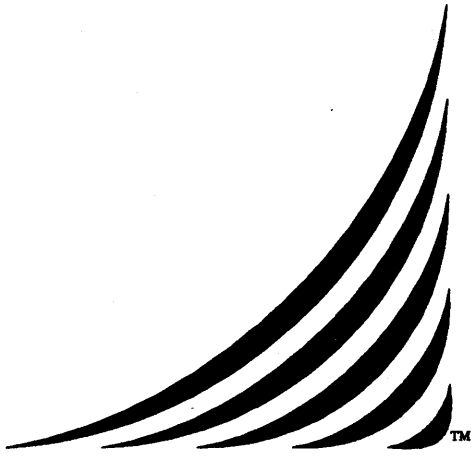
V12

MAINSAIL®

Language Manual, Volume I



MAINSAIL®

Language Manual, Part I:

Syntax and Semantics

24 March 1989

Xinak™

Table of Contents

Appendices

List of Examples

List of Tables

1. Introduction

This document is the definitive reference on MAINSAIL, a computer programming language supported and marketed by XIDAK, Inc. This manual is intended not to teach the MAINSAIL language, but rather to answer the questions of programmers who already have some knowledge of MAINSAIL. New users of MAINSAIL should consult the "MAINSAIL Documentation User's Guide and Master Index", which lists documents available from XIDAK on MAINSAIL and the MAINSAIL environment. The "MAINSAIL Overview" and the "MAINSAIL Tutorial" provide information of particular interest to the new user.

MAINSAIL system procedures, macros, and variables are described in Chapter 1 of part II of the "MAINSAIL Language Manual", which should be consulted for information on unfamiliar identifiers in this manual.

1.1. Version

This version of the "MAINSAIL Language Manual" is current as of Version 12.10 of MAINSAIL. It incorporates the "Runtime System Version 5.10 Release Note" of October, 1982; the "MAINSAIL Language Version 7.4 Release Note" and the "Runtime System Version 7.4 Release Note" of May, 1983; the "MAINSAIL Language Release Note, Version 8" of January, 1984; the "MAINSAIL Language Release Note, Version 9" of February, 1985; the "MAINSAIL Language Release Note, Version 10" of March, 1986; and the "MAINSAIL Language Release Note, Version 11" of July, 1987.

1.2. The Design of MAINSAIL

MAINSAIL is a programming system designed for the development of portable software, i.e., programs that can be transported in source form with no alterations (unless implied by the nature of the program) among all implementations. The programmer is not prevented from writing programs that are machine-dependent, but the language design is based on constructs that can be implemented on a variety of machines, so that the need for machine-dependent constructs is minimized. To provide a basis for portability it has been necessary to develop facilities beyond those normally associated with a programming language; for example, MAINSAIL provides its own intermodule linkage and module loading.

MAINSAIL is a large language. It provides a number of data types, data structuring mechanisms, and system procedures. Efficient execution and the capability to develop large software products quickly were more important considerations in the development of the language than keeping the language small.

XIDAK reserves the right to upgrade MAINSAIL from release to release. New data types, data structures, keywords, system procedures, system macros, and system variables may be introduced at each MAINSAIL release. Insofar as XIDAK deems feasible, such enhancements will be upward-compatible with previous versions of MAINSAIL.

Sizes and layouts of data types and data structures, values of system macro constants, and other system-dependent information may change from release to release and vary from system to system. For portability (both among different releases and different computers), these quantities should be specified symbolically using the features provided in the MAINSAIL language, and never hardwired into a program.

1.3. Terminology and Symbols

Abbreviations used throughout the manual are shown in Table 1.3-1. These abbreviations are often used to stand for variables of the data type being abbreviated.

```
bo          BOOLEAN
i           INTEGER
li          LONG INTEGER
r           REAL
lr          LONG REAL
b           BITS
lb          LONG BITS
s           STRING
a           ADDRESS
c           CHARADR ("c" sometimes abbreviates "character")
p           POINTER
n           Numeric (INTEGER, LONG INTEGER, REAL, LONG REAL)
v,v0,...    Variables
e,e0,...    Expressions
c,c0,...    Constant expressions
s,s0,...    Statements
```

Table 1.3-1. Abbreviations

In descriptions, the forms "(long) integer", "(long) bits", and "(long) real" mean "integer and/or long integer", "bits and/or long bits", and "real and/or long real", respectively.

MAINSAIL keywords (e.g., "BEGIN", "END") are written in all upper case in this manual, though this is not necessary in MAINSAIL programs. MAINSAIL does not distinguish identifiers or keywords on the basis of case.

"Compiletime" refers to actions that take place while a program is being compiled. "Runtime" refers to actions that take place while a program is being executed.

MAINSAIL implementations are categorized according to "processor", "operating system", and "platform". A processor is characterized by a particular instruction set; one or more operating systems may run on computers containing a given processor. In XIDAK terminology, two implementations of MAINSAIL run on different operating systems if code for the two systems must be compiled differently (even if the instruction sets are the same, operating-system-dependent logic may have to be different). Platform is a narrower category than "operating system"; i.e., every operating system includes one or more platforms. Although identical code is generated for all modules compiled for the same operating system, two different platforms within the same operating system may require different installation procedures or bootstraps, or may make slightly different operating system calls at runtime. The current lists of processors, operating systems, and platforms are given in Appendix B.

The "target system" (or "target processor", "target operating system", or "target platform") is the computer system for which a MAINSAIL program is compiled, i.e., the system on which it is to run.

"Error" refers to a situation or language construct that by default generates an error message (either at compiletime or at runtime). "Illegal" refers to some language constructs that generate errors at compiletime. "Undefined" refers to a situation or language construct that is unsafe or not logically correct, but that may or may not generate an error message. Programs making use of an undefined construct or situation may not work the same way from one MAINSAIL version or implementation to another. "Unspecified" refers to the result of a situation or operation that does not generate an error and is not logically incorrect, but that may vary from situation to situation or from implementation to implementation. For example, in operations that round numbers, the direction of rounding is often unspecified.

MAINSAIL often chooses to leave a construct undefined or unspecified for efficiency considerations. On MAINSAIL implementations, the fastest form of an operation provided by the underlying hardware or operating system is usually used, regardless of whether it provides error condition checks or high precision.

Some error messages can be suppressed; e.g., the "NOCHECK" compiler directive prevents certain situations from generating error messages. Suppressing error messages with the "NOCHECK" directive transforms the affected error situations into undefined situations.

1.4. Conventions Used in This Document

1.4.1. User Interaction

Throughout the examples in this document, characters typed by the user are underlined. "<eol>" symbolizes the end-of-line key on a terminal keyboard; this key is marked "RETURN" or "ENTER" on most keyboards. In Example 1.4.1-1, "Prompt:" is written by the computer; the user types "response" and then presses the end-of-line key.

```
Prompt: response<eol>
```

Example 1.4.1-1. How User Input Is Distinguished

1.4.2. Syntax Descriptions

Specifications of syntax often contain descriptions enclosed in angle brackets ("<" and ">"). Such descriptions are not typed literally, but are replaced with instances of the things they describe. For example, a specification of the syntax of the address on an envelope might appear as in Example 1.4.2-1.

```
<name of addressee>
<street number> <street name>
<town or city name>, <state abbreviation>  <zip code>
```

Example 1.4.2-1. Syntax of a Mailing Address

Optional elements in command or syntax descriptions are often enclosed in curly brackets ("{" and "}"). For example, a string of characters specified as "{A}B{C}" could have any one of the forms "B", "BC", "AB", and "ABC". Alternatives may be enclosed in square brackets (or curly brackets, if all alternatives are optional) and separated by vertical bars ("|"); "[A|B|C]" means "A", "B", or "C"; "{A|B}" means "A", "B", or nothing.

1.4.3. Temporary Features

Temporary features that have not acquired a final form are marked as follows:

```
TEMPORARY FEATURE:  SUBJECT TO CHANGE
```

Temporary features are subject to change or removal without notice. Programmers who make use of temporary features must be prepared to modify their code to accommodate the changes in them on each release of MAINSAIL. It is recommended that code that makes use of temporary features be as isolated from normal code as possible and thoroughly documented.

2. Basic Language Concepts

2.1. Character Set

MAINSAIL does not specify the exact character set of the system on which it runs; however, it is guaranteed that a unique character corresponds to each of the characters shown in Table 2.1-1. MAINSAIL cannot guarantee the graphics associated with each character, but they are chosen to approximate those shown.

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz
0123456789
! " # $ & ' ( ) * + , - . / : ;
< = > ? [ ] ^ _ @ \ | { } % ` ~
space (blank)
tab (horizontal tab)
eol (end-of-line)
eop (end-of-page)
$nulChar (null character)
```

Table 2.1-1. MAINSAIL Minimum Character Set

Associated with each character is an integer code. Character codes range from 0 to the predefined constant $maxChar, which has the value 255, since MAINSAIL characters occupy eight bits each. In this manual the term "character" is often used to mean "character code". The following may be assumed about the ordering of character codes:

- 'A' through 'Z' are alphabetically ordered, but not necessarily contiguous.
- 'a' through 'z' are alphabetically ordered, but not necessarily contiguous.
- '0' through '9' are numerically ordered and contiguous.

Although lowercase letters are guaranteed to have character codes, some peripherals may not provide lower case. It is recommended that portable programs not depend on upper/lower case distinctions. For example, a program that reads commands from a terminal should not use "A" for one command and "a" for another.

The three identifiers "tab", "eol", and "eop" are predefined by MAINSAIL as string constant macros.

The exact effect of a tab is peripheral-dependent, but it usually positions to the next horizontal tab stop. MAINSAIL does not define tab stops since the peripherals on which tabs have an effect may not be under MAINSAIL's control. Tab stops are often defined to be every fourth or eighth column. The tab character has the code 9 on ASCII systems and 5 on EBCDIC systems.

eol ("end-of-line") is a one-character string that indicates the end of a line of text. When written to a file it terminates a line, so that the next character is written at the start of the next line. The eol character typically has the code 10 on ASCII systems and 37 on EBCDIC systems (it is possible that other values may be encountered).

When reading a line from a text file with the system procedure "read", MAINSAIL searches for an end-of-line by searching for eol. The eol character is discarded. All characters up to the end-of-line sequence make up the line as produced by "read".

eop ("end-of-page") is a single-character string that indicates the end of a page of text. When printed to a file it terminates a page (the next character is written at the top of the next page). The eop character has the code 12 on ASCII and EBCDIC systems.

Each implementation specifies a "null character" that is by default ignored (discarded) when encountered in a text input file. The null character code is given by $nulChar. The null character code for an ASCII or EBCDIC character set is the code 0. See Section 1.259 of part II of the "MAINSAIL Language Manual" for further information about the treatment of null characters in a text input file.

The MAINSAIL compiler translates character codes (e.g., in string constants) from the host character set (the one used by the compiler) to the target character set (the one used by the compiled program). Unique characters on the host machine are translated to unique characters on the target machine, provided that the host characters are among those shown in Table 2.1-1. Other characters are used at the programmer's risk; if a character cannot be translated to the target character set, a compiletime error occurs. Characters in comments are not translated to the target character set; they affect only the portability of the source text itself.

Table 2.1-2 shows the system procedures provided to complement the minimal assumptions guaranteed above. The argument to each is an integer character code.

-1 is used in several situations to represent "no character", since no character code can be -1 (codes are guaranteed nonnegative). For example, "first(s)" returns the character code of the first character of the string s; if s is empty (i.e., contains no characters), then "first(s)" is -1.

isLowerCase(i)	True if i is the code for one of a...z.
isUpperCase(i)	True if i is the code for one of A...Z.
isAlpha(i)	True if i is the code for one of a...zA...Z.
isNul(i)	"isNul(i)" is true if i is the code for the null character, $nulChar.
prevAlpha(i)	Code of the alphabetically previous character (same case) before the one with code i. Undefined if i is not the code for one of b...zB...Z.
nextAlpha(i)	Code of the alphabetically next character (same case) after the one with code i. Undefined if i is not the code for one of a...yA...Y.
cvl(i)	If i is the code for one of A...Z, then the result is the code for the corresponding lowercase letter; otherwise it is i itself.
cvu(i)	If i is the code for one of a...z, then the result is the code for the corresponding uppercase letter; otherwise it is i itself.

Table 2.1-2. Character-Set-Independent System Procedures

2.2. Comments

A comment is used for documentation in MAINSAIL source code. A comment starts with the character "#" and extends to the end of the line; when the compiler comes upon "#", it ignores the remainder of the line. A comment may begin anywhere on a line. See Example 2.2-1.

```
a[1] := 0;              # clear first element
```

Example 2.2-1. Sample Comment

A large body of text may be "commented out" in three ways:

1. Insert "#" at the start of every line.

2. Use conditional compilation: "IFC FALSE THENC ignoredText ENDC" (see Section 14.10).

3. Use "SKIPSCAN" and "BEGINSCAN" to skip pages (see Section 14.18).

2.3. Identifiers

An identifier is (an optional dollar sign followed by) a letter followed by any number of letters and digits. The letters and digits must be contiguous (e.g., no intervening spaces).

In comparing identifiers, the compiler does not distinguish between upper- and lowercase letters; e.g., it considers the identifiers "typecode", "typeCode", and "TYPECODE" to be identical. There is no "break" character for identifiers; programmers with access to lower case may use a mixture of lower and upper case to show the structure of identifiers. For example, "sizeOfArray" is more understandable than "sizeofarray" or "SIZEOFARRAY".

Certain identifiers ("keywords" such as "BEGIN", "END", and "ARRAY") are "reserved" and cannot be declared or defined by the programmer. A list of the reserved identifiers is given in Appendix G.

Certain other identifiers (e.g., "tab", "create", "delete") are predefined by MAINSAIL and cannot be declared by the programmer.

"$" is the initial character of certain predefined and predeclared identifiers used by the MAINSAIL runtime system. This avoids conflicts with the programmer's identifiers, which must not begin with "$". When XIDAK creates a new predefined identifier, it begins with "$". A user declaration of an identifier beginning with "$" has undefined consequences.

Examples of legal identifiers are shown in Example 2.3-1.

Examples of illegal identifiers are shown in Example 2.3-2.

```
i,j,k,l,m,n              # common integer identifiers
make4sets
aVeryLongIdentifier      # same as AVERYLONGIDENTIFIER
```

Example 2.3-1. Legal Identifiers

```
array                    # reserved identifier
5th                      # starts with a number
cost-in-$                # "$" and "-" cannot be used
```

Example 2.3-2. Illegal Identifiers

2.4. Use of Semicolons and Formatters

Semicolons separate (rather than terminate) declarations and statements. They terminate compiler directives, procedure headers, and macro definitions.

Usually any number of formatters (e.g., spaces, tabs, and ends-of-line) may separate syntactic units. When in doubt, consult the description of the language construct in question.

2.5. Compiletime Evaluation

The compiler evaluates boolean, (long) integer, (long) bits, and string operations with constant operands at compiletime. A call to a system procedure declared with qualifier "COMPILETIME" is evaluated at compiletime if all the arguments are constants. The term "constant expression" refers to an expression that can be evaluated at compiletime.

All compiletime arithmetic is carried out on string representations (with a very large number of digits) so that the capabilities of the computer on which the compiler is running do not affect the results. Integer arithmetic operations that overflow and (long) bits operations that discard bits to the left may therefore not have the same result at compiletime as they would have had if the operations had been performed on the target at runtime.

String operations evaluated at compiletime produce the same result that would have been produced if they had been evaluated at runtime; i.e., they act on strings as translated to the target character set, rather than as on the host machine.

If an expression contains constant subexpressions, the subexpressions should be enclosed in parentheses to ensure that they are evaluated at compiletime. For example, "i + 2 + 4", where i is not evaluated at compiletime, should be written as "i + (2 + 4)" to ensure that the addition of 2 and 4 is done at compiletime. It might otherwise be treated as "(i + 2) + 4", which involves two additions during execution.

2.6. Storage Units and Character Units

A storage unit is the basic measure for the amount of memory required by the various data types. For example, a storage unit may represent a "byte" or "word", although these concepts are not used by MAINSAIL. Every storage unit contains a processor-dependent number of bits, given by the predefined constant $bitsPerStorageUnit.

Data files and memory in which values of the MAINSAIL data types are stored are viewed as linear sequences of storage units. Storage units are employed in situations requiring a measure of memory or file size without regard to data type, e.g., as an argument to the system procedure setPos for a data file.

The system procedure "size" can be used to determine the number of storage units occupied by a particular data type or record. The procedure $ioSize can be used to determine how many storage or character units a data type occupies in a given data file. "DSP" returns the offset in storage units from the start of a record to a field in the record.

Text files and memory in which characters are stored may be viewed as linear sequences of "character units". Character units are employed when a file or memory position must be specified as the number of characters it contains, e.g., as an argument to the system procedure setPos for a text file. A character unit is always eight bits.

Character units do not necessarily coincide with storage units, but storage units are always an integral multiple of character units. The number of bits per character unit (eight) is defined as $bitsPerChar, so the number of character units per storage unit is given by "$bitsPerStorageUnit DIV $bitsPerChar".

2.7. Type Codes

Each data type is assigned an integer type code that is used in various ways in MAINSAIL, e.g., as an argument to the compiletime system procedure "size".

Predefined integer constants for the type codes are shown in Table 2.7-1.

```
Each of these identifiers is a predefined
integer constant:

    booleanCode
    integerCode
    longIntegerCode
    realCode
    longRealCode
    bitsCode
    longBitsCode
    stringCode
    addressCode
    charadrCode
    pointerCode

For example:

    size(integerCode)

is the number of storage units in an integer.
```

Table 2.7-1. Type Codes

2.8. Garbage Collections and Memory Management

The MAINSAIL runtime system automatically reclaims the space occupied by records, arrays, and data sections (collectively known as "chunks") and by string text if the chunks or text become inaccessible. A chunk is inaccessible if no accessible pointer (local variable or pointer in an accessible chunk) references it; string text is inaccessible if no accessible string descriptor references it. Garbage collections (and other memory management operations; "garbage collections" is often used in this manual to denote a variety of memory management operations) can move chunks and string text as well as deallocate them. This is usually invisible to a program, since the referencing variables are automatically updated to point to the moved data.

Variables of the types address and charadr are not updated during a garbage collection, even if they point to structures that may be moved. The user ordinarily uses variables of these data types to point into "scratch space" (or "static space"), i.e., areas of memory in which data are not collected. Scratch space may be obtained by calling the system procedure newPage or the system procedure newScratch.

Constructs that may trigger garbage collections are noted in this manual.

Since garbage collections may take a great deal of time, the programmer may wish to prevent collections if he or she knows that few inaccessible data are being generated. This may be accomplished by incrementing the system variable $collectLock (although doing this may cause MAINSAIL to run out of memory if inaccessible data are in fact being generated). The system procedure $collect causes a collection to be performed, even when $collectLock is non-zero. Collections can also be prevented (or reduced in scope or frequency) indirectly by calling the system procedure "dispose" to deallocate data structures explicitly whenever possible.

Frequency of garbage collection can be controlled by using the utility CONF to set various parameters in a MAINSAIL bootstrap; see the "MAINSAIL Utilities User's Guide" for details.

2.9. cmdFile and logFile

cmdFile and logFile are files associated by default with a MAINSAIL execution's primary input and primary output (usually terminal input and terminal output), respectively. They are described in Section 18.12.

3. Data Types

This chapter describes MAINSAIL's eleven data types: boolean, integer, long integer, real, long real, bits, long bits, string, pointer, address, and charadr. Arrays, records, and data sections, which are "data structures" rather than "data types" in MAINSAIL terminology, are described in Chapters 7, 8, and 10, respectively.

Associated with each data type is a set of values and a set of operations that may be performed on the values. The set of values associated with each data type includes a value called the "Zero" of the data type. The memory representation of the Zero of every data type consists entirely of 0-bits.

There is no implicit data type conversion in MAINSAIL. For example, if i is an integer variable and r a real variable, then "i + r" is an illegal expression. Conversion procedures are provided to convert arguments to another data type. They are discussed in Section 3.9. "cvi", for example, is a procedure that converts its argument to an integer; "i + cvi(r)" is a legal expression.

The difference between the data types integer, real, and bits and their corresponding "long" types is the guaranteed "range" of values. For example, an integer in a portable program may have a value between -32767 and 32767, inclusive, while a long integer may have a value between -2147483647 and 2147483647, inclusive. Some machines can easily support both ranges, while others can support the long ranges only through the use of "software packages"; the programmer should employ the long forms only when necessary, since they may be less efficient.

A program that generates a value outside of the machine-dependent range of its data type behaves in an undefined fashion. Overflow and underflow are not necessarily caught; see the appropriate operating-system-dependent MAINSAIL user's guide for details.

It is an error to use a constant that cannot be represented on the target machine, e.g., an integer that is too large.

For each data type discussed in this chapter, a list of the operators that may be used with values of the data type is given. All operators are described in more detail in Section 4.8. Each data type description also includes a brief description of some of the system procedures that may be used with it. Complete system procedure descriptions are given in Chapter 1 of part II of the "MAINSAIL Language Manual".

XIDAK reserves the right to create new MAINSAIL data types, and to enhance any system procedure, macro, or variable to handle such new data types.

3.1. Boolean

Boolean values are the logical values true and false. The boolean constants are "TRUE" and "FALSE". The boolean Zero is "FALSE".

The operators shown in Table 3.1-1 may be used with boolean expressions.

```
OR   AND   NOT   =   NEQ   :=
```

Table 3.1-1. Boolean Operators

3.2. Integer and Long Integer

Integer and long integer are data types for representing mathematical whole numbers. An integer is guaranteed the range -32767 to 32767; a long integer, -2147483647 to 2147483647.

An integer constant is composed of an optional minus sign ("-") followed by decimal digits ("0" through "9"). Some examples are: "1874", "-53", and "0".

A long integer constant is like an integer constant except that it must be immediately followed by the letter "L" (or lowercase "l"), e.g., "1874L", "-53L", "0L", or "298752341".

A character enclosed in single quotes represents the integer constant of which the value is the target-machine character code of the enclosed character. For example, 'A' represents the integer constant that is the character code of the letter "A" on the target machine. Character codes are discussed in Section 2.1.

The integer Zero is "0", and the long integer Zero is "0L" (or "0l").

The operators shown in Table 3.2-1 may be used with (long) integer expressions.

```
OR      =       LEQ     :=      +                       DIV
AND     NEQ     >       MIN     - (unary & binary)      MOD
NOT     <       GEQ     MAX     *                       ^
```

Table 3.2-1. Integer and Long Integer Operators

The system procedure shown in Table 3.2-2 may operate on (long) integer expressions.

```
abs          absolute value of a (long) integer
```

Table 3.2-2. System Procedure for Integer and Long Integer

3.3. Real and Long Real

Real and long real are data types for representing "floating point" numbers. A floating point number consists of a fraction and a power-of-ten exponent; the value of the number is the product of the fraction and ten to the power of the exponent. For a real, the fraction is guaranteed to have at least six full decimal digits of significance. The exponent is guaranteed a range wide enough that at least one number less than or equal to ten to the minus 38th power (1.0E-38) can be represented as a real, and at least one number greater than or equal to ten to the plus 38th power (1.0E+38) can be represented as a real. For a long real, the fraction is guaranteed to consist of at least 11 full decimal digits, and the exponent range is guaranteed to be at least as large as that of a real exponent.

A real constant is like an integer constant except that it has either a decimal point, an exponent, or both. An exponent immediately follows the last digit (or the decimal point if it is last), and is the letter "E" (or "e") immediately followed by an integer. A nonnegative exponent may be separated from "E" by "+". Some real constants are shown in Example 3.3-1.

```
1874.56
-.78E-3 (= -.00078)
0.
1E3 (= 1e3 = 1E+3 = 1.E3 = 1.E+3 = 1000.)
```

Example 3.3-1. Real Constants

A long real constant is like a real constant except that it must be immediately followed by the letter "L" (or lowercase "l"), as in Example 3.3-2.

The real Zero is "0.", and the long real Zero is "0.L" (or "0.l").

The operators listed in Table 3.3-3 may be used with real and long real expressions.

```
12387658.5L
-.57E28L
0.0L (= 0.L)
```

Example 3.3-2. Long Real Contants

OR	=	LEQ	:=	+	/
AND	NEQ	>	MIN	- (unary & binary)	^
NOT	<	GEQ	MAX	*	

Table 3.3-3. Real and Long Real Operators

The system procedures shown in Table 3.3-4 may be used with real and long real expressions. Trigonometric functions such as sin, cos, and log are also provided; see Chapter 1 of part II of the "MAINSAIL Language Manual".

abs	absolute value of a (long) real
ceiling	smallest (long) integer not exceeded by a (long) real
floor	largest (long) integer not exceeding a (long) real
truncate	truncate a (long) real to a (long) integer

Table 3.3-4. System Procedures for Real and Long Real

3.4. Bits and Long Bits

Bits and long bits are data types for representing sequences of bits. The difference is that a bits consists of (at least) 16 bits and a long bits consists of (at least) 32 bits. Bits may take part in bit operations such as masking, shifting, and testing.

A bit has two states, 0 and 1, sometimes called "0-bit" and "1-bit" or "clear" and "set". To cause a bit to enter the 0 state is to "clear" it; to cause it to enter the 1 state, to "set" it.

A bits constant is a sequence of characters preceded by a single quote and a letter that indicates the base: "B" (or "b") for binary (base 2), "O" (or "o") for octal (base 8), or "H" (or "h") for hexadecimal (base 16). The base letter may be omitted for octal; i.e., octal is the default.

Each binary character ("0" or "1") represents a single bit. Each octal character ("0" through "7") represents three bits (000 through 111). Each hexadecimal character ("0" through "9", "A" through "F") represents four bits (0000 through 1111). The lowercase letters "a" through "f", like "A" through "F", can be used to represent the bit patterns 1010 through 1111 in hexadecimal constants.

The bits for each character are concatenated to obtain the bits of the constant. For example, "'B101011", "'O53" (or just "'53") and "'H2B" all represent the same bit sequence 101011 (ignoring leading zeros).

Other examples of bits constants are "'573", "'B10111", and "'H82A3".

A long bits constant is like a bits constant except that it must be immediately followed by the letter "L" (or lowercase "l"), e.g., "'743L" (= 'B111100110L = 'H1D6L = 'h1d6l).

The bits Zero is "'0" (or equivalently "'B0" or "'O0" or "'H0"); the long bits Zero is the bits Zero followed by "L" (or lowercase "l"), e.g., "'0L".

Bits are numbered from right to left starting with zero.

The operators shown in Table 3.4-1 may be used with bits and long bits expressions.

OR	=	NTST	:=	MSK	SHR
AND	NEQ	TSTA	IOR	CLR	!
NOT	TST	NTSTA	XOR	SHL	

Table 3.4-1. Bits and Long Bits Operators

The system procedures shown in Table 3.4-2 may operate on bits and long bits expressions.

bMask	form a bits mask (sequence of 1-bits)
lbMask	form a long bits mask (sequence of 1-bits)

Table 3.4-2. System Procedures for Bits and Long Bits

3.5. String

String is a data type for representing and manipulating sequences of characters.

A string is a variable-length sequence of characters. MAINSAIL automatically keeps track of how many characters are in a string. A string may contain up to 32766 characters, although long strings may result in serious inefficiencies when used in certain operations.

A string constant is a sequence of characters enclosed in double quotes. Some examples are shown in Example 3.5-1. A double quote is represented in a string constant with two double quotes. Each pair of double quotes stands for one double quote inside the string. For example, the last string in Example 3.5-1 contains two embedded quotes. It contains 23 characters; the two extra double quotes are not retained as part of the string constant, since they are only indicators to the compiler.

```
"Hello"
"She is 12 years old"
"The umbrella cost $2.50"
"He cried ""Wolf!"" again."
```

Example 3.5-1. String Constants

A string constant may extend across line and page boundaries; the characters that indicate the boundaries are part of the constant. For example, the string:

```
"This is a string constant that extends
across a line boundary in the source text"
```

has an embedded eol. It could also be written:

```
"This is a string constant that extends " & eol &
"across a line boundary in the source text"
```

The concatentations are performed at compiletime, since all the strings involved are constants.

The string Zero (sometimes called the "null string") is "". It is the string consisting of no characters.

"&" is the concatenation operator. "s1 & s2" is the string consisting of the characters of s1 immediately followed by the characters of s2. Thus, if s1 has the value:

```
"This is "
```

and s2 has the value:

```
"a concatenated string"
```

then the expression "s1 & s2" has the value:

```
"This is a concatenated string"
```

Substrings are described in Section 4.4, and string comparison in Section 4.8.1.

The operators shown in Table 3.5-2 may be used with string expressions.

OR	=	LEQ	:=	&
AND	NEQ	>	MIN	
NOT	<	GEQ	MAX	

Table 3.5-2. String Operators

The system procedures shown in Table 3.5-3 may operate on string expressions.

3.5.1. Low-Level String Manipulation

Strings are represented in memory as "string descriptors", composed of a length and a character address. String descriptors usually point to characters stored in a region of memory called "string space". The characters stored in string space are subject to garbage collection if they become inaccessible (i.e., no string descriptor points to them). The characters of a string allocated in scratch space or created by a foreign language procedure do not reside in string space. The user who needs to move such a string into MAINSAIL's string space may do so by means of the system procedure $getInArea.

Most programs that do not call foreign language procedures do not need to manipulate strings or string descriptors explicitly with newString or $getInArea.

More complete information on the MAINSAIL string implementation, along with suggestions for efficient use of strings, may be found in the "MAINSAIL Tutorial".

length	number of characters in a string
cvu	convert a string to upper case
cvl	convert a string to lower case
compare	return -1, 0, or 1 to indicate comparison of two strings (see Section 4.8.1). Can be made to treat upper and lower case identically, i.e., a "caseless" comparison
equ	returns true if two string arguments are equal. Like "compare", can do a caseless comparison
first	first character of a string
last	last character of a string
read	reads a value from a string
write	writes a value to a string
cRead	reads a character from a string
cWrite	writes a character to a string
rcRead	reads a character from the end of a string (reverse cRead)
rcWrite	writes a character to the front of a string (reverse cWrite)
$dup	reduplicate a string (concatenate with itself)
scan	scans a string according to a scan specification
scanSet	sets up scan bits to be used with scan
$scanSet	sets up scan integers to be used with scan
scanRel	releases scan bits or integers used with scan
newString	create a string descriptor from a charadr and a length
$getInArea	ensure that a string is in MAINSAIL's string space

Table 3.5-3. System Procedures for String

3.5.2. String Constants and Garbage Collection

The first time each string constant in a module is used, its characters may be copied into string space. This can trigger a garbage collection. Subsequent uses of the same string constant (in the same module) use the previously copied characters, and so do not cause a collection.

3.6. Pointer

Pointer is a data type for referencing records or data sections. Pointers are frequently "classified", i.e., associated with a particular class, as described in Section 8.4.

The only pointer constant is "NULLPOINTER", which is the pointer Zero. A nullPointer references no record or data section.

The operators in Table 3.6-1 may be used with pointer expressions.

OR	AND	NOT	=	NEQ	:=

Table 3.6-1. Pointer Operators

3.7. Address

Address is a data type for representing the location of a storage unit in memory. Addresses may be used for loading and storing values of any data type to and from memory. Individual characters are usually loaded and stored by means of the data type charadr.

Address is a "low-level" data type; many user programs can be written without the use of addresses.

Not every address representable on a processor is a valid MAINSAIL address; e.g., on some implementations of MAINSAIL, an address that is not a multiple of the size of the smallest data type is considered "unaligned" (or "non-data-type-aligned") and is invalid. Portable programs must therefore compute addresses as linear combinations of exact multiples of the sizes of MAINSAIL data types. Furthermore, during any particular execution of MAINSAIL, some addresses may be invalid for reading or writing because the storage units they reference are protected by the operating system. The use of an invalid address is undefined. Storage units in regions of memory allocated by the system procedures newScratch and newPage are always valid for reading or writing. Storage units in regions of memory allocated by the

system procedure "new" are also valid for reading and writing, provided that invalid values (e.g., pointers not pointing to a valid MAINSAIL data structure) are not stored into a MAINSAIL data structure. Storing into other parts of memory is undefined, since it may overwrite MAINSAIL runtime data structures.

The address of a collectable MAINSAIL data structure may change if a garbage collection occurs; an address variable is not updated in such a case. Collectable data are normally referenced with the pointer and string data types, which are updated when a garbage collection occurs.

Addresses may be classified like pointers; see Section 8.4.

The only address constant is "NULLADDRESS", which is the address Zero.

Addresses are ordered with respect to the relative position of the referenced storage units in memory. It is this order that is used when comparing addresses, or using "MIN" or "MAX" on an address.

The operators shown in Table 3.7-1 may be used with address expressions.

OR	AND	NOT	=	NEQ	<
LEQ	>	GEQ	:=	MIN	MAX

Table 3.7-1. Address Operators

System procedures used in operations with addresses are shown in Table 3.7-2.

3.8. Charadr

Charadr ("character address") is a data type for representing the address of a character unit in memory. Data other than characters are usually loaded and stored by means of the address data type.

Charadr is a "low-level" data type; many user programs can be written without the use of charadrs.

As with addresses, there may be charadr values at which the effect of performing a load or store is undefined; see Section 3.7.

The only charadr constant is "NULLCHARADR", which is the charadr Zero.

clear	clears storage units of memory
copy	copies storage units from one memory location to another
(x)Load	loads a value (of data type x) from memory
store	stores a value into memory
displace	returns an address that is displaced n (one of its arguments) storage units from another address
displacement, lDisplacement	computes the distance between two addresses
newPage	gets some memory pages
pageDispose	disposes of pages obtained with newPage
newScratch	returns the address of some memory for "scratch space"
scratchDispose	disposes of scratch space
read	reads a value from an address
write	writes a value to an address

Table 3.7-2. System Procedures for Addresses

The operators shown in Table 3.8-1 may be used with charadr expressions.

OR	AND	NOT	=	NEQ	<
LEQ	>	GEQ	:=	MIN	MAX

Table 3.8-1. Charadr Operators

System procedures used in operations with charadrs are shown in Table 3.8-2.

clear	clears character units of memory
copy	copies characters from memory starting at one charadr to memory starting at another charadr
cLoad	loads a character from memory
store	stores a character into memory
cRead	reads a character from memory
cWrite	writes a character to memory
displace	returns a charadr that is displaced n (one of its arguments) characters from another charadr
displacement	computes the distance between two charadrs
newString	makes a string (descriptor) from a charadr and an integer (length)

Table 3.8-2. System Procedures for Charadrs

3.9. Conversion Procedures

A conversion procedure converts from one data type to another. For example, if the value of the real variable r is 8., then "cvi(r)" has the integer value 8, where "cvi" is the convert-to-integer procedure. MAINSAIL does not provide implicit data type conversion; the programmer is responsible for using conversion procedures where necessary.

A conversion procedure for converting a value of type x to type y is provided for each x-y combination for which the box is marked in Table 3.9-1. The data type abbreviations listed in Table 1.3-1 are used in Table 3.9-1.

For detailed descriptions of the conversion procedures, see Chapter 1 of part II of the "MAINSAIL Language Manual".

MAINSAIL does not guarantee to catch underflow or overflow in conversions. The effect is undefined of calling one of the MAINSAIL system routines that converts a string to numeric value (e.g., read, cvi, cvli, cvr, cvlr) if the numeric value is outside the range supported by the processor.

\y / x\	i	li	r	lr	b	lb	s	a	c	p
i	*	*	*	*	*	*	*			
li	*	*	*	*	*	*	*	*		
r	*	*	*	*			*			
lr	*	*	*	*			*			
b	*	*			*	*	*			
lb	*	*			*	*	*	*	*	
s	*	*	*	*	*	*	*		*	
a		*				*		*	*	*
c						*		*	*	
p								*		*

* means the conversion procedure "cvy" is available and converts values of data type x to data type y.

Table 3.9-1. Allowed Conversions

4. Expressions

An expression provides the means of accessing and computing values.

An expression can be a constant, a variable, a call to a typed procedure (a Procedure Expression), a substring, an If Expression, an Assignment Expression, a compiletime pseudo-procedure, or a combination built up with operators and parentheses. Unless otherwise stated, the order of evaluation of the components of an expression is unspecified.

4.1. Constants

A "constant" represents a value known when the program is written. See Chapter 3 for the exact format of each data type's constants. Symbolic constants may be defined as described in Chapter 13.

4.2. Variables

A "variable" provides access to a data value. Variables are used rather than constants when the programmer knows that a value will be needed in a given computation but does not know in advance what the value will be.

The data type and other attributes of a variable are given in the variable's declaration; see Chapter 6.

The value of a variable may be changed by an Assignment Statement (Section 5.1), by an Assignment Expression (Section 5.2), or by a dotted operator (Section 4.8.5), or when used as a procedure argument corresponding to a modifies or produces parameter (Section 9.5).

A variable is either a simple variable, a subscripted variable, or a field variable.

A simple variable is an identifier associated with a single value that may be changed during program execution as governed by its data type. For example, if n is an integer variable, it may be assigned integer values during program execution.

Subscripted variables are used to access the elements of an array (see Section 7.5) and field variables are used to access the fields of a record (see Section 8.6) and the interface fields of a module (see Section 10.3).

4.3. Procedure Expression

A Procedure Expression is a procedure call used in an expression. Procedures are described in Chapter 9, procedure calls in Section 9.2. A procedure used in a Procedure Expression must be a typed procedure (see Section 9.3); i.e., it must return a value. Section 5.3 explains the difference between a procedure call in an expression and a procedure call as a statement.

A call to a typed procedure in an expression invokes execution of the procedure body and then uses the value it returns in place of the procedure call in the expression.

4.4. Substrings

A substring of a string s is a sequence of zero or more contiguous characters of s. For example, "pur", "rp", "e", "", and "purple" are substrings of "purple", but "prp" is not.

A substring of a string s is specifed by:

```
s[e1 TO e2]        or        s[e1 FOR e2]
```

where e1 and e2 are integer expressions. e1 is the start position. e2 is the stop position in the "TO" form, and the length in the "FOR" form. The characters of a string are numbered from left to right, the first position being number one. If e1 is less than one, then the effect is the same as if one were used for e1, and if e2 is greater than the length of s, the effect is the same as if the length of s were used for e2. See Example 4.4-1.

```
If:

   s = "yellow"

then:

   s[1 TO 4]  = s[1 FOR 4] = s[-3 TO 4] = "yell",
   s[4 TO 6]  = s[4 FOR 3] = "low", and
   s[7 FOR 1] = "" (since s doesn't have a 7th character).
```

Example 4.4-1. Substring Examples

A substring cannot be assigned a value, since it is not a variable.

The s in "s[e1 TO/FOR e2]" may be a string variable, a string constant, a parenthesized string expression, or a call to a string procedure. If s is a string constant, and both e1 and e2 are integer constant expressions, then the substring is evaluated at compiletime.

4.4.1. "INF"

"INF" may be used anywhere within substring brackets as an integer that represents the length of the string of which the substring is being taken. See Example 4.4.1-1.

```
If

   s = "brown"

then

   s[1 TO INF - 1] = s[1 TO length(s) - 1] = s[1 TO 4] =
     "brow".
```

Example 4.4.1-1. Use of "INF"

"INF" stands for "infinity". It gives the rightmost character position, no matter what the length of the string.

"INF" is evaluated at compiletime if the string is a constant. In this case the length returned is the same as that that would have been obtained at runtime.

4.5. If Expression

An If Expression provides a choice among several expressions. The form of an If Expression is:

```
IF e1 THEN e2 ELSE e3
```

where e1, e2, and e3 are expressions. If Expressions may be nested, e.g.,

```
IF e1 THEN e2
ELSE IF e3 THEN e4
ELSE e5
```

MAINSAIL provides the abbreviations "EF" for "ELSE IF" and "EL" for "ELSE". They allow alignment of conditions in If Expressions for clarity. Thus, the example above could be written as:

```
IF e1 THEN e2
EF e3 THEN e4
EL e5
```

In an If Expression, each possible "result expression" following "THEN" is preceded by a "condition expression" following "IF" or "EF". The condition expressions are evaluated one by one (starting with the first) until one evaluates to a non-Zero value. Its associated result expression becomes the value of the If Expression, and no further condition expressions are evaluated. If all condition expressions evaluate to Zero, the expression after the final "ELSE" (or "EL") becomes the value of the If Expression. Unselected result expressions are not evaluated.

All the result expressions must be "assignment compatible" with one another; i.e., they must evaluate to the same data type (see Section 4.9 for a definition of assignment compatibility). For example, if e2 above were an integer expression, then e4 and e5 would also have to be integer expressions. Conversion procedures (see Section 3.9) may be used to convert an expression to the proper data type.

The overall data type of the If Expression is the same as that of the result expressions.

The result expressions may be arrays. For purposes of assignment compatibility, the expression is considered to have the characteristics (type, dimension, bounds, etc.) of the first result array expression (e2 in the above example). e4 and e5 must be assignment compatible with e2.

A sample use of an If Expression used in an Assignment Statement is shown in Example 4.5-1.

```
var := IF i < 0 THEN k
       EF i = 0 THEN k + 1
       EL k * 10

is equivalent to the If Statement
(see Section 5.6):

IF i < 0 THEN var := k
EF i = 0 THEN var := k + 1
EL var := k * 10
```

Example 4.5-1. If Expression Used in Assignment Statement

If Expressions are not evaluated at compiletime.

When an If Expression is used as an operand, it must be enclosed in parentheses; see Example 4.5-2.

```
result := a + (IF b > 10 THEN c ELSE d)

CASE i := (IF j THEN k ELSE m) OFB ...
```

Example 4.5-2. If Expressions Used as Operands

4.6. Assignment Expression

An Assignment Expression assigns a value to a variable and then uses that value as the value of the expression. The value may be an array. See Example 4.6-1.

```
IF i := j + 2 THEN s...

    is equivalent to

i := j + 2; IF i THEN s...
```

Example 4.6-1. Assignment Expression

An expression that would depend on whether the variable on the left or the expression on the right of the ":=" is evaluated first is undefined. Thus, "i := 0; a[i] := (i := 2)" may assign to "a[0]", assign to "a[2]", give an error message, or produce undefined behavior. It is the programmer's responsibility to avoid Assignment Expressions that are affected by the order of evaluation of the variable and the expression.

If the variable assigned to is changed before the assigned value is utilized, the result of the Assignment Expression is undefined. For example, the results of the expressions "(v := 3) > (v := 0)" and "(a[i] := 4) > (i := 0)" are undefined. It is the programmer's responsibility to avoid undefined expressions.

4.7. Compiletime Pseudo-Procedures

"DCL", "NEEDBODY", "NEEDANYBODIES", "CHECKING", "$TYPEOF", "$CLASSOF", and "$ISCONSTANT" are compiletime pseudo-procedures used primarily with conditional compilation. Chapter 14 describes these compiletime pseudo-procedures.

"DSP" is a compiletime pseudo-procedure that returns the offset of a field in a record; see Section 14.8.

$compileTimeValue is a compiletime procedure that provides a number of miscellaneous compiletime values; see Section 1.69 of part II of the "MAINSAIL Language Manual".

4.8. Operators and Operations

Tables 4.8-1 and 4.8-2 summarize the operators that may appear in MAINSAIL expressions.

The second column of each table gives data type information in the general format:

```
t1, ..., tn -> t
```

ti gives the allowed data types for the ith operand (the leftmost operand is number one) by listing the possible data type abbreviations (see Table 1.3-1) for the ith operand, separated by dashes. t is the data type of the result.

For example, in the "shift right" operation "e1 SHR e2", e1 is the first operand, "SHR" is the operator, and e2 is the second operand. From Table 4.8-2, it can be seen that the first operand must be a (long) bits, the second must be an integer, and the result of "e1 SHR e2" is of the same data type as the first operand (i.e., a (long) bits).

"n" (standing for "numeric") is an abbreviation for "i-li-r-lr", and "all" is an abbreviation for "bo-n-b-lb-s-a-c-p-array".

If all ti must be the same data type, then only t1 is given. For example, in the "test" operation "e1 TST e2", the first and second operands (e1 and e2) must both be the same data type, either bits or long bits. If the result value t has the same data type as all the operands, then "-> t" is omitted. For example, in the concatenation operation "e1 & e2", both operands must be strings and the result is also a string.

In the tables, "e", "e1", and "e2" stand for expressions and "v" for a variable.

The operations shown in the Table 4.8-2 are evaluated at compiletime if all the operands are evaluated at compiletime, except in the following cases:

Operation	Data Types	Description of Result
NOT e	all -> bo	IF e THEN FALSE ELSE TRUE (if e is non-Zero the result is FALSE, otherwise it is TRUE)
- e	n	negation of e

Table 4.8-1. Unary Operations

- Any operands are of type (long) real.
- String comparisons other than "=" and "NEQ".
- String "MIN" and "MAX".

4.8.1. String Comparison

The relational operators ("=", "NEQ", ">", "<", "GEQ", "LEQ") compare strings based on their lengths and the characters they contain rather than on the contents of their string descriptors.

Two strings are compared according to the following definition:

1. If both strings are the null string, they are equal.
2. If one string is null and the other is not, then the non-null string is greater.
3. If both strings are non-null, and their first characters differ, then the string with the numerically greater first character code is greater.
4. If both strings are non-null, and their first characters are the same, then their comparison is the same as the comparison of the strings with the first character of each removed.

Because the uppercase and lowercase letters are alphabetically ordered (see Section 2.1), this algorithm produces an alphabetical ordering for strings that are either all uppercase or all lowercase; e.g., "ABC" is less than "ABD" or "ABCD". Strings with the same length and sequence of characters are equal. The null string is less than any other string.

Operation	Data Types	Description of Result
v := e	all	Assign e to v. The result is the value assigned. See Section 4.6. e and v must be assignment compatible (see Section 4.9).
e1 OR e2	all,all -> bo	IF e1 THEN TRUE EF e2 THEN TRUE EL FALSE (e2 is evaluated only if e1 is FALSE)
e1 AND e2	all,all -> bo	IF NOT e1 THEN FALSE EF e2 THEN TRUE EL FALSE (e2 is evaluated only if e1 is TRUE)
e1 = e2	all -> bo	TRUE if e1 is equal to e2.
e1 NEQ e2	all -> bo	TRUE if e1 is not equal to e2.
e1 < e2	n-s-a-c -> bo	TRUE if e1 is less than e2. See Section 4.8.1 regarding string comparsions.
e1 LEQ e2	n-s-a-c -> bo	TRUE if e1 is less than or equal to e2.
e1 > e2	n-s-a-c -> bo	TRUE if e1 is greater than e2. See Section 4.8.1 regarding string comparsions.
e1 GEQ e2	n-s-a-c -> bo	TRUE if e1 is greater than or equal to e2.
e1 TST e2	b-lb -> bo	TRUE if any 1-bit in e2 is a 1-bit in e1. Same as (e1 MSK e2) NEQ '0. TST stands for "test".

Table 4.8-2. Binary Operations (continued)

e1 NTST e2	b-1b -> bo	TRUE if no 1-bit in e2 is a 1-bit in e1. Same as NOT (e1 TST e2). NTST stands for "not test".
e1 TSTA e2	b-1b -> bo	TRUE if all 1-bits in e2 are 1-bits in e1. Same as (e1 MSK e2) = e2. TSTA stands for "test all".
e1 NTSTA e2	b-1b -> bo	TRUE if not all 1-bits in e2 are 1-bits in e1. Same as NOT (e1 TSTA e2). NTSTA stands for "not test all".
e1 MIN e2	n-s-a-c	Minimum of e1 and e2.
e1 MAX e2	n-s-a-c	Maximum of e1 and e2.
e1 + e2	n	Sum of e1 and e2.
e1 - e2	n	Difference of e1 and e2.
e1 IOR e2	b-1b	"Inclusive or" of e1 and e2. See Section 4.8.2.
e1 XOR e2	b-1b	"Exclusive or" of e1 and e2. See Section 4.8.2.
e1 MSK e2	b-1b	"Mask" e1 with e2; i.e., clear any bits in e1 that are 0-bits in e2. See Section 4.8.2.
e1 CLR e2	b-1b	"Clear" e2 from e1, i.e., clear any bits in e1 that are 1-bits in e2. See Section 4.8.2.
e1 ! e2	b-1b	Same as e1 IOR e2, except has higher priority (see Section 4.8.4).

Table 4.8-2. Binary Operations (continued)

e1 * e2	n	Product of e1 and e2.
e1 / e2	r-lr	Quotient (real) of e1 and e2.
e1 DIV e2	i-li	Quotient (integer) of e1 and e2. The remainder is discarded. Undefined if e1 is negative or e2 is not positive.
e1 MOD e2	i-li	Remainder of e1 divided by e2. Same as e1 - e2 * (e1 DIV e2). Undefined if e1 is negative or e2 is not positive. MOD stands for modulus, another name for remainder.
e1 SHL e2	b,i -> b lb,i -> lb	e1 shifted left by e2 bits; leftmost e2 bits are lost. 0-bits are brought in from the right. Undefined if e2 < 0 or GEQ the number of bits in the data type of e1. At compiletime, an error may occur if a 1-bit is lost at the left (this is subject to change).
e1 SHR e2	b,i -> b lb,i -> lb	e1 shifted right by e2 bits. 0-bits are brought in from the left. Undefined if e2 < 0 or GEQ the number of bits in the data type of e1.
e1 & e2	s	e1 concatenated with e2; the string consisting of the characters of e1 immediately followed by the characters of e2.

Table 4.8-2. Binary Operations (continued)

e1 ^ e2	i,i -> i li,i-> li r,i -> r lr,i -> lr r,r -> r lr,r -> lr	e1 raised to the power e2. If e1 is an integer or long integer, undefined if e2 is negative. If e2 is not a positive integer, undefined if e1 negative. Undefined if e1 and e2 both zero.

Table 4.8-2. Binary Operations (end)

Strings of mixed case may be compared in a "caseless" comparison by means of the upperCase option to the system procedure compare or equ.

4.8.2. Bitwise Operations

"IOR", "XOR", "MSK", and "CLR" perform bitwise operations on (long) bits.

Let a be a (long) bits value, and ai denote the ith bit of a; similarly for b, bi and c, ci. In the computation "c := a op b", ci is related to ai and bi as shown in Example 4.8.2-1.

		ci			
ai	bi	IOR	XOR	MSK	CLR
0	0	0	0	0	0
0	1	1	1	0	0
1	0	1	1	0	1
1	1	1	0	1	0

Example 4.8.2-1. Bitwise Operations

In words:

- "a IOR b" has a 1-bit only where either a or b has a 1-bit.
- "a XOR b" has a 1-bit only where exactly one of a or b has a 1-bit.
- "a MSK b" has a 1-bit only where both a and b have 1-bits.

- "a CLR b" has a 1-bit only where a has a 1-bit and b does not.

4.8.3. Comparison Chains

A comparison chain is a sequence of the form:

```
e1 op1 e2 op2 e3 op3 ... en-1 opn en
```

where the ei are expressions and the opi any of the operators in Table 4.8.3-1. Such a comparison chain is equivalent to the expanded form:

```
(e1 op1 e2) AND (e2 op2 e3) AND (e3 op3 ...) ...
     (en-1 opn en)
```

except that the intermediate ei (i.e., the ei other than e1 and en) are evaluated just once.

=	NEQ	<	LEQ	>
GEQ	TST	TSTA	NTST	NTSTA

Table 4.8.3-1. Operators Permitted in Comparison Chains

A chain may be composed of any combination of data types and operators, provided that the expanded form is valid. The chain is effectively enclosed in parentheses, with AND's inserted at the "shared" expressions. Thus, "NOT e1 = e2 TST e3" is equivalent to "NOT ((e1 = e2) AND (e2 TST e3))", except that e2 is evaluated once.

Consistent with the evaluation of AND, only as many ei are evaluated as necessary to determine the value of "e1 op1 e2 op2 e3 ...". For example, in "e1 < e2 = e3", e3 is evaluated only if "e1 < e2" is true (otherwise the entire expression is false, so there is no reason to proceed any further).

A comparison chain is undefined if its expanded form is undefined; e.g., "1 > v > (v := 2)" is undefined since "v > (v := 2)" is undefined.

4.8.4. Operator Precedence

Table 4.8.4-1 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 4.8.4-1. Precedence of Operators

Operators of equal precedence are associated from left to right; e.g., "a + b + c" is equivalent to "(a + b) + c", with two exceptions:

1. Assignments are associated right to left, so that "a := b := c" is associated as "a := (b := c)".

2. Comparison chains are associated as described in Section 4.8.3. For example, "a < b < c" is associated as "a < b AND b < c" rather than "(a < b) < c" or "a < (b < c)."

No exception is made for "^" (exponentiation), as in some other programming languages; i.e., "a ^ b ^ c" is equivalent to "(a ^ b) ^ c", not "a ^ (b ^ c)".

Since the order of evaluation of the operands of an operator is usually not specified, the programmer must be careful to avoid expressions that could depend on the order of evaluation. For example, in "p(a) + q(b)", where p and q are procedures, it is not specified which of p and q is called first. If it is important that p be called first, then a separate statement must be used to force the evaluation order, e.g., "t := p(a); ... t + q(b)".

The precedence of the assignment operator is illustrated in Example 4.8.4-2. The precedence of the assignment operator is slightly different in expressions and Assignment Statements.

The order in which subexpressions of an expression are evaluated may be explicitly specified with parentheses. The expression enclosed in the innermost set of parentheses is evaluated first. For example, in "((a + b) * c)", "a + b" is evaluated, then its result multiplied by c.

Parentheses may enclose any MAINSAIL expression, whether or not they are required in order to change the operator precedence that would prevail in the absence of parentheses. Redundant parentheses may be used to make source code easier to read.

```
    IF v := e1 OR e2 THEN ...

is equivalent to:

    IF (v := e1) OR e2 THEN ...

NOT equivalent to:

    IF v := (e1 OR e2) THEN ...

But the statement:

    v := e1 OR e2;

is equivalent to:

    v := (e1 OR e2);
```

Example 4.8.4-2. Precedence of the Assignment Operator in Expressions and Statements

4.8.5. Dotted Operators

Most operators may be preceded by a dot (".") to indicate that the value computed by the operator is to be assigned to the leftmost operand of the "dotted operator". The leftmost operand of the operator must be a variable. The result of the operation is the same as for the corresponding non-dotted operator.

The expression "v .op e", (where "v" is a variable, "op" is one of the binary operators that may be dotted, "e" is an expression, and "v := v op e" is well defined) is equivalent to the Assignment Expression "v := v op e", except that:

1. ".op" has the same precedence as "op".

2. If v is a non-simple variable (i.e., a subscripted or field variable), then the location of v within its data structure is evaluated just once. The evaluation of v and e must not affect this calculation; otherwise, the effects are undefined.

".- v" is a short form of "v := - v", except that if v is a non-simple variable, then the location of v within its data structure is evaluated just once.

See Example 4.8.5-1.

```
   If i, j, and k are simple variables, a is a
   one-dimensional integer array, and proc is an
   integer procedure, then:

i .+ 1        is equivalent to  i := i + 1
.- i          is equivalent to  i := - i
i .+ j * k    is equivalent to  i := i + j * k
a[proc] .+ i  is equivalent to  j := proc; a[j] := a[j] + i
              since proc is called just once.

   All operators more binding than the assignment
   operator (see Section 4.8.4) may be dotted:

MIN      MAX
+        - (binary)       IOR      XOR      MSK      CLR
*        /        &       DIV      MOD      SHL      SHR
!        ^
- (unary)
```

Example 4.8.5-1. Dotted Operators

A dotted operator has the same precedence as its corresponding non-dotted operator. An expression "v .op e" containing a dotted operator is undefined if e contains an operator that is evaluated after op; e.g., "a .* b + c" is undefined, since "+" has a lower precedence than "*".

An expression containing a dotted operator is undefined if its equivalent Assignment Expression is undefined; e.g., "(v .+ 5) = (v := 2)" is undefined. It is the programmer's responsiblity to avoid the use of such expressions.

Dotted operators can be used in Expression Statements (Section 5.2).

4.8.6. Garbage Collection

A garbage collection may occur during the concatenation operation ("&" or ".&"). It may also occur during exponentiation operation ("^" or ".^") if the exponent is a real (but not if it is an integer). Other operators cannot trigger garbage collections unless an exception occurs. An exception may occur if an operation overflows or a division by zero is attempted, provided that MAINSAIL intercepts the exception; see the appropriate operating-system-specific user's guide for details.

Other MAINSAIL language constructs that may trigger collections are the Init Statement and many system procedures; see Chapter 1 of part II of the "MAINSAIL Language Manual".

4.9. Assignment Compatibility

Two expressions are said to be "assignment compatible" if the following conditions are satisfied:

1. The expressions must be of same data type.

2. If either is an array, then both must be arrays. If both arrays are typed, then they must be of the same data type. If both arrays have dimensions, then they must have the same number of dimensions. If both are pointer arrays, then the pointer classes must be "related" as described in Section 8.9. If corresponding array bounds are declared as constants, they must be the same constant. The rules of Section 7.7 must be obeyed when one array is short and one long.

3. If either is a module, then both must be modules.

4. If either is a class, then both must be classes.

5. If the expressions are modules, classified pointers, or classified addresses, their classes must be "related" as explained in Section 8.9.

5. Statements

A statement performs an action or directs the flow of control. This chapter describes eleven of the thirteen MAINSAIL statements: Assignment, Expression, Procedure, Return, Begin, If, Case, Iterative, Done, Continue, and Empty. The other two statements are the Init Statement for initializing arrays (see Section 7.4) and the Handle Statement for handling exceptions (see Chapter 16).

Semicolons are used to separate (rather than terminate) statements.

5.1. Assignment Statement

An Assignment Statement gives a value to a variable. The form of an Assignment Statement is "v := e" where "v" is a variable, "e" is an expression, and ":=" is the assignment operator. The value of the expression e is assigned to the variable v. For example, "i := 8" is an Assignment Statement that assigns the value 8 to the variable i.

"_" (the underbar or left arrow character) may be used in place of ":=".

v and e must be "assignment compatible" as explained in Section 4.9.

The order of evaluation of v and e is not defined, so avoid Assignment Statements for which the order might make a difference. For example:

```
i := 0; a[i] := i .+ 1
```

could assign the value 1 to either "a[0]" or "a[1]", since the value of i is changed to 1 by "i .+ 1", but it is not defined which value of i (the one before or after the change) is used to evaluate "a[i]". It is the programmer's responsiblity to avoid such undefined Assignment Statements.

The precedence of the assignment operator is slightly different in expressions and Assignment Statements; see Example 4.8.4-2.

5.2. Expression Statement

An Expression Statement is a dotted expression used as a statement (see Section 4.8.5). It computes a value and assigns that value to the leftmost operand, which must be a variable. In Table 5.2-1, "v" is a variable, "op" is one of the operators that may be dotted (see Section 4.8.5), and "e" is an expression.

```
v .op e    has the same effect as    v := v op e
.- v       has the same effect as    v := - v
```

Table 5.2-1. Expression Statement Format

An Expression Statement is undefined if the expression composing it is undefined; see Section 4.8.5.

```
Assume i and j are simple variables.

Statement      Effect
.- a[i]        Negate a[i].  More efficient than a[i] :=
               - a[i].

i .+ j * 2     Same as i := i + j * 2.

i .+ j .* 2    Same as i := i + (j := j * 2).
```

Example 5.2-2. Examples of Expression Statements

5.3. Procedure Statement

A Procedure Statement is a procedure call (see Section 9.2). It invokes execution of the body of the called procedure. Procedures are described in Chapter 9.

A procedure used as a statement may be either typed or untyped; if typed, its value is discarded. A procedure used in an expression must be a typed procedure (see Section 9.3); its return value is used in the expression.

For example, the system procedure cRead is an integer procedure that returns the character code (see Section 2.1) of the first character of its string argument, and removes that character from the string. A sample call to cRead in an expression is:

```
i := cRead(s)
```

which removes the first character from s and puts its code into i. If it is desired to remove the first character from s without recording its value, then a Procedure Statement may be used:

5.4. Return Statement

A Return Statement returns from a procedure (see Chapter 9). The format is:

```
RETURN
```

for an untyped procedure (see Section 9.3), or:

```
RETURN(e)
```

for a typed procedure, where the value of the expression e is returned as the value of the procedure.

A Return Statement is not necessary in untyped procedures (Section 9.3); an untyped procedure automatically returns upon completion of the execution of the procedure body. However, a Return Statement can be used in an untyped procedure to provide a convenient "early return", much as the Done Statement (see Section 5.9) provides termination of an Iterative Statement (see Section 5.8). A Return Statement is necessary in typed procedures, since it provides the mechanism for returning a value. A runtime error occurs if a typed procedure reaches its final "END" without the execution of some "RETURN(e)".

The expression returned as the value of a typed procedure must be assignment compatible (see Section 4.9) with the data type of the procedure.

Example 5.4-1 shows typed and untyped procedures with Return Statements.

5.5. Begin Statement

A Begin Statement allows a group of statements to be treated as a single statement.

The format of a Begin Statement is the word "BEGIN" followed by a sequence of statements (separated with semicolons) followed by the word "END":

```
BEGIN s1; ...; sn END
```

A string constant may follow "BEGIN" to give a name to the Begin Statement, in which case the same string constant must also follow the "END":

```
BEGIN "name" s1; ...; sn END "name"
```

```
An untyped procedure with a Return Statement:

    PROCEDURE p;
    BEGIN
    INTEGER i;
    ...
    IF i > 0 THEN RETURN;
    ...
    END

A typed procedure with Return Statements:

    INTEGER PROCEDURE p;
    BEGIN
    INTEGER i,j;
    ...
    IF i > 0 THEN RETURN(0);
    ...
    RETURN(j) END
```

Example 5.4-1. Examples of Return Statements

The compiler issues a warning if it finds different string constants (ignoring upper and lower case distinctions) after a "BEGIN" and its matching "END". This check helps catch mismatched "BEGIN"-"END" pairs.

Declarations are not allowed in Begin Statements.

5.6. If Statement

An If Statement selects one of several statements for execution depending on the values of specified expressions.

The simplest form of If Statement is "IF e THEN s" where "e" is an expression and "s" is a statement. s can be a Begin Statement, that is, a list of statements enclosed in a "BEGIN"-"END" pair. If e evaluates to a non-Zero value, the statement s is executed. For example, "IF i THEN j := 2" assigns j the value 2 if and only if i is not zero. Similarly, "IF i = 1 THEN j := 2" assigns j the value 2 if and only if i is equal to one (for in that case the expression "i = 1" is true, which is the boolean non-Zero).

The other form of If Statement is "IF e THEN s1 ELSE s2". If e evaluates to a non-Zero value, then s1 is executed. Otherwise (if e evaluates to Zero), s2 is executed. For example, the statement "IF i = 1 THEN j := 2 ELSE k := 3" assigns j the value 2 if i has the value one, in which case the statement "k := 3" is not executed. Otherwise (if i does not equal one), k is assigned the value 3, and the statement "j := 2" is not executed.

Any statement in an If Statement can be an If Statement. Thus, an If Statement may look as shown in Example 5.6-1.

```
IF e1 THEN s1
ELSE IF e2 THEN s2
ELSE IF e3 THEN s3
ELSE s4

The expressions e1, e2, and e3 are evaluated one by one
until one of them evaluates to a non-Zero value; its
associated statement (s1, s2, or s3, respectively) is then
executed, and no further expressions are evaluated.  If
all the expressions evaluate to Zero, the statement
following the final "ELSE" (s4) is executed.
```

Example 5.6-1. If Statement within an If Statement

MAINSAIL provides the abbreviations "EF" for "ELSE IF", "EL" for "ELSE", "EB" for "ELSE BEGIN", and "THENB" for "THEN BEGIN". The first three abbreviations allow alignment of conditions in If Statements for clarity, as in Example 5.6-2.

```
IF e1 THEN s1
EF e2 THEN s2
EF e3 THEN s3
EL s4
```

Example 5.6-2. Abbreviations Used in If Statements

There is never a semicolon before an "ELSE", since semicolons are used to separate statements, and "ELSE" is not the beginning of a statement.

An "ELSE" ("EL") or "EF" is matched with the innermost unmatched "IF" or "EF". In the following statement, the "ELSE" is matched with the second "IF", and the "EL" is matched with the first "IF":

```
IF e1 THEN
    IF e2 THEN s1 ELSE s2
EL ...
```

If there were no "ELSE s2" above, the "EL" would instead be matched with the second "IF". A Begin Statement could be used as shown below to match the "EL" with the first "IF":

```
IF e1 THEN
    BEGIN IF e2 THEN s1 END
EL ...
```

This might also be written as follows, using "THENB":

```
IF e1 THENB IF e2 THEN s1 END
EL ...
```

5.7. Case Statement

A Case Statement uses an integer index to select one of several statements for execution. The simplest form of a Case Statement is shown in Example 5.7-1, where e (the index) is an integer expression, the si are statements and the ci (the selectors) are integer constant expressions. A semicolon separates a statement from the bracketed selector for the next statement. A semicolon may appear between the last statement sn and the "END", but it is not necessary.

```
CASE e OF BEGIN
    [c1]    s1;
    [c2]    s2;
    ...
    [cn]    sn # semicolon optional here
    END
```

Example 5.7-1. Sample Case Statement

Each statement is preceded by one or more selectors that specify what values of the index are to select that statement. A statement is selected if any of its selectors is satisfied. There are three forms for the selectors (Example 5.7-1 shows only the simplest form of selector); see Example 5.7-2.

```
Selector        Corresponding Statement Is Selected If
[c]             index = c.

[c1 TO c2]      c1 LEQ index LEQ c2; i.e., the index is
                between c1 and c2.  The compiler gives an
                error message if c1 exceeds c2.

[ ]             no other statement would otherwise be
                selected (catch-all selector).
```

Example 5.7-2. Three Forms for Selectors

The Case Statement shown in Example 5.7-1 has the same effect as (but is usually more efficient than) the Assignment and If Statements in Example 5.7-3.

```
t := e;       # t is an integer variable

IF t = c1 THEN s1
EF t = c2 THEN s2
...
EF t = cn THEN sn
```

Example 5.7-3. Less Efficient Form Equivalent to a Case Statement

MAINSAIL provides the abbreviation "OFB" for "OF BEGIN". A string constant may follow the "BEGIN" (or "OFB"), in which case the same string constant must also follow the "END". The compiler issues a warning if these string constants do not match (ignoring upper and lower case distinctions). This check helps catch mismatched "BEGIN"-"END" pairs, just as for the Begin Statement (see Section 5.5).

The first statement with a satisfied selector is selected for execution. This is illustrated in Example 5.7-4.

A runtime error occurs if the index selects no statement and there is no "[]" catch-all selector. A runtime error would result if num had the value 9 in the Case Statement "ex1" in Example 5.7-4. All expected values of the index must be specified in some selector, which can be the catch-all selector "[]". An Empty Statement can be used for those cases in which no action

```
CASE num OFB "ex1"
    [3]          s1;
    [1 TO 7]     s2;    s1 (and not s2) is executed
    [8]          s3     when num has the value 3.
    END "ex1"


CASE num OFB "ex2"
    [1 TO 7]     s1;
    [3]          s2;    s2 could never be selected
    [8]          s3     since num = 3 would select s1.
    END "ex2"
```

Example 5.7-4. Choice of a Selector

should be taken. If "ex1" should do nothing whenever num has a value outside of the range 1 through 8, for example, it could be written as shown in Example 5.7-5.

```
CASE num OFB
    [3]          s1;
    [1 TO 7]     s2;
    [8]          s3;
    []           # catch-all selector: do nothing
    END
```

Example 5.7-5. Use of an Empty Statement in a Case Statement

There can be no more than one catch-all selector in a Case Statement; the catch-all selector matches the same values no matter where it is placed in the Case Statement.

For each Case Statement, the compiler creates a branch table with m entries, where m = <maximum ci> - <minimum ci> + 1 (where ci ranges over all the selector bounds). If m is much greater than the number of cases with specified statements (that is, if the cases are spread sparsely over a wide range), the table results in a significant space overhead. The Case Statement shown in Example 5.7-6 would produce a table with 3000 entries. In this case, it would be better to use an If Statement.

```
CASE num OFB
     [1] [20]     j := 3;
     [980]        k := 8;
     [3000]       BEGIN j := 7; k := 9 END
     END
```

Example 5.7-6. Inefficient Case Statement

5.8. Iterative Statement

An Iterative Statement specifies a statement that is to be repeatedly executed until some condition terminates the iteration. The form of an Iterative Statement is shown in Table 5.8-1, where i is a simple local (long) 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". i, e1, and e2 must all be of the same data type, either integer or long integer.

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

Table 5.8-1. Form of Iterative Statement

The FOR-clause, WHILE-clause, and UNTIL-clause are optional clauses surrounding the required part "DO s". Thus, there are eight possible forms (ignoring the distinction between "UPTO" and "DOWNTO") depending on whether each clause is included or not, as shown in Example 5.8-2.

"DO s" alone repeatedly executes s until something in s terminates the Iterative Statement, such a Done Statement (see Section 5.9), a Return Statement (see Section 5.4), or an exception (see Chapter 16). The other forms are explained in Table 5.8-3. To get the equivalent forms for "DOWNTO", replace "LEQ" with "GEQ", ".+" with ".-" and "UPTO" with "DOWNTO" on the righthand side of Table 5.8-3.

"DOB" is equivalent to "DO BEGIN".

In accordance with Table 5.8-3, the second expression (e2) in a FOR-clause is evaluated just once, before the iterations begin. Furthermore, the use of the largest (long) integer as e2 in an

```
DO s
DO s UNTIL e4
WHILE e3 DO s
WHILE e3 DO s UNTIL e4
FOR i := e1 UPTO e2 DO s
FOR i := e1 UPTO e2 DO s UNTIL e4
FOR i := e1 UPTO e2 WHILE e3 DO s
FOR i := e1 UPTO e2 WHILE e3 DO s UNTIL e4
```

Example 5.8-2. Eight Possible Forms

Form	Equivalent Form
DO s UNTIL e4	DOB s; IF e4 THEN DONE END
WHILE e3 DO s	DOB IF NOT e3 THEN DONE; s END
WHILE e3 DO s UNTIL e4	WHILE e3 DOB s; IF e4 THEN DONE END
FOR i := e1 UPTO e2 DO s	i := e1; t := e2; WHILE i LEQ t DOB s; i .+ 1(L) END
FOR i := e1 UPTO e2 DO s UNTIL e4	FOR i := e1 UPTO e2 DOB s; IF e4 THEN DONE END
FOR i := e1 UPTO e2 WHILE e3 DO s	FOR i := e1 UPTO e2 DOB IF NOT e3 THEN DONE; s END
FOR i := e1 UPTO e2 WHILE e3 DO s UNTIL e4	FOR i := e1 UPTO e2 WHILE e3 DOB s; IF e4 THEN DONE END

Table 5.8-3. Explanation of Forms

"UPTO" FOR-clause or of the most negative (long) integer in a "DOWNTO" FOR-clause is undefined. Such forms may result in (possibly undetected) arithmetic overflow.

In accordance with Table 5.8-3, the value of the iterative variable after the Iterative Statement terminates is one greater (for "UPTO") or one less (for "DOWNTO") than e2, unless the Iterative Statement is terminated early (e.g., by means of a Done Statement), or unless the iterative variable is explicitly modified within the loop.

FOR-clause increments or decrements other than 1 or 1L are not provided. To get the effect of some other increment e, use the equivalent form shown above with ".+ e" in place of ".+ 1(L)", where e is the desired increment.

A sample Iterative Statement with a FOR-clause is shown in Example 5.8-4.

```
FOR i := 1 UPTO 3 DO
    ttyWrite("i is ",i,"; i squared is ",i * i,"." & eol)

    writes to primary output:

i is 1; i squared is 1.
i is 2; i squared is 4.
i is 3; i squared is 9.
```

Example 5.8-4. Sample Iterative Statement

A string constant may follow "DO" to give a name to the Iterative Statement. This name may then be used in a Done Statement (see Section 5.9) or a Continue Statement (see Section 5.10) within s. If an Iterative Statement is not given a name in this manner, but s (the iterated statement) is a named Begin Statement or Case Statement, then s's name is used as the name of the Iterative Statement.

An "UNTIL" is matched with the innermost unmatched "DO". In the following, the "UNTIL" in "UNTIL e1" is matched with the "DO" in "DO s1", and the last "UNTIL" is matched with the first "DO":

```
DO
    DO s1 UNTIL e1
UNTIL ...
```

If there were no "UNTIL e1" above, a Begin Statement (see Section 5.5) could be used as shown below to match "UNTIL ..." with the first "DO":

```
DOB
    DO s1 END
UNTIL ...
```

5.9. Done Statement

A Done Statement terminates an Iterative Statement and must occur within an Iterative Statement. The form of a Done Statement is:

```
DONE
```

which terminates the innermost enclosing Iterative Statement, regardless of its name (if any), or:

```
DONE c
```

which terminates the innermost Iterative Statement with name c (see Section 5.8), where c is a string constant expression. To terminate an Iterative Statement means that the iterations are stopped, and execution continues with the statement following the Iterative Statement (if any).

```
sum := 0;

DOB ttyWrite("Next integer (type 0 to stop): ");
    IF NOT t := cvi(ttyRead) THEN DONE;
    sum .+ t END;

ttyWrite("The sum is ",sum,eol)
```

Example 5.9-1. Sample Use of "DONE"

A sample use of "DONE" is shown in Example 5.9-1. This Iterative Statement keeps typing "Next integer (type 0 to stop):" to the terminal, adding all the numbers input from the terminal. When the end of the list is signified by an input of "0", the Done Statement terminates the Iterative Statement, and the sum is written to the terminal.

5.10. Continue Statement

A Continue Statement "continues" an Iterative Statement. This means that the current iteration is stopped as if the statement being iterated had completed, and then the usual increments, decrements and tests are applied prior to the next iteration (if any). For example, if the Iterative Statement has an UNTIL-clause, execution continues with the UNTIL-clause test (which may or may not terminate the Iterative Statement).

The form of a Continue Statement is:

```
CONTINUE
```

which continues the innermost enclosing Iterative Statement, regardless of its name (if any), or:

```
CONTINUE c
```

which continues the innermost Iterative Statement with name c, where c is a string constant expression.

An Iterative Statement with a "CONTINUE" in it can look like the one in Example 5.10-1, where s1 and s2 are statements and e is an expression. Whenever the If Statement finds e to be non-Zero, the current iteration terminates (s2 is not executed) and a new iteration is begun.

```
DOB s1; IF e THEN CONTINUE; s2 END
```

Example 5.10-1. Iterative Statement with a Continue Statement

The If Statement may be a more convenient means of controlling execution than the Continue Statement; for example, the Iterative Statement with a Continue Statement of Example 5.10-1 could also be written as in Example 5.10-2.

```
DOB s1; IF NOT e THEN s2 END
```

Example 5.10-2. Iterative Statement with If Statement instead of a Continue Statement

5.11. Empty Statement

The Empty Statement consists of nothing at all.

An Empty Statement is allowed wherever any statement may occur.

For example, a semicolon between the final statement and the "END" of a Begin Statement is not required, since semicolons are used to separate statements, not to terminate them. Nevertheless, a semicolon is accepted there by the compiler; the semicolon indicates that an Empty Statement follows the semicolon. Therefore:

```
BEGIN s1; s2; s3; END
```

has the same effect as:

```
BEGIN s1; s2; s3 END
```

The Empty Statement can also be used, for example, for those cases in which no action is to be taken in a Case Statement, as in Example 5.11-1, in which an Empty Statement follows the "[]" catch-all selector.

```
CASE i OFB
    [0]             j := i;
    [1 TO 4]        BEGIN j := i + k; k .MAX i END;
    [5]             j := (i + k) DIV j;
    []              # Empty Statement: do nothing
    END
```

Example 5.11-1. Example of an Empty Statement

6. Declarations

A declaration presents an identifier to the compiler so that the compiler recognizes the identifier until it reaches the end of the scope of the declaration. A declaration associates with an identifier attributes such as its data type, structure (e.g., array or class), and/or qualifiers that govern its use. All identifiers must be declared before they are referenced, except that class identifiers may be referenced before they are declared under certain circumstances.

The attributes supplied by a declaration do not change; unless the identifier is redeclared, they are associated with the identifier for the remainder of the compilation, and they may not be changed at runtime.

Things that may be declared are simple variables, array variables, procedures, classes, and modules. Any declaration may occur as an "outer declaration", i.e., a declaration between the initial "BEGIN" and final "END" of a module outside of any procedure within the module. In addition, simple variables and array variables may be declared within a procedure after the initial "BEGIN" of a procedure and before the first statement in the procedure; such declarations are called "local declarations", since they are local to the procedure.

Macros are considered to be "defined" rather than declared. A macro definition both declares the macro identifier to the compiler and gives it a value. Macro definitions may appear where ordinary declarations may not. See Chapter 13 for a complete discussion of macros.

Declarations are separated from one another with semicolons. A declaration followed by a statement in a procedure body (see Section 9.1) is separated from the statement with a semicolon. See Example 6-1.

An empty declaration consists of nothing at all and may occur wherever other declarations may occur. Empty declarations permit extra semicolons to appear in program text to make it easier to read.

This chapter describes simple variable declarations, the scope of identifiers, and qualifiers that may be used in declarations to provide additional information about the entities being declared. Array declarations are described in Section 7.1, procedure declarations in Section 9.1, class declarations in Section 8.2, and module declarations in Section 10.2.

```
BEGIN "modNam"

<class declaration>; # outer declaration

; # empty declaration

PROCEDURE p; # (outer) procedure declaration
BEGIN
<variable declaration>; # local variable

# a class, module, or procedure declaration is illegal
# here inside the procedure

<statements of procedure body>
# the scope of the local declarations for p ends here
END;

<module declaration>; # outer declaration

<class declaration>; # outer declaration

<procedure declaration>; # procedure declaration; may
                         # contain local declarations

<variable declaration>; # outer declaration

<procedure declaration> # outer declaration

END "modNam"
```

Example 6-1. Where Declarations May Occur

6.1. Scope of Identifiers

A module's "outer declarations" associate each declared identifier with an entity (variable, procedure, class, or module) that can be accessed throughout the rest of the module. Variables declared among a module's outer declarations are referred to as "outer variables".

Each entity declared in an outer declaration must have a unique name among all entities declared in outer declarations in the same module.

An entity declared with an identifier v inside a procedure cannot be accessed outside the procedure. Such an entity is said to be "local" to the procedure, and variables declared locally are called "local variables". If the identifier v is also declared in the outer declarations, the v accessed within the procedure is the one declared in the procedure. When the end of the procedure body occurs, all locally declared entities such as v "disappear", and if v had been declared in the outer declarations, the outer v is once again visible throughout the rest of the module.

Each entity declared in a local declaration must have a unique name within the procedure.

6.2. Simple Variable Declarations

A simple variable declaration declares one or more variables. The values of the variables may be used during program execution as governed by their data type. The general form of a simple declaration is:

```
type v1, ..., vn
```

where the vi are identifiers, and type is "BOOLEAN", "INTEGER", "LONG INTEGER", "REAL", "LONG REAL", "BITS", "LONG BITS", "STRING", "POINTER", "ADDRESS", or "CHARADR". "ADDRESS" and "POINTER" may be followed by a parenthesized class name as described in Section 8.4.

For example, "INTEGER i,t,num" declares three variables (i, t, and num) that may be assigned integer values.

The qualifier "OWN" may precede a variable declaration, in which case the effect is as described in Section 6.4.

6.3. Qualifiers

A qualifier is used in a declaration to provide additional information about the entity being declared.

If any qualifiers are used in a declaration, they precede all other parts of the declaration. When more than one qualifier is used, the order of the qualifiers themselves is unimportant.

The "OWN" qualifier (described below) may be used only in simple variable and array declarations.

The following qualifiers may be used only in procedure declarations: "FORWARD", "INITIAL", "FINAL", "GENERIC", "COMPILETIME", "INLINE", "SPECIAL", and "$ALWAYS". They are described in Section 9.8.

The following qualifiers may be used only in procedure parameter declarations: "USES", "PRODUCES", "MODIFIES", "OPTIONAL", and "REPEATABLE". They are described in Section 9.5.

6.4. "OWN" Qualifier

An "own variable" is a variable of which the value is retained until the data section of the module in which it is declared is deallocated (see Section 10.6). All outer variables are therefore own variables. The values of local variables declared without the "OWN" qualifier are lost when the associated invocation of the procedure in which they are declared is terminated.

A local variable declared with the "OWN" qualifier retains its value from execution to execution of the procedure, like an outer variable, but its identifier may not be used in the source text outside the procedure in which it is declared. Such local variables are referred to as "local own variables".

Own variables are initialized to Zero when the data section they are in is allocated. Local variables are not automatically initialized; use of their values before they are explicitly initialized has undefined effects.

```
PROCEDURE p;
BEGIN
OWN INTEGER n;
n .+ 1;
...
END
```

Example 6.4-1. Sample Use of a Local Own Variable

In Example 6.4-1, n counts how many times the procedure p has been called. If n's declaration were not qualified with "OWN", its value would be lost whenever the procedure was exited. n could have been declared outside the procedure, but that would obscure the fact that it is used only within the procedure.

Declaring an outer variable with the "OWN" qualifier is legal but has no effect on the way the variable is treated.

7. Arrays

An array is a collection of values, or "elements", all of the same data type, which are accessed by subscripts (described below).

MAINSAIL arrays differ from similar data structures in many other languages in that they must be explicitly allocated at runtime, as described in Section 7.2.

7.1. Array Declarations

The form of an array declaration is:

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

or:

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

where type is the data type of the arrays, li and ui specify the lower and upper bounds, respectively, of the ith dimension, and the vi are the identifiers of the arrays being declared. The first form above declares a "short array", the second a "long array".

Each li or ui is either a (long) integer constant expression, with li less than or equal to ui, or an asterisk ("*"). li and ui must be integers, not long integers, if the array is a short array. If one is a long integer, both must be long integers. The asterisk indicates that the bound is not known at the point of declaration. A single "*" may be used in place of "* TO *".

The data type and/or the parenthesized bounds list may be omitted from the array declaration, in which case the array cannot be used for element access. Such a "typeless" or "dimensionless" array may be passed as a parameter (see Section 9.7) or be assigned to or compared with some other array (see Sections 7.7 and 7.8).

MAINSAIL supports arrays of up to three dimensions. The number of dimensions is the number of "bound pairs" specified in parentheses after the word "ARRAY" in the array declaration.

"NULLARRAY" specifies the Zero array. It is typeless and dimensionless.

7.2. Array Allocation

It is the programmer's responsibility to allocate an array before an attempt is made to access its elements. This is accomplished with the system procedure "new". The general form of a call to new for array allocation is:

```
new(v,l1,u1,...,ln,un)
```

where v is the array variable to be allocated, and li and ui are (long) integer expressions for the lower and upper bounds of the ith dimension. new clears (sets to Zero) all the elements of the newly allocated array.

Any bound declared as a constant may be omitted from new (in which case all remaining arguments must also be omitted) as long as all subsequent bounds were also declared as constants; the compiler fills in the missing bounds from the information given in the declaration. The compiler issues an error message if some bound is declared as a constant, but the corresponding argument to new is specified as some other value. See Example 7.2-1.

An array may be allocated any number of times, though usually it is allocated just once. Each call to new replaces the old array; no elements are copied. A one-dimensional array's upper bound may be changed during program execution by use of the system procedure newUpperBound, which does copy the values to the reallocated array. The system procedure "copy" may be used to copy elements from one array to another.

7.3. Array Disposal

The system procedure "dispose" is used to deallocate arrays. "dispose(arr1)" allows the storage associated with arr1 to be immediately reused. It is not necessary to dispose an array explicitly, since an array that becomes inaccessible is automatically collected by the MAINSAIL garbage collector; however, disposing of arrays no longer in use may improve program performance. The programmer must not use a disposed array in any way (unless it has been re-allocated with new).

7.4. Array Initialization

The Init Statement may be used to initialize an array. The array must be allocated with new before it is initialized. Untyped arrays cannot be initialized with the Init Statement.

The form of the Init Statement is:

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

```
If arr1 is declared as:

    INTEGER ARRAY(1 TO *) arr1

and n is an integer expression, then
these calls to new are legal:

    new(arr1,1,10)
    new(arr1,1,n)

and these are illegal:

    new(arr1,n,20)
    new(arr1)

But if arr1 were declared:

    INTEGER ARRAY(1 TO 15) arr1

then:

    new(arr1)

would be legal and equivalent to:

    new(arr1,1,15)
```

Example 7.2-1. Specifying Array Bounds to the Procedure "new"

where v is an array variable and the ci are initialization specifiers. The simplest form of initialization specifier is a constant expression of v's data type. The ith initialization value (after application of any replications, as described below) initializes the ith element of v. See Example 7.4-1.

```
STRING ARRAY(1 TO 3) cmds;

new(cmds); INIT cmds ("view","clear","next");
```

Example 7.4-1. Init Statement for a One-Dimensional Array

The programmer is responsible for ensuring that multidimensional arrays are properly initialized. Arrays are stored with the last dimensions varying most rapidly. For example, a two-dimensional array is stored by rows (first row immediately followed by second row, and so forth). See Example 7.4-2. A two-dimensional array is initialized in Example 7.4-3. Example 7.4-4 shows what the initialized array arr3 of Example 7.4-3 would look like when viewed as a matrix.

```
An array declared as

type ARRAY(1 TO 3, 1 TO 4) a

has its elements stored in the order

a[1,1], a[1,2], a[1,3], a[1,4], a[2,1], a[2,2],
a[2,3], a[2,4], a[3,1], a[3,2], a[3,3], a[3,4].
```

Example 7.4-2. How Arrays Are Stored

```
INTEGER ARRAY(1 TO 3, 1 TO 4) arr3;

new(arr3); INIT arr3 (2,8,7,5,3,9,8,7,1,3,5,7);
```

Example 7.4-3. Init Statement for a Two-Dimensional Array

```
          col 1   col 2   col 3   col 4
        +-------------------------------+
row 1   |   2       8       7       5   |
row 2   |   3       9       8       7   |
row 3   |   1       3       5       7   |
        +-------------------------------+
```

Example 7.4-4. Array arr3 as a Matrix

An error occurs if there are more initialization values than elements of v. There may be fewer initialization values than elements of v, in which case any elements for which an initial value is not specified are set to the proper Zero value.

An array may be initialized with the Init Statement many times. The Init Statement is executed each time it is encountered.

An initialization specifier may consist of a bracketed integer constant expression ("replication") followed by an initialization value. The replication specifies the number of consecutive elements the initialization value is to initialize. If the replication is less than or equal to zero, the initialization value is ignored. See Example 7.4-5.

```
INTEGER ARRAY(1 TO 4, 1 TO 2) arr4;

new(arr4);
INIT arr4 ([3] 8, [2] 7, 9, [2] 6)

would initialize arr4 as shown below:

              col 1   col 2
            +---------------+
    row 1   |   8       8   |
    row 2   |   8       7   |
    row 3   |   7       9   |
    row 4   |   6       6   |
            +---------------+
```

Example 7.4-5. Use of Replications

A garbage collection may occur during the execution of an Init Statement.

It is common to allocate and initialize an array in the initial procedure (see Section 10.9) and deallocate it in the final procedure (see Section 10.10).

7.5. Accessing an Array Element

An array element is accessed with a "subscripted variable", i.e., an array variable or parenthesized expression followed by a bracketed list of (long) integer expressions. There is one subscript for each dimension of the array, and the subscripts are separated by commas, e.g., "a[e1]", "a[e1,e2]", "a[e1,e2,e3]".

All the subscripts of a short array must be integers. The subscripts of a long array may be integers or long integers and need not be the same data type as used for the corresponding bounds in the declaration.

To refer to the eighth element of a one-dimensional array "a" with a lower bound of one, the subscripted variable "a[8]" is used ("a[8L]" may be used instead if a is a long array). If a two-dimensional array "b" with both lower bounds equal to one is viewed as a matrix, "b[8,3]" is the element in the eighth row and third column.

Each subscript must be within the bounds declared for its dimension (see Section 7.1). If checking is in effect (see Section 14.3), an error message is issued at runtime if, for any subscript, this is not the case.

An array must be allocated before an attempt is made to access its elements (or pseudo-fields; see Section 7.10). If checking is in effect an error message is issued at runtime if the subscripted variable has not been allocated.

7.6. Clearing an Array

The system procedure "clear" can clear (set to Zero) any number of elements of an array. For example, if an array called arr1 has 50 elements, then "clear(arr1)" clears the entire array, and "clear(arr1,20)" clears the first 20 elements of arr1.

An array is automatically cleared when it is allocated by new.

7.7. Array Assignment

One array may be assigned to another if they are assignment compatible (see Section 4.9), i.e., if they are of the same data type (except that either could be untyped) and dimension (except that either could be dimensionless), and corresponding constant bounds are the same. A short array may be assigned to a long array, but a long array may not be assigned to a short array. Array assignment does not copy elements. Instead, both arrays are made to point to the same data structure, and so refer to the same elements.

The conversion procedure "cvAry(a,b)" converts a long array a to a short array b. The effect is undefined if a does not satisfy the short-array rule of Section 7.9. The conversion is purely syntactic; no elements are copied and no storage is allocated, so a and be continue to reference the same elements.

Table 7.7-1 shows the rules for short and long array parameters. The rules are derived by viewing an argument arg passed to a parameter parm as an assignment "parm := arg" if parm is a uses parameter, "arg := parm" if parm is a produces parameter, and both if parm is a modifies parameter.

An array is really a pointer to a data structure, and can be manipulated as a pointer. Information in the declaration of an array such as the type, dimensions, and bounds is necessary

Argument	Parameter	When Allowed
short array	short array	always
short array	long array	uses parameter
long array	short array	produces parameter
long array	long array	always

Table 7.7-1. Array Arguments and Parameters

only if the array is to be allocated, or if elements of the array are to be accessed, in the scope of the declaration.

It is the programmer's responsibility to ensure that an array assignment makes sense. It is possible to write syntactically correct array assignments that are logically invalid and therefore have undefined effects.

```
ARRAY arr1;
INTEGER ARRAY(1 TO 6) arr3;
INTEGER ARRAY(1 TO 2,1 TO 3) arr4;
```

Example 7.7-2. Array Declarations

In Example 7.7-2, arr1 is assignment compatible with both arr3 and arr4, but arr3 and arr4 are not assignment compatible with each other, since they have different dimensions. The following assignments are accepted by the compiler since each assignment involves assignment compatible arrays:

```
arr1 := arr3; arr4 := arr1
```

The effect is the same as assigning arr3 to arr4, which would be invalid since arr3 and arr4 are not assignment compatible. The effect of the statements above is therefore undefined. In particular, this approach does not allow the same array to be accessed as both a one- and a two-dimensional array, since the data structures to which the array pointers point are different.

7.8. Array Comparison

Two arrays may be compared with the relational operators "=" and "NEQ". Short arrays may be compared with long arrays.

The arrays are considered equal if and only if they are the same pointer, and thus share the same elements. In other words, an array comparison compares just the pointers. To check whether two distinct arrays have the same element values, use an Iterative Statement (Section 5.8) to compare corresponding elements one by one.

7.9. The Short-Array Rule

The short-array rule is used to determine whether a particular set of array bounds is allowed for a short array, or whether a long array must be declared instead. The rule is determined by the manner in which subscript calculations are performed, and is independent of the size of the elements of the array. The relevant parts of the subscript calculations for multidimensional arrays are shown in Table 7.9-1. The calculation is performed from left to right; i.e., first i is multiplied by e2, then j is added, then the result is multiplied by e3, and finally k is added.

```
a[i,j]:      (i * e2) + j

a[i,j,k]:    ((i * e2) + j) * e3 + k

(ei is the number of elements in dimension i)
```

Table 7.9-1. Multidimensional Subscript Calculation

The short-array rule, which is used to determine whether an array can be declared as a short array, is given in Table 7.9-2.

```
The bounds must be in the range -32767 to 32767,
inclusive.

The subscript calculation must not overflow the range
-32767 to 32767, inclusive, at any step for any valid
subscripts.
```

Table 7.9-2. Short-Array Rule

The short-array rule is easily applied by first using all lower bounds in place of i, j, and k in Table 7.9-1, and then using all upper bounds. If neither calculation overflows the range, then the array may be declared as a short array. If none of the bounds is negative, then only the upper bound calculation need be performed.

An error message is given when the short-array rule is violated:

- At compiletime, when a short array is declared, based on any constant bounds.
- At runtime, when a short array is allocated with the MAINSAIL system procedure new or newUpperBound.

The long-array rule states that a long array must satisfy the same rules as the short array except that the range -2147483647 to 2147483647, inclusive, is used in place of -32767 to 32767, inclusive. This rule is not enforced; violations produce undefined results.

If an array variable is made not to satisfy the rule implied by its declaration, access to its elements is undefined, since it may cause (possibly undetected) overflow during the subscript calculation, with indeterminate results.

Short arrays should be used whenever possible due to the potential performance penalty incurred by long arrays on some machines, especially for multidimensional arrays.

Some examples of the application of the short-array rule are given in Example 7.9-3.

```
Bounds                          Short-Array Rule          Short OK
1 TO 10000,1 TO 3               10000 * 3 + 3 = 30003       yes
1 TO 10000,1 TO 4               10000 * 4 ... (overflow)    no
1 TO 180,1 TO 180               180 * 180 + 180 = 32580     yes
1 TO 181,1 TO 181               181 * 181 + 181 = 32942     no
1 TO 10000,2001 TO 2003         10000 * 3 + 2003 = 32003    yes
1 TO 10000,3001 TO 3003         10000 * 3 + 3003 = 33003    no
-1000 TO -1,-32 TO -1           -1000 * 32 + -32 = -32032   yes
-1000 TO -1,-832 TO -801        -1000 * 32 + -832 = -32832  no
-10000 TO -1,301 TO 304         -10000 * 4 ... (overflow)   no
1 TO 31,1 TO 31,1 TO 31         (31*31+31)*31 + 31 = 30783  yes
1 TO 32,1 TO 32,1 TO 32         (32*32+32)*32 + 32 = 33824  no
```

Example 7.9-3. Examples of the Short-Array Rule

7.10. Array Pseudo-Fields

The bounds and name of an array are available as "pseudo-fields" of the array. Syntactically, the pseudo-fields of an array a are accessed by means of a "pseudo-field variable" of the form "a.<pseudo-field name>". The pseudo-fields available are shown in Table 7.10-1. The value of

the integer pseudo-fields is undefined if the corresponding long integer pseudo-field cannot be represented as an integer.

Reference to the pseudo-fields representing second- and third-dimensional bounds generates a compiletime error message for a one-dimensional array, and reference to the pseudo-fields representing third-dimensional bounds generates an error for a two-dimensional array. Reference to any bound pseudo-field generates an error for a dimensionless array.

```
Field   Description
name    string name of the array
lb1     integer lower bound of 1st dimension
ub1     integer upper bound of 1st dimension
lb2     integer lower bound of 2nd dimension (if any)
ub2     integer upper bound of 2nd dimension (if any)
lb3     integer lower bound of 3rd dimension (if any)
ub3     integer upper bound of 3rd dimension (if any)
$lb1    long integer lower bound of 1st dimension
$ub1    long integer upper bound of 1st dimension
$lb2    long integer lower bound of 2nd dimension (if any)
$ub2    long integer upper bound of 2nd dimension (if any)
$lb3    long integer lower bound of 3rd dimension (if any)
$ub3    long integer upper bound of 3rd dimension (if any)
$arrayType
        type code of array's data type
$dimension
        number of dimensions of array
```

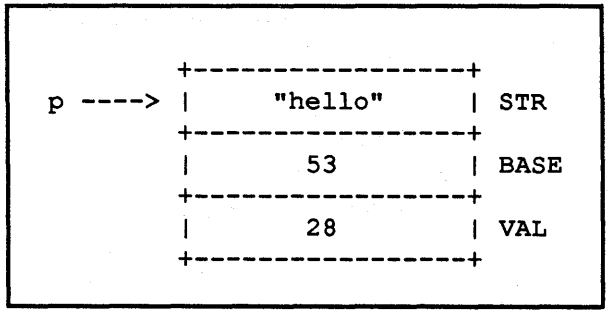
Table 7.10-1. Array Pseudo-Fields

8. Classes and Records

8.1. Records

A record is a data structure of which the components (called "fields") may be of differing data types and are accessed by field names.

For example, a record with three fields, a string named "str", an integer named "base", and another integer named "val", may be imagined as three adjacent boxes, the first holding the value of field str, the second holding the value of base, and the third holding the value of val. If str = "hello", base = 53, and val = 28, then the record may be picured as shown in Example 8.1-1.



Example 8.1-1. A Record with Three Fields

A field of a record is accessed by means of a "field variable" (as described in Section 8.6), which is a pointer to the record followed by a period and the field name. If p points to the record described in Example 8.1-1, then "p.str", "p.base", and "p.val" have the values shown in Example 8.1-2.

```
p.str = "hello"
p.base = 53
p.val = 28
```

Example 8.1-2. Field Variables

Records are not created at compiletime. Classes may be created at compiletime or at runtime, and function as templates for records, which must be allocated at runtime. Pointers are used to access records once they have been created.

8.1.1. The Layout of Fields within a Record

A knowledge of the order in which fields of a MAINSAIL record are stored is necessary in order to pass a classified pointer or address to a foreign language (see the description of the Foreign Language Interface in the "MAINSAIL Compiler User's Guide"), since the fields of the foreign record must be located at the same offsets from the start of the record as the corresponding MAINSAIL fields. The order of the fields of a MAINSAIL record is, however, subject to change, and code that depends upon this order should be avoided in contexts other than the passing of records to a foreign language.

At present, consecutively declared fields of a class are stored in consecutive memory locations within each record of that class. Each field occupies exactly the number of storage units given by "size(typeCode)", where typeCode is the integer type code for the field's data type; no padding or packing is done. For example, if a class is declared as:

```
CLASS xyz (
    INTEGER i;
    POINTER(xyz) p;
    LONG INTEGER li;
);
```

and, on the machine where the record is stored:

```
size(integerCode) = 2
size(pointerCode) = 4
size(longIntegerCode) = 4
```

then:

```
the field  i is stored at offset 0 from the start of the record
the field  p is stored at offset 2 from the start of the record
the field li is stored at offset 6 from the start of the record
```

XIDAK reserves the right to change this layout of record fields.

8.2. Classes

A class describes records, data sections, or storage templates for accessing regions of static memory. Only record classes and storage templates are described here; the use of classes for data sections is described in Section 10.4.

Classes may be created at runtime (with the system procedure $createClassDscr), but are usually created by compiletime declarations. The most common form of a class declaration is:

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

where v is an identifier for the name of the class, and the field declarations are separated with semicolons. The field declarations have the form of simple or array variable declarations or empty declarations, as described in Chapter 6. An example of a class declaration is given in Example 8.2-1.

```
CLASS mix (STRING s; INTEGER val,i; STRING t)

declares a class called "mix" that has four fields:
a string, followed by two integers, followed by another
string.
```

Example 8.2-1. Sample Class Declaration

Class declarations may occur only in the outer declarations of a module (see Chapter 10); i.e., they may not occur within procedures.

The fields of a class can be of any data type. The order in which they occur in the class declaration is the same as the address order in which they are stored in records belonging to that class.

A field name must not be the same as the name of another field in the same class or the same as the class name. It may have the same name as a local or outer variable or the field of another class. No confusion can occur, since whenever a field name is used in a field variable, it is preceded by a pointer that determines its class.

A class c can obtain its initial fields from another class pc by means of a declaration of the form:

```
CLASS(pc) c (<declarations of additional fields>)
```

pc is called a "prefix class" of c, as discussed in Section 8.8.

8.3. Record Allocation and Disposal

Any number of new records of a class may be created at runtime by calls to the system procedure "new" or "$createRecord". new initializes to Zero the fields of the record it creates, and returns a pointer to the record. A call to new such as "p := new(c)" creates a record belonging to the class c and assigns the pointer to the newly created record to p.

The system procedure "dispose" disposes of existing records when they are no longer needed. Thus, "dispose(p)" disposes of the record pointed to by p. It is not necessary to dispose records explicitly, since a record that becomes inaccessible (no longer pointed to by any pointer) is automatically collected by the MAINSAIL garbage collector; however, disposing of records no longer in use may improve program performance. The programmer must not use a pointer referencing a disposed record in any way.

8.4. Classified Pointers and Addresses

If a pointer is to be used only to access records of a particular class (the most common case), the name of the class may appear, enclosed in parentheses, following the word "POINTER" in the pointer variable declaration. Pointers so declared are called "classified". For example, "POINTER(list) p" declares a pointer variable p to reference records that belong to the class called "list".

The compiler ensures that classified pointers are not mistakenly used to refer to records of unrelated classes (see Section 8.9).

A pointer declaration can use a class that has not yet been declared (sometimes called a "forward class"). This allows each of two classes to contain a pointer field to the other. For example, the declarations of Example 8.4-1 declare two classes, a and b, each of which has a pointer field for referencing records belonging to the other class. (Note that when a is declared, b has not yet been declared.)

If the fields of a class are not referenced in a module, and the class is not used as the prefix class of another class in the module, then the class does not have to be declared in the module.

"DCL" returns false if given a forward class name if its class declaration has not yet been encountered.

Like pointers, addresses may be classified. Classified addresses are not usually used to refer to records allocated by the system procedure new, but rather to provide a template for storage in scratch space allocated by the programmer with the system procedure newScratch, $newScratchChars, or newPage. Syntactically, a classified address declaration looks like a

```
CLASS a (POINTER(b) ptr; INTEGER i; ...);

CLASS b (POINTER(a) q; STRING str; ...)
```

Example 8.4-1. Classes Referring to Each Other

classified pointer declaration, except that the word "ADDRESS" replaces the word "POINTER". The rules for assignment compatibility that apply to classified pointers also apply to classified addresses. Example 8.4-2 shows the use of a classified address.

```
CLASS c (INTEGER i; BITS b; ADDRESS(c) link);
ADDRESS(c) a,list;
POINTER(dataFile) f;
...
# Create a linked list in scratch space
list := NULLADDRESS;
DOB a := newScratch(size(c));
    read(f,a.i,a.b); a.link := list; list := a;
    ... END;
```

Example 8.4-2. Use of a Classified Address

8.5. Unclassified Pointers and Addresses

An unclassified pointer or address is useful when a pointer may refer to records of different classes or an address to different storage templates at different times. An unclassified pointer or address declaration omits the parenthesized class name following the keyword "POINTER" or "ADDRESS". For example, "POINTER q" declares a pointer variable q that can be used to point to records of any class, since unclassified pointers are considered to be related (see Section 8.9) to all pointers in all classes.

The programmer must be especially careful when using unclassified pointers since class checking is not provided for them; the security of the language can be violated, as in the last statement shown in Example 8.5-1.

The effect of the last two statements in Example 8.5-1 would be the same as that of the previous statement "p1 := p2". That is, they effectively assign a pointer (p2) that can access

```
POINTER(c1) p1;     # assume class c1 is not related to
POINTER(c2) p2;     # class c2
POINTER p;

p1 := p2;           # compiler reports a class
                    # compatibility error

p := p2;
p1 := p;            # invalid, but compiler does not
                    # report an error
```

Example 8.5-1. Use of an Unclassified Pointer

records belonging to class c2 to a pointer (p1) that can access records belonging to an unrelated class c1. The consequences of using a pointer that has been made to point to a class unrelated to its declaration are undefined. It is the programmer's responsibility to avoid such use of unclassified pointers.

Unclassified addresses are more common than unclassified pointers, since a variety of system procedures operate on unclassfied addresses.

8.6. Accessing Fields of Records and Storage Templates

A field of a record or storage template is accessed by means of a "field variable", which has the form "p.f" where p is called the "base part" and f the "field part". The base and field parts are separated by a period, which need not be immediately adjacent to either part; e.g., "p.f" could be written "p . f".

The base part must be a classified pointer or address of which the associated class contains a field named by the field part. The base part may be a simple pointer or address variable, an element of a pointer or address array, another field variable, a procedure call, or a parenthesized pointer or address expression. As a special case, it may be an array variable, in which case the field part must be one of the special pseudo-fields described in Section 7.10.

The field part is the name of a field of the record pointed to by the pointer base part or the storage template pointed to by the address base part. The field name must have been declared to be a field of the class associated with the base part.

Data section fields are accessed by means of field variables, as with records; however, a data section field name may be a procedure name if the class contains procedure fields (see Section 10.3).

```
CLASS c2 (INTEGER num; STRING name; BOOLEAN fin);
CLASS c3 (POINTER(c2) p);
CLASS c4 (POINTER(c3) PROCEDURE proc (INTEGER i));

POINTER(c2) p;
POINTER(c3) q;
POINTER(c4) r;
INTEGER t;

...

p := new(c2); # new initializes the fields of the record
              # to Zero, i.e., p.num = 0, p.name = "",
              # and p.fin = FALSE

# Change the fields of the record pointed to by p:
p.num := t; p.name := "MAXIMUM"; p.fin := TRUE;

q := new(c3);
q.p := p;     # This is legal and unambiguous; "q.p" now
              # refers to the same record as "p".

# Change the fields of the record pointed to by p again,
# this time going through q:
q.p.num .+ 4; q.p.name .& " 2"; q.p.fin := FALSE;

...

# Now change both q and p:
p := (q := r.proc(t)).p;
```

Example 8.6-1. The Use of Field Variables

The data type of a field variable is the data type of the field part. The data type of "p.num" in Example 8.6-1 is integer.

The programmer must ensure that the base part is not Zero (i.e., not nullPointer or nullAddress). If checking is in effect (see Section 14.3), code is output for each field variable to generate an error if a base part is nullPointer (no check is made for nullAddress).

When the base part of a field variable is itself a field variable, the base part is evaluated first; i.e., constructs such as "p.f.g.h" are evaluated as "((p.f).g).h".

8.7. Explicit Classes in Field Variables

A class name may be specified explicitly in a field variable by following the base part with a colon followed by the class identifier. That is, if the class "c" has a field "f", and "p" is a pointer or address, then "p:c.f" is a legal field variable. For example, "p:c2.num" could have been used in place of "p.num" in Example 8.6-1. However, since the compiler knows from p's declaration that p is a pointer for referencing records belonging to the class c2, specifying ":c2" is redundant and unnecessary.

Explicitly specifying a class name is required when an unclassified pointer or address is used as the base part of a field variable, since the base part of a field variable must be a classified pointer or address. Specifying a class has the effect of temporarily classifying an unclassified pointer or address.

Sometimes the programmer may want to specify a class different from the class declared for the pointer or address, in which case the pointer or address is used as if it belongs to the specified class. The specified class is usually related (as described in Section 8.9) to the pointer's or address's class. See Section 8.8.1. The effects of overriding a pointer's class are undefined if the field accessed does not exist in the record referenced.

An explicit class may be specified with a pointer or address even if it is not used as a field variable. For example, if a file f is declared as:

```
POINTER(file) f;
```

then it is permissible to say:

```
read(f:textFile,...)
```

to force the generic mechanism to select a textFile form of read.

8.8. 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(prefixClass) id (<declarations of additional fields>)
```

where id is the name of the class being declared (the "prefixed class") and prefixClass is the name of its prefix class. The parenthesized declaration list after id may be omitted if there are no additional fields.

The prefix class contributes its fields to the prefixed class. For example, the declarations:

```
CLASS      c1 (INTEGER i; STRING s);
CLASS(c1)  c2 (REAL r);
```

declare two classes, c1 and c2. c1 does not have any prefix classes. c2, a prefixed class, has a prefix class, c1, and has three fields: an integer i, a string s, and a real r. The first two fields are inherited from c1.

Prefix classes permit several related classes to have their initial fields "abstracted out" into a separate class so that records in each of the related classes can be manipulated by procedures or statements that access just the initial fields of the common prefix class.

A prefix class may itself have a prefix class. For example, given the declarations:

```
CLASS    a (...);
CLASS(a) b (...);
CLASS(b) c (...);
```

a has no prefix classes, a is the only prefix class of b, and c has two prefix classes, a and b. The fields of c are the fields of a followed by the additional fields contributed by b's and c's declarations.

8.8.1. Accessing Prefix Fields

```
Given these declarations:

    CLASS       c1 (INTEGER f);
    CLASS(c1)   c2 (STRING  s);
    POINTER(c1) p1;
    POINTER(c2) p2;

these field variables are valid:
```

Example 8.8.1-1. Prefix Classes and Pointers (continued)

```
    p1.f  p2.f  p2.s

and this is invalid:

    p1.s
```

Example 8.8.1-1. Prefix Classes and Pointers (end)

A pointer or address to a prefixed class may be used without an explicit class to access the fields of its prefix classes. In Example 8.8.1-1, p1 can access fields only in c1, whereas p2 can access fields in both c1 and c2.

If the programmer knows that p1 points to a record that has all of c2's fields, then the field variable "p1:c2.s" can be used to access the field s; thus, the information provided in class declarations may be overridden. Accessing fields not present in the record referenced produces undefined results.

8.9. Related Classes

Two classes are said to be "related" if one is a prefix class of the other or if they are the same class. Two pointers are said to be assignment compatible (see Section 4.9) if their classes are related or if one or both of them are unclassified. See Example 8.9-1.

8.10. "Safe" and "Unsafe" Assignment of Pointers

An assignment of the form "p := q", where p's class is a prefix class of q's, is considered to be a "safe" assignment, but an assignment in the opposite direction, i.e., "q := p", is considered "unsafe". Both are legal since the classes of the pointers are related, but the latter allows the programmer subsequently to write syntactically correct but logically invalid field variables, and the assignment itself may sometimes be logically incorrect. This "loophole" in MAINSAIL covers those cases where "safe" assignments are too restrictive. See Example 8.10-1.

8.11. Alignment of Chunks

```
TEMPORARY FEATURE:  SUBJECT TO CHANGE
```

```
CLASS         a (...);
CLASS(a)     b1 (...);
CLASS(a)     b2 (...);
CLASS(b1)    c  (...);

Each class is related to itself.
In addition,

    a  is related to b1, b2, and c,
    b1 is related to a and c,
    b2 is related to a,
    c  is related to a and b1,
    b1 is not related to b2,
    b2 is not related to c.
```

Example 8.9-1. Related Classes

Chunks (i.e., records, arrays, and data sections) are currently aligned to a multiple of a system-dependent number of storage units, usually at least the size of an address. This means that appropriate arrangement of fields within a record can, on some processors, reduce the access time to some of the fields.

Typical cases include the VAX-11, IBM System/370, and M68000, on which memory is organized into eight-bit bytes, and where the MAINSAIL data types have the sizes shown in Table 8.11-1. All these processors permit a 4-byte quantity to be fetched from an address that is 2-byte-aligned (i.e., they do not require it to be 4-byte-aligned); however, on at least some models of all of these processors, it is slower to fetch a 4-byte quantity from such an address than from one that is 4-byte-aligned.

For example, if a record of the class:

```
CLASS c (
    INTEGER i;
    LONG INTEGER ii;
);
```

is aligned on a 4-byte boundary, then the field ii is fetched from a non-4-byte-aligned address. A better arrangement of c would be:

```
CLASS     a (STRING name; INTEGER num);
CLASS(a)  b (INTEGER val);
CLASS(a)  c (STRING sample);

POINTER(a) pa;
POINTER(b) pb;
POINTER(c) pc;

pa := new(b);              # Legal

pb := pa; pb.val := 0;     # Valid, since pa points to a
                           # record of class b

pa := new(a);              # Also legal, replaces previous
                           # record pointed to by pa

pb := pa; pb.val := 0;     # Invalid; pa points to a record
                           # of class a.  There is no "val"
                           # field in the record.  Execution
                           # of these statements will have
                           # undefined consequences.

pc := new(c);

pa := pc; pb := pa;        # Legal

pb.val := 0;               # Invalid.  The effect is the same
                           # as that of "pc.val := 0", which
                           # the compiler would flag as an
                           # error.
```

Example 8.10-1. Examples of Safe and Unsafe Assignments

```
CLASS c (
    LONG INTEGER ii;
    INTEGER i;
);
```

since ii is now on a 4-byte boundary (i need be aligned only to a 2-byte boundary).

A rule of thumb that ensures optimal field access on all processors for which MAINSAIL has been implemented so far is to declare string and long real fields of a class first (in any order),

Data type	Size
boolean	2 bytes
integer	2 bytes
long integer	4 bytes
real	4 bytes
long real	8 bytes
bits	2 bytes
long bits	4 bytes
string	8 bytes
address	4 bytes
charadr	4 bytes
pointer	4 bytes

Table 8.11-1. Typical Data Type Sizes

followed by long integer, real, long bits, address, charadr, and pointer fields (in any order), and finally boolean, integer, and bits fields (in any order). In the case of prefix classes, it may be convenient to add a padding field if the size of the class is not a multiple of 4 bytes on the typical processors, so that the first new field of a prefixed class is 4-byte-aligned.

The alignment of records and the size of data types are subject to change, so this strategy may not always result in optimal field access. XIDAK is also considering the possibility of introducing padding fields into a record automatically to ensure that all data types are properly aligned; however, it may not prove feasible to add such fields.

9. Procedures

A procedure associates an executable unit (the "procedure body") with an identifier so that later occurrences of the identifier can be used in a procedure call to cause the procedure body to be executed.

9.1. Procedure Declarations

The form of a procedure declaration (header and body) is shown in Table 9.1-1.

```
A procedure with parameters is declared:

    qualifiers type PROCEDURE v
        (declaration list for parameters);
    procedure body

A procedure without parameters may be declared:

    qualifiers type PROCEDURE v;
    procedure body

or:

    qualifiers type PROCEDURE v ();
    procedure body

"type" is omitted if the procedure is untyped, and
"qualifiers" if the procedure has no qualifiers.
```

Table 9.1-1. Format of a Procedure Declaration

In Table 9.1-1, the identifier v is name of the procedure, and the parenthesized list of parameter declarations appears only if the procedure has parameters (see Section 9.4). The type (e.g., "INTEGER") is present if and only if the procedure returns a value. In this case the procedure is said to be a "typed" procedure (see Section 9.3) and must return a result with a Return Statement (see Section 5.4).

A procedure body may have two forms:

1. A statement.

2. The keyword "BEGIN" followed by an optional list of local declarations (see Section 6.1) followed by a list of statements separated by semicolons followed by the keyword "END".

See Example 9.1-2. The di cannot include a procedure declaration; that is, procedures cannot be statically nested. The di may not include class or module declarations either. The "BEGIN"-"END" pair in the second form of procedure may be given a name with a string constant as described in Section 5.5.

For purposes of name scoping, the parameters of a procedure are also considered to be local variables of the procedure. Parameters may be declared to be "uses", "modifies", or "produces"; uses and modifies parameters are initialized by the corresponding argument, and modifies and produces variables set the value of the corresponding argument variable. The initial values of local variables are undefined, except that uses and modifies parameters are initialized by their arguments (see Section 9.5). It is the programmer's responsibility to ensure that local variables and produces parameters are initialized before their values are accessed (for this purpose, a produces parameter is considered to be "accessed" on procedure return (unless it is an omitted optional parameter)); otherwise, the consequences are undefined.

```
A procedure body may look like:

  s

or:

  BEGIN
  d1; ... dm;    # declarations
  s1; ... sn     # statements (";" after sn would be OK)
  END

where the di are declarations and the si are statements.
```

Example 9.1-2. Two Procedure Body Forms

9.2. Procedure Calls

A procedure call causes execution of the procedure body. It can also entail transfer of argument values to and from the procedure, and the transfer of a result value from the procedure.

The form of a procedure call is shown in Table 9.2-1.

```
A procedure with parameters is called with:

    p(e1,...,en)

A procedure without parameters is called with:

    p;

or:

    p();
```

Table 9.2-1. Procedure Call Formats

In Table 9.2-1, p is the procedure to be called and the ei are the arguments. The order of evaluation of the ei is not specified (see Section 9.6). See Section 9.4 for further explanation of arguments. Example 9.2-2 shows sample procedure declarations and calls.

9.3. Typed and Untyped Procedures

Procedures are either "typed" or "untyped". If a data type name does not precede the word "PROCEDURE" in the declaration, then the procedure is said to be "untyped". Untyped procedures do not return values. They may be called only as statements, not in expressions, as described in Sections 4.3 and 5.3.

If a data type name (e.g., "INTEGER") precedes the word "PROCEDURE" in the procedure declaration, then the procedure is said to be "typed". When a typed procedure is called, it must return a value of its declared data type with a Return Statement (see Section 5.4). In the typed procedure shown in Example 9.3-1, p1 returns either 0 or j depending on the value of i at the If Statement.

```
Suppose p2 is declared as:

    PROCEDURE p2 (INTEGER i; STRING str);
    BEGIN
    ...
    END

Then sample calls for p2 are (assuming j is an
integer and s a string):

    p2(j,s); p2(j + 3,"error"); p2(10,s & "xxx")
```

Example 9.2-2. Procedure Declaration and Calls

```
INTEGER PROCEDURE p1;
BEGIN
INTEGER i,j;
...
IF i > 0 THEN RETURN(0);
...
RETURN(j) END
```

Example 9.3-1. Example of a Typed Procedure

Typed procedures may be called either in an expression or as a statement, depending on whether or not the returned value is to be used. A call to a typed procedure in an expression uses the returned value in the expression. When a typed procedure is called as a statement, the returned value is discarded. Typed procedures called as statements are called for the actions they perform rather than the results they return.

Section 5.3 shows a typed procedure that is called either in an expression or as a Procedure Statement, depending on whether or not the returned value is needed.

9.4. Parameters to Procedures

As shown in Table 9.1-1, a parenthesized list of parameter declarations (separated with semicolons) may follow the procedure name in the procedure declaration. The parameter

declarations specify the characteristics of arguments passed to and from the procedure when it is called.

A parameter is either a simple variable (Section 6.2), an array (array parameters are discussed in Section 9.7), a module (Section 10.2), or a class (Section 8.2). The programmer cannot declare procedures with module or class parameters; such parameters are used only by system procedures. The "OWN" qualifier cannot be used in parameter declarations.

An argument specified in a procedure call must be "assignment compatible" with the corresponding parameter in the procedure declaration; i.e., they must have the same data type. See Section 4.9 for a definition of assignment compatibility. See Example 9.4-1.

Except for procedures with parameters declared with the "OPTIONAL" or "REPEATABLE" qualifiers (see Section 9.5), the number of arguments in a procedure must be the same as the number of parameters in the procedure declaration.

```
If the procedure procEx is declared as:

    PROCEDURE procEx (INTEGER i,j; STRING s)

and k and m are integer variables, s1 a string variable,
and r a real variable, then:

    procEx(k,m,s1)
    procEx(1,8,"go")
    procEx(k,k,s1)
    procEx(m,7,s1)

are all legal procedure calls, but:

    procEx(r,m,s1)
    procEx(k,2.7,"go")
    procEx(k,m,r)
    procEx(k)
    procEx(k,m,s1,r)

are all illegal.
```

Example 9.4-1. Parameters and Arguments

9.5. Parameter Qualifiers

There are five parameter qualifiers: "USES", "PRODUCES", "MODIFIES", "OPTIONAL", and "REPEATABLE".

"USES", "PRODUCES", and "MODIFIES" do not affect the use of a parameter within a procedure, but indicate whether the parameter is initialized by the argument ("USES" and "MODIFIES"), and whether the parameter value is transmitted back to the argument upon completion of the procedure ("PRODUCES" and "MODIFIES").

Any parameter (whether uses, modifies, or produces) may be used within a procedure as if it were a normal local variable; i.e., its value may be both examined and modified. The parameter qualifiers specify only whether a parameter is initialized by an argument, and whether its final value is sent back to the argument.

"OPTIONAL" and "REPEATABLE" do not affect the use of a parameter within a procedure, but govern how many arguments may be given for the parameter in the procedure call.

9.5.1. "USES"

A uses parameter is passed the value of the argument; i.e., the argument value initializes the parameter.

The procedure uses the value of the argument, but does not otherwise access the argument. Thus any changes to the parameter (which is a local variable) have no effect on the argument. Parameters declared with no qualifier are uses parameters.

9.5.2. "PRODUCES"

A produces parameter passes its value back to the argument upon return from the procedure, but is not initialized by the argument value. The argument is assigned the value of the parameter upon procedure return. The argument must be a variable, except that if the parameter is declared "OPTIONAL", it can be omitted (see Section 9.5.4). In this case the returned value is discarded.

Since produces parameters are not automatically initialized, they should be assigned a value before they are accessed.

9.5.3. "MODIFIES"

A modifies parameter combines the effect of uses and produces parameters. It is passed the value of the argument, and also passes its value back to the argument upon return from the procedure.

The argument corresponding to a modifies parameter must be either a variable or, if the parameter is declared "OPTIONAL", an omitted argument. If the argument is omitted, the Zero of the appropriate data type is passed to the procedure (if the parameter is uses or modifies), and the returned value is discarded.

```
A procedure "proc" that uses an integer value,
produces a real value, modifies the value of a string
argument and returns a bits would have the header:

BITS PROCEDURE proc
    (INTEGER i; PRODUCES REAL r; MODIFIES STRING s)

where it is understood that i is a uses parameter,
since "USES" is the default.
```

Example 9.5.3-1. Example Using Parameter Qualifiers

9.5.4. "OPTIONAL"

A parameter may be qualified with "OPTIONAL" to indicate that its argument may be omitted in procedure calls. All parameters following an optional parameter must also be declared "OPTIONAL". If the corresponding argument is omitted, the compiler substitutes the Zero value of the appropriate data type instead. See Example 9.5.4-1.

An optional argument may be omitted only if all subsequent arguments are also omitted. If all of a procedure's parameters are optional and all the arguments are omitted in a procedure call, the argument parentheses may also be omitted as if the procedure had no declared parameters. See Example 9.5.4-2.

9.5.5. "REPEATABLE"

The last one or more parameters of an untyped procedure may be qualified with "REPEATABLE" to indicate that a call may give more than one set of arguments ("repeated

```
Given the declaration:

   PROCEDURE p (INTEGER v1; OPTIONAL INTEGER v2)

the call:                p(e)

is treated as:           p(e,0)
```

Example 9.5.4-1. Use of Optional Argument

```
Given the declaration:

   PROCEDURE p2 (OPTIONAL INTEGER i; OPTIONAL REAL r)

the call:                p2

is treated as:           p2(0,0.)
```

Example 9.5.4-2. Use of Optional Arguments, Omitting All Arguments

arguments") for the repeatable parameters. The compiler treats a call to a procedure with repeatable parameters as a series of calls, each expansion call with one set of the repeated arguments passed for the repeatable parameters.

All parameters following a repeatable parameter must also be repeatable. If the last n parameters of a procedure are declared repeatable (but not optional), then each expansion call to the procedure passes n more arguments for the repeatable parameters, in addition to the non-repeated arguments that are evaluated once (except for simple variables passed as modifies or produces arguments) and passed each time.

Before expanding the call to a procedure with repeatable parameters, non-variable non-repeated arguments are evaluated and stored as temporaries. The stored values of these arguments are used on each expansion call, so that such arguments are evaluated only once. Any variable non-repeated arguments may or may not be re-evaluated on each expansion call. In particular, the use of non-simple variables as produces or modifies non-repeated arguments has undefined results. Simple variables as produces or modifies non-repeated arguments are re-evaluated on each expansion call. Each repeated argument is evaluated immediately before the expansion call on which it is used.

Optional repeatable arguments are governed by the rule that an optional argument is assumed omitted only if all arguments have been used; see Example 9.5.5-2.

Repeatable parameters are forbidden in declarations of typed procedures.

```
With the declaration:

   PROCEDURE p (INTEGER v1; REPEATABLE REAL v2)

the call:                p(e,e1, ..., en)

is treated as:           p(e,e1); ... p(e,en)

or, if e is evaluated only once, as:
                         t := e; p(t,e1); ... p(t,en)

With the declaration:

    PROCEDURE drawLine (REPEATABLE INTEGER x1,y1,x2,y2);
    # draw line from (x1,y1) to (x2,y2)

the call:

    drawLine(1,2,3,4,100,200,300,400,i,j,k,l)

draws three lines.

If a procedure has the header:

    PROCEDURE foo
        (POINTER(device) gp; REPEATABLE REAL x,y);

the call:

    foo(gp,x1,y1,x2,y2,x3,y3);

is equivalent to:

    foo(gp,x1,y1); foo(gp,x2,y2); foo(gp,x3,y3);

where gp may be evaluated just once.
```

Example 9.5.5-1. Use of Repeatable Argument

```
Given the declaration:

   PROCEDURE p
       (REPEATABLE INTEGER i; OPTIONAL REPEATABLE REAL r);

the calls:

    p(1,2.0);
    p(3,4.0,5,6.0);
    p(7,8.0,9);

are legal, but:

    p(10,11,12.0);

is not.
```

Example 9.5.5-2. Interaction of "OPTIONAL" and "REPEATABLE" Qualifiers

9.6. Order of Argument Evaluation

The order in which the arguments to modifies and produces parameters and the procedure result are assigned values upon return from a procedure call is unspecified. Calls in which the same variable is used for more than one such assigned value have undefined results. See Example 9.6-1.

```
If v is passed for a modifies or
produces parameter, then

    v := p(...,v,...)

and

    p(...,v,...,v,...)

assign undefined values to v.
```

Example 9.6-1. Calls of Which the Results Are Not Well-Defined

9.7. Array Parameters

An array parameter is a reference to the argument array; i.e., the argument array variable (a pointer) is assigned to the parameter. The assignment does not copy the elements; the array parameter points to the same array elements as the argument array. Any changes made to the elements of the parameter within the procedure are made to the argument array elements.

The "USES", "PRODUCES", and "MODIFIES" qualifiers apply to the array variable itself rather than to the elements of the array. For example, a modifies array parameter is initialized to point to the argument array and upon return the array parameter is assigned to the argument array. The procedure may have assigned a different array to the parameter.

In Example 9.7-1, the procedure "aryPrint" prints the values of the first n (where n is a parameter) elements of the one-dimensional integer array (with a lower bound of 1) specified as its first argument.

```
PROCEDURE aryPrint (INTEGER ARRAY(1 TO *) a; INTEGER n);
BEGIN
INTEGER i;
FOR i := 1 UPTO n DO write(logFile,a[i],eol)
END
```

Example 9.7-1. Use of an Array Parameter

The array parameter declaration "INTEGER ARRAY(1 TO *) a" signifies that the first argument of any call to the procedure aryPrint must be a one-dimensional integer array of which the lower bound is declared as 1. The assignment compatability checking done by the compiler (see Section 4.9) attempts to ensure that each argument conforms to these requirements.

In Example 9.7-2, the procedure doubleSize increases the upper bound of the array ary by a factor of two. ary must be a modifies parameter; otherwise, the argument to doubleSize would be unaffected.

```
PROCEDURE doubleSize (MODIFIES ARRAY(1 TO *) ary);
newUpperBound(ary,2 * ary.ub1);
```

Example 9.7-2. A Modifies Array Parameter

9.8. Procedure Qualifiers

One or more procedure qualifiers may be used in a procedure declaration to provide additional information about the procedure being declared. The qualifiers must precede the data type name (in a typed procedure) or the keyword "PROCEDURE" (in an untyped procedure). The order of the qualifiers is unimportant, except as noted.

The procedure qualifiers are "FORWARD", "INITIAL", "FINAL", "GENERIC", "COMPILETIME", "INLINE", "SPECIAL", "$BUILTIN", and "$ALWAYS".

"COMPILETIME", "$BUILTIN", and "SPECIAL" apply only to system procedures; the programmer cannot use them. They are described in Sections 1.1, 1.2, and 1.3 of part II of the "MAINSAIL Language Manual".

"INITIAL" and "FINAL" are explained in Sections 10.9 and 10.10.

"FORWARD" and "GENERIC" are described in Sections 9.10 and 9.12.

"$ALWAYS" and "INLINE" are explained in Section 9.11.

9.9. Recursion

Any procedure may be invoked recursively, i.e., called again before it has returned from a previous call. Each invocation creates a new copy of the non-own local variables; i.e., any existing non-own locals are not affected.

Recursion may occur when a procedure calls itself, or when it calls another procedure that causes it to be called. A recursive calculation of Fibonacci numbers appears in Example 9.9-1. The procedure fibonacci calls itself. Mutual recursion is shown in Example 9.10.1-1.

```
LONG INTEGER PROCEDURE fibonacci (LONG INTEGER i);
RETURN(
    IF i LEQ 1L THEN i
    EL fibonacci(i - 2L) + fibonacci(i - 1L));
```

Example 9.9-1. A Recursive Calculation of Fibonacci Numbers

Recursive procedures that are called too many times before returning may cause a stack overflow. See Section 9.13. In particular, a procedure that calls itself unconditionally on each

invocation produces an "infinite recursion" (see Example 9.9-2); calling such a procedure always results in a stack overflow.

No special qualifier is required in MAINSAIL to allow a procedure to be invoked recursively.

```
PROCEDURE p;
p;
```

Example 9.9-2. Infinite Recursion

9.10. Forward Procedures

The keyword "FORWARD" serves two functions:

1. It permits a procedure to be called before its body has been seen. This makes possible mutual recursion between procedures, or a convenient ordering of procedures if the programmer does not want to remember which procedures were declared before which.

2. It can indicate the source file in which the body of a procedure may be found.

9.10.1. "FORWARD" for Mutual Recursion

A procedure must be declared before it can be called. If two procedures call each other, one of the procedures must first be given a "forward" procedure declaration, which is like a normal procedure declaration except that it is qualified with "FORWARD", and just the procedure header (not the body) is given. Later, the procedure is declared as usual (the "FORWARD" qualifier is not used, and a body is given); the compiler automatically figures out that the later declaration redeclares the previous forward procedure. The type of the result and parameter types and qualifiers must be the same in the forward declaration as in the body declaration, but the parameter names may differ. Calls to the procedure may appear at any point after the forward declaration. See Example 9.10.1-1.

An interface procedure declaration (see Section 10.2) for the current module serves as a forward declaration of the procedure. There is no need to provide a separate forward declaration in the event that calls are made to an interface procedure before its body has been declared, provided that the module declaration has been seen.

```
FORWARD PROCEDURE p (INTEGER i);

PROCEDURE q (REAL x);
BEGIN
...
p(1);
...
END;

PROCEDURE p (INTEGER i);
BEGIN
...
q(1.4);
...
END;
```

Example 9.10.1-1. Example of Forward Procedure

If a forward procedure is not called (i.e., the compiler does not encounter it in a call), the forward declaration is ignored; i.e., no error message is issued if the body of the procedure declared forward is never encountered. This rule does not apply to interface procedures, since they may be called from outside the module, and must therefore always be given a body.

9.10.2. "FORWARD" for Source Library Declarations

A related use of "FORWARD" is to indicate the file in which the procedure (header and body) is declared. In this case the form is:

```
FORWARD(c) PROCEDURE p ( ... )
```

where c is a string constant expression giving the name of the file that contains p's full declaration. If at the end of compilation the procedure has been called, but no body has been declared for it, the compiler automatically compiles the indicated file, expecting to encounter p's declaration; an error occurs if it does not.

For example, if the following forward procedure declaration is given:

```
FORWARD("lib") PROCEDURE p ( ... )
```

and p is called, but a body is not declared for it, the compiler compiles the file named "lib" in order to get p's declaration. "lib" may employ the compiletime pseudo-procedures

"NEEDBODY" and "NEEDANYBODIES" (see Section 14.19) and conditional compilation (Section 14.10) to ensure that the compiler "sees" only those procedure bodies that are actually needed. The compiler repeatedly compiles all files declared to contain forward procedures until either all necessary procedure bodies have been obtained, or it detects an infinite loop.

The compiletime system procedure $thisFileName is useful for declaring forward procedures in the current file.

9.11. Inline Procedures

A procedure call may be implemented either by passing its arguments to the procedure and calling the procedure, or by expanding the procedure call inline. Calls implemented by the first method are called "closed" procedure calls; calls implemented by the second, "inline" calls. The semantics of using an inline call are the same as if a closed call had been used instead.

Using an inline call has the advantage of eliminating the overhead of a call and return. Often the arguments to the call can be substituted directly for their corresponding parameters, which eliminates the overhead of copying procedure arguments. Also, constant folding can sometimes be done on constant arguments that are directly substituted for their parameters, for further improvements in the generated code (such improvements may be made even if the "OPTIMIZE" compiler option is not specified).

The main disadvantage of an inline call is that it may consume more code space if it is expanded many times. Inline procedures nested several layers deep may make a module so large that the MAINSAIL compiler runs out of memory attempting to compile it. Also, the debugger is unable to jump into a procedure at an inline call, and it is unable to set breakpoints, single-step, or examine local variables within a procedure with no closed body. If all calls to a procedure are expanded inline, then no closed body is generated for the procedure.

The programmer controls which calls are done inline by means of the keyword "INLINE". "INLINE" may be used either in a procedure declaration, in which case all calls to that procedure are affected, or at a given call, in which case only that call is affected. The keywords used at a given call to a procedure override the keywords used in the procedure's declaration, but only for that call.

"$ALWAYS INLINE" (the two keywords used together) indicates that calls to a procedure should be inline, regardless of what compiler options are in effect. "$ALWAYS", if present, must immediately precede "INLINE". The macro "$ALWAYSINLINE" (one word) is defined for convenience to be "$ALWAYS INLINE".

A procedure declared "INLINE" but not "$ALWAYSINLINE" indicates that calls to the procedure are to be inline unless the procedure is compiled debuggable, in which case the calls should be closed so that the procedure can be debugged. A procedure declared "$ALWAYSINLINE" is compiled inline even if compiled debuggable.

If a procedure is not declared "INLINE", calls to it are closed.

If "$ALWAYSINLINE" or "INLINE" is used in a procedure declaration, it appears before the procedure header, where other procedure qualifiers would appear. The order of the procedure qualifiers is unimportant, except that "$ALWAYS", if present, must appear immediately before "INLINE". Examples of the use of "$ALWAYSINLINE" and "INLINE" as procedure qualifiers are:

```
INLINE BOOLEAN PROCEDURE eof (POINTER(file) f);

COMPILETIME $ALWAYSINLINE BITS PROCEDURE bMask
    (INTEGER lowBit,highBit);
```

"$ALWAYSINLINE" and "INLINE" may appear in the declaration of a procedure's body without also appearing in a forward or interface declaration for the procedure.

If "$ALWAYSNLINE" and "INLINE" are used in a call, they appear immediately before the name of the procedure being called, e.g.:

```
INLINE eof(f)

$ALWAYSINLINE bMask(0,n)
```

The compiler rarely disregards the programmer's instructions on whether to make a given call inline or closed. The exception is for recursive calls. If a recursive call is designated as inline, then the compiler cannot keep expanding the procedure's body indefinitely. Instead, if it is expanding one or more calls inline and encounters another call to one of the procedures of which bodies are currently being expanded, it forces the call to be closed. This has the effect of reducing the number of closed calls made to the recursive procedure without unduly increasing the module's code size.

Only small procedures, or procedures called just once, should be called inline to avoid the generation of excessive code.

9.12. Generic Procedures

A generic procedure allows a single identifier to represent several procedures (the "instance procedures" of the generic procedure). On each call to a generic procedure (a "generic call"), a call to one or more of the instance procedures is generated. The instance procedure is selected at compiletime based on the data types and number of the arguments to the generic call. A single generic name can thus be used for several related procedures with different parameter declaration lists.

For example, the single procedure name "new" provides a number of related services (it allocates new records, new arrays, and new data sections). "new" is actually a generic procedure, so that its arguments determine which of its instances is used in a particular call.

A generic procedure is not really a procedure since it has no procedure body and no parameters of its own; it is more like a special kind of macro than a procedure (macros are described in Chapter 13). The declaration of a generic procedure must appear in the outer declarations of each module that calls it.

The form of a generic procedure declaration is:

```
GENERIC PROCEDURE id exp
```

where id is an identifier and exp is a string constant expression containing a list of procedure names separated by commas, e.g.,

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

There is nothing special about the pi; i.e., they are normal procedures declared elsewhere as if they did not appear in a generic declaration. A procedure may appear in any number of generic declarations.

When 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., a parameter of p1 is a different data type from that of the corresponding argument in the procedure call), 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 produces an error message if no such pi is found.

Any pi may itself be a generic procedure, thereby recursively invoking the generic mechanism, except that the effect of including p in its own instance list is undefined. Any pi may also be of the form "m.f" where m is a module and f is an interface procedure of that module. The form "m.f" is described in Section 10.3.

The pi need not have been declared when the generic declaration is encountered, since the string constant in the generic declaration is not examined until p is used in a procedure call. In fact, if while processing a generic call the compiler finds a pi that has not yet been declared, it proceeds to the next pi.

A generic procedure may be used as a field, i.e., may be preceded by a pointer or module identifier and a period; see Section 10.11.

9.12.1. Sample Generic System Procedure

Many of the system procedures are generic. For example, cos, the procedure that computes the cosine of its argument in radians, is declared as (except that XIDAK reserves the right to use different instance names from "rCos" and "lrCos"):

```
GENERIC PROCEDURE cos "rCos,lrCos"
```

and the headers of the procedures rCos and lrCos are:

```
REAL      PROCEDURE rCos  (REAL      r)
LONG REAL PROCEDURE lrCos (LONG REAL r)
```

When the identifier "cos" occurs in a procedure call, either rCos or lrCos is invoked, depending on the data type of the argument. For example, "cos(1.4)" results in "rCos(1.4)". The compiler first tries to process "cos(8.76582L)" as "rCos(8.76582L)", but an error occurs, since the parameter to rCos must be real, but the argument 8.76582L is a long real. The compiler then tries "lrCos(8.76582L)", which compiles without error, so lrCos is the instance procedure selected.

XIDAK reserves the right to change the instance procedure names of generic system procedures at any time without notice. Programmers must never make explicit use of an instance name of a generic system procedure.

9.12.2. Generic Procedure Instance Selection Algorithm

When the compiler encounters a generic identifier, it searches the associated procedure declarations for one with parameters that "match" the arguments in the generic call.

For each instance procedure in the generic instance list, starting with the first, the compiler determines whether the procedure has been declared; if not, it skips to the next instance. Otherwise, it compares the parameters one by one with the corresponding arguments in the generic call until either:

- it finds an assignment compatibility error (see Section 4.9) or mode error between a parameter and the corresponding argument, or
- it runs out of arguments, or
- it runs out of parameters.

A mode error occurs if anything other than a variable is passed for a modifies or produces parameter.

If there is an assignment compatibility or mode error, the compiler knows that the procedure it is checking is inappropriate, and so it goes on to check the next instance.

If there are more parameters than arguments, the compiler checks the "extra" parameters to see if they are declared "OPTIONAL". If so, an appropriate instance procedure has been found, and a call to that procedure, with all the given arguments plus appropriate Zero values for the optional parameters, is generated. If the "extra" parameters are not optional, then the procedure being checked is inappropriate, and so the compiler goes on to check the next instance procedure.

If it runs out of parameters and there are not any more arguments, the compiler has found the appropriate procedure, so it stops its search and generates a call to the current instance procedure.

If it runs out of parameters but there are more arguments in the generic call, the compiler checks the last parameters of the current instance procedure. If they are not repeatable parameters, the compiler rejects the procedure as inappropriate, and goes on to check the next instance. If the last parameters are repeatable, then an appropriate procedure has been found; a call to the procedure, with all the arguments compared so far, is generated. Then a new generic call is processed (starting from the beginning of the instance procedure list), this time with all the arguments of the generic call resulting from the last step except for the last ones compared (the one matching the repeatable parameters).

The compiler issues an error message if it searches all the procedures and doesn't find any that are appropriate to call.

The order in which the procedure names are given in the generic procedure declaration is important, since it determines the order in which the procedures are checked. For example, if the generic procedure gen were declared:

```
GENERIC PROCEDURE gen "proc1,proc2,proc3"
```

and proc1, proc2, and proc3 were declared:

```
PROCEDURE proc1 (REPEATABLE INTEGER i); ...
PROCEDURE proc2 (REPEATABLE REAL r); ...
PROCEDURE proc3 (INTEGER i; REPEATABLE REAL r); ...
```

then proc3 would never be called, since any combination of integer or real parameters would match with proc1 or proc2. However, if proc3 appeared first in the generic declaration, then the call:

```
gen(1,2.0,3.0)
```

would call proc3 twice, first with arguments 1 and 2.0, then with arguments 1 and 3.0. The call:

```
gen(1,2,3,4.0,5.0,6.0)
```

would call proc1 twice, first with an argument of 1, then an argument of 2, and then call proc3 three times, with argument pairs 3, 4.0; 3, 5.0; and 3, 6.0. The call:

```
gen(1.0,2,3.0,4,5.0)
```

would call proc2 with an argument of 1.0, then proc3 with arguments 2 and 3.0, then proc1 with an argument of 2, and finally proc3 with arguments 4 and 5.0.

The results are undefined if the generic mechanism is used in conjunction with repeatable parameters to generate more than one instance call with a single generic call if any of the instances is a typed procedure.

9.12.3. Generic Procedure Extension

A generic procedure may be extended (i.e., may have new instances added to its instance list) by redeclaring it. The instances in the new declaration are added to the front of the instance list from previous declarations. For example, if a procedure p is originally declared as:

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

then if p is subsequently redeclared as:

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

the effect is as if p had been declared as:

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

9.13. Stack Overflow

Stack overflow may occur if too many procedure calls are simultaneously active, or if procedures with too many local variables are called. Stack overflow is not necessarily detected by MAINSAIL. Its effects are undefined, although on some systems, the predefined $stackOverflowExcpt can make a stack overflow easier to recognize.

On some operating systems, the MAINSAIL utility "CONF" allows the user to set the initial coroutine's stack size. On such systems, if a stack overflow occurs in a program, it may be necessary to build a MAINSAIL bootstrap with a larger stack size in order to run the program

successfully. The stack sizes of coroutines other than the initial coroutine may be set with the system procedure $createCoroutine.

See the appropriate operating-system-dependent MAINSAIL user's guide for more information.

10. Modules and Data Sections

A module is the smallest separately compilable unit of MAINSAIL code. A MAINSAIL program is composed of one or more modules, some of which may be contributed by the programmer and some by the MAINSAIL runtime system.

Modules communicate at runtime through "interface fields", which are the variables and procedures of each module declared by the programmer to be accessible from other modules.

A module is written in the general form shown in Table 10-1. The outer declarations sections may be empty; see Chapter 6.

"modNam" is the name of the module, enclosed in double quotes, and must be an identifier of six characters or fewer. The name is used in the declaration of the module, if the module is explicitly declared (see Section 10.2). Since the runtime system uses a module's name to identify it, every module in a program must have a unique name. The programmer's choice of module names must not conflict with the names of standard runtime modules.

The "outer declarations" declare all identifiers (except those predefined by MAINSAIL) that are to be accessible from the point of declaration to the end of the module, but not from any other modules. The outer declarations of a module m must include declarations of all modules referenced by m. In addition, m must declare itself if it has any interface fields. Module declarations are described in Section 10.2.

A module cannot contain another module; i.e., modules cannot be nested. All modules are on equal footing; there is no explicitly declared "main" or "controlling" module, though the module first given control during a particular execution might be considered the "main" module for that execution.

The division of a program into modules is entirely up to the programmer. MAINSAIL has no portable rule by which to determine when a module is too large, but each machine on which MAINSAIL is implemented may place an upper limit on the size of a module it can execute. If such a limit exists, it may be found in the appropriate operating-system-specific MAINSAIL user's guide.

Unlike most programming languages, MAINSAIL does not use a "link" step prior to execution. Instead, the modules are brought into memory as needed during execution. The MAINSAIL runtime system provides all the facilities for intermodule communication. The programmer need never specify beforehand what modules make up a program; a program is an open-ended collection of modules the identity of which is not determined until execution time. The dynamic binding of modules provides a degree of flexibility lacking in statically linked systems.

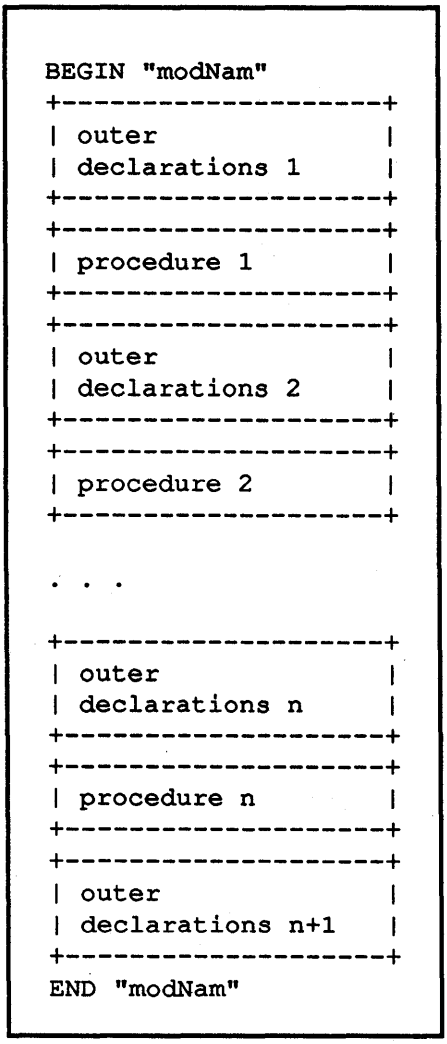


Table 10-1. A MAINSAIL Module

The MAINSAIL runtime system automatically verifies that module interfaces are consistent with one another when linkage between modules is established. See Section 10.8.

10.1. Bound and Nonbound Data Sections

Own variables (including interface variables) of a module are stored in a data structure called a "data section". After an object module (the compiled form of a module) has been brought into

memory for execution, it is referred to as a "control section". Each control section may be associated with at most one "bound data section" and zero or more "nonbound data sections". Bound data sections are created by means of the system procedure "bind"; nonbound data sections are created by several forms of the system procedure "new". Bound and nonbound data sections are identical in format, but the interface fields of a bound module can be accessed "indirecty" as described below.

Data sections differ from records in that each data section is associated with a control section. A procedure field may be accessed by means of a pointer to a data section, but not by means of a pointer to a record; modules may contain procedures, but records may not.

Interface fields associated with bound data sections may be accessed by means of implicit module pointers; such access is called "indirect access". Interface fields associated with either bound or nonbound data sections may be accessed by means of explicitly declared pointers to the data sections; such access is called "direct access".

10.2. Module Declaration

Modules communicate by means of "interface fields", i.e., those variables and procedures specified by the programmer (in module declarations) as accessible by other modules. Interface data fields are called "interface variables" and are own variables (see Section 6.4); i.e., they reside in the data section of the module (see Section 10.6).

The outer declarations of a module m must include, in any order (except that interface variables must be declared before use, and interface procedures must be declared as interface procedures before their bodies appear), declarations for all modules of which interface fields are indirectly accessed by m. In addition, m itself must be declared if it has any interface fields, even if they are not referenced within m, because the compiler must know the interface fields of the module it is compiling. If m has no interface fields, its declaration may be omitted, in which case the compiler declares it automatically as a module with no interface fields; see Example 10.2-1.

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 its header. It serves as a forward declaration (see Sections 9.10 and 14.19) for the procedure. The procedure body must be given within the module v, where the procedure is declared as usual. Interface variables of v are declared only in v's module declaration; they are not redeclared in v after the module declaration has been seen.

A sample module declaration is shown in Example 10.2-2. The sample declares a module named "parse" with several interface fields.

```
    BEGIN "m"
    outer declarations          # not including one for m
    ...
    END "m"

is equivalent to

    BEGIN "m"
    outer declarations
    ...
    MODULE m;
    END "m"
```

Example 10.2-1. A Module That Does Not Explicitly Declare Itself

```
MODULE parse (
    INTEGER                  val,lineNum,index;
    STRING                   token,line;
    INTEGER ARRAY(1 TO 10)   order;
    PROCEDURE                getToken;
    STRING PROCEDURE         msgTxt (INTEGER page,msgNum);
    )
```

Example 10.2-2. Sample Module Declaration

A module m that accesses fields of a module n need declare only a prefix of n's interface if it does not access all of n's fields. No error occurs (see Section 10.8), provided that m's view of n matches a prefix of n's view of itself. See Example 10.2-3.

10.3. Indirect Access to Interface Fields

An interface field f of a module m may be indirectly accessed with the field variable "m.f" (which may also be written "(m).f"). As described below, the simpler form "f" may often be used. The interface field is associated with the module's bound data section. Because of the syntactic simplicity of indirect access, a module's bound data section may be thought of as its "default" data section (the one used when no explicit pointer is used).

```
If the module n has two fields, and is declared in
n as:

    MODULE n (INTEGER field1; PROCEDURE field2);

and the module m uses only the first field (field1) of n,
then n may be declared in m as:

    MODULE n (INTEGER field1);
```

Example 10.2-3. A Module That Declares Only a Prefix of Another's Interface

The term "field variable" is used for both interface procedures and interface variables.

For every module m declared by means of the keyword "MODULE" in a module n (and actually used in n), n maintains a hidden "implicit module pointer" to m's bound data section. It is this implicit module pointer that is used in indirect access to interface fields of m.

For example, if m1 is declared as:

```
MODULE m1 (STRING name,quest; INTEGER val; PROCEDURE findVal)
```

then the interface fields of m1 can be indirectly accessed as the field variables "m1.name", "m1.quest", "m1.val", and "m1.findVal". The interface variables of m1 may be altered by other modules; i.e., there are no "protected" or "read-only" fields.

"m.f" can be written as "f" if m is the current module, or if m is the only module declared in the current module with a field named f. In either case, the compiler effectively provides the "m." prefix. The compiler generates the same code regardless of whether or not the abbreviated form is used (i.e., there is no efficiency penalty or advantage for including the "m.").

As an example, many of the MAINSAIL system procedures (see Chapter 1 of part II of the "MAINSAIL Language Manual") are interface fields of various system modules. The programmer need not know the names of the runtime modules. To call a system procedure, the programmer specifies only the procedure name. All the system procedures have been given unique names, and hence the compiler can figure out the module name. If the programmer declares an interface variable with the same name as a system procedure, the name of the system procedure alone may no longer suffice to produce an unambiguous reference. System procedure names must not be explicitly prefixed with their module names, since XIDAK reserves the right to change the module in which a system procedure resides.

10.4. Classes with Procedures

The fields of a module can be supplied using a class, with a declaration such as:

```
MODULE(c) m
```

where c is a class name. The module declaration may declare the module to be of a prefixed class by adding extra fields to an existing class, in the form:

```
MODULE(c) m (declarations of additional fields)
```

The declaration of the module PARSE of Example 10.4-1 is equivalent to the declaration of Example 10.2-2.

```
CLASS parseCls (
    INTEGER                  val,lineNum,index;
    STRING                   token,line;
    INTEGER ARRAY(1 TO 10)   order;
    PROCEDURE                getToken;
    STRING PROCEDURE         msgTxt (INTEGER page,msgNum);
    );

MODULE(parseCls) parse;
```

Example 10.4-1. Sample Module Declaration Using a Class

10.5. Direct Access to Interface Fields

An explicit pointer to a module's data section may be declared if the interface of the module is declared as a class. Allocation of a record of a class c by means of "new(c)", where c contains procedure fields, results in a record in which the procedure fields are not valid; the compiler issues a warning message if such an allocation is performed. The effect of calling a procedure in c using such a record is undefined. Procedure as well as variable interface fields may be accessed with the pointer only if a data section of the class is allocated instead of a record.

Access by means of an explicit pointer to fields associated with a data section is referred to as "direct access" to the data section. Both bound and nonbound data sections may be directly accessed.

The system procedure "bind" and some forms of the system procedure "new" allocate data sections instead of records. These forms specify the name of the object module to be associated with the data section. The module must have the interface specified by the class of the pointer used to access its data section; otherwise, the effects of accessing fields with the pointer is undefined.

Example 10.5-1 shows field variables that are valid if a pointer points to a data section of a given class.

10.6. Module Allocation and Disposal

The allocation of a module is the allocation and clearing of its data section and the invocation of its initial procedure, if it has an initial procedure (see Section 10.9). A module may be allocated by a call to bind or new, as shown in Example 10.5-1. An indirectly accessed module may be automatically allocated, as described in Section 10.7.

When a data section is disposed (see Section 1.127 of part II of the "MAINSAIL Language Manual"), MAINSAIL automatically invokes the module's final procedure, if there is one. When a module is disposed, MAINSAIL disposes all the data sections for that module, then releases the control section for the module (so that if a new data section is allocated for the module, the search for the module's control section proceeds from scratch as described in Section 12.2). At the end of a MAINSAIL execution, MAINSAIL normally executes the final procedures associated with all data sections and then closes any open files.

The exception $disposedDataSecExcpt is raised when an attempt is made to return to a procedure invocation associated with a disposed data section.

When a module is unbound (see Section 1.370 of part II of the "MAINSAIL Language Manual"), the bound data section is disposed, but the control section is not released.

10.7. Establishing Module Linkage

A module m may at any time call an interface procedure in another module n. However, m can indirectly access an interface variable in n only if a bound data section already exists for n. When a bound data section for n is created, implicit module pointers to n in all modules that use n are initialized. If a module m is allocated and a bound data section for n exists, m's implicit module pointer to n is initialized. When the implicit module pointer from m to n is initialized, m is said to "have linkage to" n.

If checking is in effect, MAINSAIL raises the exception $unboundModuleExcpt if an attempt is made to access an interface variable of a module to which linkage is not yet established (i.e., for which no bound data section exists).

```
CLASS parseCls (
    INTEGER                  val,lineNum,index;
    STRING                   token,line;
    INTEGER ARRAY(1 TO 10)   order;
    PROCEDURE                getToken;
    STRING PROCEDURE         msgTxt (INTEGER page,msgNum);
    );

POINTER(parseCls) p;
```

If there exists a module PARSE of parseCls, p may be made to point to the bound data section of PARSE by means of:

```
    p := bind("PARSE");
```

or, if PARSE is declared in a module declaration:

```
    p := bind(parse);
```

The call to bind allocates the bound data section if it has not already been allocated.

p may be made to point to a nonbound data section of PARSE by means of:

```
    p := new("PARSE");
```

or, if PARSE is declared in a module declaration (like "MODULE(parseCls) parse"):

```
    p := new(parse);
```

The call to new always allocates a new nonbound data section.

If p is made to point to a data section of the class parseCls, the follwing directly accessed field variables are valid:

Example 10.5-1. Accessing Data Section Fields with a Pointer (continued)

```
p.val
p.lineNum
p.index
p.token
p.line
p.order
p.getToken
p.msgTxt(...)
```

Example 10.5-1. Accessing Data Section Fields with a Pointer (end)

In the future, MAINSAIL may support automatic establishment of linkage upon interface data access; this has not been done so far because of execution overhead considerations.

10.8. Intermodule Consistency Checking

If a module A indirectly accesses a module B, the compiler requires that A include a module declaration for B and that B include a module declaration for itself. When A gets linkage to B, MAINSAIL checks these module declarations for consistency. A module inconsistency is detected if A's view of B is not a prefix of B's view of itself. If the interfaces are inconsistent, an error occurs.

The consistency check verifies that the declarations of corresponding interface variables and procedures are compatible. Corresponding interface variables must have the same data type; if the variables are arrays, they must have the same bounds and must be either both short arrays or both long arrays. Corresponding interface procedures must have equivalent declarations; corresponding parameters must be of the same data type, and must be both uses, both modifies, or both produces. It is not checked that corresponding identifiers have the same name or that corresponding pointers have the same class.

10.9. Initial Procedure

Each module may contain a single typeless and parameterless procedure that is to be called whenever a data section for the module is allocated. This procedure is qualified with "INITIAL", in accordance with the syntax described in Section 9.8. The initial procedure may also be called explicitly.

The "top level" module of a program (the one specified to the MAINSAIL executive as the module to gain control first) must have an initial procedure, or it returns immediately without

performing any action. The execution of the top level module's initial procedure leads to execution of the entire program.

A module need not have an initial procedure. When a module not intended to be used as a top level module has an initial procedure, it usually performs initialization of the module's outer and interface variables.

The programmer does not have to give the initial procedure a name, in which case the compiler supplies the name "initialProc". See Example 10.9-1.

```
    INITIAL PROCEDURE;
    BEGIN ... END

is equivalent to

    INITIAL PROCEDURE initialProc;
    BEGIN ... END
```

Example 10.9-1. Default Name of the Initial Procedure

The "INITIAL" qualifier need be specified only in a procedure declaration that includes a procedure body; i.e., it does not have to be given in forward or interface field declarations. However, if it is given in a forward or a field declaration, it must also be given in the body declaration.

10.10. Final Procedure

Each module may contain a single typeless and parameterless procedure that is automatically invoked when the module is disposed. This procedure is qualified with "FINAL", in accordance with the syntax described in Section 9.8. The final procedure may also be called explicitly.

Final procedures are often used to dispose of arrays and modules that are no longer needed, to release scan bits, or to close files that should not be left open. A final procedure often "undoes" the work of the initial procedure.

The programmer does not have to give the final procedure a name, in which case the compiler supplies the name "finalProc", as in Example 10.10-1.

```
    FINAL PROCEDURE;
    BEGIN ... END

is equivalent to

    FINAL PROCEDURE finalProc;
    BEGIN ... END
```

Example 10.10-1. Default Name of the Final Procedure

The "FINAL" qualifier need be specified only in a procedure declaration that includes a procedure body; i.e., it does not have to be given in forward or interface field declarations. However, if it is given in a forward or a field declaration, it must also be given in the body declaration.

When a MAINSAIL execution terminates, all data sections are disposed, and all open files are closed. The order in which data sections are disposed is unspecified. The programmer must not access a data field of another module in a final procedure without first ensuring that the other module is bound. The programmer must not perform I/O to a file in a final procedure without ensuring that the file is open. Bugs that are difficult to track may occur because of dependencies on the order of execution of final procedures.

10.11. Generic Procedures as Field Variables

Generic procedures may be used as field variables; i.e., in p.f or m.f (p is a pointer and m a module), f may be a generic procedure name. The generic name is substituted according to the usual rules (see Section 9.12) for f. For example, given the declarations:

```
CLASS c1 (PROCEDURE p1 (INTEGER i));

PROCEDURE p2 (REAL r);

CLASS c3 (PROCEDURE p3 (STRING s));

GENERIC PROCEDURE p "p1,p2,p3";

POINTER(c1) ptr1;
POINTER(c3) ptr3;
```

the code:

```
ptr1.p(2);
p(2.0);
ptr3.p("2")
```

would be equivalent to:

```
ptr1.p1(2);
p2(2.0);
ptr3.p3("2")
```

10.12. Control Sections and Module Swapping

Whenever a data section is allocated for a module, the corresponding objmod is brought into memory if necessary; an objmod brought into memory for execution is called a "control section". Only the currently executing module's control section need be in memory; MAINSAIL automatically swaps out control sections for which there is not enough memory. This may be thought of as an "automatic overlay" facility. MAINSAIL attempts to keep the most recently used control sections in memory. MAINSAIL does not swap out any data structures other than control sections; thus, MAINSAIL supports "virtual code space" but not "virtual data space".

When a control section that was obtained from an individual objmod file is first removed from memory, it is written to a "swap file". When a control section obtained from a currently open objmod library is removed from memory, it is not written to the swap file, since it may be read back in from the objmod library.

10.13. Compilation of Several Modules in One File

More than one source module can appear in the same file. If no text (other than blank, tab, end-of-page and end-of-line characters) appears between the final "END" of one module and the "BEGIN" of the following module, the modules are compiled just as if they were separated into different files and each file compiled separately.

Symbols that are defined between the "END" of one module and the "BEGIN" of the next (with "DEFINE" or "REDEFINE") normally belong to the PRECEDING module, not the following. The compilation of a new module does not begin until its "BEGIN" is seen (if something other than "BEGIN" or a macro definition or compiler directive is seen, it is treated as end-of-file). Compiling a file with the following contents therefore gets an error:

```
REDEFINE moduleName = "firstM";

BEGIN moduleName
END moduleName

REDEFINE moduleName = "second";

BEGIN moduleName
END moduleName
```

since the second definition of moduleName belongs to the first module; when the second "BEGIN" is seen, the compilation of the second module is begun, and the previous definition of moduleName is forgotten. The use of "moduleName" therefore results in a compiler error message.

Symbols MAY be defined after the initial "BEGIN" of a module. The above could be rewritten as:

```
REDEFINE moduleName = "firstM";

BEGIN moduleName
END moduleName

BEGIN REDEFINE moduleName = "second";
    moduleName
END moduleName
```

This would produce two modules, FIRSTM and SECOND, without error. The "$GLOBALREDEFINE" directive could also be used in place of "REDEFINE" in the original example to make it compile correctly.

10.14. Nonbound-Invocation Modules

```
TEMPORARY FEATURE:  SUBJECT TO CHANGE
```

A module may be a "nonbound-invocation module", meaning that when $invokeModule is called it creates a nonbound data section rather than a bound data section (if the module's data section is allocated with "bind" or by indirect access of an interface procedure, a bound data section is created, as usual). This allows several invocations of a module to coexist, provided that they are created by $invokeModule.

A module is specified to be nonbound-invocation module with the compiler subcommand "UNBOUND". This subcommand may be used in a "$DIRECTIVE" directive; i.e., a module containing:

```
$DIRECTIVE "UNBOUND";
```

is specified to be a nonbound-invocation module.

In the MAINSAIL compiler, the module COMPIL is a nonbound-invocation module (all the other compiler modules are also allocated as nonbound data sections, created with new). If, during a compilation, the compiler is invoked (e.g., with a call to $invokeModule, or at an "error response" prompt, or from MAINEX or MAINEDIT, all of which use $invokeModule), a new instance of the compiler is created which does not interfere with any other instance. If a bound data section of COMPIL were used, then a second invocation of the compiler would destroy the first, since both would be using the bound instance of COMPIL.

XIDAK is considering other, more flexible, approaches to allowing simultaneous invocations of the same program.

11. Intmods

Intmods are compiler symbol table files created by compiling MAINSAIL modules. The symbols defined in intmods may be used in the compilation of other modules or by MAINSAIL system programs like MAINDEBUG or the MAINSAIL disassembler.

An intmod contains several kinds of information, including:

- outer symbols
- local symbols
- procedure parse trees (a parse tree is a representation of the statements and expressions in the procedure)
- procedure instruction maps (instruction maps relate objmod offsets to source code)
- additional information allowing the debugger to examine variables and call procedures

The outer symbols may be used by other modules that "open" the intmod during compilation. The other contents of an intmod are for MAINSAIL system programs.

An intmod is generated (or updated) during a module's compilation only when a compiler option is in effect that requires information to be stored in the intmod. By default, no intmod is generated. Any of the following options cause an intmod to be generated (or updated):

Option	Information Stored in Intmod
ALIST	instruction maps
DEBUG	instruction maps, symbol tables, debug information
INCREMENTAL	parse trees for inline procedures, symbol tables
MONITOR	instruction maps (needed by PERSTMT)
PERSTMT	instruction maps
SAVEON	parse trees for all procedures, symbol tables

In each case, only information required by the options is put into the intmod.

11.1. Intmod Directives

Several directives are provided for dealing with intmods:

```
$DIRECTIVE "OPENMODULE s";
$DIRECTIVE "MAKEMODULEVISIBLE m1 ... mn";
$DIRECTIVE "MAKEMODULENOTVISIBLE m1 ... mn";
$DIRECTIVE "MAKEVISIBLE s1,...,sn";
$DIRECTIVE "MAKENOTVISIBLE s1,...,sn";
RESTOREFROM "s";
SAVEON "s";
```

11.1.1. Opening Intmods and Accessing Symbols

In the "$DIRECTIVE" directive "OPENMODULE s", s is the name of a module or intmod file; the intmod must have been made with the "SAVEON" option in effect. The named intmod is opened, along with any "supporting modules", i.e., intmods that were open when the "SAVEON" that created the intmod was done.

Identifiers from intmods may be specified using the syntax:

```
<intmod module name>$<identifier>
```

For example, an identifier "id" from an intmod FOO is specified using the compound, or qualified, identifier "foo$id". "foo$id" refers to the "id" in the intmod FOO, even if there is also an "id" in the current module or in another open intmod. The compound identifier is considered a single identifier for purposes of macro substitution; i.e., if a macro "bar" has been defined, or if "bar" is a macro parameter, its definition is not expanded in "bar$id" or "mod$bar". Whenever a compound identifier is found, the compiler automatically opens the specified module if it is not already open; e.g., if "baz$xxx" is encountered and BAZ is not an open module, the compiler acts as if it had seen:

```
$DIRECTIVE "OPENMODULE baz";
```

11.1.2. Module Visibility

The "$DIRECTIVE" directive "MAKEMODULEVISIBLE" makes identifiers in one or more open modules "visible", i.e., accessible without the need for qualification by the module-name-and-dollar-sign prefix (the modules themselves are also said to be visible). The mi are module names (never file names). Each module is opened, if not already open, as by "OPENMODULE" (so each intmod must have been made with the "SAVEON" option in effect). The identifiers in a visible module can still be qualified, if desired. The effect of making a module visible can be undone by the "$DIRECTIVE" directive "MAKEMODULENOTVISIBLE". These two directives can be used any number of times for the same module, alternately making it visible, then invisible.

11.1.3. Individual Symbol Visibility

The individual symbol visibility directives provide a way to control the visibility of individual symbols in an intmod. These directives are useful only if the intmod is saved. Only those symbols individually visible are made visible in another module by "MAKEMODULEVISIBLE".

Each intmod has a "visibility default" that can have one of the two values "visible" and "invisible". At the start of compilation a module's visibility default is initialized to "visible". This default applies to all symbols not explicitly marked by "MAKEVISIBLE" or "MAKENOTVISIBLE".

The "$DIRECTIVE" directive "MAKEVISIBLE s1,...,sn" does the following:

- If the visibility default is "visible" then change it to "invisible" and clear the visibility list.
- Add the symbols s1, ..., sn to the visibility list.

The "$DIRECTIVE" directive "MAKENOTVISIBLE s1,...,sn" does the following:

- If the visibility default is "invisible" then change it to "visible" and clear the visibility list.
- Add the symbols s1, ..., sn to the visibility list.

These directives are not available as compiler subcommands.

The symbols si may be the names of identifers declared or defined anywhere in the outer block of the current module. An si may also be of the form <class name>.<field> or <module name>.<field>, assuming the field is declared in the current module. There is no way to affect the visibility of individual symbols from other intmods.

The si are not processed until the compiler has read the entire module, so the si may refer to identifiers declared after the visibility directive. A visibility directive can appear in a procedure body, but it cannot refer to symbols local to the procedure.

After the compiler has read the entire module, each referenced symbol on the visibility list is "marked", so that the intmod contains the marked symbols. The visibility default is stored in the intmod.

When an intmod is made visible, only the following outer symbols are actually visible:

- If the visibility default stored in the intmod is "visible", then all symbols are visible except those on the visibility list.

- If the visibility default stored in the intmod is "invisible", then no symbols are visible except those on the visibility list.

A symbol can always be used with a compound identifier, e.g., "modNam$symbolName", even if "symbolName" is not visible.

All symbols in the current module are always visible, regardless of the symbol visibility directives.

If a symbol x from an intmod B is visible in a module A, then A cannot declare or define x without a module prefix, since the definition from B would conflict. However, x may be declared in A as "a$x". If a programmer wishes to redeclare a predefined identifier, e.g., "integerCode", in a module FOO, he or she must use the form "foo$integerCode". After the point of definition in FOO, "integerCode" refers to foo$integerCode, not the system macro integerCode.

11.2. Visibility from Supporting Intmods

Symbols from an intmod C used by an intmod B used by the current module A have the same visibility as they had at the end of the compilation of B, regardless of any individual symbol directives in B. For example, if all symbols in C are visible at the end of the compilation of B, and B is currently visible in A, then all symbols in C are currently visible in A. This is true regardless of the status of B's visibility list, e.g., even if no symbols from B are visible.

11.3. "RESTOREFROM" and "SAVEON"

The directive:

```
RESTOREFROM "s";
```

where s is a module name or file name, first performs:

```
$DIRECTIVE "OPENMODULE s","MAKEMODULEVISIBLE m";
```

where m is the name of the module in the intmod specified by s. In addition, all modules that were open or visible when the "SAVEON" for m occurred ("supporting modules for m") are also made open or visible, i.e., are restored to their status at the time of the "SAVEON". By contrast, "MAKEMODULEVISIBLE" makes only the specified modules visible; it makes sure that modules that were open when the specified intmods were made are open, but does not make them visible, regardless of whether or not they were visible when the intmod was made.

The directive:

```
SAVEON "s";
```

makes an intmod with the "SAVEON" option in effect for the current module, when the compilation is complete (saving a partial module is not possible). The intmod contains all the information required to support the "OPENMODULE" directive. s is the name of the file on which the saveon is stored. s may be omitted, i.e.:

```
SAVEON;
```

in which case a default file name is used, based on the name of the module being compiled (or the intmod is put in an intmod library if the appropriate compiler subcommands are in effect; see the "MAINSAIL Compiler User's Guide" for details).

The compiler does not permit an intmod to be opened unless it was created with the "SAVEON" option in effect. An intmod created in the absence of "SAVEON" does not contain enough information to support its use by the compiler, although it may be usable by other tools such as the debugger and disassembler.

Typically, a "SAVEON" directive occurs in a "definitional module", one for which no code is generated and which is used only as a repository of definitions and declarations. The MAINSAIL language does not make a distinction between a definitional module and an "executable module", one for which code is generated and executed. An executable module can serve as a definitional module, or vice versa.

11.4. Unqualified Identifier Search Rules

The MAINSAIL compiler searches for an unqualified identifier in the following order:

1. It searches the MAINSAIL keywords.

2. It searches the symbols defined in the current procedure, if compiling a procedure body.

3. It searches the outer identifiers of the current module.

4. It searches visible intmods, in the order most recently made visible to least recently made visible.

5. It searches the global symbol table (in which symbols defined by "$GLOBALREDEFINE" reside).

The first identifier found by searching in the above order is the one used by the compiler. No warning is given if the same identifier occurs in another open intmod. Compilation may become slower as more intmods are made visible, since there are more symbol tables to be searched. For this reason, it may be a good idea to keep as many identifiers as possible in a single intmod, if the identifiers are to be used without qualification. There is no compilation slow-down if several intmods are opened but not made visible, and all references to identifiers in the intmods are qualified; however, this puts the burden on the user of remembering each identifier's declaring module, which may not be convenient.

11.5. Use of Symbols from an Intmod

Interface procedures and variables from an open intmod are processed as intermodule references, as usual. Referenced non-interface procedures and outer variables from an open intmod are "copied" out of the intmod; i.e., the procedures are compiled into the current module as if they were forward procedures, and the outer variables are treated as if they were declared in the current module. Macros from an open intmod are expanded in the usual way.

Procedures called by procedures copied into the current module are also copied into the current module. The compiler remembers the module in which it found each procedure, so that the procedures that would have been called at the point of compilation in the original intmod are called, not procedures of the same name in the current module. See Example 11.5-1.

Procedures extracted from definitional modules are more quickly compiled than forward procedures, since the source text has already been parsed and converted into the compiler's internal representation.

11.6. Intmod Search Rules

When searching for an intmod, MAINSAIL looks in the specified file name, if processing a compiler directive that specifies a file name. If processing a directive that specifies a module name, or if looking for an intmod for some program other than the compiler, it follows the search rules described in Section 12.2. If unsuccessful, it tries treating the given name as a file name instead of a module name, and attempts to open the named file. If it still does not find the intmod, and the module it is looking for is a supporting module (a module used during compilation by some other module), it last attempts to find the intmod file under the file name specified when it was used during compilation (which may be different from the file name actually used, e.g., if a logical file name or searchpath was in effect).

11.7. Changing an Intmod

If an intmod is remade, all intmods that reference it must also be remade; otherwise, undefined errors may result when the changed intmod is opened.

```
If a module A contains:

        BEGIN "a"

        SAVEON;

        PROCEDURE p1;
        <body for p1>

        PROCEDURE p2;
        BEGIN ... p1; ... END;

        END "a"

and a module B contains:

        BEGIN "b"

        RESTOREFROM "a";

        PROCEDURE p1;
        <body for B's p1>

        INITIAL PROCEDURE;
        BEGIN ... p2; ... END;

        END "b"

then the call to p2 in B's initial procedure calls the
procedure p2 copied from A, i.e., a$p2, which calls the
procedure p1 copied from A, i.e., a$p1, rather than the
p1 in B, since a$p1, not b$p1, would have been called at
the point where p2's body was encountered when it was
compiled (in module A).
```

Example 11.5-1. The Compiler Is Not Confused by Procedures of the Same Name in the Wrong Module

If the date on an intmod is older than the date on one of its supporting intmods, a message is issued when the supporting intmod is opened, the system program opening the intmod enters a dialogue to confirm that the supporting intmod really should be used. If the date on the supporting intmod file is wrong (e.g., if the file has been copied, or the system clock is inaccurate), but the contents are correct, the supporting intmod should be used; but if the

supporting intmod actually has been changed since it was used in the compilation of the first intmod or objmod, undefined errors may result.

11.8. Sample Use of Intmods

An intmod may be used as an alternative to "header" files, i.e., sourcefiled files containing declarations common to several modules (see Section 14.2). Declarations from intmods are processed more quickly by the compiler than header files.

Suppose that several modules each require three "header" files "hdr1", "hdr2", and "hdr3" as sourcefiles. Each of the modules could use the "SOURCEFILE" directive to obtain the information in the header files, with each header file being recompiled. But it would be more efficient (assuming the header files are not being changed) to compile the header files once and then save the state of the symbol table. Each module could then restore the symbols from the saved intmod, thereby giving the effect of having just compiled the header files.

A convenient way to create the saved file is to compile a file with the contents shown in Example 11.8-1. Each of the modules would then be written as shown in Example 11.8-2.

```
BEGIN "hdr"
SAVEON;
SOURCEFILE "hdr1";
SOURCEFILE "hdr2";
SOURCEFILE "hdr3";
END "hdr"
```

Example 11.8-1. A Source File Compiled to Produce an Intmod

```
BEGIN "modNam"
RESTOREFROM "hdr";
...
END "modNam"
```

Example 11.8-2. A Module Using an Intmod

12. Objmods, Intmods, Libraries, and Search Rules

MAINSAIL uses two types of compiled modules, objmods (object modules) and intmods (intermediate modules).

The MAINSAIL compiler, by default, outputs the executable form of a MAINSAIL module into a file (an "objmod file") of which the name is formed from the name of the module compiled.

Intmods are not executable. They contain information used during compilation of other modules or used by MAINSAIL system programs. See Chapter 11 for more details.

Both objmods and intmods may be stored into libraries, files containing several modules. Objmods are stored into objmod libraries, and intmods into intmod libraries; either type of library may be referred to as a "module library". Modules may be compiled directly into module libraries (see the "MAINSAIL Compiler User's Guide") or added to module libraries with the utilities MODLIB and INTLIB (see the "MAINSAIL Utilities User's Guide").

MAINSAIL may execute either directly from objmod files or from objmod libraries. Objmod libraries have several advantages over objmod files:

- When a system consists of several modules, putting them into a library eliminates the need for individual objmod files, thus reducing clutter in the file system.

- An objmod library opened for execution is opened just once. Each module that does not reside in an objmod library is individually opened during execution when it is first accessed. Opening a file is a time-consuming operation on many systems.

- Sometimes it is necessary to discard modules from memory in order to make more room. A module that does not reside in memory must first be written to a "swap" file (see Section 10.12) before its space is used. A module that resides in a module library open for execution need not be written to the swap file.

More information on module libraries may be found in the "MAINSAIL Utilities User's Guide".

12.1. Objmod and Intmod File Names

The default intmod file name has the form:

```
<1st 3 characters of $systemNameAbbreviation>-int:<module name>
```

For example, a module FOO compiled for a VAX-11 UNIX system (where $systemNameAbbreviation is "uvax") is compiled into an intmod file named "uva-int:foo". A searchpath is usually set up for intmod file names. Bootstraps distributed by XIDAK specify a searchpath (see the "MAINSAIL Utilities User's Guide" for a description of the MAINEX "SEARCHPATH" subcommand) of the form:

```
SEARCHPATH *-int:* *2-*1.int
```

unless otherwise noted in the system-specific documentation; consult the system-specific MAINSAIL user's guide for information. This searchpath would map "uva-int:foo" into "foo-uva.int", where the intmod for FOO would be stored.

The default objmod file name has the form:

```
<1st 3 characters of $systemNameAbbreviation>-obj:<module name>
```

For example, a module BAR compiled for an M68000 UNIX system (where $systemNameAbbreviation is "um68") is compiled into an objmod file named "um6-obj:bar". A searchpath is usually set up for objmod file names. Bootstraps distributed by XIDAK specify a searchpath of the form:

```
SEARCHPATH *-obj:* *2-*1.obj
```

unless otherwise noted in the system-specific documentation; consult the system-specific MAINSAIL user's guide for information. This searchpath would map "um6-obj:bar" into "bar-um6.obj", where the objmod for BAR would be stored.

These file names and searchpaths are subject to change; those shown are correct for the current release, unless otherwise documented in the system-specific MAINSAIL user's guide.

12.2. Objmod and Intmod Search Rules

There are three types of searches that MAINSAIL makes to find an intmod or objmod:

- intSearch: an intmod search.
- objSearch: a non-executable objmod search, or just "objmod search", as when recompiling or disassembling.
- exeSearch: an executable objmod search, or just "executable search", as when executing or debugging a module.

A search is always for a particular target system. For example, an executable search is always for the host system. A cross-compilation requires intmods and objmods for the system for which the compilation is done.

The distinction between objmod searches and executable searches allows one version of an objmod to be executing while another version (possibly for a different target system) is used for another purpose, such as incremental recompilation.

By default, all searches first look in all open libraries (intlibs for intmods, objlibs for objmods), more recently opened libraries first, and then try to open a file with the default name, as described in Section 12.1. This search order may be reversed with the MAINEX "INTFILE", "OBJFILE", or "EXEFILE" subcommand. The MAINSAIL system objlib is initially open for exeSearches.

An unsuccessful library search is always immediately followed by a foreign module search in the case of exeSearches. The foreign module table is constructed with the aid of the Foreign Language Interface; see the "MAINSAIL Compiler User's Guide".

Each type of search is governed by a separate list:

- The intList governs intmod searches.
- The objList governs non-executable objmod searches.
- The exeList governs executable objmod searches (in conjunction with the module-to-module association list; see below).

Each entry on one of these lists indicates that a particular intmod or objmod is to be found in a particular file or library (or that the objmod is really another objmod, in the case of the module-to-module association list). The lists are empty unless an entry is specifically created (normally by a MAINEX subcommand, or by a MAINEX subcommand passed to $getSubcommands, as described in the "MAINSAIL Utilities User's Guide"). In the absence of any entries, a default search occurs, which is often sufficient; the lists are used only to indicate exceptions to the default search mechanism. If a list indicates that a module is in a particular library, then when a search is made for that module, the indicated library is automatically opened if necessary.

Each entry on the lists contains the following information:

- the module name,
- the file name in which it is to be found,
- whether the file is a library or a file that contains just one module,
- and the target system to which the entry applies.

An exeSearch first checks the module-to-module-assocation list, as made by calling the procedure "setModName" from a MAINSAIL module or by specifying the MAINEX "SETMODULE" subcommand (see the "MAINSAIL Utilities User's Guide"). After the module-to-module substitution is made, if any, an exeSearch behaves like the other two types of search (there is no equivalent of the module-to-module association list for intSearches and objSearches). Each type of search checks its list to see whether the target module has an entry (for the target system), and if so, insists on finding the intmod or objmod in the file or library specified by the entry. A list can have at most one entry with a given module name and target system.

For each type of search, MAINEX subcommands are provided to add an entry to the list, remove an entry from the list, print (part of) the list, and to alter the normal order of search (libraries before files or vice versa). These subcommands are described in detail in the "MAINSAIL Utilities User's Guide"; they are summarized in Table 12.2-1.

Command	Arguments	Description
INTFILE	m{=f} ...	intSearch: get m's intmod from file f
OBJFILE	m{=f} ...	objSearch: get m's objmod from file f
EXEFILE	m{=f} ...	exeSearch: get m's objmod from file f
INTLIB	m{=f} ...	intSearch: get m's intmod from intlib f
OBJLIB	m{=f} ...	objSearch: get m's objmod from objlib f
EXELIB	m{=f} ...	exeSearch: get m's objmod from objlib f
INTDEFAULT	m ...	intSearch: default search for m's intmod
OBJDEFAULT	m ...	objSearch: default search for m's objmod
EXEDEFAULT	m ...	exeSearch: default search for m's objmod
INTSHOW	{m ...}	show entries {for m ...} in intList
OBJSHOW	{m ...}	show entries {for m ...} in objList
EXESHOW	{m ...}	show entries {for m ...} in exeList
INTFILE		intSearch: search files before intlibs
OBJFILE		objSearch: search files before objlibs
EXEFILE		exeSearch: search files before objlibs
INTLIB		intSearch: search intlibs before files
OBJLIB		objSearch: search objlibs before files
EXELIB		exeSearch: search objlibs before files

Table 12.2-1. MAINEX Search List Subcommands Summary

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 that was specified when the macro was defined. This chapter describes macro definitions and macro calls.

13.1. "DEFINE"

A macro equate associates an identifier (the "macro name" or the "macro") with text or a constant expression (the "macro body") that is substituted by the compiler for subsequent occurrences of the identifier ("macro calls"). A macro definition consists of the keyword "DEFINE" (or the keyword "REDEFINE"; see Section 13.2) followed by a series of one or more macro equates, as follows:

```
DEFINE macroEquate1,macroEquate2,...,macroEquateN;
```

The form of a simple macro equate is:

```
v = macroBody
```

where v is an identifier and macroBody is a constant or "bracketed text" (Section 13.3).

If macroBody is a constant, the identifier defined is called a "macro constant". macroBody may be a constant expression of any data type. For example:

```
DEFINE maxNum = 10;
```

defines the identifier maxNum to be 10. During compilation, any subsequent occurrences of maxNum in the module are replaced with the constant 10 (as if the number 10 appeared instead of maxNum).

The form of a macro equate with parameters is:

```
v(v1, ..., vn) = [bracketed 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 (vn may optionally be preceded by the keyword "REPEATABLE"; see Section 13.6.1). 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. Macro arguments are described in Section 13.5.

A form of macro equate that involves compiletime interaction with the programmer is described in Section 13.4.

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

Macro identifiers may be used anywhere, even in subsequent macro definitions. For example, if upperLimit is defined as:

```
DEFINE upperLimit = 100;
```

then a subsequent macro definition:

```
DEFINE threeTimesUpperLimit = 3 * upperLimit;
```

is equivalent to:

```
DEFINE threeTimesUpperLimit = 3 * 100;
```

A macro definition within a procedure body defines new macros that are accessible only within the body of the procedure. After the end of the procedure body, any earlier definitions (or declarations) of the macro identifiers are again in effect.

13.2. "REDEFINE"

"REDEFINE" may be used to change the body of a previously defined macro.

In a macro definition headed by "DEFINE", each identifier defined must not have been previously defined. This restriction is not applied to macro definitions headed by "REDEFINE". Macro identifiers in macro definitions headed by "REDEFINE" are given new bodies whether the identifiers were previously defined or not.

A "REDEFINE" within a procedure body may change the body of a macro defined outside the procedure body. The new macro body remains in effect throughout the rest of the module (or until a new "REDEFINE" of the macro is encountered).

"REDEFINE" may be used to increment a counter as shown in Example 13.2-1. The macro x is defined to be 0 originally. Whenever a call to the macro def occurs, x is redefined to have a

```
DEFINE
    x       =    0,
    def(y)  =    [REDEFINE x = x + 1; DEFINE y = x;];
```

Example 13.2-1. Use of "REDEFINE"

value one greater than its previous value, and the argument to def is defined to have this new value of x. Thus the macro calls:

```
def(case1)
def(case2)
def(case3)
```

result in casei being defined as i. $def is a more sophisticated, pre-defined version of def; see Section 14.21.

13.3. Bracketed Text

Bracketed text is a sequence of characters enclosed in matching brackets ("[" and "]"). It is used in a macro body to define a macro as almost arbitrary text. The characters are taken just as is when building the bracketed text; e.g., macro calls are not expanded and compiler directives are ignored.

The use of bracketed text is shown in Example 13.3-1.

Brackets may appear within the text if they are matched; i.e., each left bracket must be followed by a matching right bracket, and each right bracket must be preceded by a matching left bracket.

A macro constant definition such as:

```
DEFINE bound = 100;
```

could be written with the same effect using bracketed text as:

```
DEFINE bound = [100];
```

but the former is more efficiently compiled.

```
The macro definition

    DEFINE verOk = [testSkill(2 * skNum,5,15)];

allows a programmer to use

    verOk

to stand for

    testSkill(2 * skNum,5,15)

throughout the scope of the definition.
```

Example 13.3-1. Example of Bracketed Text

13.4. Interactive Definition

A macro equate may omit the "=" and subsequent macro body, in which case the compiler prompts for and reads a line from cmdFile and uses this line to define the body of the macro. For example:

```
DEFINE v1, ..., vn;
```

causes the compiler to write to logFile for each identifier vi:

```
DEFINE Vi =
```

(where Vi is uppercase for vi). It then reads a line from cmdFile. The text:

```
DEFINE Vi = <line read from cmdFile>;
```

is then compiled as if it had appeared in the source file.

Another option is to supply a string constant expression that is written to logFile in place of "DEFINE Vi = ". An example is:

```
DEFINE v1 c1, ..., vn cn;
```

where the ci are string constant expressions. In this case, for each vi, the compiler writes ci instead of the standard "DEFINE Vi = " message.

For example, when the compiler encounters:

```
DEFINE maxNumInput,debug "debugging version (TRUE or FALSE)? ";
```

it first types:

```
DEFINE MAXNUMINPUT =
```

If the user types "10", for example, the effect is the same as if:

```
DEFINE maxNumInput = 10;
```

had occurred in the program.

The compiler then types:

```
debugging version (TRUE or FALSE)?
```

to which the user replies either "TRUE" or "FALSE". If "TRUE" is typed, for example, the effect is the same as if:

```
DEFINE debug = TRUE;
```

had occurred in the program. Thus, this form of macro definition allows the programmer to interact with the compilation.

Any mixture of the various forms of macro equate can occur with the same macro definition, as shown in Example 13.4-1.

```
REDEFINE
    debug    =        TRUE,
    callFoo(i) =      [foo(i,1)],
    version,
    compileAllModules
        "Compile all modules (TRUE or FALSE): ",
    upperBound =      10;
```

Example 13.4-1. Using Various Forms of Macro Equate

It is possible to have an interactive define of a macro header that contains parameters, e.g.:

```
DEFINE xxx(yyy) "xxx(yyy): ";
```

When the compiler prompts for the definition of this macro, the user's response must be bracketed text.

13.5. 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 (see Section 13.1), a parenthesized list of macro arguments (see Section 13.6) separated with commas may appear after the macro identifier. Fewer arguments may be supplied than parameters, in which case the compiler supplies no text (i.e., acts as if an empty pair of brackets were supplied) for each unspecified argument. No parentheses are needed if no arguments are specified.

The macro arguments replace all occurrences of the corresponding parameter identifiers in the macro body, as in Example 13.2-1.

13.6. Macro Arguments

Most macro arguments may consist of the intended text with no special delimiters. But if the macro argument is a text "fragment" (e.g., if it contains unmatched parentheses), then it must be enclosed in brackets. An argument with unmatched brackets is not allowed.

The text of each macro argument starts with the first character (other than the "white space" characters space, tab, eol, or eop) following the previous terminating comma (or the opening left parenthesis of the argument list).

If the first character of a macro argument is not a left bracket, then the text of the argument is terminated with the next comma (or the closing right parenthesis) except that nesting counts are kept of parentheses and brackets; the argument text does not terminate until each nesting count is zero. That is, each time a left parenthesis (bracket) is encountered, the parenthesis (bracket) nesting count is incremented by one and each time a right parenthesis (bracket) is encountered, the appropriate count is decremented by one. The macro argument scan does not terminate until both counts are zero and a comma or right parenthesis is encountered. Trailing characters such as space, tab, eol, and eop are removed from the argument text. Comments are discarded; i.e., if "#" is encountered, the remainder of the line is removed from the argument text.

String constants are treated as a unit; i.e., when a double quote is found, the compiler immediately scans for the end of the string constant (as described in Section 3.5). Parentheses, commas, or brackets that occur in the string constant are not specially processed.

Commas may appear within properly nested parentheses, brackets, or string constants.

If the first character of a macro argument is a left bracket ("["), then the argument is the sequence of characters up to the next matching right bracket (a nesting count, as described above, is kept for brackets, and the argument text terminates when the count is zero). This allows almost arbitrary text to be used as a macro argument; i.e., no attention is paid to parentheses, commas, string quotes, or comments within square brackets.

13.6.1. Repeatable Macro Parameters, $numArgs, $arg, and $sArg

The last parameter of a macro may be declared repeatable:

```
DEFINE foo(a,b,REPEATABLE v) = [...];
```

In a call to such a macro, the arguments that correspond to the repeatable parameter are treated as if they had been enclosed in square brackets, i.e., as if they were a single argument:

```
foo(i,j,k,l,m,n)   =>   foo(i,j,[k,l,m,n])
```

Each occurrence of the repeatable parameter in the macro body is replaced by the bracketed text. Thus, an occurrence of "v" in foo's body expands to "k,l,m,n".

Usually, however, it is desired to deal with one at a time of the arguments passed to the repeatable parameter, not with all of them at once.

Three special-purpose macros are provided for accessing the individual arguments of a repeatable parameter:

- "$numArgs(v)" is the number of arguments passed to repeatable parameter v. For example, in the call to foo above, "$numArgs(v)" is 4 since [k,l,m,n] was passed to v.

- "$arg(v,i)" is the ith argument passed to repeatable parameter v (the first argument is numbered one). For example, "$arg(v,3)" above is m, since m was the third argument in the [k,l,m,n] passed to v. If i is less than 1 or greater than the number of arguments passed to v, $arg expands to no characters, i.e., [].

- "$sArg(v,i)" is like "$arg(v,i)" except that it expands to a string constant containing the text that "$arg(v,i)" would have returned ([] is treated as ""). For example, "$sArg(v,3)" above would be the string constant "m".

$sArg is useful when the text of a macro is to be used as if it were a string constant. With a normal parameter, the effect can be obtained as follows:

```
DEFINE print(a) = [ttyWrite("a" & eol)];
```

Thus:

```
print(Hello)
```

causes "Hello<eol>" to be written. If the parameter a were made repeatable with the same macro body:

```
DEFINE print(REPEATABLE a) = [ttyWrite("a" & eol)];
```

the effect of "print(x,y,z)" is not the same as "print(x); print(y); print(z)" since the former writes "x,y,z<eol>" while the latter writes "x<eol>y<eol>z<eol>". The following:

```
DEFINE print(REPEATABLE a) = [ttyWrite("$arg(a,i)" & eol)];
```

writes "$arg(x,y,z,i)<eol>", since $arg is not recognized inside a string constant. Use $sArg to get the desired behavior for print:

```
DEFINE print(REPEATABLE a) =
    [$FORC i = 1 UPTO $numArgs(a) $DOC
        ttyWrite($sArg(a,i) & eol) ENDC];
```

In this form, "print(x,y,z)" writes "x<eol>y<eol>z<eol>".

In fact, $numArgs, $arg, and $sArg are implemented in a way that allows them to be used with non-repeatable parameters:

- $numArgs really just counts how many macro arguments it has. For example, "$numArgs(a,b,c)" is 3.

- These macros see their arguments after any macro parameters of enclosing macros have been expanded. When used as above, "$numArgs(v)" expands to "$numArgs(k,l,m,n)", since v is [k,l,m,n].

- $arg really just selects its ith argument, where i is its last argument; similarly for $sArg. For example, "$arg(a,b,c,2)" is b. If i is less than 1 or greater than the number of preceding arguments, $arg expands to [] and $sArg expands to "". Thus "$arg(a,b,c,0)" is [] and "$sArg(a,b,c,4)" the null string.

- If $arg (or $sArg) has just one argument, it behaves as if there were a second argument (the index) with value 1; i.e., the first (and only) argument is selected.

A repeatable macro argument is really just a convenience so the programmer does not have to enclose the repeated arguments in brackets. This allows the syntax of a macro to look like that of a procedure, so that the programmer need not know which is really being called. Defining foo as:

```
DEFINE foo(a,b,c) = [...];
```

and then invoking it as:

```
foo(i,j,[k,l,m,n])
```

has the same effect as the above example using a repeatable parameter and no brackets in the call. In particular, $numArgs, $arg, and $sArg can be used to "look inside" any bracketed arguments, not just those declared repeatable.

For example:

```
MODULE xProcs (
    DEFINE defProc(t,p) = [t PROCEDURE p] & [Proc (t parm);];
    $FORC j = 1 UPTO 4 $DOC
        REDEFINE
            x = $arg([INTEGER,i],[LONG INTEGER,li],[REAL,r],
                     [LONG REAL,lr],j);
        defProc($arg(x,1),$arg(x,2)) ENDC
    );
```

expands to:

```
MODULE xProcs (
    INTEGER PROCEDURE iProc (INTEGER parm);
    LONG INTEGER PROCEDURE liProc (LONG INTEGER parm);
    REAL PROCEDURE rProc (REAL parm);
    LONG REAL PROCEDURE lrProc (LONG REAL parm);
    );
```

The following is an example of a "compiletime case expression":

```
DEFINE pdfChrs(a) =
    [cvli($arg(pdfBoChars,pdfiChars,pdfLiChars,pdfrChars,
              pdfLrChars,pdfbChars,pdfLbChars,
              pdfiChars + length(a),pdfaChars,pdfcChars,
              MESSAGE "pdfChrs: unexpected type"; 0,
              $TYPEOF(a)))];
```

Using a parameter a in string quotes ("a") fails if a's argument contains undoubled string quotes; $sArg provides a safe, general way to convert an ordinary macro argument into a string constant:

```
DEFINE foo(a,b) = [...write(f,$sArg([a],1))...];
```

is like:

```
DEFINE foo(a,b) = [...write(f,"a")...];
```

except that the latter is incorrect if the argument to "a" contains undoubled string quotes. The argument a was enclosed in brackets when used as the argument to $sArg because a alone could expand to what appears to be several macro arguments.

13.7. Determining Whether a Macro Argument Has Been Omitted

A bracketed macro parameter identifier used in the governing expression of an "IFC" can be used within a macro definition to determine whether or not the macro parameter is empty (i.e., has been omitted), assuming that the argument passed for the parameter is syntactically correct (if it is not omitted). For example, consider a macro designed to assert a condition that issues a message if the condition is false. It has a default message but allows that message to be overridden. The macro could be defined as:

```
DEFINE assert(condition,msg) = [
    BEGIN
    IF NOT (condition) THEN
    IFC [msg] THENC errMsg(msg)
    ELSEC errMsg("Assertion failed") ENDC END];
```

in which case the calls:

```
assert(x = y);
assert(i > 20,"i is too small")
```

expand to:

```
BEGIN IF NOT (x = y) THEN errMsg("Assertion failed") END;
BEGIN IF NOT (i > 20) THEN errMsg("i is too small") END
```

assert may not work if msg contains, e.g., unmatched "[" or "]", since the parameters are expanded before the macro bodies are parsed.

13.8. Bracketed Text in Constant Expressions

Bracketed text can be used in constant expressions as an operand of "NOT", "AND", "OR", "=", "NEQ", and "&". In the cases of "=", "NEQ", and "&", one operand may be bracketed text and the other bracketed text or a string constant. This use of bracketed text is useful with macro arguments.

```
DEFINE
    m1(a,b) = [IFC NOT [a] OR NOT [b] THENC
               MESSAGE "Missing argument"; ENDC
               ... ],

    m2(a,b) = [IFC [a] = [b] THENC ... ELSEC ... ENDC],

    m3(a,b) = [IFC [a] = "b" THENC ... ELSEC ... ENDC];
```

Example 13.8-1. Bracketed Text Operands

```
    DEFINE a = [id] & "1";

has the same effect as:

    DEFINE a = [id1];

(unless the definition occurs within a macro body, and
"id" is a macro parameter for which text is substituted;
the substitution is made in the first case but not the
second). This capability is useful when macro bodies are
built up from other macros.
```

Example 13.8-2. Concatentation of Bracketed Text and String Constants

In Example 13.8-1, the use of "b" in the definition of m3 instead of [b] fails if the argument b contains unpaired double quotes (e.g., if it is or contains a string constant, although it could be repaired with $sArg as described above). The use of [b] in m2 fails if b is a source text fragment containing unmatched square brackets (whereas "b" would work in that case if the argument contained no unpaired double quotes).

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" is a compiler directive that writes a string at compiletime to a new line of logFile.

The form of a "MESSAGE" directive is "MESSAGE c;" where c is a string constant expression, or "MESSAGE c,c2;", where c is an arbitrary string constant expression and c2 is one of "error" or "warning" (case is ignored). c is written to logFile when the "MESSAGE" directive is encountered during compilation; if c2 is present, then c is included in a warning message if c2 is "warning" or an error message if c2 is "error". For example, to give a compiletime error message if the character set is unknown:

```
IFC $charSet = $ascii THENC ...
$EFC $charSet = $ebcdic THENC ...
ELSEC MESSAGE "Unknown char set","error"; ENDC
```

14.2. "SOURCEFILE"

"SOURCEFILE" directs the compiler to compile another file as if it appeared in place of the directive.

The form of a "SOURCEFILE" directive is "SOURCEFILE c;" where c is a string constant expression that specifies a file name. "SOURCEFILE" causes the compiler to save the state of the current source file (that is, the one it is currently compiling) and then begin compiling the file named by c, as if its text had appeared in place of the directive. When compilation of the file c is complete, compilation resumes immediately following the "SOURCEFILE" directive.

A file that was itself obtained with "SOURCEFILE" may also use "SOURCEFILE" to get additional files; i.e., SOURCEFILE's may be nested.

A sourcefile name may be read from cmdFile by means of an interactive define (see Section 13.4):

```
DEFINE defFile "Name of file with definitions: ";
SOURCEFILE defFile;
```

The "SOURCEFILE" directive may be used to maintain a set of declarations common to a number of modules in a single file that is sourcefiled by all of them.

14.3. "CHECK", "NOCHECK", and "CHECKING"

"CHECK", "NOCHECK", and "CHECKING" govern the generation of code to check certain conditions at runtime that cannot be determined at compiletime. They are described in detail in Chapter 15.

14.4. "$DIRECTIVE"

The directive "$DIRECTIVE" permits certain compiler subcommands and other directives to be specified inside the source text for a module. Its format is:

```
$DIRECTIVE s1,...,sn;
```

where the si are string constants that are the names of compiler subcommands (followed by arguments, if applicable). The case of the the compiler subcommands in si does not matter. The subcommands currently accepted by "$DIRECTIVE" are:

```
ABORT ACHECK ACHECKALL ALIST CHECK CHECKALL CLOSEINTLIB
CLOSEOBJLIB CMDLINE DEBUG GENCODE GENINLINES INCREMENTAL
INOBJFILE INOBJLIB LOG MODTIME MONITOR OPENINTLIB OPENOBJLIB
OPTIMIZE OPTIMIZEALL OUTINTFILE OUTINTLIB OUTOBJFILE
OUTOBJLIB PERMOD PERPROC PERSTMT PROCS PROCTIME REDEFINE
RESPONSE SAVEON UNBOUND NOACHECK NOACHECKALL NOALIST NOCHECK
NOCHECKALL NODEBUG NOGENCODE NOGENINLINES NOINCREMENTAL
NOINOBJLIB NOMONITOR NOOPTIMIZE NOOPTIMIZEALL NOOUTINTLIB
NOOUTOBJLIB NOPROCS NOREDEFINE NORESPONSE NOSAVEON NOUNBOUND
LIBRARY OUTPUT NOLIBRARY NOOUTPUT
```

The "$DIRECTIVE" directives that are not compiler subcommands are:

```
MAKEMODULENOTVISIBLE MAKEMODULEVISIBLE MAKENOTVISIBLE
MAKEVISIBLE OPENMODULE PUSHACHECK PUSHCHECK POPACHECK
POPCHECK
```

"MAKEMODULENOTVISIBLE", "MAKEMODULEVISIBLE", "MAKENOTVISIBLE", "MAKEVISIBLE", and "OPENMODULE" are described in detail in Chapter 11.

"PUSHACHECK", "POPACHECK", "PUSHCHECK" and "POPCHECK" are described in detail in Chapter 15.

"$DIRECTIVE" directives apply only to the current module, i.e., are not sticky.

The directives:

```
$DIRECTIVE "CHECK";
```

and:

```
$DIRECTIVE "NOCHECK";
```

are equivalent to:

```
CHECK;
```

and:

```
NOCHECK;
```

respectively.

The current setting of many "$DIRECTIVE" directives can be examined with the system procedure $compileTimeValue.

14.5. "SAVEON" and "RESTOREFROM"

"SAVEON" and "RESTOREFROM" allow symbols from an intmod to be used during a compilation; see Chapter 11.

14.6. "ENCODE"

The "ENCODE" directive is used with the Foreign Language Interface (FLI, described in the "MAINSAIL Compiler User's Guide") to supply target-dependent names to be written to the generated assembly language file in place of the MAINSAIL procedure identifiers. The FLI ordinarily uses some transformation (as specified in the appropriate operating-system-specific MAINSAIL user's guide) of a MAINSAIL procedure name as the foreign procedure name. In some cases the foreign name cannot be derived from the default transformation. The "ENCODE" directive allows the programmer to supply an arbitrary string as the foreign procedure name.

The form of the "ENCODE" directive is:

```
ENCODE p1 s1, ..., pn sn;
```

where the pi are interface procedure identifiers and the si are string constants. The string si is used as the foreign procedure name corresponding to pi when pi is compiled with the FLI compiler.

In Example 14.6-1, "streamPutRec" is the MAINSAIL procedure identifier used for the foreign procedure "stream_Put$Rec". The FLI code generator outputs "stream_Put$Rec" to the assembly file as the name of the foreign procedure.

```
ENCODE streamPutRec "stream_Put$Rec";
```

Example 14.6-1. Use of the "ENCODE" Directive

The FLI is further described in the "MAINSAIL Compiler User's Guide".

14.7. "$GLOBALREDEFINE"

Sometimes it is useful to carry over information from one compilation to the next (within the same compiler session). This can be accomplished with the keyword "$GLOBALREDEFINE", which introduces a "global macro definition". The syntax and semantics of "$GLOBALREDEFINE" are just like those of "REDEFINE", except that the defined identifiers are entered into a "global symbol table" that persists from one compilation to the next in the same invocation of the compiler. A macro defined in one module can be accessed in a subsequently compiled module.

If an identifier has been defined in a global definition, then a "DEFINE" or "REDEFINE" of the identifier defines or redefines a non-global (local to the current module or procedure) occurrence of the identifier. Subsequent references to the identifier (within the scope of the non-global definition) reference the non-global definition. If the non-global definition is local to a procedure, then after the body of that procedure the global definition is once more visible.

Compiler subcommands are available for doing global redefinitions and also removing identifiers from the global symbol table.

The module shown in Example 14.7-1 compiles into an indefinite number of empty modules named FOO1, FOO2, FOO3, etc.

```
IFC NOT DCL(modNum) THENC
$GLOBALREDEFINE modNum = 1;
ELSEC
$GLOBALREDEFINE modNum = modNum + 1;
ENDC

$GLOBALREDEFINE modName = "foo" & cvs(modNum);

BEGIN modName

END modName

SOURCEFILE $thisFileName;
```

Example 14.7-1. Generating an Aribitrary Number of Empty Modules with "$GLOBALREDEFINE"

14.8. "DSP"

"DSP" is a compiletime pseudo-procedure that returns as an integer the displacement to a field of a class or module. It takes a single argument of the form "class.field" or "module.field". See Example 14.8-1. XIDAK reserves the right to change the layout of record fields in memory.

```
    CLASS cell
        (POINTER(cell) left,right;
         STRING name;
         INTEGER value)

Suppose pointers and integers each require 1 storage unit,
and strings require 2.  Then:

    DSP(cell.left)      is 0
    DSP(cell.right)     is 1
    DSP(cell.name)      is 2
    DSP(cell.value)     is 4
```

Example 14.8-1. Use of "DSP"

14.9. "$LEGALNOTICE"

The "$LEGALNOTICE" directive causes a "legal notice" to be put into the objmod and/or intmod for the module being compiled. A legal notice is typically a legal notification such as a copyright or trade secret paragraph. The compiler directive:

```
$LEGALNOTICE s;
```

where s is any string constant expression, is used to specify a legal notice. Only the first "$LEGALNOTICE" text encountered in a module's source text is put into the output file(s). "$LEGALNOTICE" texts from referenced intmods are not put into the object file; it must appear in the source file when it is compiled. A semicolon is required after s.

14.10. Conditional Compilation: "IFC", "THENC", "$EFC", "ELSEC", and "ENDC"

Conditional compilation allows the programmer to specify under what conditions indicated parts of the source file are to be compiled or ignored.

The conditional:

```
IFC c THENC text1 ENDC
```

causes the compiler to compile text1 if c (a constant expression evaluable at compiletime; see Section 2.5) is non-Zero, and to ignore text1 otherwise. text1 is any source text that is valid (e.g., statement(s), declaration(s), macro definition(s)) where the conditional appears. "THENC" separates the condition c from the text to be conditionally compiled, and "ENDC" marks the end of that text.

A sample conditional is:

```
IFC doDebug THENC ttyWrite("current value is ",val); ENDC
```

The "ttyWrite ..." is compiled if doDebug is non-Zero; otherwise, it is ignored.

The general conditional form is:

```
IFC c1 THENC text1
{$EFC c2 THENC text2
 {$EFC c3 THENC text3
  ...
  {$EFC cm THENC textm}}}
{ELSEC textn}
ENDC
```

The ci are constant expressions evaluable at compiletime. If c1 is non-Zero, then text1 is compiled; otherwise, if c2 is present and non-Zero, then text2 is compiled; otherwise, if c3 is present and non-Zero, then text3 is compiled, etc. If none of the ci is non-Zero, and "ELSEC" is present, then textn is compiled. The texti that are not compiled are ignored.

"Ignored" text is really scanned (except that macros are not expanded, and compiler directives are ignored) but not parsed, which means that basic constructs such as constants must be properly constructed, and IFC's, $EFC's, ELSEC's, and ENDC's must be properly matched. Otherwise, the text need not be syntactically correct.

IFC's may be nested to any depth. That is, constructs such as that shown in Example 14.10-1 are allowed. The "C" in "IFC", "$EFC", "THENC", "ELSEC", and "ENDC" stands for "conditional".

```
IFC c1 THENC
    ...
    IFC c2 THENC text1 ELSEC text2 ENDC
    ...
ELSEC text3 ENDC
```

Example 14.10-1. Nested IFC's

In Example 14.10-1, if c1 is Zero, then text3 is compiled (the rest of the conditional is scanned, but not parsed). If c1 and c2 are both non-Zero, then text1 is compiled. If c1 is non-Zero but c2 is Zero, then text2 is compiled.

14.11. "$CASEC": Compiletime Case

"$CASEC" provides for compiletime case selection, and is useful as a shorthand for some forms of "IFC" directives. The syntax is similar to that of the Case Statement:

```
$CASEC x OF
    [x1]          text1
    [x2 TO x3]    text2
    [x4][x5]      $BEGINC text3 ENDC
    [ ]           text4
    ENDC
```

"OFB" may be used in place of "OF" (it does not take a matching "END"); "OFB" and "OF" are therefore exactly equivalent in this context.

14.11.1. Selectors

The case selection expression x is a constant expression of any data type; it need not be an integer as it would for a Case Statement. All the expressions used in the case selectors (the xi) must be of the same data type as x's.

The bracketed selectors are evaluated in the order encountered, and as soon as one occurs that matches (see Section 14.11.2) the selection expression, the corresponding text is gathered into a string (and so must not exceed the maximum string length), the remainder of the "$CASEC" directive (down to the terminating "ENDC") is discarded, and then the gathered text is compiled. The selected text is terminated by a "bare" left bracket (see Section 14.11.3) or "ENDC".

The catch-all selector "[]" matches any selection expression. All selections after it are ignored, so it should come last (this differs from the treatment of the catch-all selector in the Case Statement).

14.11.2. Selector Matching Rules

A selector of the form "[a]" matches x if "x = a", i.e., if x and a evaluate to the same value.

A selector of the form "[a TO b]" matches x depending on the data type of x, a, and b (all must be the same type) according to the rules shown in Table 14.11.2-1.

14.11.3. Delimiters of Selected Text

The selected text (the texti above) is arbitrary text to be compiled. If it contains a "bare" left bracket ("[" or "{"), i.e., not inside of a string constant ("...[..."), character constant ('['), or comment (#...[...<eol>), the left bracket (and optionally surrounding text) must be enclosed in a "$BEGINC"-"ENDC" pair. For example:

Type of x,a,b	[a TO b] matches x if
boolean	(IF a THEN 1 EL 0) LEQ (IF x THEN 1 EL 0) LEQ (IF b THEN 1 EL 0)
(long) integer (long) real string	a LEQ x LEQ b
(long) bits	cvli(a) LEQ cvli(x) LEQ cvli(b)
pointer address charadr	always matches since the only constant for these types is Zero. $CASEC's of these types are not very useful

The above computations are carried out using the host machine characteristics; avoid "$CASEC" if cross-compiling with non-portable values for x or any xi.

Table 14.11.2-1. "$CASEC" Selector Matching Rules

```
$CASEC i OF
    [...]       ...
    [bitsCode]  $BEGINC b := bAry[k]; ENDC
    [...]       ...
    ENDC
```

This causes the compiler not to treat the left bracket as the start of the next selector. If the "$BEGINC"-"ENDC" pair had been left out of the above example, the compiler would have gathered "b := bAry" as the text corresponding to [bitsCode], and considered [k] to be the next selector (with just ";" as its selected text), which would probably have caused a syntax error.

The gathering of the selected text applies to the text AFTER any macro parameters have been replaced with their arguments. If any of the text contains macro parameters which could possibly be passed an argument with a bare left bracket, the text (or at least each such macro parameter) must be enclosed in a "$BEGINC"-"ENDC" pair (it never hurts to enclose the text in a "$BEGINC"-"ENDC" pair).

14.12. "$BEGINC"

Matched "$BEGINC"-"ENDC" pairs are useful within the selected text governed by "$CASEC". In other places, matched "$BEGINC"-"ENDC" pairs are permitted, but have no effect.

14.13. "$DOC", "$DONEC", "$CONTINUEC", "$FORC": Compiletime Iteration

14.13.1. "$DOC iteratedText ENDC"

Text between "$DOC" and its terminating "ENDC" is repeatedly compiled until the loop is terminated, e.g., by "$DONEC". The compiler gathers the iterated text into a single string, and hence it cannot exceed the maximum string length. The user must take care to avoid putting the compiler into an infinite loop.

14.13.2. "$DONEC" and "$CONTINUEC"

"$DONEC" and "$CONTINUEC" are the compiletime analogues of the Done and Continue Statements.

"$DONEC" may be used in a "$DOC" body to terminate the iterations. The compiler discards the remainder of the loop body and resumes compilation beyond the "ENDC" matching the terminated "$DOC".

"$CONTINUEC" may be used in a "$DOC" body to continue with the next iteration. The compiler resumes compilation at the top of the loop body (immediately after the "$DOC").

By default, "$DONEC" and "$CONTINUEC" apply to the innermost "$DOC" loop. A string constant name may be associated with a "$DOC" and used by "$DONEC" or "$CONTINUEC" within the "$DOC" body to terminate or continue the named loop. For example:

```
$DOC("outer") ...
$DONEC("outer") ...
$CONTINUEC("outer") ...
ENDC
```

The name must be enclosed in parentheses, and the left parenthesis must immediately follow the keyword (with no intervening text, not even white space; otherwise, the parenthesized name is considered part of the iterated text). Unlike END's associated with named DO's, a name cannot be associated with a matching "ENDC". Specifying the null string is equivalent to not

specifying a name, so if used with a "$DONEC" or "$CONTINUEC", it specifies the innermost loop.

Examples:

```
DEFINE i = 1;
$DOC write(f,i); REDEFINE i = i + 1;
     IFC i > 5 THENC $DONEC ENDC ENDC

DEFINE i = 1;
$DOC("outer")
    ...
    $DOC write(f,i); REDEFINE i = i + 1;
        IFC i > 5 THENC $DONEC("outer") ENDC ENDC
    ... ENDC
```

14.13.3. "$FORC"

In the form:

```
$FORC var = start UPTO/DOWNTO stop $DOC ... ENDC
```

var is an iteration identifier that is redefined as described below, start is a (long) integer constant expression, and stop is a constant expression of the same type as start.

Each form is expanded by the compiler to an "equivalent" form, as shown below:

```
$FORC var = start UPTO stop $DOC ... ENDC
REDEFINE var = <start-1>; # ($FORC expansion)
$DOC REDEFINE var = var + 1; # ($FORC expansion)
     IFC var > stop THENC $DONEC ENDC # ($FORC expansion)
     ... ENDC

$FORC var = start DOWNTO stop $DOC ... ENDC
REDEFINE var = <start+1>; # ($FORC expansion)
$DOC REDEFINE var = var - 1; # ($FORC expansion)
     IFC var < stop THENC $DONEC ENDC # ($FORC expansion)
     ... ENDC
```

If start and stop are long integers, the compiler uses "1L" in place of "1".

The expanded form is compiled in place of the original text. Any errors in the expanded form show the expanded text as if it were in the source file. The comments are included to help the

user recall that the text has been created by the compiler, should the text be shown in an error message.

The first example of the previous section can be rewritten as follows:

```
$FORC i = 1 UPTO 5 $DOC write(f,i); ENDC
```

An example using "$FORC", "$CONTINUEC", "$DONEC" and nested loops:

```
$FORC i = 1 UPTO 2 $DOC("outer")
    $FORC j = 1 UPTO 5 $DOC
        IFC j DIV 2 THENC $CONTINUEC; ENDC
        ...
        IFC ... THENC $DONEC("outer") ENDC
        ... ENDC
    ENDC
```

14.14. "DCL"

"DCL" is a compiletime pseudo-procedure. "DCL(identifier)" is true if the identifier has been declared or defined (by the programmer, or as a standard MAINSAIL identifier), and false otherwise. "DCL" is useful in conjunction with conditional compilation. For example:

```
IFC NOT DCL(switch) THENC DEFINE switch = FALSE; ENDC
```

If switch has been declared or defined then "DEFINE switch = FALSE;" is ignored; otherwise, it is compiled.

14.15. "$TYPEOF"

"$TYPEOF(x)" returns the integer constant type code (one of the values shown in Table 14.15-1) of the expression, class name, or module name x. The compiler parses x, determines the type of the result, then discards the resulting parse information (x is not actually evaluated).

booleanCode	integerCode	longIntegerCode	realCode
longRealCode	bitsCode	longBitsCode	stringCode
addressCode	charadrCode	pointerCode	$classCode
$moduleCode			

Table 14.15-1. Type Codes As Returned by "$TYPEOF"

```
CLASS cls (STRING s; POINTER(c) link);
MODULE          m;
INTEGER         t,u;
POINTER(cls)    p;

Expression                  Result
$TYPEOF(t)                  integerCode
$TYPEOF(t + u)              integerCode
$TYPEOF(p)                  pointerCode
$TYPEOF(p.s)                stringCode
$TYPEOF(p.link)             pointerCode
$TYPEOF(sin(1.))            realCode
$TYPEOF(cls)                $classCode
$TYPEOF(m)                  $moduleCode

# increment an integer or long integer
DEFINE inc(a) =
        [a .+ IFC $TYPEOF(a) = integerCode THENC 1
              ELSEC 1L ENDC];
```

Example 14.15-2. Sample "$TYPEOF" Values

If x is an array, "$TYPEOF(x)" is the base type of the array, i.e., the type of the elements (0 if untyped). If x is a class or module name, "$TYPEOF" returns $classCode or $moduleCode, respectively.

14.16. "$CLASSOF"

"$CLASSOF(x)" returns the string constant class name (upper case) for the expression x if x evaluates to a classified pointer or address; otherwise, it returns the null string. The compiler parses x as an expression, determines the class name of the result (if a classified pointer or address), then discards the resulting parse information (x is not actually evaluated).

14.17. "$ISCONSTANT"

"$ISCONSTANT(x)" returns true if and only if the expression x evaluates to a constant. The compiler parses x as an expression, determines whether it evaluates to a constant, then discards the resulting parse information (x is not actually evaluated).

```
CLASS cls (STRING s; POINTER(c) link);
MODULE          m;
INTEGER         t,u;
POINTER(cls)    p;

Expression                  Result
$CLASSOF(t)                 "" (null string)
$CLASSOF(t + u)             ""
$CLASSOF(p)                 "CLS"
$CLASSOF(p.s)               ""
$CLASSOF(p.link)            "C"
$CLASSOF(sin(1.))           ""

CLASS rec1 (...; POINTER(rec1) nextRec1);
CLASS rec2 (...; POINTER(rec2) nextRec2);

DEFINE nextRec(p) = # make it work for rec1 and rec2
        [p := IFC $CLASSOF(p) = "REC1" THENC p.nextRec1
              ELSEC p.nextRec2 ENDC];
```

Example 14.16-1. Sample "$CLASSOF" Values

For example, suppose calls to the local procedure fooProc (one argument) are desired to be inline if the argument is a constant. Define foo as below, then use "foo" instead of "fooProc":

```
DEFINE foo(x) =
    [IFC $ISCONSTANT(x) THENC INLINE ENDC
        fooProc(x) ENDC];
```

14.18. Scanning Directives

The scanning directives are "BEGINSCAN", "SKIPSCAN", and "DONESCAN".

"SKIPSCAN" allows the compiler to skip quickly over pages in the source file. The form of a "SKIPSCAN" directive is:

```
SKIPSCAN c;
```

where c is a string constant "scan name". This directive causes the compiler to begin skipping pages until it finds one that starts with the keyword "BEGINSCAN" followed by the same scan

name. Compilation then resumes on that page. Pages are delimited by eop characters (see Section 2.1). Upper and lower case are not distinguished in examining the scan name.

If c is Zero, i.e., if:

```
SKIPSCAN "";
```

is encountered, the compiler stops at the next "BEGINSCAN c", regardless of the value of c.

Macros are not expanded within text skipped over by "SKIPSCAN", and compiler directives (other than a matching "BEGINSCAN") are ignored.

"BEGINSCAN" serves as a stopping point for a "SKIPSCAN" search.

The form of a "BEGINSCAN" directive is:

```
BEGINSCAN c;
```

where c is a string constant "scan name". It is used by the "SKIPSCAN" search to determine whether or not the search should stop at this "BEGINSCAN".

"BEGINSCAN" must appear as the very first text on a page, not even preceded by blank or tab; this allows the "SKIPSCAN" search to be fast since it need examine only the first line of each page. The compiler ignores the "BEGINSCAN" directive except during a "SKIPSCAN" search.

If c is Zero, the "BEGINSCAN" directive stops any "SKIPSCAN" search; i.e.:

```
BEGINSCAN "";
```

stops any "SKIPSCAN" search.

"DONESCAN" terminates compilation of the current file as if the end of the file were reached. It may be used to return from a sourcefile of a file that is being used as a repository for several sources.

The form of a "DONESCAN" directive is:

```
DONESCAN;
```

The scanning directives may be used to compile modules selectively in a file that contains more than one module. That is, proper use of the scanning directives can direct the compiler to compile just some of the modules in the file, depending on which ones are specified by the user.

14.19. "NEEDBODY" and "NEEDANYBODIES"

The compiletime pseudo-procedures "NEEDBODY" and "NEEDANYBODIES" are used in conjunction with the "FORWARD" qualifier (see Section 9.10) 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. "NEEDBODY" may also be used to determine whether an interface procedure (whether called or not) needs a body.

The form:

```
NEEDBODY(id)
```

is true if and only if id is the name of a procedure that either has been declared forward and has appeared in a procedure call or is an interface procedure, but has not been declared with a body.

"NEEDANYBODIES" has two forms, one followed by a parenthesized file name, and one not. The form:

```
NEEDANYBODIES(c)
```

where c is a string constant expression for a file name, is true if and only if some procedure p was declared with the qualifier "FORWARD(c)", and "NEEDBODY(p)" is currently true. In other words, "NEEDANYBODIES(c)" tells whether 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 as explained in Section 9.10. The implied c is the "top-level" file that was sourcefiled, not necessarily the current file (additional "SOURCEFILE" directives may occur in the top-level file).

The $compileTimeValue argument "HASBODY" may be used to determine whether a procedure has been given a body, regardless of how the procedure was declared.

14.20. $compileTimeValue

$compileTimeValue is a compiletime procedure that provides a number of miscellaneous compiletime facilities. It is described in detail in Section 1.69 of part II of the "MAINSAIL Language Manual".

14.21. $def

The macro $def is an aid to defining a consecutive sequence of (long) integers, or a sequence of (long) bits each shifted one more bit to the left than the previous value. "$def(a,v)" redefines a to be v, and then:

- if v is a (long) integer, redefines v to be "v + 1(L)"
- if v is a (long) bits, redefines v to be "v SHL 1"

The user must define v's starting value before using $def.

Example:

```
# define a = 1, b = 2, c = 3, d = 4

DEFINE v = 1;
$def(a,v)
$def(b,v)
$def(c,v)
$def(d,v)
```

The second argument, v, may be omitted, in which case the identifier $defVal is used in place of v. $defVal is an identifier which has been set aside for this purpose. In this case, the user must initialize $defVal with "REDEFINE" since it may already be defined at the start of compilation. It is expected that most uses of $def will omit the second argument and use $defVal; the second argument would be useful, however, if multiple sequences need to be defined in parallel.

Example:

```
# define a = '1, b = '2, c = '4, d = '10

REDEFINE $defVal = '1;
$def(a)
$def(b)
$def(c)
$def(d)
```

"REDEFINE" allows the type of a macro to be changed; for example, if $defVal is defined to be an integer constant, then:

```
REDEFINE $defVal = '1L;
```

changes it to be a long bits constant.

There is nothing "magic" about $def; i.e., it could just as easily be provided by the user. $def is defined as follows:

```
DEFINE
    $def(a,v) =
        [IFC [v] THENC
            REDEFINE
                a = v,
                v = IFC $TYPEOF(v) = integerCode THENC v + 1
                    $EFC $TYPEOF(v) = longIntegerCode THENC
                                v + 1L
                    ELSEC v SHL 1 ENDC;
        ELSEC
            REDEFINE
                a = $defVal,
                $defVal =
                    IFC $TYPEOF($defVal) = integerCode THENC
                        $defVal + 1
                    $EFC $TYPEOF($defVal) = longIntegerCode
                          THENC $defVal + 1L
                    ELSEC $defVal SHL 1 ENDC;
        ENDC];
```

The exact definition of $def is subject to change; the above is presented only as an example.

15. Optimization and Checking

Optimization and checking govern the quality of object code produced and the amount of checking of various runtime error conditions. Facilities are provided to optimize or check whole modules or individual procedures, and to check parts of procedures. The facilities for arithmetic checking parallel the standard checking facilities.

15.1. Optimization

Optimization causes the compiler to use a variety of code improvement strategies to emit better code for MAINSAIL objmods. Optimized code takes longer to compile than non-optimized code, but usually runs faster, sometimes significantly faster. Optimization may be governed by compiler subcommands (see the "MAINSAIL Compiler User's Guide") or, with finer control, by directives in a source module.

At any point in a module, the current default optimization status, optimizeStatus, is given by:

```
$compileTimeValue("OPTIMIZE")
```

It may have one of four values at any given point in a compilation:

```
"OPTIMIZE"          "NOOPTIMIZE"
"OPTIMIZEALL"       "NOOPTIMIZEALL"
```

Each of these four values is also a "$DIRECTIVE" directive. An optimization directive has different effects depending on whether it is specified as a compiler subcommand, in a module outside a procedure body, or within a procedure body. See Tables 15.1-1 and 15.1-2.

At the start of each compilation optimizeStatus is set according to the compiler subcommands in effect (the initial default is "NOOPTIMIZE").

Once "OPTIMIZEALL" or "NOOPTIMIZEALL" is in effect, any compiler directives that set optimizeStatus are ignored. This allows an "OPTIMIZEALL" or "NOOPTIMIZEALL" subcommand to override any directives in the source text.

After the compiler parses the entire module, it does the following for each procedure for which a body is to be compiled into the module:

- If optimizeStatus is "OPTIMIZEALL", mark it to be optimized (and ignore the lists).

"$DIRECTIVE" Directive or Subcommand	Effect
OPTIMIZEALL	optimizeStatus := "OPTIMIZEALL"
NOOPTIMIZEALL	optimizeStatus := "NOOPTIMIZEALL"
OPTIMIZE	optimizeStatus := "OPTIMIZE"
NOOPTIMIZE	optimizeStatus := "NOOPTIMIZE"
OPTIMIZE p1 ... pn	add the pi to the list of procs to be optimized, and if necessary, remove them from the list of procs not to be optimized
NOOPTIMIZE p1 ... pn	add the pi to the list of procs not to be optimized, and if necessary, remove them from the list of procs to be optimized

Table 15.1-1. Effects of Optimization Directives outside Any Procedure Body or Specified as a Compiler Subcommand

"$DIRECTIVE" Directive	Effect
OPTIMIZEALL	error (not allowed inside a procedure)
NOOPTIMIZEALL	error (not allowed inside a procedure)
OPTIMIZE	same as "OPTIMIZE p"
NOOPTIMIZE	same as "NOOPTIMIZE p"
OPTIMIZE p1 ... pn	same as outside p
NOOPTIMIZE p1 ... pn	same as outside p

Table 15.1-2. Effects of Optimization Directives inside a Procedure p

- If optimizeStatus is "NOOPTIMIZEALL" then mark it not to be optimized (and ignore the lists).
- If optimizeStatus is anything else:
    - If it is on the list of procedures to be optimized, mark it to be optimized.

- If it is on the list of procedures not to be optimized, mark it not to be optimized.

- If it is on neither list, don't mark it (or don't change the marking if it has been brought in from an intmod and is already marked).

An intmod records each procedure's marking. The default value of optimizeStatus used during an incremental recompilation of a procedure is the marked value of the procedure, if the procedure is marked; otherwise, the value of optimizeStatus as of the end of the original compilation is the default.

When code is generated for a procedure, it is not optimized if the procedure is debuggable. Otherwise, it is optimized or not, as marked; if unmarked, it is optimized if and only if the value of optimizeStatus at the end of the module was "OPTIMIZE".

At present, a procedure expanded inline is optimized if and only if the procedure in which it is expanded is optimized. This is subject to change.

15.1.1. $compileTimeValue("OPTIMIZE")

As noted above, the current value of optimizeStatus is given by:

```
$compileTimeValue("OPTIMIZE")
```

The expression:

```
$compileTimeValue("OPTIMIZE p")
```

where p is a procedure name, yields the value given by:

```
IF  optimizeStatus = "OPTIMIZEALL" OR
    optimizeStatus NEQ "NOOPTIMIZEALL" AND
        (<p is on the list of procs to be optimized> OR
         <p is not on the list of procs not to be
            optimized> AND optimizeStatus = "OPTIMIZE"))
    THEN "TRUE" EL ""
```

The value of "$compileTimeValue("OPTIMIZE p")" does not necessarily indicate whether p really will be optimized since the value can vary from one point in the module to another as the values it depends on vary.

MODLIB and INTLIB directory commands display the "O" option if the value of optimizeStatus at the end of the compilation was "OPTIMIZE" or "OPTIMIZEALL".

15.2. Checking

The standard checking directives cause the compiler to emit code to issue error messages for certain runtime conditions that cannot be determined at compiletime. This causes more code to be generated, and thus results in slower execution. Sections of code on which such checking is performed are said to have "checking in effect". It can be very difficult to track bugs in code where checking is not in effect; error messages are suppressed, and the error condition has undefined effects.

The conditions checked when checking is in effect are that:

1. Array subscripts are within bounds.

2. An array used in a subscripted variable is not Zero.

3. A pointer used as the base part of a field variable is not Zero.

4. Linkage has been established to a module of which an interface variable is indirectly accessed (see Section 10.7).

Checking is controlled by the directives "CHECK" and "NOCHECK", the compiletime pseudo-procedure "CHECKING", and by several "$DIRECTIVE" directives.

Checking status is governed by two values, checkingStatus and localCheckingStatus, which interact to determine what code is checked and what is not. Checking is specifiable at the expression level. checkingStatus governs checking throughout a module, whereas the directives affecting the value of localCheckingStatus apply only within a procedure.

Checking directives may surround a subscripted variable, preceding the array expression and following the closing square bracket, e.g.:

```
IF CHECK; a[i] NOCHECK; THENB ...
```

(in which case only "a[i]" is checked) or a field variable, preceding the pointer or module and following the last field, e.g.:

```
IF NOCHECK; p.f.g CHECK; THENB ...
```

(in which case only "p.f.g" is not checked). The effect of a checking directive placed within a subscripted or field variable (e.g., "a NOCHECK; [i] CHECK;") is undefined. Checking directives may appear anywhere in the source between subscripted variables or field variables; it is often convenient to place them between statements or procedures.

checkingStatus indicates the default for checking. It can have one of four values at any given point in a compilation:

"CHECK"	"NOCHECK"
"CHECKALL"	"NOCHECKALL"

Each of these four values is also a "$DIRECTIVE" directive. At the start of each compilation checkingStatus is set according to any compiler subcommands in effect (the initial default is "CHECK").

localCheckingStatus indicates the current local (to a procedure) default for checking. It can have one of three values at any given point in a procedure:

"CHECK"	"NOCHECK"	""

localCheckingStatus is set to the null string at the start of compilation and at the end of each procedure (so it is always "" outside of a procedure).

As a compilation proceeds, individual expressions as well as the procedures containing them may be marked with a checking status. Each expression is marked when encountered with the current value of localCheckingStatus:

- The expression is marked to be checked if localCheckingStatus is "CHECK".
- The expression is marked not to be checked if localCheckingStatus is "NOCHECK".
- The expression is not marked if localCheckingStatus is the null string.

Procedures are marked if they contain "CHECKALL" or "NOCHECKALL" (the last value specified is the value used, if conflicting directives are specified), or if checkingStatus is "CHECKALL" or "NOCHECKALL" at the end of the module, as described below. checkingStatus determines, at the end of the compilation, how code is generated for unmarked procedures and expressions.

A checking directive has different effects depending on whether it is specified as a compiler subcommand, in a module outside a procedure body, or within a procedure body. See Tables 15.2-1 and 15.2-2. The directives "CHECK" and "NOCHECK" are equivalent to "$DIRECTIVE("CHECK")" and "$DIRECTIVE("NOCHECK")", respectively.

There is a stack of localCheckingStatus values, the localCheckingStatus stack, that is cleared at the beginning of each procedure. It may be used within a procedure to preserve the localCheckingStatus; the recommended way to set or clear checking temporarily within a procedure body without affecting later code is:

"$DIRECTIVE" Directive or Subcommand	Effect
CHECKALL	checkingStatus := "CHECKALL"
NOCHECKALL	checkingStatus := "NOCHECKALL"
CHECK	checkingStatus := "CHECK"
NOCHECK	checkingStatus := "NOCHECK"
DEFAULTCHECK	no effect as directive; not allowed as subcommand
PUSHCHECK	no effect as directive; not allowed as subcommand
POPCHECK	no effect as directive; not allowed as subcommand

Table 15.2-1. Effects of Optimization Directives outside Any Procedure Body or Specified as a Compiler Subcommand

"$DIRECTIVE" Directive	Effect
CHECKALL	mark this procedure to be checked
NOCHECKALL	mark this procedure not to be checked
CHECK	localCheckingStatus := "CHECK"
NOCHECK	localCheckingStatus := "NOCHECK"
DEFAULTCHECK	localCheckingStatus := ""
PUSHCHECK	push localCheckingStatus onto stack
POPCHECK	pop stack into localCheckingStatus

Table 15.2-2. Effects of Checking Directives inside a Procedure p

```
$DIRECTIVE "PUSHCHECK","{NO}CHECK";
<code to be affected by above "CHECK" or "NOCHECK">
$DIRECTIVE "POPCHECK";
```

It is an error to use "POPCHECK" if the localCheckingStatus stack is empty.

Once checkingStatus is set to "CHECKALL" or "NOCHECKALL", any compiler directives that set checkingStatus are ignored. This allows a "CHECKALL" or "NOCHECKALL" subcommand to override any directives in the source text.

Before code is generated for the module, the following occurs:

- If checkingStatus is "CHECKALL", mark each procedure to be checked.
- If checkingStatus is "NOCHECKALL", mark each procedure not to be checked.

Code is generated for each procedure according to its marking, or, if the procedure is unmarked, code is generated for the individual expressions in accordance with their markings. If neither the expression nor its procedure is marked, then the value of checkingStatus determines whether checking is performed; checking is performed if checkingStatus is "CHECK", not performed if it is "NOCHECK".

The marking of procedures and expressions is recorded in intmods, so that if a procedure is pulled in from an intmod, it is marked as it was when originally compiled. The final value of checkingStatus is recorded in an intmod and is used as the initial value of checkingStatus if the intmod is used for incremental recompilation.

Inline procedures are marked just like closed procedures. When an inline procedure is expanded during code generation, the checking code generated in the inline procedure is independent of the checking status of the calling procedure.

15.2.1. $compileTimeValue("CHECKINGSTATUS"), $compileTimeValue("LOCALCHECKINGSTATUS"), and "CHECKING"

checkingStatus is returned by:

```
$compileTimeValue("CHECKINGSTATUS")
```

localCheckingStatus is returned by:

```
$compileTimeValue("LOCALCHECKINGSTATUS")
```

The compiletime pseudo-procedure "CHECKING" returns the value given by:

```
checkingStatus = "CHECKALL" OR
checkingStatus NEQ "NOCHECKALL" AND
    (<current procedure is marked to be checked> OR
     <current procedure is not marked not to be checked> AND
     (localCheckingStatus = "CHECK" OR
      localCheckingStatus = "" AND checkingStatus = "CHECK"))
```

if in a procedure, or:

```
checkingStatus = "CHECKALL" OR
checkingStatus NEQ "NOCHECKALL" AND checkingStatus = "CHECK"
```

outside of a procedure.

"CHECKING" does not necessarily indicate whether checking is done at the point of call since checkingStatus can change before the end of the compilation, and only its final value affects the generated code.

MODLIB and INTLIB directory commands show the "C" option if the final value of checkingStatus is "CHECK" or "CHECKALL".

15.3. Arithmetic Checking

Arithmetic checking enables the generation of extra code to detect arithmetic overflow in (long) integer operations. On some processors, no extra code is required to detect certain kinds of (long) integer overflow; on such machines, an overflow exception may be generated whether or not the "ACHECK" subcommand was in effect when the overflowing code was compiled. Where extra instructions are required to detect overflow, arithmetically checked code may be significantly larger and slower.

Arithmetic checking facilities exactly parallel the standard runtime checking facilities, except that:

- "CHECK" in "$DIRECTIVE" directives and $compileTimeValue arguments is replaced by "ACHECK",
- the default for arithmetic checking is "NOACHECK",
- and there are no equivalents of the non-$DIRECTIVE directives "CHECK" and "NOCHECK" or of the "CHECKING" compiletime pseudo-procedure.

Specifically, the "$DIRECTIVE" directives for arithmetic checking are:

ACHECKALL	NOACHECKALL
ACHECK	NOACHECK
DEFAULTACHECK	
PUSHACHECK	POPACHECK

Arithmetic checking directives may apply to individual expressions. The effect is undefined unless the expression affected is surrounded by parentheses:

```
IF $DIRECTIVE "ACHECK"; (a .+ b) $DIRECTIVE "NOACHECK";
    THENB ...
```

MODLIB and INTLIB directory commands show the "A" option if the final value of acheckingStatus is "ACHECK" or "ACHECKALL".

16. Exceptions

An exception is an unusual or erroneous condition that occurs during a program's execution. When an exception occurs, it causes the execution of a statement called an exception handler (see Section 16.1). 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.

Table 16-1 lists system procedures, variables, and macros for dealing with exceptions.

Identifier	Function
$raise	Raise an exception
$raiseReturn	Return from $raise, if possible
$exceptionName	Name of current exception
$exceptionPointerArg	Pointer argument to $raise for current exception
$exceptionStringArg1	First string argument to $raise for current exception
$exceptionStringArg2	Second string argument to $raise for current exception
$exceptionCoroutine	Raiser coroutine of current exception
$exceptionBits	Bits describing current exception
$newException	New name for an exception
errMsg	Write an error message and/or raise an exception
$registerException	Register an exception so that it can be raised from errMsg
$deregisterException	Undo $registerException

Table 16-1. System Procedures, Variables, and Macros for Exceptions

In order to be an exception, a condition must have been caused by the operations being performed when the condition occurred. Asynchronous interrupts and events occurring in other

tasks or processes are not considered to be exceptions, and are not dealt with by the MAINSAIL exception mechanism.

Exceptions are divided into two categories: those predefined by MAINSAIL (see Section 16.8) 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.

Exceptions are denoted by strings. Section 16.7 describes conventions supported by MAINSAIL to help ensure that exception names are unique.

A program can "register" its exceptions with MAINSAIL using the system procedure $registerException. The system procedure errMsg allows a user, in response to the "Error response:" prompt, to obtain a list of the exceptions that have been registered and to raise any registered exception.

By default, the system procedure errMsg raises a predefined exception before writing its message to logFile. This may cause the execution of a handler that recovers from the exception by aborting the call to errMsg, in which case the message is not written to logFile. If no such handler is found, the message is written as if no exception had been raised.

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
```

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

The abbreviations "$HANDLEB" for "$HANDLE BEGIN" and "$WITHB" for "$WITH BEGIN" are provided.

When a Handle Statement is executed, its statement s1 is initiated. If an exception occurs during s1's execution (which may involve 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 (unless the exception disallows it), 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 an explicit 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 an explicit 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 using 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, of which the execution 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; see Section 16.6.

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. The next handler is the handler associated with the most recently executed Handle Statement the s1 part of which began execution before the current Handle Statement and which has not yet completed. 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 $exceptionName. The current exception is the exception that occurred most recently for which no handler has yet been found or for which the handler is still executing. Other information about the exception (e.g., whether or not the handler can resume execution at the place where the exception occurred) can be obtained by calling $exceptionBits.

When an exception is caused by means of the system procedure $raise, more information about the exception can be passed by means of the parameters exceptionStringArg1, exceptionStringArg2, and exceptionPointerArg. A handler can access the values passed for these parameters as $exceptionStringArg1, $exceptionStringArg2, and $exceptionPointerArg, respectively.

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 (as shown in Example 16.5-1), the execution of the previous handler's Handle Statement is also aborted.

```
$HANDLE
    $HANDLE $raise("first exception")
    $WITH $raise("second exception")
$WITH;  # Abort both Handle statements
```

Example 16.5-1. Sample Nested Handle Statements

16.6. Aborting Procedures

Immediately before MAINSAIL aborts the execution of one or more procedures, it raises the predefined exception denoted by the identifier $abortProcedureExcpt. This exception differs from other exceptions in the following ways:

- MAINSAIL initiates handlers only in procedures that are about to be aborted. When the exception is propagated to a handler in a procedure that is not being aborted, MAINSAIL intercepts it, marks it as handled, and finishes aborting the procedures' execution.

- MAINSAIL requires each handler for this exception to propagate the exception to the next handler. If a handler attempts to handle the exception either by calling the system procedure $raiseReturn or by terminating the execution of its Handle Statement, a runtime error occurs.

- It is an error for the user to raise $abortProcedureExcpt explicitly.

If a procedure must perform certain actions to "clean up" after itself before it is aborted, such as closing files, disposing of arrays or modules, or releasing scan bits, it should contain a handler that catches the exception denoted by $abortProcedureExcpt, does the required cleaning up, and propagates the exception to the next procedure being aborted, as shown in Example 16.6-1.

```
PROCEDURE compiler;
BEGIN
POINTER(textFile) inputFile;

inputFile := NULLPOINTER;
$HANDLE
    WHILE open(inputFile,"compile: ",input!prompt!errorOk)
    DOB compileModule(inputFile); close(inputFile) END
$WITHB
    IF $exceptionName = $abortProcedureExcpt THEN
        IF inputFile THEN close(inputFile);
    $raise END END;
```

Example 16.6-1. Sample Procedure Needing Cleanup

16.7. Exception Naming Conventions

In systems composed of separately developed modules, two distinct exceptions may mistakenly be denoted by the same string. To help avoid such conflicts, MAINSAIL supports a convention whereby each string denoting an exception is assumed to consist of one or more "phrases" separated by colons. Table 16.7-1 illustrates the general form of such a string, in which each si is a phrase.

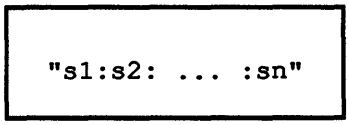


Table 16.7-1. General Form of Exception String

According to this convention, the last phrase describes the actual exception and the first phrases serve to distinguish the exception from other exceptions having the same last phrase. Typically, the first phrases would contain the name of the company owning the rights to the system in which the exception is embedded, or the name of a product of the company, as illustrated in Example 16.7-2.

```
"MAINSAIL compiler: Abort compilation"
```

Example 16.7-2. A Sample Exception Name

All predefined exceptions begin with the substring "MAINSAIL". Programmers should avoid choosing exception names that begin with that substring.

The system procedure $newException returns a string consisting of a unique decimal integer followed by a colon. To avoid conflicts with names created from strings returned by $newException, users should avoid choosing an exception name that begins with a decimal integer and a colon unless that prefix was obtained by calling $newException.

16.8. Predefined Exceptions

Predefined exceptions are exceptions known to MAINSAIL. A predefined exception may occur as a result of an operation other than a call to the system procedure $raise. The strings denoting predefined exceptions can be referred to by means of identifiers defined by MAINSAIL.

The predefined exceptions and the identifiers by which they are known are given in Appendix C.

When the exception denoted by $abortProgramExcpt is raised and no other handler handles it, execution of the current program is aborted and control passes to the MAINSAIL component that initiated it, e.g., the MAINSAIL executive, MAINEDIT, MAINDEBUG, or MAINPM.

The exception denoted by $systemExcpt includes all errors not covered by the other predefined exceptions, in which case the specific error is described by the strings returned by $exceptionStringArg1 and $exceptionStringArg2.

The exceptions denoted by $abortProgramExcpt and $systemExcpt are the only predefined exceptions that may be handled by resuming execution at the place where the exception occurred.

The exception denoted by $abortProgramExcpt is the only predefined exception that is registered.

XIDAK reserves the right to create new predefined exceptions. MAINSAIL programmers should be aware that system predefined exceptions, or undocumented exceptions used internally by the MAINSAIL runtime system, may be raised by many MAINSAIL constructs. Exception handlers must therefore be written to check that they are actually handling the exceptions they expect to handle. Issuing a $raiseReturn or terminating a handler on an undocumented exception has undefined effects; an unrecognized exception should always be propagated by calling $raise with no arguments.

16.9. errMsg Response Abbreviations

In response to the errMsg "Error response:" prompt, distinctions between upper and lower case letters are ignored when comparing a specified response to one of the expected responses.

Responses may be abbreviated, but an abbreviation must be unambiguous. A response's text is divided into "phrases" separated by colons. By convention, the last phrase describes the action to be taken as a result of the response. Any preceding phrases serve to distinguish the response from other responses having the same last phrase, as illustrated in Example 16.9-1.

```
The response "MAINSAIL: Abort program" consists
of the phrases "MAINSAIL" and "Abort program".
```

Example 16.9-1. Phrases in a Sample errMsg Response

Responses may be abbreviated by omitting leftmost phrases (and the colons that follow them). The phrases present may be abbreviated by omitting trailing words and, within the words present, by omitting trailing characters. A blank or tab between words in a specified response is equivalent to any number of blanks or tabs in an expected response.

If a specified response is an abbreviation for more than one expected response, then:

- If the specified response exactly matches one of the expected responses, it is assumed to denote the expected response it exactly matches.

- If a phrase in the specified response exactly matches a corresponding phrase in only one expected response and none of its other phrases exactly matches the corresponding phrases in any of the other expected responses, then the specified response is assumed to denote the one expected response.

- Otherwise, the specified response is ambiguous, in which case errMsg writes to logFile the expected responses for which the specified response is an abbreviation and reads another response from cmdFile.

Example 16.9-3 illustrates valid and invalid abbreviations for the set of expected responses shown in Example 16.9-2.

```
MAINSAIL: Abort program
MAINSAIL compiler: Abort compilation
Abort
```

Example 16.9-2. Sample Expected Responses

```
a p       denotes "MAINSAIL: Abort program".

m c:      denotes "MAINSAIL compiler: Abort compilation".

m:a       ambiguous abbreviation.  It is an abbreviation
          for both "MAINSAIL: Abort program" and
          "MAINSAIL compiler: Abort compilation".

abort     denotes "Abort".  It is also an abbreviation for
          each of the other expected responses, but is
          assumed to denote "Abort" since "abort" and
          "Abort" match exactly.

m         invalid abbreviation; there is no response the
          rightmost phrase of which begins with "m" or "M".
```

Example 16.9-3. Valid and Invalid Abbreviations

16.9.1. Sample Use of Registered Exceptions

Suppose a program is controlling the execution of several processes by some undisclosed means and wants to allow a user to kill any of them from errMsg by typing "kill p", where p is the name of the process, or to kill the current process by typing just "kill". Then it could register the exception "KILL" as follows:

```
$registerException
    ("KILL","Kill process p",useKeyWord,"p");
$registerException
    ("KILL","Kill current process",$doNotMatch);
```

If errMsg is called and the user types "?", errMsg displays:

```
KILL p          Kill process p
KILL            Kill current process
```

There are two instances of the exception "KILL", but the second one (the one registered with $doNotMatch set) is used only by errMsg when it displays the registered exceptions.

If the user types "k main<eol>", "k" matches the exception "KILL" and "main" is set aside as the response's argument. errMsg raises the exception "KILL" and passes "main" as the $raise argument exceptionStringArg1. The program is expected to have a handler for the exception "KILL" and gets the name of the process to be killed by calling $exceptionStringArg1. If the user types just "k", "" is passed as the $raise argument exceptionStringArg1.

Example 16.9.1-1 illustrates a call to $registerException.

```
$registerException("DEBUG","to enter the debugger")
```

Example 16.9.1-1. A Sample Call to $registerException

In order for an exception to be raised from errMsg, it must have been registered; exceptions that are not to be raised from errMsg need not be registered.

The system procedure $deregisterException causes an exception to be no longer registered.

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. MAINSAIL puts no limit on the number of coroutines, other than the limits implied by the size of memory on the target system. A "stack" is allocated for each coroutine to provide storage for the coroutine's procedure "frames". A procedure frame contains the storage for parameters and local non-own variables.

Table 17-1 lists system procedures, macros, and variables that deal with coroutines.

Identifier	Function
$createCoroutine	create a coroutine
$osdStackPages	default coroutine stack size
$resumeCoroutine	continue or start execution in a coroutine
$killCoroutine	get rid of a coroutine
$killedCoroutine	determine whether a coroutine has been killed
$moveCoroutine	change location of coroutine in coroutine tree
$findCoroutine	return a pointer to a coroutine record, given its name
$thisCoroutine	current coroutine
$exceptionCoroutine	coroutine in which an exception occurred

Table 17-1. System Procedures, Macros, and Variables for 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 explicitly resume other coroutines; i.e., coroutines do not execute in parallel, and there is no automatic resumption of coroutines. However, a program may resume coroutines in such a way as to give the illusion of parallel execution.

A coroutine is created by means of a call to $createCoroutine. The data section and the name of the procedure (the "initializing procedure") in which the new coroutine is to start execution are specified in the call. A call to $resumeCoroutine is used to transfer execution to the new coroutine; the new coroutine may then use $resumeCoroutine to transfer control to the original (or any other) coroutine. When a coroutine determines that another coroutine will no longer be used, it may kill the other coroutine with a call to $killCoroutine, provided that the coroutine is not itself or one of its ancestors. A coroutine may kill itself by specifying the delete bit in a call to $resumeCoroutine.

The exception $coroutineExcpt is raised if a coroutine attempts to return from its initializing procedure. The initializing procedure must terminate by calling $resumeCoroutine.

Example 17-2 shows a use of coroutines. The procedure generateNextNode generates a node in some data structure, and the procedure processNextNode processes the node.

```
POINTER(node) nextNode;
POINTER($coroutine) generator,processor;

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

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

Example 17-2. Generator/Processor Coroutines

The procedure processNextNode in the processor coroutine 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 processor coroutine (in an application where the generator coroutine could be resumed from several other coroutines, the application would have to keep track of which coroutine the generator was to resume).

Coroutines must transmit data to one another by means of outer or interface variables, since there is no way to pass parameters into or out of a coroutine using, e.g., $resumeCoroutine.

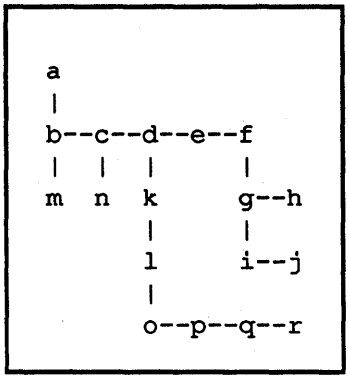
Coroutines provide stacks for procedure frames, but do not create new data sections for modules; i.e., procedures executing in different coroutines may access the same data section. If a separate data section is desired, it must be explicitly created with the system procedure "new" and the newly created data section specified as an argument to $createCoroutine.

17.1. Coroutine Implementation

A coroutine consists of a stack to hold the procedure frames and a record of the predeclared class $coroutine that 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. Example 17.1-1 shows a coroutine tree, where the letters represent coroutine records, the vertical bars represent the $up and $down links, and the dashes represent the $left and $right links. Only the $up links for the oldest children are shown; nodes c, d, e, and f have $up links to node a, h to f, j to g, and p, q, and r to l. a's $up link is nullPointer since it is the root coroutine; this coroutine is called "MAINSAIL" and is created when the MAINSAIL runtime system initializes itself.

The $up, $down, $left, and $right links are "structural" links in that they depend on the order of coroutine creation (unless coroutines have been moved about in the coroutine tree, e.g., with the system procedure $moveCoroutine). In addition, $coroutine records are maintained on a "dynamic" list by means of the $prev and $next links. Each living coroutine appears exactly once on this list. Each time a coroutine is resumed, it is moved to the head of the 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, if that coroutine is still living (this may not be a coroutine within the current application, however; creation of coroutines by the MAINSAIL runtime system and searches for exception handlers also reorder the dynamic coroutine list). The list is thus ordered from most recently resumed to least recently resumed



Example 17.1-1. Coroutine Tree

(where a search for a handler in a coroutine is considered a resumption of the coroutine); newly created but not yet resumed coroutines are put at the tail of the list.

The $coroutine record includes the following fields of interest to the user:

- $name: the name of the coroutine
- $prev: a pointer to the dynamically "previous" coroutine
- $next: a pointer to the dynamically "next" coroutine
- $up: a pointer to the parent coroutine
- $down: a pointer to the oldest child coroutine
- $left: a pointer to the left (next older) sibling coroutine
- $right: a pointer to the right (next younger) sibling coroutine
- $userHook: an unclassified pointer reserved for a user program to point to some data structure associated with the $coroutine record.

Modifying fields other than $userHook or accessing undocumented fields of a $coroutine record has undefined effects.

17.2. Coroutines and Exceptions

Unhandled exceptions are propagated up the coroutine tree through the $up link, so that a coroutine can handle exceptions in itself and its descendants but not, for example, in its siblings or ancestors.

If an exception is raised in a coroutine (the "raisee coroutine"), any active handlers in the raisee coroutine are given control as usual. If none of them handles the exception, the exception is simulated in the raisee coroutine's parent to give an opportunity for active handlers there to handle the exception. If the exception is still not handled, the exception is simulated in the raisee coroutine's grandparent (all traces of the simulated exception in the parent are erased), and so forth. This propagation continues until either the exception is handled, or a $raiseReturn occurs in the coroutine to which the exception has been propagated, or there is no handler in the root node "MAINSAIL".

The system macro $exceptionCoroutine returns a pointer to the "raiser coroutine" (the coroutine in which $raise was called) for the current exception (different from the raisee coroutine only if the exceptionCoroutine argument to $raise denoted a coroutine other than the raiser coroutine).

$raiseReturn in a coroutine to which the exception has been propagated first erases any trace of the propagated exception, then resumes the raiser coroutine (which may have already been resumed several times by various handlers). If $raiseReturn is not allowed by the original exception (e.g., implicit exception such as divide-by-zero or $raise with $cannotReturn set in ctrlBits), an error occurs if the raiser coroutine is ever resumed, whether by $raiseReturn or by $resumeCoroutine. The program should kill the raiser coroutine if it can never be resumed.

If the exception is handled, procedures are aborted in the handling coroutine as usual, and execution continues in the handling coroutine. The raiser coroutine is not killed, but is left in a such a state that, if resumed, it continues from the call to $raise as if $raiseReturn had been called from a handler. Thus, execution can be resumed from a call to $raise handled by a handler in a different coroutine from the raiser coroutine. An error occurs if the raiser coroutine is resumed but the exception does not allow a $raiseReturn.

The use of $resumeCoroutine to resume a coroutine that propagated the current exception (before it is handled) has undefined effects.

The programmer should be careful to kill coroutines that will never be resumed in order to free the space occupied by the coroutines.

If $raiseReturn or an argumentless call to $raise occurs in some coroutine other than the raisee coroutine or the one to which the exception has been propagated, then that $raiseReturn or $raise applies only to the active exception in its coroutine (an error if there is no such

exception); i.e., exceptions are maintained on a per-coroutine basis except for the propagation described above.

When a coroutine is killed, $descendantKilledExcpt is raised in its ancestors to inform the coroutines that their descendant has died.

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. The correspondence between files and strings may not be one-to-one.

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.

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 the program no longer intends to use it (unless the program reopens the file later).

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

The system procedures shown in Table 18-1 may be used to manipulate files.

18.1. File Names

The format of a file name is machine-dependent. Different operating systems impose different limits on the length of a system-dependent file name and on the characters a file name may contain.

The MAINSAIL device module delimiter (see Section 18.11) may be different on different operating systems.

Procedure	Function
open	open a file
close	close a file
$reOpen	open same file pointer with different bits
$createUniqueFile	create a file with a unique name
$delete	delete a file
$rename	rename a file
read	read a value from a file
write	write a value to a file
cRead	read a character from a file
cWrite	write a character to a file
$copyFile	copy (part of) one file to another
$truncateFile	truncate a file to a given length
$storageUnitRead	read a number of data from a file
$storageUnitWrite	write a number of data to a file
$characterRead	read a number of characters from a file
$characterWrite	write a number of characters to a file
$pageRead	read whole pages of data from a file
$pageWrite	write whole pages of data to a file
setPos	set a file position
getPos	get the current file position
relPos	set a relative file position
eof	true if the file pointer is at or beyond the end of the file
$gotValue	true if last read was successful; better way of determining end-of-file

Table 18-1. System Procedures for Files (continued)

scan	scan a file as directed by scan specifications
fldRead	read a string with specified width from a file
fldWrite	write a string with specified width to a file
ttyRead	read a line from the terminal or primary input
ttyWrite	write values to the terminal or primary output
ttycWrite	write characters to the terminal or primary output
$fileInfo	return information about a file

Table 18-1. System Procedures for Files (end)

On every operating system, the name "TTY" (case is not distinguished) refers to the operating-system-dependent primary input or output file, usually the user's terminal. "TTY" is a text file that may be opened either for sequential input or sequential output.

Identifiers (other than "TTY") of six or fewer characters are guaranteed to be valid operating-system-dependent file names on any system. It is not guaranteed whether the operating system distinguishes case; e.g., "ABC" and "abc" may or may not refer to distinct operating system files. The $attributes bit $fileNamesAreCaseSensitive is set if the operating system treats files names of different case as names of different files.

18.2. The Classes file, textFile, and dataFile

A file is manipulated as a pointer to one of two predeclared classes "textFile" or "dataFile". The pointer is initialized by a call to the system procedure open. The predeclared class "file" is a prefix class of both textFile and dataFile.

The only fields of the class file available to a user program are the fields "name" and "$strArea". $strArea is described in Chapter 20. name is set to be the name specified to the open procedure (or the substituted name if a logical name or searchpath substitution occurs) when the file is opened; see Example 18.2-1. The name may be further changed by some device modules to reflect the "real" name of the file opened; the documentation for any device module that modifies the name field explains the modifications made. The effect of altering this field or of accessing other fields of the classes file, textFile, and dataFile is not defined.

```
if outFile is a pointer declared as

    POINTER(textFile) outFile

and a file named "RESULT" is opened as

    open(outFile,"RESULT",...)

then

    outFile.name = "RESULT"
```

Example 18.2-1. The Field name of the Class file

18.3. Text Files

A text file contains only characters.

When (long) integer, (long) real, or (long) bits data are written to a text file (with the system procedure "write"), an automatic conversion is made to the appropriate string representation of those data, and then the string is written to the file. For example, if r is a real variable with value 123.8, and f is a text file, then "write(f,r)" converts the number 123.8 to the string "123.8" and writes the string to the file. Only the five characters of the string representation are written; i.e., MAINSAIL does not automatically write any spaces or end-of-line characters to separate numbers.

A string is written to a text file by writing its characters into the file. For example, write(f,"a4$") writes the three characters of the string "a4$" into the text file f. An eol is not automatically appended; eol may be explicitly written to a text file like any other string.

When the non-string data types are read from a text file (by means of 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. For example, if i is an integer variable, then "read(f,i)" causes a scan of the file referenced by f for a proper string representation of an integer (that is, one or more digits, possibly preceded by "-"). If the digits found are "123", then the number 123 is assigned to the integer variable i. The scan skips over characters that do not form part of the numeric representation.

A string may be read from a text file using read, fldRead, or scan.

18.4. Data Files

A data file is used for storing data in a machine-dependent internal ("binary") format. That is, a data file consists of boolean, (long) integer, (long) real and/or (long) bits data stored in a compact form identical to the internal representation within the computer.

Since no conversion to string is necessary, input and output of boolean, (long) integer, (long) real, and (long) bits values to data files are usually more efficient than to text files.

Characters stored in data files are stored as integers, one character per integer. This representation is often less compact than that used in a text file. See Sections 1.82 and 1.104 of part II of the "MAINSAIL Language Manual".

18.5. Input and Output

If a file is opened for input, data may be read from the file; if a file is opened for output, data may be written to the file.

A file opened for sequential access may be opened for either input or output, but not both at once. A file opened for random access may be opened for input or output or both.

18.6. Sequential and Random Access

When a file is created and opened for sequential access, it can be opened only for output. Each write to a sequential file appends to the current end of the file, starting with the first position in the file.

When a file already exists and is opened for sequential access, it can be opened only for input (unless the entire file is replaced). Each read from the file obtains data starting from the current position (starting with the beginning position of the file) and then sets the file pointer to the position following the data read, as determined by the data type being read.

A random file allows access to any position within the file, as specified by the positioning procedures setPos, relPos, and getPos. The positioning may be interspersed with normal sequential reads and/or writes. A random file is like a sequential file except that it may allow both reads and writes, an existing file can be altered without replacing the entire file, and the position within the file can be controlled by explicit positioning procedures.

A file opened for random output is automatically extended if the current position is set to be beyond the end of the file.

Not all file formats allow random access. An error message is generated when a random open is attempted on a file that does not permit random access.

The contents of unwritten locations in a random access file are Zero.

Some operating system file systems have separate file organizations for sequential and random files. It may not be possible to open a sequentially organized file for random access. MAINSAIL guarantees to use the random file organization only if a file is opened for random access when it is originally created.

18.7. Opening a File

A file is opened by specifying the file name and how the file is to be used (for input or output, accessed sequentially or randomly, etc.) to the system procedure open. For example:

```
open(inFile,"notes",input)
```

opens the file named "notes" for sequential text input. input is a predefined bits constant used by the open procedure. The open procedure produces as its first argument a pointer belonging to one of the predeclared classes "textFile" or "dataFile". That pointer is used for subsequent references to the file. For example, if s is a string variable, then:

```
read(inFile,s)
```

reads the next line from the file "notes" into the string s.

A file name can be specified during execution. If an error occurs while a file is being opened, then by default, MAINSAIL prompts for and reads a new file name from cmdFile. The "prompt" bit may also be specified as described in Section 1.259 of part II of the "MAINSAIL Language Manual".

The same file may be opened more than once (i.e., several calls to open may be made for the same file, resulting in several different file pointers referring to the same file), provided that none of the file pointers is opened for write access; otherwise, the effects are undefined.

Some operating systems do not permit the creation of a zero-length file. Thus, if a file is opened for create access but no data are written to it before it is closed, the file is not guaranteed to exist after it is closed.

File names may be logical names. A real file name is substituted for a logical file name when the file is opened. Logical file names allow redirection of specified files and the use of system-independent file name forms. Logical names are established by a call to enterLogicalName and examined by a call to lookUpLogicalName.

Any string may be used as a logical file name. XIDAK's own logical file names are written enclosed in parentheses. To avoid conflict with XIDAK logical names, it is suggested that each organization adopt its own convention; e.g., XYZ Corporation might choose names like:

```
XYZ:main library
```

Logical file names may contain spaces, although some XIDAK utilities use space as a separator character. Real file names must be specified to such utilities if the corresponding logical file name contains a space.

18.8. Closing a File

A file is closed with the system procedure "close". This causes the file to be disengaged from the runtime and operating systems, if necessary. It is good practice to close files as soon as I/O no longer needs to be performed on them, since open files may occupy scarce resources.

MAINSAIL automatically closes all files that remain open at the end of a MAINSAIL execution.

18.9. End-of-File

Many file systems organize files as fixed-size blocks, and remember only the number of blocks in a file, so that the exact end-of-file position (that is, the last position in the file to which a value was written) cannot be determined. Some types of files (e.g., terminals on some operating systems) may not have end-of-file positions. The system procedure eof returns true if the current file position is beyond the end of the file, and may return true if the current file position is at the end of the file, depending on the file organization and operating system. $gotValue is a better test of end-of-file position, since on files where end-of-file is detected exactly, $gotValue always becomes false after the first read beyond end-of-file is attempted. Portable programs should not rely on eof or $gotValue, if possible, since operating systems may not record the end-of-file position of a file exactly.

18.10. Terminal I/O and Primary Input and Output

The system procedures ttyRead, ttyWrite, and ttycWrite are used for explicit communication with the primary input and output files (the operating-system-dependent standard input and output files). These files are established by most operating systems before MAINSAIL is invoked. On most operating systems, these files correspond by default to the user's keyboard and terminal screen. Both the primary input file and the primary output file have the MAINSAIL file name "TTY". ttyRead reads a line from "TTY", ttyWrite writes a string to "TTY", and ttycWrite writes individual characters to "TTY".

Terminal input and output may have operating-system-dependent characteristics; consult the appropriate operating-system-dependent MAINSAIL user's guide.

Most operating systems provide editing commands (e.g., a backspace key to correct mistakes) that may be applied to a line before it is entered from a keyboard, but MAINSAIL does not guarantee that this is the case.

If an end-of-file is detected by ttyRead, and no text was read by the invocation of ttyRead before the end-of-file, then the exception $ttyEofExcpt is raised. If it is not handled and $raiseReturn is not called, then MAINSAIL exits with the message:

```
Eof on TTY: exiting
```

This helps avoid infinite loops in batch jobs that mistakenly loop while reading beyond the end of the batch script. To deal explicitly with a terminal end-of-file in a program, do something like the following:

```
$HANDLE s := ttyRead
$WITHB
    IF $exceptionName NEQ $ttyEofExcpt THEN $raise;
    s := "" END; # and fall out of handler
```

18.11. Device Modules

A file name used in a MAINSAIL program may indicate a "device module" to be used for file manipulation. If a device module is not specified, a default module appropriate for the operating system is used.

The device module name is specified as a part of the file name. It is separated from the rest of the file name with the device module break character, $devModBrk. For example, if $devModBrk is the character ">", the file name "foo>s" specifies that the device module FOO be used with the file "s".

Some device modules accept special file name syntax; refer to the documentation on the device module (e.g., the descriptions of "MEM" and "NUL" in the "MAINSAIL Utilities User's Guide") for details.

Some files use a prefix that looks syntactically like a device module prefix, but is in fact mapped to a module of another name. This is transparent to the user of the device module. The terms "device prefix" and "device module" are used interchangeably.

18.12. cmdFile and logFile and MAINSAIL Standard Input and Output

cmdFile (command file) and logFile (logging file) are the standard input and output files used by the MAINSAIL runtime system and by most MAINSAIL utilities (e.g., the MAINSAIL compiler). cmdFile is a text input file, and logFile is a text output file. Both files are initially opened to "TTY".

cmdFile and logFile are the "standard" input and output, i.e., the main medium of interaction (or simulated interaction) with the user, as established within MAINSAIL. They are distinct from "TTY", which is "primary" input and output, i.e., the main medium of interaction with the user, as established by the operating system when MAINSAIL is invoked from the operating system level. If "TTY" is redirected, it must be done at the operating-system level; however, cmdFile and logFile can be redirected within MAINSAIL.

cmdFile and logFile are used to "redirect" the MAINSAIL standard input or output stream by opening some other text input file as cmdFile, or by opening some other text output file as logFile. For example, a program may use the calls:

```
open(cmdFile,"Command file: ",prompt!input);
open(logFile,"Logging file: ",prompt!create!output)
```

to allow the user to specify the files to use for MAINSAIL standard input and output.

MAINEX provides subcommands to redirect cmdFile and logFile; see the "MAINSAIL Utilities User's Guide".

The system procedures ttyRead, ttyWrite, and ttycWrite communicate directly with "TTY", and cannot be redirected by MAINSAIL.

Closing cmdFile or logFile has the effect of reopening it to "TTY". When end-of-file is reached on cmdFile, the following occurs unless the configuration bit $noAutoCmdFileSwitching is set (see the description of CONF in the "MAINSAIL Utilities User's Guide"):

1. The predefined exception $cmdFileEofExcpt is raised.

2. If the exception is handled, execution proceeds with no change to the status of cmdFile.

3. If the exception is not handled:

   - If cmdFile is "TTY", MAINSAIL exits.

   - Otherwise, cmdFile is closed and reopened to "TTY".

18.13. errorOK and File I/O

The errorOK bit suppresses most file I/O error messages. Errors that may produce error messages even when the errorOK bit is set when a file I/O procedure is called include:

- Invalid open bits, e.g., the create bit set without the output bit.
- A file closed twice, or I/O attempted on a closed file.
- An invalid format prefix, e.g., a "VAR" prefix for a file opened for random output.
- A file that cannot be closed.
- Buffered I/O attempted on an unbuffered file.
- Unable to perform a read or write on an open file not at end-of-file.

18.14. cmdFile and logFile Echoing

```
TEMPORARY FEATURE:  SUBJECT TO CHANGE
```

The configuration bit $echoIfRedirected (value 'H80) causes input to cmdFile to be echoed to "TTY" if cmdFile or logFile is not the file "TTY", and output to logFile to be echoed to "TTY" if logFile is not the file "TTY". The configuration bit $echoCmdFile (value 'H100) always echoes cmdFile to logFile.

The file "TTY" is the MAINSAIL file "TTY". If operating-system-dependent standard input is redirected for a MAINSAIL process, MAINSAIL still considers that the operating-system-dependent standard input is the MAINSAIL file "TTY". The $echoIfRedirected bit does not take effect unless logFile and cmdFile are redirected from within MAINSAIL.

There are some instances when input routines look one character ahead to determine that an input operation has terminated. This character is then input the next time a read occurs from the file. In such cases, the character is echoed twice, once when it was used in the one-character lookahead, and once when it was actually read. This can occur at present (if f is cmdFile) for "read(f,x)", where x is a (long) integer, (long) real or (long) bits, and for "scan(f,...,proceed...)". This bug may be fixed in a future release.

18.15. Caching of Files

```
TEMPORARY FEATURE:  SUBJECT TO CHANGE
```

18.15.1. Introduction

The file cache is a temporary feature that permits greater control of I/O. Facilities provided by the file cache include:

- the ability to specify that a file not be cached
- the ability to cache non-bytestream files opened for random access
- the ability to cache buffers privately for a particular file
- the ability to remove all of a file's buffers from the cache
- the ability to control whether or not dirty buffers are written when removed from the cache
- more efficient buffer lookup

The file cache improves I/O performance for buffered random access files; sequential files are not cached.

A file cache is a collection of file buffers that are maintained in memory rather than on a device. All such buffers are referred to as cached buffers. Since information which resides in memory can be accessed faster than information which resides on a device, I/O performance is improved.

A buffered random access file can be globally cached, privately cached, or not cached, except that if its buffer size is not the same as the size of the buffers in the global cache, it cannot be globally cached. The global cache reuses buffers among all globally cached files while a private cache reuses only a particular file's cached buffers. A buffered random access file is automatically globally cached if its buffer size is the same as the size of the buffers in the global cache; otherwise, it is automatically privately cached.

The major benefit of the global cache is that it does not force an application to predict file usage. Which buffers reside in the global cache is dynamically adjusted to file usage; i.e., more buffers are cached for more heavily-accessed files, and fewer buffers are cached for less

heavily-accessed files. A private cache is meant for use only for key files that are very heavily used and for files with a buffer size other than that of the buffers in the global cache. An application program should therefore normally let bytestream files be globally cached.

All buffers (including the current buffer) for a cached file reside in a hash table associated with that file. The hash function is based on the file position. All buffers except the current buffer also reside in a list ordered from most-recently-used to least-recently-used, subsequently referred to as an LRU list. The cache parameters requestedMinCacheSize and requestedMaxCacheSize refer to the size of the LRU list (a file can be cached even though requestedMaxCacheSize is Zero, since requestedMaxCacheSize refers to the LRU list and the current buffer is cached but does not reside in the LRU list).

A cached file is implicitly updated when a cached buffer is reused and when the file is closed. It can be explicitly updated by calling a procedure which clears the cache.

Maintenance of a cache is governed by three parameters: 1) the requested minimum number of buffers, 2) the requested maximum number of buffers, and 3) the requested cache hit percent. Default parameters for the global cache are defined in the MAINSAIL system module but can be altered programmatically. Default values for private cache parameters are local to the file cache module and are used if private cache parameter values are not provided by the application program. The number of buffers in the LRU list is kept between the requested minimum and maximum size whenever possible; the LRU list size may be smaller than the requested minimum size but is not allowed to grow larger than the requested maximum size.

When a request for a new buffer is made, the cache is searched (this is a hash table lookup). If the requested buffer is found, it becomes the file's current buffer. If it is not found, either a new cache entry is created, the least-recently-used cache entry is reused, or the file's current buffer is reused. Which action is taken is governed by the file cache parameters.

18.15.2. File Cache Procedures

The procedures $queryFileCacheParms, $setFileCacheParms, and $clearFileCache may be used to manipulate the file cache.

$queryFileCacheParms returns information about the cache, optionally associated with a particular file.

$setFileCacheParms sets the parameters for the cache, optionally associated with a particular file.

$clearFileCache removes some or all of a file's buffers from the LRU list, optionally writes dirty buffers, and optionally uncaches the file.

18.16. Partial Data Reads

```
TEMPORARY FEATURE:  SUBJECT TO CHANGE
```

A partial data read is an input operation where the number of storage units remaining in the file is less than the size of the data type being read; i.e., if the size of a long bits is four bytes and only three bytes remain in the file f, then:

```
read(f,bb)
```

where bb is a long bits variable, causes a partial data read.

When a partial data read occurs, an error message is issued. The user can handle the error and obtain information about what happened. In particular, the errMsg call that occurs if a data read gets only part of a data type is:

```
errMsg($partialDataRead,
    "of " & cvs(i) & " chars from file " & f.name,msgMyCaller);
```

where $partialDataRead = "partial data read".

To handle partial data read errors, put a Handle Statement around any suspect read. In the handler, check for:

```
$exceptionName = $systemExcpt AND
    $exceptionStringArg1 = $partialDataRead
```

If this is true, then "cvi($execptionStringArg2)" is the number of characters actually read, and the last "word" in $exceptionStringArg2 is the file name (if the file name contained no spaces). In order to obtain the partial datum, the handler must call $raiseReturn, which will cause the error message not to be printed, and then the read will return the partial datum in the produces variable (left or right justified depending upon the host machine's characteristics). "$gotValue(f)" will be false and the file will be positioned immediately after the characters that were read (if possible; on processors where storage units are larger than character units, the position may be rounded down to the nearest storage unit).

19. Date and Time Facilities

MAINSAIL provides a large set of procedures for obtaining dates, times, and CPU times in various formats and for performing arithmetic on date and time values. The system procedures shown in Table 19-1 may be used to manipulate files. An additional facility for timing MAINSAIL programs for performance analysis purposes is MAINPM, described in the "MAINPM User's Guide".

19.1. Representation of Dates and Times

MAINSAIL numeric dates and times are represented as long integers. They may be local dates and times, Greenwich Mean Time (or GMT; also known as Coordinated Universal Time) dates and times, or date and time differences (intervals). The three possibilities are distinguished by being represented by values located in three different parts of the long integer range.

Local dates and times can be used, e.g., by programs that want to print out the current date and time. GMT dates and times are useful for timestamping data that may be shipped across time zone boundaries. Date and time differences represent a length of time, e.g., the amount of time a program took to run (as would be recorded in a benchmark). Local and GMT dates and times are called "absolute", since a date/time pair represents a single well-defined moment in time.

Absolute dates are represented as:

```
the number of days since 17 November 1858
```

added to an appropriate bias (there is one bias value for GMT dates and another for local dates); absolute times are represented as:

```
the number of seconds since midnight
```

added to the same bias as for dates.

The guaranteed date range is from 1 January, 32,766 B.C. to 31 December, 32,766 A.D., according to the Gregorian calendar (which is the calendar now in use in all Western countries and, for secular purposes, in most other countries as well; it is extrapolated for MAINSAIL's purposes backwards or forwards to the limiting dates of the guaranteed range even though it was not actually used anywhere until 1582 A.D., and may or may not be in use in 32,766 A.D.).

The absolute time of day range is from 0:00:00 to 23:59:59 on a 24-hour clock.

Procedure	Function
$date	Return current date
$time	Return current time of day
$dateAndTime	Return current date and time of day
$dateToStr	Convert numeric date to string
$timeToStr	Convert numeric time to string
$dateAndTimeToStr	Convert numeric date/time pair to string
$strToDate	Convert string to numeric date
$strToTime	Convert string to numeric time
$strToDateAndTime	Convert string to numeric date/time pair
$assembleDate	Make numeric date from day, month, year
$assembleTime	Make numeric time from hour, minute, second
$assembleDateAndTime	Make numeric date/time pair from components
$disassembleDate	Convert numeric date to day, month, year
$disassembleTime	Convert numeric time to hour, minute, second
$disassembleDateAndTime	Convert numeric date/time pair to components
$convertDateAndTime	Convert date/time pair from local to GMT or vice versa
$dateFormat	Whether numeric date is local, GMT, or difference
$timeFormat	Whether numeric time is local, GMT, or difference

Table 19-1. System Procedures and Macros for Date and Time (continued)

```
$addToDateAndTime
          Date and time addition
$dateAndTimeDifference
          Date and time subtraction
$dateAndTimeCompare
          Date and time comparison

$timeSubcommandsSet
          See if MAINSAIL date/time parameters available
          (required on many systems for correct GMT)

$setTheDate
          Set the date if unavailable from the operating
          system

$cpuTime    Return CPU time used by current process
$cpuTimeResolution
          Ticks per second for $cpuTime

$timeout    Suspend execution for specified time
```

Table 19-1. System Procedures and Macros for Date and Time (end)

Differences are absolute numbers with no bias; e.g., a date difference of 23L represents 23 days, and a time difference of 23L represents 23 seconds. Subtractions may produce a negative difference if the presumed later date or time is actually the earlier. The guaranteed range of date and time differences is large enough to accommodate the subtraction of the earliest guaranteed absolute date or time from the latest or vice versa.

The procedures $dateFormat and $timeFormat may be used to determine the format (i.e., local, GMT, difference, or invalid) of any date or time value.

19.2. Information Required by MAINSAIL

The following information needs to be available to MAINSAIL in order to return GMT values and convert local dates and times to GMT and vice versa:

- The offset to Greenwich Mean Time of standard time in the current time zone.

- Whether the local time zone is susceptible to daylight savings time (or some other form adjusted time), and if so how large the adjustment is (usually one hour, at least in the United States in recent years).

- The rules for the starting and ending dates and times of local daylight savings time, if applicable. In the United States in recent years, these rules are of the form:

```
2 am on <nth/last> Sunday of <month>
```

- The string name of the current time zone, both for standard and daylight savings times.

This information can be made available to the MAINSAIL runtime system by issuing the appropriate MAINEX subcommands. It is intended that the subcommands be set once, at MAINSAIL installation, in a file called "site.cmd" on the MAINSAIL directory. This file, if it exists, is read by MAINSAIL each time it is initialized.

The MAINEX subcommands may not be required on those operating systems that provide sufficient time zone information to user programs. At present, no operating system provides as much information as the MAINEX subcommands, although some operating systems do provide part of the information.

The MAINEX time subcommands are documented in full in the "MAINSAIL Utilities User's Guide"; here is a summary:

- "GMTOFFSET": Specify the offset of the (standard) local time zone from GMT.

- "DSTOFFSET": Specify the offset from GMT of the local time zone adjusted for daylight savings time (if applicable).

- "STDNAME" and "DSTNAME": Specify name (usually an abbreviation) used for local time zone during standard and (if applicable) daylight savings time.

- "DSTSTARTRULE" and "DSTENDRULE": Specify the dates and times when daylight savings time (or whatever the adjustment is called locally) starts and ends (if applicable).

- "DEFINETIMEZONE": Define the names and offsets from GMT of other time zones to be recognized by $strToDateAndTime (if desired).

If the MAINEX subcommands are not given, the local date and time facilities continue to work correctly; however, procedures that accept or return GMT values may not work correctly. The boolean procedure $timeSubcommandsSet is provided to allow a program to deal with this situation.

19.3. GMT Conversions and $timeSubcommandsSet

Any procedure that does a conversion (explicit or implicit) from GMT to local time or vice versa acts as if local time were Greenwich Mean Time if the GMT MAINEX subcommands have not been issued and the operating system does not provide the required information. This is the same behavior produced if all subcommand values are set to Zero.

The procedure $timeSubcommandsSet returns true if and only if if any of the relevant MAINEX subcommands has been issued; if it returns true, then GMT conversions may be expected to be successful. If no subcommands describing the local time zone are given, $timeSubcommandsSet returns false, and any procedure attempting to convert from GMT to local time or vice versa returns the value it would have returned in the Greenwich Mean Time zone unless the operating system also provides the information.

19.4. Conversion Caveats at the Start and End of Daylight Savings Time (or Other Adjusted Time)

It is not specified which of the two possible GMT times is returned by any procedure that converts (implicitly or explicitly) from local to GMT times during the ambiguous transition period from daylight savings time to standard time. If the conversion is from GMT to local time, however, it is always possible to determine the correct local time if the correct time zone and daylight savings time algorithm are provided by the appropriate MAINEX subcommands.

19.5. MAINEX Time Subcommand Values Appropriate to the Forty-Eight Contiguous United States

For the time zones of the contiguous United States, the appropriate MAINEX subcommand values are shown in Table 19.5-1.

Time Zone	GMTOFFSET	STDNAME	DSTNAME
Eastern	18000	EST	EDT
Central	21600	CST	CDT
Mountain	25200	MST	MDT
Pacific	28800	PST	PDT

Table 19.5-1. MAINEX Time Subcommand Values for the Contiguous United States

The subcommands shown in Table 19.5-2 should be given everywhere that the standard time zone abbreviations for the forty-eight contiguous United States are used. Additional subcommands may be desirable if other time zone abbreviations are commonly referred to.

```
DEFINETIMEZONE EST 18000
DEFINETIMEZONE EDT 14400
DEFINETIMEZONE CST 21600
DEFINETIMEZONE CDT 18000
DEFINETIMEZONE MST 25200
DEFINETIMEZONE MDT 21600
DEFINETIMEZONE PST 28800
DEFINETIMEZONE PDT 25200
```

Table 19.5-2. Subcommands Defining the Names of the Time Zones in the Forty-Eight Contiguous United States

"DSTOFFSET 3600" should be specified where daylight savings time is used, and "DSTOFFSET 0" where it is not. As of 8 July 1986, the following should be specified where daylight savings time is in effect (and will be ignored if it is not, i.e., if "DSTOFFSET 0" is set):

```
DSTSTARTRULE April Sunday 1 2:00
DSTENDRULE October Sunday 5 2:00
```

Complete sample sets of time zone subcommands for the forty-eight contiguous United States (except for those shown in Table 19.5-2) are shown in Tables 19.5-3, 19.5-4, 19.5-5, 19.5-6, 19.5-7, and 19.5-8. These subcommands should appear in a file "site.cmd" on the MAINSAIL directory, which should be created when a new version of MAINSAIL is installed.

```
GMTOFFSET 18000
DSTOFFSET 3600
STDNAME EST
DSTNAME EDT
DSTSTARTRULE April Sunday 1 2:00
DSTENDRULE October Sunday 5 2:00
```

Table 19.5-3. Subcommands for the Eastern Time Zone: from the Atlantic Seaboard West through Michigan, Eastern Kentucky, Eastern Tennessee, Georgia, and Florida Exclusive of the Panhandle

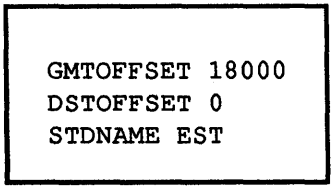


Table 19.5-4. Subcommands for Indiana except Parts of the Extreme West

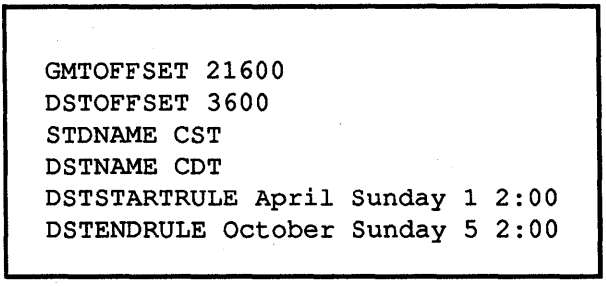


Table 19.5-5. Subcommands for the Central Time Zone: Wisconsin, Illinois, Parts of Extreme Western Indiana, Western Kentucky, Western Tennessee, Alabama, the Florida Panhandle, Mississippi, Louisiana, Arkansas, Missouri, Iowa, Minnesota, Eastern North Dakota, Eastern South Dakota, Eastern Nebraska, Kansas except Parts of the Extreme West, Oklahoma, and Texas except the Extreme West

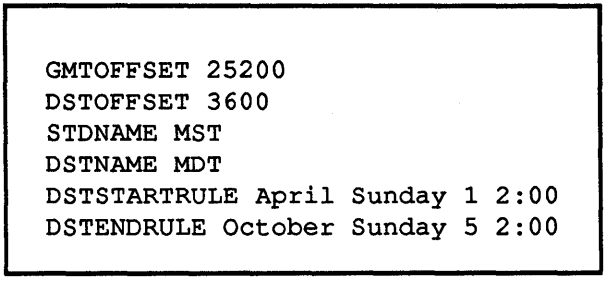


Table 19.5-6. Subcommands for the Mountain Time Zone: Western North Dakota, Western South Dakota, Western Nebraska, Parts of Extreme Western Kansas, Extreme Western Texas, New Mexico, Colorado, Wyoming, Montana, Southern Idaho, Parts of Extreme Eastern Oregon, and Utah

```
GMTOFFSET 25200
DSTOFFSET 0
STDNAME MST
```

Table 19.5-7. Subcommands for Arizona

```
GMTOFFSET 28800
DSTOFFSET 3600
STDNAME PST
DSTNAME PDT
DSTSTARTRULE April Sunday 1 2:00
DSTENDRULE October Sunday 5 2:00
```

Table 19.5-8. Subcommands for the Pacific Time Zone: Northern Idaho, Washington, Oregon except Parts of the Extreme East, Nevada, and California

20. Areas

Areas provide a method for more precise control of memory management. Correct explicit use of areas is relatively complex, and requires considerable care in order to avoid introducing bugs that may be difficult to track. Programmers need not understand areas in order to write correct programs, and programs that do not allocate significant amounts of memory will not benefit from explicit use of areas. This chapter may therefore be considered optional reading for programmers not involved in writing such programs.

An "area" is a dynamically growing collection of "chunks" (collectable data structures, i.e., records, arrays, and data sections) and string text, which can be treated as a unit. An area can be disposed, which immediately causes all of the area's memory (an integral number of pages) to be released; this simplifies the task of the MAINSAIL runtime system's memory manager, which can more efficiently reclaim free pages than "free chunks" (unallocated or deallocated memory from which chunks may be allocated) or inaccessible text in string space. The memory management algorithms applied to an area can be explicitly specified by a program, so that unnecessary memory management does not take place. In some programs, the explicit use of areas can substantially improve the efficiency of memory management; however, bugs introduced by the improper use of areas can be very difficult to track.

20.1. Examples and Motivation

The exact details of memory management and the MAINSAIL compiler implied by this section are subject to change, although the explanation of the advantages of areas is expected to be broadly applicable to future releases of MAINSAIL.

An area could contain a data structure, such as a linked-list structure, that is built up, processed, and then no longer needed. If all of the structure's data are allocated in a single area, disposing the area when the structure is no longer needed frees all the memory used by the structure in a single operation. This has several advantages over individually disposing each chunk:

- It is faster.
- Entire pages become free, rather than individual chunks.
- It is easy to be sure that entire data structure has been freed.

If the chunks are disposed individually, they are put on free lists (based on size). Even if two free chunks are contiguous, they are not coalesced until a chunk garbage collection occurs. Disposing a large number of chunks results in a large number of free chunks, rather than free

pages. If another data structure consisting of chunks of the same size is built up soon afterward, then the free chunks are reallocated, and the free lists have played their role well. However, if different-sized chunks or non-chunk objects (e.g., string space, control sections, file buffers, etc.) are needed, the free chunks are wasting space, and a garbage collection is needed to coalesce the chunks and free the pages consisting only of free chunks. Freeing an entire area immediately frees all the pages, making them available for any kind of use.

An area can also be used as a "bag" containing unrelated chunks and data structures, all of which should be disposed at the same time. A "program" (e.g., the MAINSAIL compiler) can allocate all its chunks in the same area, then dispose the area when it is finished; this reclaims the whole "bag". This is an especially convenient way to clean up when an exception occurs that causes a program to abort. The MAINSAIL compiler disposes several areas when a compilation is aborted; in this way, the code to deallocate all current data structures can be simple and does not need to understand in detail any of the data structures.

20.2. Area Facilities

Any number of areas can be created during a MAINSAIL execution. Facilities are provided for allocating chunks and string text in specified areas. System procedures are provided to:

- create and specify attributes of an area ($newArea)
- clear an area ($clearArea)
- clear the string space part of an area ($clearStrSpc)
- dispose of an area ($disposeArea)
- dispose only the data sections in an area ($disposeDataSecsInArea)
- allocate a chunk in a specified area (new)
- use a specified area for string text (various procedures)
- determine whether a pointer or string references an area ($inArea)
- determine which area is referenced by a pointer or string ($areaOf)
- get a chunk or string text into an area ($getInArea)
- find an area with a specified title ($findArea)

20.2.1. Allocation, Clearing, and Disposal

An area is specified by a pointer to a record of the class $area, which contains information about an area. $newArea, which creates areas, returns a pointer to the $area record for the newly created area, which is "empty" (it actually contains the $area record itself and some supporting data structures). A name (or "title") may be given to an area when it is created, to help identify the area in various situations during execution.

An area automatically grows to accommodate new chunks and string text. The memory pages occupied by an area are typically not contiguous. There is a single default area, $defaultArea (with the title "Default area"), into which all chunks and string text are put unless explicitly specified otherwise (so that programs that make no explicit use of areas use only $defaultArea). Another area, entitled "Dscr area", contains all chunk descriptors. Other areas may be created by the MAINSAIL runtime system or utilities. The effect of disposing or clearing any system areas is undefined. The titles of the system areas are subject to change.

Clearing an area returns it to its state immediately after allocation (i.e., empty except for the $area record and supporting structures). This is useful when the current contents of an area are no longer needed, but more data are to be put into the area. It is slightly more efficient than disposing and reallocating the area; also, if several pointers are pointing at the $area record, all of them would have to be made to point to the new $area record if the old area were disposed and then re-allocated (since the $area record is also disposed when the area is disposed). The string space part of an area can be cleared separately with $clearStrSpc.

Most system procedures that generate string text can be given an $area parameter to specify the area into which the text is to be put. Alternately, in the case of a file, the file record's $strArea field can be set to an area that is to receive input text by default:

```
STRING s;
POINTER(textFile) f;
POINTER($area) myArea;
...
f.$strArea := myArea;
read(f,s); # input text goes into myArea
```

20.2.2. Specifying Memory Management Attributes of an Area

Chunk collections, chunk compactions, and string collections are performed only on areas with memory management attributes permitting these operations. The memory management attributes of an area are established when the area is allocated by $newArea.

Chunk collection of an area is useful when many chunks become inaccessible, i.e., when many chunks are not referenced by any accessible pointer. Chunk compaction is useful when chunks

become fragmented by a mixture of free and allocated chunks, and such fragmentation causes inefficient use of memory due to the inability to find free chunks to satisfy allocation requests (if all chunks in an area are the same size, fragmentation is not a problem). String collection (which marks all accessible text, then compacts it within string space) is useful when there is a lot of inaccessible string text, i.e., text not referenced by any accessible string descriptor (the charadr and length that represent a string). Each of these operations can take quite some time in a large address space, and can be particularly slow when virtual memory must be swapped in from disk, since they involve an essentially random pattern of many memory accesses. It saves time to avoid collections and compactions that do not reclaim much space.

By default, when an area is allocated, it is not marked as collectable or compactible. The assumption is that an area managed by a single program typically does not contain enough "garbage", or become sufficiently fragmented, to justify the time spent collecting or compacting. It is expected that the programmer knows enough about the anticipated use of the area to know whether this assumption holds. For example, an area used to build up a binary tree, process it, and then dispose it may never contain any garbage, or even any free chunks. If the assumption that collections and compactions are unnecessary in an area does not hold, the area could build up enough garbage to cause the MAINSAIL execution to abort due to lack of memory.

The programmer can control the memory management attributes of an area by specifying the following predefined long bits constants for $newArea's attr argument:

attr Bit	Description
$collectableChkSpc	collect area's chunks
$compactableChkSpc	compact area's chunks
$collectableStrSpc	collect area's string text

For example, $defaultArea is allocated as if by the call:

```
$newArea("Default area",
    $collectableChkSpc!$compactableChkSpc!$collectableStrSpc)
```

Even if an area A is not marked $collectableChkSpc or $compactableChkSpc, all the pointers it contains must be examined when one or more other areas are being chunk collected or compacted in order to determine whether A has any pointers into any of the collected or compacted areas (each chunk that A references must be marked as accessible). Likewise, for string collections, all A's strings must be examined to determine whether they reference string-collected areas. However, the programmer may know that a particular area contains no pointers into any area that could be chunk collected or chunk compacted, or no string variable referencing text in any area that could be string collected. In this case, the area need not be examined at all when a collection or compaction takes place (provided the area is not itself collectable or compactible); this may be specified in attr bits to $newArea, which may result in significantly faster collections and compactions. For example, suppose an area contains a symbol table that consists of records in which all the pointer and string fields reference chunks

and string text in the same area. A garbage collection or compaction can safely ignore the area, assuming the area itself is marked so as not to take part in the collection or compaction. As another example, an area may consist entirely of records with no string fields, and thus there is never a need to examine the area when collecting strings.

The following predefined long bits constants for $newArea's attr argument are provided for indicating this kind of information:

attr Bit	Description
$noCollectablePtrs	this area contains no pointers into $collectableChkSpc areas, and hence its ptrs need not be examined when collecting chunks
$noCompactablePtrs	this area contains no pointers into $compactableChkSpc areas, and hence its ptrs need not be examined when compacting chunks
$noCollectableStrs	this area contains no string dscrs into $collectableStrSpc areas, and hence its string descriptors need not be examined when collecting strings

Be very careful when specifying these bits; if the conditions specified by the bits do not hold, effects are undefined and the resulting bugs can be strange, unpredictable, and delayed, and thus difficult to track down.

An area that contains a data section must not be marked $noCollectablePtrs or $noCompactablePtrs.

20.3. Area Caveats

Read this section carefully before making use of areas!

When an area is disposed or cleared, all pointers and strings outside the area that reference data inside the area are "dangling", i.e., reference invalid data (when just the string space part of the area is cleared, only strings referencing the area are dangling). The effects of using a dangling pointer or string are undefined, just as in the case of a dangling pointer that points to a chunk that has been individually disposed. The MAINSAIL memory manager has been designed to ignore dangling pointers and strings; in particular, the garbage collector does not fail if it encounters a dangling pointer or string.

A program must not use a dangling pointer or string. Whenever a chunk or string text is allocated in an area, the programmer must be certain that it does not outlive the area, i.e. that the program does not attempt to reference the chunk or string text after the area has been disposed.

Special care must be taken when putting string text into an area, to avoid dangling strings. Consider the following:

```
read(f,myArea,s);      # read line from f, put text into myArea
open(g,s,...);         # open file g with name in s
$disposeArea(myArea); # dispose of myArea
```

The open routine may store s into the field "g.name". When myArea is disposed, "g.name" becomes a dangling string, which may cause I/O errors to the file g, or may result in unintelligible error messages regarding g, or may produce some other failure. One might think that "open" should copy s into the default area before assigning it to "g.name", but this kind of copying would have to occur in thousands of places in the MAINSAIL runtime system. Instead, the rule is established that a program must never pass to a system procedure or macro or assign to a system variable a string or pointer referencing data allocated in an area that will be disposed before the MAINSAIL execution terminates. Put a string into an area only if its use is so restricted that no reference can occur to the string's text after the area is disposed. A string passed to a MAINSAIL system procedure or macro may be referenced by the runtime system at any later point in the execution (e.g., you must assume that "g.name" above may be referenced even after g is closed).

Dangling pointers may produce bugs as difficult to track as those produced by dangling strings, but are perhaps easier to manage conceptually, since the programmer always explicitly allocates and deallocates chunks, so the problems of dangling chunks exist even when areas are not used if the programmer ever calls "dispose".

In summary, the safety rules for areas are:

1. Do not use a pointer or string referencing data in a disposed or cleared area, or a string referencing data in an area of which the string space has been cleared.

2. Do 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.

21. Portable Data Format (PDF)

21.1. Introduction

Distributed processing involves the sharing of data across different kinds of machines. The use of a common data representation avoids each machine having to cope with the data representation of every other kind of machine. MAINSAIL's portable data format (PDF) is provided for this purpose.

The term "PDF data" refers to data formatted according to PDF. Table 21.1-1 shows the PDF representation for each MAINSAIL data type. The characters composing the PDF data representation are not human-readable.

"PDF text", "PDF characters", and "PDF strings" are just like host text, characters, and strings except that the characters are translated to the PDF character set. This translation does nothing if the host and PDF character sets are identical, as they are on many ASCII machines; see Appendix F.

Data Type	Chars	PDF Representation
BOOLEAN	2	TRUE is integer 1; FALSE is integer 0
INTEGER	2	2's-complement
LONG INTEGER	4	2's-complement
REAL	4	IEEE floating point
LONG REAL	8	IEEE floating point
BITS	2	16 bits
LONG BITS	4	32 bits

Chars is the number of 8-bit characters (bytes).

All values are stored with the high-order byte first ("big-endian" representation).

PDF characters are represented as ASCII characters.

Appendix F gives the character set translation between host character sets and the PDF character set.

Table 21.1-1. Portable Data Format (PDF) Representation of Data

PDF text and data can be manipulated in the following ways:

- A file can be opened in such a way that reads and writes automatically use PDF (see Section 21.2).
- PDF text and data can be written to, and read from, strings or scratch memory. These facilities are provided by an intmod, PDFMOD, described in detail in the "MAINSAIL Utilities User's Guide".
- The Structure Blaster can read and write structure images in a format that uses PDF. This facility is described in detail in the "MAINSAIL Structure Blaster User's Guide".

21.2. PDF I/O

The MAINSAIL I/O system allows an application program to store and retrieve PDF data from a file with the conversion between host data and PDF data carried out automatically by the reads and writes. The application program need not be aware that such conversions are taking place.

When a data file is opened for PDF I/O, all data read as individual MAINSAIL data type values from the file are interpreted as PDF data and all data written to the file individual MAINSAIL data type values are written as PDF data. Values read as characters or text are interpreted as PDF characters, and values written as characters or text are written as PDF characters. Data written en masse as storage units or pages undergo no change. If the PDF data format is the same as the host data format for all data types and the host character set is the same as the PDF character set, then host and PDF data files are treated identically.

When a datum is read from or written to a data file opened for PDF I/O, the effect is undefined of reading or writing a value outside the MAINSAIL guaranteed range for the datum's data type.

When a text file is opened for PDF I/O, all data read as individual MAINSAIL data type values from the file are read by scanning for the appropriate string representation and all data written to the file as individual MAINSAIL data type values are written as strings. The strings are in the PDF character set rather than the host character set. Values read as characters or text are interpreted as PDF characters, and values written as characters or text are written as PDF characters. If the host character set is the same as the PDF character set, then host and PDF text files are treated identically.

When text is interpreted as PDF characters on input, it is translated to the host character set; when text in the host character set is written as PDF characters, a translation is also performed. A program reading and writing PDF files may therefore be written to deal only with the host character set, provided that it uses a subset of the character set that has PDF equivalents.

Some of the I/O procedures provide a way to suppress the translation that normally takes place.

The choice of whether or not to perform PDF I/O on a file is made at execution time when the file is opened. By default, the host format is used. PDF I/O is specified by including additional information in the file name as described below.

21.3. Opening a File for PDF I/O

A file with a name starting with the device prefix "PDF" is opened for PDF I/O. Alternately, the bit $pdf may be specified in the openBits parameter to "open", which forces a file to be opened for PDF I/O whether its name contains the "PDF" device prefix or not.

A file opened for PDF I/O is sometimes referred to as a "PDF file".

"PDF" must appear in the file name before any other device prefix specifications, e.g., where the $devModBrk character is ">", "PDF>LIB(foo.lib)>/baz".

Table 21.3-1 lists the procedures for which PDF I/O is supported; i.e., if the file is opened for PDF I/O, then all forms of these procedures access the data in the file as PDF text (for character and string operations and for text files) or PDF data (for non-character, non-string operations on data files). Character and string operations are considered to be cRead, cWrite, fldRead, fldWrite, scan, and the string forms of read and write.

read	cRead	$characterRead	fldRead	scan
write	cWrite	$characterWrite	fldWrite	

Table 21.3-1. File I/O Procedures for Which PDF I/O Is Supported

For data files, $pageRead, $pageWrite, $storageUnitRead, and $storageUnitWrite read or write the requested amount of data without modification or translation.

By default, $characterRead and $characterWrite translate characters read from or written to PDF files. This translation can be suppressed by setting the $noTranslate bit in ctrlBits (an optional bits parameter to both $characterRead and $characterWrite).

21.4. Positions in a File Opened for PDF I/O

When a file is opened for PDF I/O, all file positions are in terms of character units; i.e., the units used for positioning by relPos and setPos are character units, and the positions returned by getPos and $getEofPos are character positions.

21.5. $ioSize

An application that does not know how data are formatted in a file must be careful when positioning within the file. For example, if a data file has been opened for PDF I/O, positions are character units; otherwise, they are storage units. Even if a storage unit is equal to a character unit, the sizes of host data and PDF data may differ. $ioSize is provided to simplify the writing of such applications.

"$ioSize(f,x)", where x is a MAINSAIL data type code, returns the size of x based on the format of the data in a data file f. For example, if f contains host data, "$ioSize(f,x)" returns the same value as "size(x)", but if f contains PDF data, "$ioSize(f,x)" returns the same value as "pdfChars(x)" (see the description of PDFMOD in the "MAINSAIL Utilities User's Guide").

$ioSize returns 0 if f is a text file since there is no fixed size for the string representation of a data type.

21.6. PDF Example

PDF is designed so that as many programs as possible can be written to operate on either PDF data or host data without any special logic to handle the two different cases; in other words, as many procedures as possible "do the right thing" for each case.

Example 21.6-1 shows a sample program FVIEW that displays data in a file. It is written to work independently of the format of the data in the file. Table 21.6-2 shows how to run FVIEW.

```
BEGIN "fView"

# fView is an interactive program that lets the user
# view the contents of a file.
#
# The use of $ioSize when positioning makes the program
# independent of the type of data stored in the file.

INTEGER iSize;

PROCEDURE examine (POINTER(dataFile) d);
BEGIN
INTEGER         t;
LONG INTEGER    l;
BITS            b;
l := getPos(d);
fldWrite(logFile,cvs(l),8,' '); write(logFile,"/   ");
read(d,t); b := cvb(t);
write(logFile,t,"  '",b,"  'H",cvs(b,hex),eol);
setPos(d,(getPos(d) - cvli(iSize)) MAX 0L) END;



INITIAL PROCEDURE;
BEGIN
INTEGER             t,lastCmd,cmd;
LONG INTEGER        tt;
STRING              s;
POINTER(dataFile)   d;

write(logFile,
    "File viewer (? for help)" & eol &
    "File to view: ");

read(cmdFile,s); open(d,s,random!input);
iSize := $ioSize(d,integerCode); examine(d); lastCmd := 0;

DOB write(logFile,"FVIEW>"); read(cmdFile,s);
    CASE cmd := cvu(first(s)) OFB
        ['Q'] DONE;
```

Example 21.6-1. Data-Format-Independent I/O (continued)

```
    [-1] BEGIN
        IF NOT relPos(d,iSize,errorOK) THEN
            write(logFile,
                "Cannot position beyond end of file"
                & eol);
        examine(d) END;

    ['0' TO '9'] BEGIN
        tt := cvli(s);
        IF NOT setPos(d,cvli(cvlb(tt)),errorOK) THEN
            write(logFile,
                "Cannot position beyond end of file"
                & eol);
        examine(d) END;

    ['^'] BEGIN
        IF getPos(d) THEN relPos(d,- iSize,errorOK);
        examine(d) END;

    ['+']['-'] BEGIN
        IF length(s) = 1 THEN cWrite(s,'1');
        t := cvi(s) * iSize;
        IF NOT relPos(d,t,errorOK) THEN
            write(logFile,
                "Cannot position beyond end of file"
                & eol);
        examine(d) END;

    ['?']
        write(logFile,
            "n   Examine position n" & eol &
            "eol Step forward through file" & eol &
            "^   Step backward through file" & eol &
            "+n  Forward n integers" & eol &
            "-n  Backward n integers" & eol &
            "Q   Quit" & eol);

    [ ] write(logFile,"Type ? for help" & eol);
    END;

lastCmd := cmd END;
```

Example 21.6-1. Data-Format-Independent I/O (continued)

```
close(d) END;

END "fView"
```

Example 21.6-1. Data-Format-Independent I/O (end)

```
How to use FVIEW to examine a file that contains host
data:

*fview<eol>
File viewer (? for help)
File to view: fview.dat<eol>
<FVIEW displays the contents of fview.dat; fview.dat
contains host data>

How to use FVIEW to examine a file that contains PDF
data:

*fview<eol>
File viewer (? for help)
File to view: pdf>fview.dat<eol>
<FVIEW displays the contents of fview.dat; fview.dat
contains PDF data>
```

Table 21.6-2. How to Run FVIEW

Appendix A. Type Codes

The type code identifiers and their values are listed in Table A-1.

The programmer must use the identifiers rather than the associated values, since XIDAK reserves the right to change the values. If necessary, values may be examined with MAINDEBUG.

Type Code Identifier
booleanCode
integerCode
longIntegerCode
realCode
longRealCode
bitsCode
longBitsCode
stringCode
addressCode
charadrCode
pointerCode
Two identifiers have more limited use than the above:
$classCode
$moduleCode

Table A-1. Type Codes

Appendix B. Target Platform, Operating System, and Processors

The target platform names, abbreviations, and numbers are listed in Table B-1; the target operating systems in Table B-2, and the target processors in Table B-3.

The lists of platforms, operating systems, and processors change frquently without notice from release to release. XIDAK regularly adds support for new platforms and drops support for old platforms, depending on customer demand. The name and abbreviation strings associated with each processor, operating system, and platform are subject to change as manufacturers change the names of their products. Some of the names included in the list may not be commercially available implementations of MAINSAIL; indeed, some platforms may be implementations for which support has already been dropped or for which a MAINSAIL implementation is only proposed.

The programmer must use the identifiers rather than the associated values, since XIDAK reserves the right to change the values. If necessary, the values may be examined using MAINDEBUG.

Number	Abbrev.	Full Name
$a20	a20	Apollo MC68020/FPA
$clp	clp	CLIPPER
$elx	elx	ELXSI System 6400
$i38	i38	Intel 80386
$ibm	ibm	IBM System/370
$m20	m20	MC68020/MC68881
$m68	m68	M68000
$mv	mv	ECLIPSE MV
$pri	pri	PRISM
$rdg	rdg	Ridge 32
$spa	spa	SPARC
$vax	vax	VAX-11
$w38	w38	Intel 80386/WTL 1167
$xa	xa	IBM System/370 Extended Architecture

Table B-3. Target Processors

Number	Abbrev.	OS Num.	Full Name
$aeg	aeg	$aeg	Apollo's Aegis on Motorola M68000
$aix	aix	$uxa	IBM's AIX on IBM System/370
$alnt	alnt	$um68	Alliant's CONCENTRIX on Motorola M68000
$cms	cms	$cms	IBM's VM/SP CMS on IBM System/370
$dgux	dgux	$udg	Data General's DG/UX on Data General ECLIPSE MV
$emb	emb	$emb	ELXSI's EMBOS on ELXSI System 6400
$hp20	hp20	$um20	HP's HP-UX on Motorola MC68020/MC68881
$hp38	hp38	$ui38	SCO's XENIX on HP Vectra with Intel 80386
$hpux	hpux	$um68	HP's HP-UX on Motorola M68000
$ip32c	ip32c	$uclp	Intergraph's System V UNIX on Interpro 32C
$ipsc2	ipsc2	$ui38	Intel's iPSC/2 System V UNIX on Intel 80386
$ix20	ix20	$um20	Apollo's DOMAIN/IX on Motorola MC68020/MC68881
$ixfpa	ixfpa	$ua20	Apollo's DOMAIN/IX on Motorola MC68020/Weitek FPA
$ixpri	ixpri	$upri	Apollo's DOMAIN/IX on Apollo PRISM
$mvux	mvux	$umv	Data General's MV/UX on Data General ECLIPSE MV
$ros	ros	$urdg	Ridge's ROS on Ridge 32
$spix	spix	$urdg	Bull's SPIX on Ridge 32
$sun2	sun2	$um68	Sun Microsystems' SunOS on Motorola M68000
$sun3	sun3	$um20	Sun Microsystems' SunOS on Motorola MC68020/MC68881
$sun38	sun38	$ui38	Sun Microsystems' SunOS on Intel 80386
$sun4	sun4	$uspa	Sun Microsystems' SunOS on SPARC
$sw38	sw38	$uw38	Sun Microsystems' SunOS on Intel 80386/WTL 1167
$ultrx	ultrx	$uvax	DEC's ULTRIX-32 on VAX-11
$uts5	uts5	$uibm	Amdahl's System V UNIX on IBM System/370
$vms	vms	$vms	DEC's VAX/VMS on VAX-11
$xcms	xcms	$xcms	IBM's VM/XA SP CMS on IBM System/370

Table B-1. Target Platforms

Number	Abbrev.	Proc.	Full Name
$aeg	aeg	$m68	Aegis
$aos	aos	$mv	AOS/VS
$cms	cms	$ibm	VM/SP CMS
$emb	emb	$elx	EMBOS
$ua20	ua20	$a20	Apollo MC68020/FPA UNIX
$uclp	uclp	$clp	CLIPPER UNIX
$udg	udg	$mv	ECLIPSE DG/UX
$ui38	ui38	$i38	Intel 80386 UNIX
$uibm	uibm	$ibm	IBM System/370 UNIX
$um20	um20	$m20	MC68020/MC68881 UNIX
$um68	um68	$m68	M68000 UNIX
$umv	umv	$mv	ECLIPSE MV UNIX
$upri	upri	$pri	PRISM UNIX
$urdg	urdg	$rdg	Ridge 32 UNIX
$uspa	uspa	$spa	SPARC UNIX
$uvax	uvax	$vax	VAX-11 UNIX
$uw38	uw38	$w38	Intel 80386/WTL 1167 UNIX
$uxa	uxa	$xa	IBM System/370 Extended Architecture UNIX
$vms	vms	$vax	VAX/VMS
$xcms	xcms	$xa	VM/XA SP CMS

Table B-2. Target Systems

Appendix C. Predefined Exception Names

The predefined exception name identifiers and their values are shown in Table C-1.

The programmer must use the identifiers rather than the associated values, since XIDAK reserves the right to change the values.

```
Identifier: Exception string

$abortProgramExcpt: "MAINSAIL: Abort program"
$abortProcedureExcpt: "MAINSAIL: Abort procedure"
$systemExcpt: "MAINSAIL: System exception"
$stackOverflowExcpt: "MAINSAIL: Apparent stack overflow"
$arithmeticExcpt: "MAINSAIL: Arithmetic error"
$exponentExcpt:
    "MAINSAIL: IntegerOrLongInteger ^ i: i < 0"
$subscriptExcpt: "MAINSAIL: Subscript error"
$nullArrayExcpt: "MAINSAIL: NULLARRAY access"
$nullPointerExcpt: "MAINSAIL: NULLPOINTER data access"
$nullCallExcpt: "MAINSAIL: NULLPOINTER call"
$unboundModuleExcpt: "MAINSAIL: Unbound MODULE access"
$caseIndexExcpt: "MAINSAIL: Case index error"
$returnExcpt: "MAINSAIL: Fell out of a typed procedure"
$coroutineExcpt: "MAINSAIL: Fell out of coroutine"
$cmdFileEofExcpt: "MAINSAIL: cmdFile eof"
$descendantKilledExcpt:
    "MAINSAIL: Coroutine descendant killed"
$disposedDataSecExcpt:
    "MAINSAIL: Return to disposed dataSec"
$ttyEofExcpt: "MAINSAIL: tty eof"

The following are considered temporary features:

$almostOutOfMemoryExcpt: "MAINSAIL: Almost out of memory"
$overheadTooHighExcpt: "MAINSAIL: Overhead too high"
```

Table C-1. Predefined Exceptions

Appendix D. Target System Attributes

This appendix describes the bits that may be set in the system macro $attributes. Their values are shown in Table D-1. The programmer must use the identifiers rather than the associated values, since XIDAK reserves the right to change the values. If necessary, values may be examined with MAINDEBUG.

Bit
$fileNamesAreCaseSensitive
$halfDuplex
$hasFileVersions
$onesComplement

Table D-1. Target System Attribute Bit Values

$fileNamesAreCaseSensitive is set if case is distinguished on operating system file names, as on UNIX; case is considered to be distinguished if it is possible, for example, for "FOO" and "foo" to refer to different files.

$halfDuplex is set if full-duplex terminal I/O is not supported.

$hasFileVersions is set if the target file system supports file version numbers, as on VAX/VMS.

$onesComplement is set if negative integers are represented in ones'-complement form (the two's-complement form is far more common).

Appendix E. Character Set Identifiers

The character sets supported are shown in Table E-1. The identifiers denote possible values of the system macro $charSet.

The programmer must use the identifiers rather than the associated values, since XIDAK reserves the right to change the values. If necessary, the values may be examined with MAINDEBUG.

Character Set	Identifier
ASCII	$ascii
EBCDIC	$ebcdic

Table E-1. Supported Character Sets

Appendix F. PDF Character Set Translation Tables

F.1. Translation between the ASCII and PDF Character Sets

The ASCII character set is used as the PDF character set. Thus, no character set translation is necessary on machines that use the ASCII character set, except possibly for eol.

The PDF eol character is linefeed (LF, ASCII 10). Translation between the host eol character and the PDF eol character is done if the host eol character is not linefeed; the host linefeed character is also translated to PDF eol in this case. When converting from PDF to the host character set, PDF eol is always translated to the host eol.

A file converted from an ASCII host character set to PDF and back to the ASCII host character set thus remains unchanged only if the host eol is PDF eol or the original file contained no host linefeed characters. For portable text files, therefore, avoid writing linefeed into an ASCII text file if linefeed is not the host eol character.

F.2. Translation between the EBCDIC and PDF Character Sets

Table F.2-1 shows how MAINSAIL translates characters from the PDF character set to the EBCDIC character set.

Table F.2-2 shows how MAINSAIL translates characters from the EBCDIC character set to the PDF character set. EBCDIC characters that have no corresponding ASCII equivalent translate to character code 0.

Char	PDF	EBCDIC	Char	PDF	EBCDIC	Char	PDF	EBCDIC
NUL	0	0	+	43	78	V	86	229
SOH	1	1	,	44	107	W	87	230
STX	2	2	-	45	96	X	88	231
ETX	3	3	.	46	75	Y	89	232
EOT	4	55	/	47	97	Z	90	233
ENQ	5	45	0	48	240	[	91	173
ACK	6	46	1	49	241	\	92	224
BEL	7	47	2	50	242	]	93	189
BS	8	22	3	51	243	^	94	95
HT	9	5	4	52	244	_	95	109
LF	10	37	5	53	245	`	96	121
VT	11	11	6	54	246	a	97	129
FF	12	12	7	55	247	b	98	130
CR	13	13	8	56	248	c	99	131
SO	14	14	9	57	249	d	100	132
SI	15	15	:	58	122	e	101	133
DLE	16	16	;	59	94	f	102	134
DC1	17	17	<	60	76	g	103	135
DC2	18	18	=	61	126	h	104	136
DC3	19	19	>	62	110	i	105	137
DC4	20	60	?	63	111	j	106	145
NAK	21	61	@	64	124	k	107	146
SYN	22	50	A	65	193	l	108	147
ETB	23	38	B	66	194	m	109	148
CAN	24	24	C	67	195	n	110	149
EM	25	25	D	68	196	o	111	150
SUB	26	63	E	69	197	p	112	151
ESC	27	39	F	70	198	q	113	152
FS	28	28	G	71	199	r	114	153
GS	29	29	H	72	200	s	115	162

Table F.2-1. PDF to EBCDIC Character Set Translation Table (continued)

RS	30	30	I	73	201	t	116	163
US	31	31	J	74	209	u	117	164
SPACE	32	64	K	75	210	v	118	165
!	33	90	L	76	211	w	119	166
"	34	127	M	77	212	x	120	167
#	35	123	N	78	213	y	121	168
$	36	91	O	79	214	z	122	169
%	37	108	P	80	215	{	123	192
&	38	80	Q	81	216	|	124	106
'	39	125	R	82	217	}	125	208
(	40	77	S	83	226	~	126	161
)	41	93	T	84	227	DEL	127	7
*	42	92	U	85	228			

Table F.2-1. PDF to EBCDIC Character Set Translation Table (end)

Char	EBCDIC	PDF	Char	EBCDIC	PDF	Char	EBCDIC	PDF
NUL	0	0	CU2	43	0		86	0
SOH	1	1		44	0		87	0
STX	2	2	ENQ	45	5		88	0
ETX	3	3	ACK	46	6		89	0
PF	4	0	BEL	47	7	!	90	33
HT	5	9		48	0	$	91	36
LC	6	0		49	0	*	92	42
DEL	7	127	SYN	50	22	)	93	41
GE	8	0		51	0	;	94	59
RLF	9	0	PN	52	0	# ^	95	94

Table F.2-2. EBCDIC to PDF Character Set Translation Table (continued)

SMM	10	0	RS	53	0	-	96	45
VT	11	11	UC	54	0	/	97	47
FF	12	12	EOT	55	4		98	0
CR	13	13		56	0		99	0
SO	14	14		57	0		100	0
SI	15	15		58	0		101	0
DLE	16	16	CU3	59	0		102	0
DC1	17	17	DC4	60	20		103	0
DC2	18	18	NAK	61	21		104	0
TM	19	19		62	0		105	0
RES	20	0	SUB	63	26	|	106	124
NL	21	0	SPACE	64	32	,	107	44
BS	22	8		65	0	%	108	37
IL	23	0		66	0	_	109	95
CAN	24	24		67	0	>	110	62
EM	25	25		68	0	?	111	63
CC	26	0		69	0		112	0
CU1	27	0		70	0		113	0
IFS	28	28		71	0		114	0
IGS	29	29		72	0		115	0
IRS	30	30		73	0		116	0
IUS	31	31	cent	74	0		117	0
DS	32	0	.	75	46		118	0
SOS	33	0	<	76	60		119	0
FS	34	0	(	77	40		120	0
	35	0	+	78	43	`	121	96
BYP	36	0	|	79	124	:	122	58
LF	37	10	&	80	38	#	123	35
ETB	38	23		81	0	@	124	64
ESC	39	27		82	0	'	125	39
	40	0		83	0	=	126	61
	41	0		84	0	"	127	34
SM	42	0		85	0			

Table F.2-2. EBCDIC to PDF Character Set Translation Table (continued)

Char	EBCDIC	PDF	Char	EBCDIC	PDF	Char	EBCDIC	PDF
	128	0		171	0	O	214	79
a	129	97		172	0	P	215	80
b	130	98	[	173	91	Q	216	81
c	131	99		174	0	R	217	82
d	132	100		175	0		218	0
e	133	101		176	0		219	0
f	134	102		177	0		220	0
g	135	103		178	0		221	0
h	136	104		179	0		222	0
i	137	105		180	0		223	0
	138	0		181	0	\	224	92
	139	0		182	0		225	0
	140	0		183	0	S	226	83
	141	0		184	0	T	227	84
	142	0		185	0	U	228	85
	143	0		186	0	V	229	86
	144	0		187	0	W	230	87
j	145	106		188	0	X	231	88
k	146	107	]	189	93	Y	232	89
l	147	108		190	0	Z	233	90
m	148	109		191	0		234	0
n	149	110	{	192	123		235	0
o	150	111	A	193	65		236	0
p	151	112	B	194	66		237	0
q	152	113	C	195	67		238	0
r	153	114	D	196	68		239	0
	154	0	E	197	69	0	240	48
	155	0	F	198	70	1	241	49
	156	0	G	199	71	2	242	50
	157	0	H	200	72	3	243	51

Table F.2-2. EBCDIC to PDF Character Set Translation Table (continued)

	158	0	I	201	73	4	244	52
	159	0		202	0	5	245	53
	160	0		203	0	6	246	54
~	161	126		204	0	7	247	55
s	162	115		205	0	8	248	56
t	163	116		206	0	9	249	57
u	164	117		207	0	|	250	0
v	165	118	}	208	125		251	0
w	166	119	J	209	74		252	0
x	167	120	K	210	75		253	0
y	168	121	L	211	76		254	0
z	169	122	M	212	77	EO	255	0
	170	0	N	213	78			

Table F.2-2. EBCDIC to PDF Character Set Translation Table (end)

Appendix G. Reserved Identifiers

A reserved identifier (or "keyword") is one that is part of MAINSAIL's syntax, and cannot be declared or defined by the programmer. The reserved identifiers are shown in Table G-1. Identifiers beginning with dollar signs should never be declared by the programmer, whether they appear in this list or not.

$ALWAYS	$BEGINC	$BUILTIN
$CASEC	$CLASSOF	$CONTINUEC
$DIRECTIVE	$DOC	$DONEC
$EFC	$EXPR	$FORC
$GLOBALREDEFINE	$HANDLE	$HANDLEB
$ISCONSTANT	$KINDOF	$LEGALNOTICE
$SHARED	$STMT	$STMTB
$THISFILENAME	$TYPEOF	$UNDCL
$WITH	$WITHB	ADDRESS
AND	ARRAY	BEGIN
BEGINSCAN	BITS	BOOLEAN
CASE	CHARADR	CHECK
CHECKING	CLASS	CLR
COMPILETIME	CONTINUE	DCL
DEFINE	DIV	DO
DOB	DONE	DONESCAN
DOWNTO	DSP	EB
EF	EL	ELSE
ELSEC	ENCODE	END
ENDC	FALSE	FINAL

Table G-1. Reserved Identifiers (continued)

FOR	FORWARD	GENERIC
GEQ	IF	IFC
INF	INIT	INITIAL
INLINE	INTEGER	IOR
LEQ	LONG	MAX
MESSAGE	MIN	MOD
MODIFIES	MODULE	MSK
NEEDANYBODIES	NEEDBODY	NEQ
NOCHECK	NOT	NTST
NTSTA	NULLADDRESS	NULLARRAY
NULLCHARADR	NULLPOINTER	OF
OFB	OPTIONAL	OR
OWN	POINTER	PROCEDURE
PRODUCES	REAL	REDEFINE
REPEATABLE	RESTOREFROM	RETURN
SAVEON	SHL	SHR
SKIPSCAN	SOURCEFILE	SPECIAL
STRING	THEN	THENB
THENC	TO	TRUE
TST	TSTA	USES
UNTIL	UPTO	WHILE
XOR		

Table G-1. Reserved Identifiers (end)

Appendix H. Predefined Non-Reserved Identifiers without Dollar Signs

The identifiers in Table H-1 are predefined by MAINSAIL but do not begin with the dollar sign character. Not all of these identifiers are intended for use by the programmer, but the programmer must avoid declaring an identifier with a name that conflicts with one listed in Table H-1.

AAREAD	AAWRITE	ABS
ACLEAR	ACOPY	ACOS
ADDRESSCODE	ADISPLACE	ADISPLACEMENT
ALCLEAR	ALCOPY	ALDISPLACE
ALDISPLACEMENT	ALOAD	ALTEROK
APPEND	ARYCLEAR	ARYCOPY
ARYDISPOSE	ASIN	ASTORE
ATAN	BAREAD	BAWRITE
BDREAD	BDWRITE	BFSCAN
BINARY	BIND	BITSCODE
BLOAD	BMASK	BOAREAD
BOAWRITE	BOLOAD	BOOLEANCODE
BOSTORE	BREAK	BSREAD
BSSCAN	BSTORE	BSWRITE
BTREAD	BTTYWRITE	BTWRITE
CALOAD	CAREAD	CASTORE
CAWRITE	CCLEAR	CCLOAD
CCOPY	CCREAD	CCSTORE
CCWRITE	CDISPLACE	CDISPLACEMENT
CEILING	CFREAD	CFWRITE

Table H-1. Non-Reserved Identifiers without Dollar Signs (continued)

CHARADRCODE	CLASSSIZE	CLCLEAR
CLCOPY	CLEAR	CLOAD
CLOSE	CLOSELIBRARY	CMDFILE
CMDMATCH	COMPARE	CONFIRM
COPY	COS	COSH
CREAD	CREATE	CSREAD
CSWRITE	CVA	CVAC
CVALB	CVALI	CVAP
CVARY	CVARYP	CVB
CVBI	CVBLB	CVBLI
CVBS	CVC	CVCA
CVCL	CVCS	CVCU
CVI	CVIB	CVILB
CVILI	CVILR	CVIR
CVIS	CVL	CVLB
CVLBA	CVLBB	CVLBI
CVLBLI	CVLBS	CVLI
CVLIA	CVLIB	CVLII
CVLILB	CVLILR	CVLIR
CVLIS	CVLR	CVLRI
CVLRLI	CVLRR	CVLRS
CVP	CVPA	CVR
CVRI	CVRLI	CVRLR
CVRS	CVS	CVSB
CVSC	CVSI	CVSL
CVSLB	CVSLI	CVSLR
CVSR	CVSU	CVU
CWRITE	DATAFILE	DATAOPEN
DELETE	DISCARD	DISPLACE
DISPLACEMENT	DISPOSE	ENTERLOGICALNAME

Table H-1. Non-Reserved Identifiers without Dollar Signs (continued)

EOF	EQU	ERRMSG
ERROROK	EXIT	EXP
EXPONENT	FASTEXIT	FATAL
FILE	FIRST	FIXED
FLDREAD	FLDWRITE	FLOOR
FORMATTED	GETPOS	HEX
IABS	IAREAD	IAWRITE
IDREAD	IDWRITE	ILOAD
INPUT	INTEGERCODE	ISALPHA
ISLOWERCASE	ISNUL	ISREAD
ISTORE	ISUPPERCASE	ISWRITE
ITREAD	ITTYWRITE	ITWRITE
KEEPNUL	LAST	LBAREAD
LBAWRITE	LBDREAD	LBDWRITE
LBLOAD	LBMASK	LBSREAD
LBSTORE	LBSWRITE	LBTREAD
LBTTYWRITE	LBTWRITE	LDISPLACEMENT
LENGTH	LIABS	LIAREAD
LIAWRITE	LIDREAD	LIDWRITE
LILOAD	LISREAD	LISTORE
LISWRITE	LITREAD	LITTYWRITE
LITWRITE	LN	LOG
LOGFILE	LONGBITSCODE	LONGINTEGERCODE
LONGREALCODE	LOOKUPLOGICALNAME	LRABS
LRACOS	LRAREAD	LRASIN
LRATAN	LRAWRITE	LRCEILING
LRCOS	LRCOSH	LRDREAD
LRDWRITE	LREXP	LRFLOOR
LRLN	LRLOAD	LRLOG
LRSIN	LRSINH	LRSQRT

Table H-1. Non-Reserved Identifiers without Dollar Signs (continued)

LRSREAD	LRSTORE	LRSWRITE
LRTAN	LRTANH	LRTREAD
LRTRUNCATE	LRTTYWRITE	LRTWRITE
MODBIND	MODDISPOSE	MODUNBIND
MSGME	MSGMYCALLER	NEW
NEWARRAY	NEWDATASECTION	NEWPAGE
NEWRECORD	NEWSCRATCH	NEWSTRING
NEWUPPERBOUND	NEXTALPHA	NORESPONSE
OCTAL	OMIT	OPEN
OPENLIBRARY	OUTPUT	PAGEDISPOSE
PAREAD	PAWRITE	PCLEAR
PCOPY	PDISPLACE	PDISPOSE
PLOAD	POINTERCODE	PREVALPHA
PROCEED	PROMPT	PSTORE
RABS	RACOS	RANDOM
RAREAD	RASIN	RATAN
RAWRITE	RCEILING	RCOS
RCOSH	RCREAD	RCWRITE
RDREAD	RDWRITE	READ
REALCODE	RELFILENAME	RELMODNAME
RELPOS	RETAIN	REXP
RFLOOR	RLN	RLOAD
RLOG	RSIN	RSINH
RSQRT	RSREAD	RSTORE
RSWRITE	RTAN	RTANH
RTREAD	RTRUNCATE	RTTYWRITE
RTWRITE	SAREAD	SAWRITE
SBIND	SCAN	SCANREL
SCANSET	SCRATCHDISPOSE	SDISPOSE
SETFILENAME	SETMODNAME	SETPOS

Table H-1. Non-Reserved Identifiers without Dollar Signs (continued)

SFREAD	SFSCAN	SFWRITE
SIN	SINH	SIZE
SLOAD	SNEWDATASECTION	SQRT
SSREAD	SSSCAN	SSTORE
SSWRITE	STORE	STRINGCODE
STTYWRITE	SUNBIND	TAB
TAN	TANH	TEXTFILE
TEXTOPEN	THISDATASECTION	TRUNCATE
TTYCWRITE	TTYREAD	TTYWRITE
TYPESIZE	UNBIND	UPPERCASE
USEKEYWORD	WARNING	WRITE

Table H-1. Non-Reserved Identifiers without Dollar Signs (end)

Appendix I. Synonyms

Table I-1 shows synonymous forms supported by the MAINSAIL compiler. In the case of curly brackets, the compiler does not require that curly brackets be matched only by curly brackets and square brackets only by square brackets; e.g., if a is a one-dimensional array and i an integer, "a{i]" and "a[i}" are legal and equivalent.

Usual Form	Synonymous Form
:=	_
^	**
NEQ	<>
LEQ	<=
GEQ	>=
SHL	<<
SHR	>>
[	{
]	}
DOB	DO BEGIN
EL	ELSE
EF	ELSE IF
EB	ELSE BEGIN
THENB	THEN BEGIN
OFB	OF BEGIN
$HANDLEB	$HANDLE BEGIN
$WITHB	$WITH BEGIN

Table I-1. MAINSAIL Synonyms

Appendix J. Restrictions

This appendix lists restrictions on sizes of MAINSAIL objects and values allowed in operations. It is not a complete list. Each operating-system-dependent MAINSAIL user's guide may contain additional implementation-dependent restrictions. The operating-system-dependent MAINSAIL user's guide also lists the size of the data types on that implementation.

J.1. Portable Data Type Ranges and Data Structure Size Limits

The portable ranges of the MAINSAIL data types and size limits of the MAINSAIL data structures are shown in Table J.1-1. The limits listed are inclusive. The sizes listed for strings and arrays are subject to the availability of sufficient memory to contain the data structure.

A class or module may have no more than 4095 entries, where an entry is a series of fields all of the same data type. The data section size includes all own variables, not just interface variables, and possibly hidden variables not declared by the programmer as well.

J.2. Interface Procedures in a Module

A limit as low as 1600 interface procedures per module may be imposed. A compiler error message is issued if the limit is exceeded.

J.3. Local Variable Limitations

A limit as low as 255 parameters and local variables per procedure may be imposed. A compiler error message is generated if this limit is exceeded. Temporaries generated by the compiler are counted in the limit.

J.4. String Constants in a Module

The number of different string constants allowed in a module is 4095. A string constant may be no longer than 16383 characters (string variables may be up to 32766 characters).

Type or Structure	Portable Range
Boolean	TRUE or FALSE
Integer	-32767 to +32767
Long Integer	-2147483647 to +2147483647
Real	1.0E-38 to 1.0E+38, six decimal digits
Long Real	1.0E-38 to 1.0E+38, eleven decimal digits
String	0 to 32766 eight-bit characters
Short Array	
one dimension	bounds -32767 to +32767
two or three dimensions	see Section 7.9
Long Array	
one dimension	bounds -2147483647 to +2147483647
two or three dimensions	see Section 7.9
Record	up to 32767 storage units
Data Section	up to 16383 storage units
Class or Module	up to 4095 entries

Table J.1-1. Portable Data Type Ranges and Data Structure Size Limits

J.5. Size of a Procedure

Each implementation may place an upper limit on the size of a procedure. Such limits are usually reasonably large, but not necessarily lavish. It is difficult to translate such limits into a quantity readily observable by the programmer, e.g., number of statements.

J.6. Number of Cases in a Case Statement

A Case Statement may contain any number of selectors in the range -32767 to 32767, provided the maximum procedure size is not exceeded.

J.7. Uninitialized Variables

Operations using the values of uninitialized variables have undefined results. Especial care is required for (long) real values, since some bit patterns may not represent valid (long) real values.

J.8. Init Statement Counts

Init Statement bracketed counts are limited only by the range of an integer.

J.9. Init Statement Constants

The total number of different constants in an Init Statement (where a number of consecutive elements specified by a bracketed count are considered to be a single constant, not different constants) may not exceed 32767 for non-string arrays. For string arrays, the different constants in an Init Statement may not total more than 16383 characters.

J.10. copy and clear Addresses

The instances of the generic procedures clear and copy that deal with addresses must take a number of storage units that is a multiple of the size of the smallest data type on the host machine. The effect of specifying other values is not defined.

J.11. FOR-Clause Limit Values

The largest (long) integer must not be used as a limit value in an "UPTO" FOR-clause, nor the most negative (long) integer as a limit value in a "DOWNTO" FOR-clause.

Appendix K. Modules Shipped in a Standard System

Table K-1 lists the objmods that may be shipped in a standard MAINSAIL system (i.e., all modules contained in the union of all XIDAK products). The list may include some modules not shipped in every system, and may not contain modules shipped in some systems. Table K-2 shows objmods shipped in runtime-only systems (the union of all such systems offered by XIDAK). Users should avoid creating modules with the same name as a MAINSAIL system module. All the standard module names are subject to change. In a future release, XIDAK will introduce facilities to make it easier for users to avoid name conflicts with system modules.

Modules marked with an asterisk are maintenance utilities intended for XIDAK use or "non-critical" utilities described in the "MAINSAIL Utilities User's Guide". They may be deleted from well-debugged systems shipped for runtime-only purposes without adverse effect. Modules marked with a plus sign are display modules (primarily for use with MAINEDIT), and may be deleted if no program that uses the display modules is to be run.

ADR*	AM60+	ARYMOD	AT386+	BIGSUN+	CALLS*	CBK
CLOSEF*	CONCHK*	CONF	COPIER	CVRLR	D100+	D400+
D460+	D460C+	DATAME+	DATDIO	DATE*	DATTIO	DELFIL
DIR	DISK	DISPSE	DTIMOD	DVIEW	ELSEX	ENCMOD
ENDX	ERRMOD	FILCCH	FILDIF*	FILMOD	FILMRG*	FNDPRC*
GCCHP	GENTBL	HEATH+	HP300H+	HPTERM+	HSHMOD	IFX
INTLIB	INTSCH	KERMOD	KILLCO	LIB	LIBEX	LINCOM
LINDPY+	LRMATH	MAINEX	MEM	MEMDPY+	MM*	MODLIB
NUL	OPENF*	PACK*	PATSCN*	PDFDIO	PDFMOD	PDFTIO
PMERGE	PRMAP*	PRNTCO	PRNTRE*	PRTREE*	RAISE	RANMOD
RECIO	RECMOD	REMTAB*	RMATH	RNMFIL	RSMCO	SRTMOD
STAMP	STRHDR	SUBCMD	SUN+	SUN3+	SUN46+	SUNSCR
SYS	TARGET	TELEVI+	TOHEX*	TREEIO	TRMCAP	TRNCAT*
TTY	TVI925+	TVI950+	TVIEW	TXTDIO	TXTTIO	UM2CNF
UNPACK*	UTYMOD	VIS200+	VIS550+	VT100+	VT102+	VT102M+
WRDCOM	WY43+	WY50+	WY5043+	WY75+	XREF	XRFMRG

Table K-2. Objmods Shipped in Runtime-Only MAINSAIL Systems

A20D	A20G	ADR*	AEGCNF	AEGFCG	AEGFPG	AEGTCG
AEGTPG	AM60+	ARYMOD	AT386+	BCSTR	BIGSUN+	CALLS*
CBK	CLOSEF*	CLPD	CLPG	CMPHDR	CMSCNF	CMSF7G
CMSFAG	CMSFFG	CMST7G	CMSTAG	CMSTFG	COMPIL	CONCHK*
CONF	COPIER	CVRLR	D100+	D400+	D460+	D460C+
DATAME+	DATDIO	DATE*	DATMGR	DATTIO	DEBUG	DELFIL
DIR	DISASM	DISK	DISPSE	DPYEXE	DTIMOD	DVIEW
E29	ED9	EDIT	ELSEX	ELXD	ELXG	EMBCNF
EMBFEG	EMBTEG	ENCMOD	ENDX	ERRMOD	FCSTR	FILCCH
FILDIF*	FILMOD	FILMRG*	FNDPRC*	GCCHP	GENTBL	HEATH+
HP300H+	HPTERM+	HPTRAN	HSHMOD	I38D	I38G	I96D
I96G	IBMD	IBMG	IFX	INTERP	INTLIB	INTMGR
INTSCH	ITFXRF	IXTRAN	KERMOD	KILLCO	LIB	LIBEX
LINCOM	LINDPY+	LRMATH	M20D	M20G	M68D	M68G
MAINED	MAINEX	MAINPM	MAINVI	MEDT	MEM	MEMDPY+
MM*	MODLIB	MVD	MVG	NUL	NULMGR	OPENF*
OPT	PACK*	PAKMOD	PASS1	PASS2	PATS	PATSCN*
PDFBLT	PDFDIO	PDFMOD	PDFTIO	PMERGE	PRMAP*	PRNTCO
PRNTRE*	PRTREE*	RAISE	RANMOD	RDGD	RDGG	RECIO
RECMOD	RECOM	REMTAB*	RMATH	RNMFIL	RSMCO	SCRFIL
SCSTR	SPAD	SPAG	SRTMOD	STAMP	STRBLT	STRCHK
STRHDR	STRTXT	SUBCMD	SUN+	SUN3+	SUN46+	SUNSCR
SYS	TARGET	TELEVI+	TM9	TOHEX*	TREEIO	TRMCAP
TRNCAT*	TTY	TVI925+	TVI950+	TVIEW	TXTDIO	TXTMGR
TXTTIO	UA2CNF	UA2FPG	UA2FXG	UA2TPG	UA2TXG	UCLCNF
UCLFCG	UCLTCG	UDGCNF	UDGTCG	UI3CNF	UI3FCG	UI3TCG
UI9CNF	UI9FCG	UI9TCG	UIBCNF	UIBFCG	UIBTCG	UM2CNF
UM2FCG	UM2FHG	UM2FPG	UM2FXG	UM2TCG	UM2THG	UM2TPG
UM2TXG	UM6CNF	UM6FAG	UM6FCG	UM6FFG	UM6FHG	UM6TAG
UM6TCG	UM6TFG	UM6THG	UMVCNF	UMVTCG	UNPACK*	UPDDPY
URDCNF	URDFCG	URDTCG	USPCNF	USPFCG	USPTCG	UTYMOD
UVACNF	UVAFCG	UVAFVG	UVATCG	UVATVG	UXACNF	UXAFCG
UXATCG	V10BLT	VAXD	VAXG	VIS200+	VIS550+	VMSCNF
VMSFCG	VMSFVG	VMSTCG	VMSTVG	VT100+	VT102+	VT102M+
WRDCOM	WY43+	WY50+	WY5043+	WY75+	XAD	XAG
XCMCNF	XCMF7G	XCMFAG	XCMFFG	XCMT7G	XCMTAG	XCMTFG
XREF	XRFMRG					

Table K-1. Standard MAINSAIL System Objmods

Index

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