processor systems and high-speed networks to speed up their applications.

STREAMS also provides a large set of procedures for low-level I/O and server management. Typically, programmers use this level of STREAMS only to control special devices (like terminals), since the RPC mechanism is a better way of performing most server functions than low-level STREAMS facilities.

Master Index

24 March 1989

28. Master Index

The MAINSAIL master index combines the separate indices in each MAINSAIL document into one large index. It lists an abbreviation for the relevant document name as well as the page in the document with each index entry. At the beginning of the index is the list of documents covered, including title, abbreviation, and date of issue.

Because MAINSAIL documents are updated from time to time, the page number in the master index may not coincide exactly with the page number in the document if the index and document were issued on different dates. The date of any document may be found on its cover page.

If you find that a topic you wish to look up is not listed in the master index, you may send a "User Change Request" (UCR) form to XIDAK asking that the topic be covered in the documentation. UCR forms are available from XIDAK upon request. For more urgent problems, XIDAK's customer service personnel may be contacted by telephone.

Master Index

```
Abbreviations:
  KSTRMU =  STREAMS and MAINKERMIT User's Guides
    (24 March 1989)
  M1 = MAINSAIL Language Manual Part I
    (24 March 1989)
  M2 = MAINSAIL Language Manual Part II
    (24 March 1989)
  MEDTU = MAINEDIT User's Guide
    (24 March 1989)
  MTUT1 = MAINSAIL Tutorial, Part I
    (24 March 1989)
  MTUT2 = MAINSAIL Tutorial, Part II
    (24 March 1989)
  TOOLU = MAINSAIL Tools User's Guides
    (24 March 1989)
```

XIDAK, Inc., 530 Oak Grove Avenue, M/S 101, Menlo Park, CA 94025, (415) 324-8745