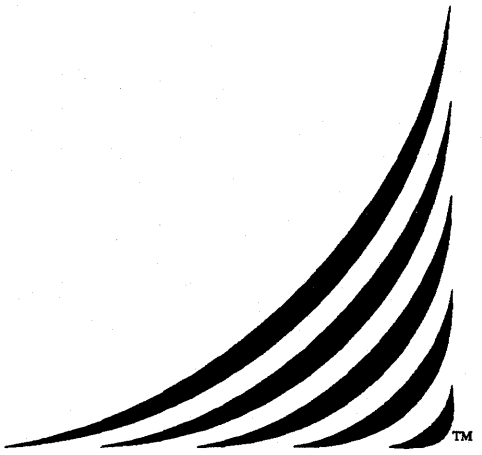
V12

MAINSAIL®

Tools User's Guides



MAINSAIL®

Tools User's Guides

24 March 1989

Table of Contents

Appendices

List of Examples

List of Figures

List of Tables

1. MAINSAIL Tools User's Guides

This document contains descriptions of the programs provided by XIDAK as a part of the MAINSAIL environment. Some of the programs are included in the basic runtime package; others must be purchased separately. For complete information on pricing and availability, consult a current "XIDAK Product Catalog".

Some of the facilities described herein are meant to be run interactively; some are invoked from a program; and some may be invoked either way. Those that are invoked from a program are not included in the "MAINSAIL Language Manual" because either they are purchased separately from the basic runtime package or they require the program to include special directives (usually "SOURCEFILE" or "RESTOREFROM" directives) or explicit allocation of certain modules in order to have access to the facilities.

1.1. Conventions Used in This Document

1.1.1. 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 1.1.1-1, "Prompt:" is written by the computer; the user types "response" and then presses the end-of-line key.

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

Example 1.1.1-1. How User Input Is Distinguished

1.1.2. 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 1.1.2-1.

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

Example 1.1.2-1. Syntax of a Mailing Address

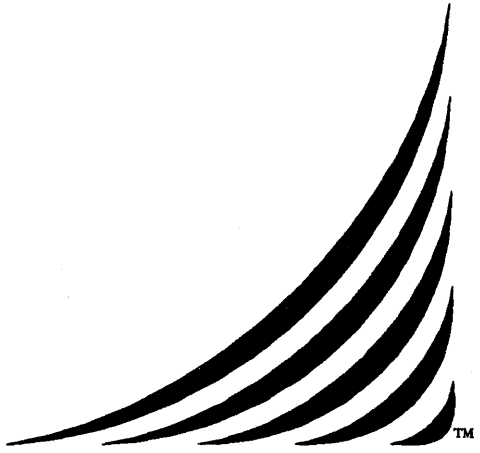
Optional elements in command or syntax descriptions are often enclosed in curly brackets ("{" and "}"). For example, a string of characters specified as "{A}B{C}" could have any one of the forms "B", "BC", "AB", and "ABC". Alternatives may be enclosed in square brackets (or curly brackets, if all alternatives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

1.1.3. 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.



MAINSAIL® Compiler

User's Guide

24 March 1989

2. Introduction

This document describes the MAINSAIL compiler and related modules. The MAINSAIL compiler consists of a set of modules that are independent of the host and target machine, and a set of code generation modules that are independent of the host machine but know about the target machine.

A MAINSAIL program is a collection of modules. The modules are not linked before execution into a single unit as is common for most programming languages. Instead, MAINSAIL dynamically brings modules into memory as needed and takes care of intermodule communication in a machine-independent manner.

The chief outputs of the MAINSAIL compiler are:

- objmod (object module): a data file that contains the executable code.
- intmod (intermediate module): a text file that encodes information about the module, used by programs such as the compiler, disassembler, and debugger.

MAINSAIL directly executes objmods (they are not linked with a linkage editor). An intmod is produced only if certain non-default compiler options are set, as described later in this document.

Intmods are portable files; an intmod produced on one system may be used (for the same target) on another system. Intmods are target-specific; an intmod compiled for one target operating system cannot be used in compiling for another operating system. Some care must be used in shipping intmods from one system to another, as they may contain special characters that do not appear in ordinary text files; they should therefore be shipped as binary or (in MAINKERMIT) "PTEXT" files.

Intmods may be stored in intmod libraries, called intlibs, as described in the INTLIB section of the "MAINSAIL Utilities User's Guide". When the compiler looks for the intmod for some module FOO, it first searches all open intlibs for FOO. If it does not find the intmod in any open library, it then looks for the file named "int-xxx:foo", where "xxx" is the first three letters of the target operating system name abbreviation (as described later, a searchpath is often used to map this file name to another).

"Host machine" refers to the machine on which the compiler is running; "target machine" refers to the machine for which code is being generated.

2.1. Version

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

3. Invoking the MAINSAIL Compiler

COMPIL is the name of the top-level compiler module. To compile a module, invoke COMPIL as you would any other MAINSAIL module. The invocation of MAINSAIL is described in the appropriate operating-system-dependent MAINSAIL documentation, and the invocation of MAINSAIL modules from the MAINSAIL executive, MAINEX, is described in the "MAINSAIL Utilities User's Guide".

When the compiler is invoked it prints a herald identifying itself and prompts for input. Table 3-1 lists valid responses to this prompt (the list is displayed when "?" is typed). When compilation is complete, the compiler prompts for the next file to be compiled. To exit the compiler, type <eol> to this prompt.

```
<srcFile>      compile the module(s) in <srcFile>
<srcFile>,     same as above, and enter subcommand mode
<modName>,     recompile module, enter subcommand mode
<intmodFile>,  recompile module, enter subcommand mode
=              use previous response and subcommands
=,             as as above, and enter subcommand mode
,              enter subcommand mode
# ...          comment (line is discarded)
<eol>          stop compiling
```

Table 3-1. Compiler Commands

<srcFile> is the name of a file that contains the source code that is to start the compilation. If <srcFile> cannot be opened, the compiler prints an error message and returns to the "compile (? for help):" prompt. The comma after <srcFile> indicates that subcommand mode is to be entered. Subcommands provide a means of specifying compiler options; they are described in Chapter 5.

<modName> and <intmodFile> are used when a module is incrementally recompiled, as described in Chapter 4.

Responding to the compiler prompt with just "=<eol>" or "=,<eol>" tells the compiler to use the same <srcFile>, <modName>, or <intmodFile> as it did for the previous compilation. This

avoids typing the same file name in the event that it has been modified and is to be compiled again. Nonsticky subcommands are remembered from the previous compilation, and need not be specified again; sticky subcommands remain in effect, as usual (see Chapter 5 for a discussion of "sticky" and "nonsticky" subcommands).

By default, the objmod and intmod are put into files with names derived from the name of the compiled module. If the compiled module is FOO and the first three letters of the operating system name abbreviation are "xxx", then the compiler uses "xxx-obj:foo" and "xxx-int:foo" for the names of the objmod and intmod, respectively. For example, on VAX/VMS, the file names for FOO are "vms-obj:foo" and "vms-int:foo". Searchpaths are expected to have been set up to map these names into actual file names. A typical searchpath supplied by XIDAK maps "xxx-int:foo" into "foo-xxx.int" and "xxx-obj:foo" into "foo-xxx.obj"; if the mapping differs on a particular system, it is described in the appropriate operating-system-dependent MAINSAIL documentation. The default searchpaths allow the same module to be compiled for different targets without the output files getting mixed up. If you always compile for the same target, you might consider changing the searchpaths so that the default file names are shorter, e.g., "foo.int" and "foo.obj". Searchpaths are described in the MAINEX section of the "MAINSAIL Utilities User's Guide" under the "SEARCHPATH" subcommand entry.

Subcommand mode can be entered without specifying a file or module by typing just a comma followed by <eol>. In this case the specified options apply to the next compilation even if they are nonsticky. This is useful in scripts that want to specify compiler options independently of which module is compiled next.

Example 3-2 shows how to compile a module FOO in the file "foo.msl". The example assumes the host system is MC68020/MC68881 UNIX, which has system name abbreviation "UM20".

The intmod for FOO is not produced since there are no compiler options in effect that cause it to be generated. Unless the user requests otherwise, the compiler displays the names and page numbers of the source files and the names of forward procedures compiled from source libraries as they are compiled.

When the compiler finds an error in the source text, it issues an error message by calling the system procedure errMsg (unless the "NORESPONSE" option is in effect). The following responses to the "Error response:" prompt are often appropriate:

- "QUIT": exit MAINSAIL.
- "MAINSAIL: Abort program": exit the compiler. Any unique abbreviation will do ("a p" is usually sufficient).
- "MAINSAIL compiler: Abort compilation": abort only the current compilation ("a c" is usually sufficient). The compiler prompts for the next file to compile.

```
MAINSAIL (R) Version 12.10 (? for help)
Copyright (c) 1984, 1985, 1986, 1987, 1988, and 1989 by
 XIDAK, Inc., Menlo Park, California, USA.
*compil<eol>

MAINSAIL (R) Compiler
Copyright (c) 1984, 1985, 1986, 1987, 1988, and 1989 by
 XIDAK, Inc., Menlo Park, California, USA.

compile (? for help): foo.msl<eol>
Opening intmod for $SYS...

foo 1
Objmod for FOO stored on foo-um2.obj
Intmod for FOO not stored

compile (? for help): <eol>
```

Example 3-2. Compiler Example

- <eol>: continue compiling the same module. Although valid output is not generated if a compiler error occurs, it is often convenient to see the remaining errors in the module so that they can all be corrected before recompilation is attempted.

- "MAINSAIL compiler: Recompile procedure" (which can usually be abbreviated to "r p"): back up to the start of the current procedure and start over again. The compiler closes the source file and issues a message telling the user to repair the procedure. When it gets an <eol> response, the compiler reopens the source file, positions to where it remembered the start of the procedure to be, and makes sure the procedure is really still there; if so, it resumes compilation at the start of the procedure. The effect is undefined if changes were made in the module's source text before the start of the affected procedure. In order to repair the source text for which the compiler is waiting, the user must either invoke (or already be running) MAINEDIT, in which case it is a simple matter to edit any files and save them before continuing the compilation, or there must be some operating-system-dependent way of running a text editor to do the repair before returning to the MAINSAIL compiler session.

- "MAINSAIL compiler: Delayed recompile procedure" (which can usually be abbreviated to "d r p"): same as above, except the compiler continues until the end of

the current procedure, at which point it goes into the dialogue and then backs up as described above. This is appropriate when you want to find all errors in the current procedure before backing up. However, sometimes the error is such that it is better to back up immediately rather than deal with all subsequent error messages, which may in fact be due to the first error (e.g., an undeclared local variable).

Using the "recompile procedure" responses makes it possible to get an error-free compilation in a single compilation, unless a change must be made at a point before the current procedure.

No intmod or objmod is generated if the compiler detects any errors in the first pass that are not repaired as described above. If the compiler detects errors during the code generation phase, the intmod and objmod are produced, but they are probably not valid.

4. Incremental Recompilation

The MAINSAIL compiler can "incrementally recompile" a module on a per-procedure basis (the term "recompile" is sometimes used to mean "incrementally recompile" when the intent is clear).

An incremental recompilation is indicated by specifying <modName> or <intmodFile> to the compiler prompt, followed by a comma to enter subcommand mode. One of the subcommands must be "RECOMPILE", which specifies the procedure(s) to be recompiled. The compiler must be able to find the old intmod, and, if code is generated, the old objmod.

Unless otherwise specified using "OUTINTFILE" or "OUTINTLIB", the new intmod and objmod are stored under the default file names; they do not automatically replace the old intmod and objmod.

The advantage of incremental recompilation over full compilation is faster compilation, since only the specified procedures are processed. The disadvantage is that the new intmod and objmod are larger than the old ones, since the code for the recompiled procedures is added to the end of the objmod, and the information for the recompiled procedures is added to the end of the intmod, but the space occupied by the original code and information is still present. Since incremental recompilation is used during program development, the larger intmod and objmod should not be a major drawback. At any time a compilation of the entire module can be used to produce an objmod and intmod with no unused space.

In order to be able to recompile a module, it must be initially compiled with the "INCREMENTAL" option. This tells the compiler to put sufficient information into the intmod to allow recompilation. The "RECOMPILE" subcommand tells the compiler that an incremental recompilation is to be done. This subcommand takes a list of names, separated by spaces, of those procedures that have been changed. The effects of changes to the source text of the module outside the bodies of these procedures are undefined. The compiler issues an error message if it detects a change outside of the specified procedures, but does not necessarily detect all such changes. The user must be careful to remember and include in the list all procedures that were changed.

It is somewhat inconvenient that, in order to use incremental recompilation, the user must tell the compiler what procedures have been changed; it would be easier if the compiler could automatically figure this out. It is also inconvenient that if any change, however slight (e.g., even to a comment), is made outside of any procedure, then incremental recompilation cannot be used. These restrictions may be relaxed in a future release.

The options specified by the following compiler subcommands are cleared if the "RECOMPILE" or "INCREMENTAL" option is set:

```
MODTIME MONITOR PERMOD PERPROC PERSTMT PROCTIME NOMONITOR
```

In other words, a module cannot be compiled with the "INCREMENTAL" option, or recompiled, and also have a performance monitoring option in effect. This is due to the difficulty of incrementally altering the effect of these options, and this restriction may be lifted in a future release.

When recompiling, the options affected by the following compiler subcommands are set from the old intmod and the current settings are ignored for the current compilation:

```
ALIST DEBUG GENCODE GENINLINES INCREMENTAL SAVEON
NOALIST NODEBUG NOGENCODE NOGENINLINES NOINCREMENTAL
NOSAVEON
```

The treatment of subcommands governing optimization, checking, and arithmetic checking is described in the "MAINSAIL Language Manual".

An error is reported at the start of a compilation if the "INCREMENTAL" or "RECOMPILE" subcommand is in effect and the "FLI" subcomand is also in effect.

All other compiler subcommands have their normal effect during a recompilation. All sticky subcommands have their normal effects on later compilations, even if ignored on the current compilation because the "INCREMENTAL" or "RECOMPILE" subcommand is in effect.

If a procedure is recompiled, and was expanded inline in the module being recompiled (either because it was declared "INLINE" or because it was invoked "INLINE"), then expansions of the procedure in non-recompiled procedures cannot be updated. Whenever the compiler detects this situation it prompts for whether it should abort or proceed with code generation.

5. Compiler Subcommands

There are many subcommands available for use with the compiler. Subcommands affect the objmod and/or intmod generated for the compilation and the information that the compiler displays during the compilation. Subcommand mode is entered by terminating the input line typed to the compiler prompt with a comma.

When subcommand mode is entered, the subcommand prompt ">" is displayed. Any number of subcommands may be entered at this point, one per line. Subcommand mode is exited by typing <eol> to the ">" prompt.

Multi-line subcommands may be entered by terminating each line except the last with a backslash ("\"). The backspace and following eol are discarded, and all the lines entered are concatenated to form the subcommand.

File names in subcommands cannot contain space characters.

Valid subcommands are listed in Table 5-1. In this table, commands that do not have "(nonsticky)" in their description remain in effect for all subsequently compiled modules until either another subcommand turns it off or the compiler exits (unless they perform a single, one-time-only action; e.g., "ABORT", "REDEFINE"). Nonsticky subcommands apply only to the next module compiled.

Any sticky subcommand may be made temporarily nonsticky by immediately following the subcommand (before any arguments) with an asterisk; e.g., "NOGENCODE *" indicates that object module generation is to be omitted only for the next compilation.

XIDAK reserves the right to create compiler subcommands for internal use only. Such subcommands are not documented here and are subject to change and/or removal without notice. Compiler subcommands may change from release to release of MAINSAIL.

Mixed case is used in the subcommands to show the minimum permitted abbreviation of compiler subcommands. The uppercase portion of each compiler subcommand in mixed case is a sufficient abbreviation for that subcommand (as if the upperCase bit were given to cmdMatch, as described in the "MAINSAIL Language Manual").

All of the compiler subcommands except the following may be invoked from MAINSAIL source text with the "$DIRECTIVE" directive, as described in the "MAINSAIL Langauge Manual":

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

Table 5-1. MAINSAIL Compiler Subcommands (continued)

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

Table 5-1. MAINSAIL Compiler Subcommands (continued)

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

For backward compatibility:

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

Table 5-1. MAINSAIL Compiler Subcommands (end)

```
FLDXREF FLI ININTLIB ITFXREF RECOMPILE SLIST TARGET
VERSION NOFLDXREF NOININTLIB NOITFXREF NOSLIST
```

The "$DIRECTIVE" directive is never sticky, i.e., never applies to more than one module, even in the same compiler session.

5.1. "ABORT"

"ABORT" aborts the current compilation while in subcommand mode. This is useful if subcommands have been entered that are realized to be incorrect for the current compilation, and rather than try to undo them, it might be easier just to abort and start over. It is also useful as a "$DIRECTIVE" directive if a condition occurs at compiletime that requires that a compilation be aborted.

5.2. "ACHECK"/"NOACHECK", "ACHECKALL"/"NOACHECKALL"

Default: "NOACHECK"; sticky

The "ACHECK" option 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, achecked code may be significantly larger and slower.

"NOACHECK" turns the "ACHECK" option off. It suppresses the generation of extra instructions; it does not necessarily mean that arithmetic exceptions are not caught.

"ACHECKALL" and "NOACHECKALL" override "ACHECK" and "NOACHECK" directives in the source text of a module. Arithmetic checking, and the subcommands and directives used to control it, are described in more detail in the "MAINSAIL Language Manual".

5.3. "ALIST"/"NOALIST"

Default: "NOALIST"; sticky

"ALIST" causes the compiler to put information into the intmod required if the disassembler is to be used to generate an intermixed source/code listing. The objmod is not affected.

"NOALIST" turns off "ALIST".

5.4. "CHECK"/"NOCHECK", "CHECKALL"/"NOCHECKALL"

Default: "CHECK"; sticky

"CHECK" causes the compiler to generate code that checks for runtime errors (the list of errors checked for is the same as the list governed by the MAINSAIL language "CHECK" directive; consult the "MAINSAIL Language Manual" for details). This option should be used when debugging and testing programs. Once a program is debugged, it can be compiled "NOCHECK" to reduce code size and increase execution speed.

"NOCHECK" turns off "CHECK".

"CHECKALL" and "NOCHECKALL" override "CHECK" and "NOCHECK" directives in the source text of a module. The rules for checking, and the subcommands and directives used to control it, are described in more detail in the "MAINSAIL Language Manual".

5.5. "CMDLINE s"

Not sticky

"CMDLINE s" concatenates the string:

```
s & eol
```

to the end of the current cmdLine string. Whenever the compiler needs to read a line from file cmdFile, it first checks whether cmdLine is non-Zero, and if so, it reads the line from cmdLine rather than cmdFile. For example, interactive defines utilize cmdLine. Thus, responses to interactive defines can be specified in "CMDLINE" subcommands. This is sometimes useful in compilation scripts; e.g., if a compilation may or may not do a single interactive define (based on compiletime evaluation), the response can be put in a "CMDLINE" subcommand so that it does not get in the way even if the interactive define does not occur.

5.6. "DEBUG"/"NODEBUG"

Default: "NODEBUG"; sticky

"DEBUG" causes the compiler to put information into the intmod that is used by the MAINSAIL debugger. A module that is to take part in debugging must be compiled with this option. See the "MAINDEBUG User's Guide" for more information. "DEBUG" slightly increases the size and execution time of the objmod.

Procedures declared "INLINE" (but not "$ALWAYSINLINE") are not expanded inline if "DEBUG" is in effect. Procedures declared "$ALWAYSINLINE" are expanded inline even if "DEBUG" is in effect. Thus, a module that uses inline procedures may be larger and slower for this reason also.

"NODEBUG" turns off "DEBUG".

As a side effect, "DEBUG" sets the "INCREMENTAL" and "GENINLINES" options, and "NODEBUG" clears them. "DEBUG *" has no effect on the current sticky subcommand setting values used in subsequent compilations, and may therefore be more suitable for compilation scripts in which both debuggable and non-debuggable modules are compiled.

5.7. "FLDXREF"/"FLDXREF f"/"NOFLDXREF"

Default: "NOFLDXREF"; not sticky

The "FLDXREF" and "FLDXREF f" options are similar to the corresponding "ITFXREF" options, except that they cause field references of the form "p.f", where "p" is a pointer or address variable, to be listed ("ITFXREF" is for "m.f" only where "m" is a module name). Class names enclosed in angle brackets are listed in place of the module names for the "ITFXREF" option. Both "ITFXREF" and "FLDXREF" can be specified; the output goes to the same listing file if no output file is specified for either (".xrf" file), or if the same output file is specified for both.

The interface cross-reference merger, XRFMRG, is used in conjunction with the "ITFXRF" and "FLDXRF" compiler sobcommands. It is described in Chapter 7.

5.8. "FLI s"

Sticky

The "FLI s" subcommand specifies the direction and language of a foreign module interface compilation. The string s has the form "xy", where x is a character indicating the direction of the FLI compilation and y is a character indicating the foreign language. "FLI ?" displays a list of valid xy combinations. The supported values for x and y are shown in Table 5.8-1 (unsupported values, if any are provided, are not documented).

Thus, "TP" means "To Pascal (from MAINSAIL)", and "FC" means "From C (to MAINSAIL)". The "TARGET" subcommand must also be used if the target is different than the host. At present, only selected foreign-language code generators have been implemented, and some combinations make no sense (e.g., "TV" used on a target processor other than the

```
Field Cross-Reference for Module TOPCMP

<$DEVICECLS>.$DEVNEWBUF          (m:syslib) SFWRITE 13.13
<$MICONF>.$CONFBITS              (m:tops20) STTYWRITE 4.8
<FILE>.$DEVICE                   (m:syslib) SFWRITE 13.11
                                 13.13
      .$REMSIZE                  (m:syslib) SFWRITE 13.13
                                 13.16*
      .$STATUSBITS               (m:syslib) SFWRITE 13.8
                                 13.16*
<SYMBOL>.LINK                    (miscPF) COMPILE 5.96 5.100
<TEXTFILE>.$NEXTTEXT             (m:syslib) SFWRITE 13.15*
```

Figure 5.7-1. "FLDXREF" Sample Output

x	Description
T	To <foreignLanguage>
F	From <foreignLanguage>

y	Description
C	the language "C"
F	FORTRAN
P	Pascal
V	VAX-11 Calling Standard
X	DOMAIN/IX Standard
A	Alliant C or AIX
H	HP/UX C
7	FORTRAN77

Table 5.8-1. The Format of Arguments to the Compiler "FLI" Subcommand

VAX-11). If a non-existent or unavailable combination is selected, the compiler issues an error message to the effect that it cannot open the FLI code generator module, the name of which is formed from the target, x, and y. A list of the FLI code generators that have been implemented may be found in the current "XIDAK Product Catalog".

"FLI<eol>" (i.e., no s specified) turns off the "FLI" subcommand.

5.9. "GENCODE"/"NOGENCODE"

Default: "GENCODE"; sticky

"GENCODE" specifies that an objmod is to be generated.

"NOGENCODE" specifies that no objmod is to be generated. This is useful for verifying that the syntax of a module is correct, or for a module for which just the intmod is desired.

5.10. "GENINLINES"/"NOGENINLINES"

Default: "NOGENINLINES"; sticky

"GENINLINES" specifies that bodies are to be generated for each inline procedure, though calls to inline procedures are still expanded inline. This option is useful when debugging to make procedures declared "$ALWAYSINLINE" callable from the debugger. If some additional procedure bodies are generated due to this option, the objmod is larger.

"DEBUG" sets this option, and "NODEBUG" clears it, so it is rarely useful to use these subcommands explicitly (though they could be used to force code to be generated for an inline procedure for which a disassembly was to be obtained in order to examine the generated code).

"NOGENINLINES" turns off "GENINLINES".

5.11. "INCREMENTAL"/"NOINCREMENTAL"

Default: "NONICREMENTAL"; sticky

"INCREMENTAL" specifies that the intmod is to contain all the information required to support incremental recompilation. This subcommand must be given if the module is ever to be incrementally recompiled. The objmod is not affected.

"DEBUG" sets this option, and "NODEBUG" clears it.

"NOINCREMENTAL" turns off "INCREMENTAL".

5.12. "ININTLIB f"/"NOININTLIB"

Default: "NOININTLIB"; sticky

"ININTLIB f" specifies that the input intmod is in the intlib f. This information is used during an incremental recompilation where just the module name, say FOO, is given to the compiler prompt, and FOO resides in the intlib f. As a side effect, the intlib f remains open for further intmod searches.

"NOININTLIB" turns the "ININTLIB" option off, so that the default intmod search rules are subsequently followed.

5.13. "INOBJFILE f"

Not sticky

"INOBJFILE f" specifies that the object module being recompiled is in the file f. This option is useful only when recompiling, since the compiler must then find the objmod for the module being recompiled. This option need not be given if the objmod is in a file with the default objmod file name, or is in an open objmod library.

There is no corresponding "ININTFILE" subcommand, since the name of the input intmod file can be specified in response to the compiler prompt. If the intmod file has the default name, then the module name can be given in response to the compiler prompt, and the compiler automatically finds the intmod file (and the objmod file if it also has the default name or is in an open objmod library).

5.14. "INOBJLIB f"/"NOINOBJLIB"

Default: "NOINOBJLIB"; sticky

"INOBJLIB f" specifies that the input objmod is in the objmod library f. This information is used during an incremental recompilation, since the compiler must locate the objmod for the module being recompiled. As a side effect, the objmod library f remains open for further objmod searches.

"NOINOBJLIB" turns the "INOBJLIB" option off, so that the default objmod search rules are subsequently followed.

5.15. "ITFXREF"/"ITFXREF f"/"NOITFXREF"

Default: "NOITFXREF"; not sticky

"ITFXREF" and "ITFXREF f" cause the compiler to generate an interface field cross-reference listing that gives the file, name of containing procedure, page, and line of every occurrence of an interface field referenced (both implicitly and explicitly) by the module being compiled. In addition, an asterisk ("*") indicates that a reference alters the (data) field. Procedure and data fields as well as the module's references to its own interface fields are reported. A field "f" referenced using "p.f", where "p" is a pointer variable rather than a module name, is not included in the listing. Use the "FLDXREF" option, described in Section 5.7, to include such references.

If the file name for the cross-reference listing is not specified, the output is written to a file "m.xrf", where "m" is the name of the module being compiled.

Once the "ITFXREF" option is specified, it remains in effect until either the "NOITFXREF" option is specified or the compiler is exited. The "ITFXREF f" option does not remain in effect for subsequently compiled modules.

Figure 5.15-1 is a sample cross-reference listing. At the left is an alphabetic list of <module>.<field>, where <module> is the name of the referenced module and <field> is the name of the referenced field. The <module> part is specified only the first time so that references to the same module stand out as a group.

At the right are the source file, name of the containing procedure, page and line occurrences of each reference to the corresponding field, and asterisk if the field is altered (e.g., assigned to) by the reference. The source file name is enclosed in parentheses and given before all references in that file. References are listed in the order encountered by the compiler. The name of the containing procedure is repeated only as often as necessary; i.e., it applies to all following items up to the next time a procedure name is given. "SOURCEFILE" directives and implicit sourcefiles due to "FORWARD" procedures can cause more than one source file to appear to the right.

5.16. "LIBRARY f"/"NOLIBRARY"

Default: "NOLIBRARY"; sticky

"LIBRARY f" is equivalent to the two options "GENCODE" and "OUTOBJLIB f".

"NOLIBRARY" is equivalent to "NOOUTOBJLIB".

```
Interface Cross-Reference for Module TOPCMP

COMPIL.COMPILEBITS          (miscPF) COMPILE 5.13 5.24* 5.25
                            5.53 5.54 5.60* SUBCOMMAND 41.43
                            41.47* 41.49* 41.50* 41.51*
                            41.54* 41.55* 41.56* 41.57*
                            41.60* 41.61* 41.62* 41.63*
      .COMPVERSIONBITS      (miscPF) COMPILE 5.15 5.16
      .SYSVERSIONBITS       (miscPF) COMPILE 5.18 5.19
      .TARGETSYSTEM         (miscPF) COMPILE 5.56 5.57
PASS1.FIRSTPASS             (miscPF) COMPILE 5.63
PASS2.SECONDPASS            (miscPF) COMPILE 5.75
TOPCMP.COMMANDSTRING        (miscPF) CMPREAD 3.8*
                            PROGCOMPILE 35.8*
      .DISPOSEDSYMLIST      (miscPF) COMPILE 5.47* 5.95
      .ENCODELIST           (miscPF) COMPILE 5.44* 5.99
                            5.100* 5.100
      .ERRFILE              (miscPF) COMPILE 5.30 5.30*
      .ERRS                 (miscPF) COMPILE 5.31* 5.68
                            5.76* PROGCOMPILE 35.12
```

Figure 5.15-1. "ITFXREF" Sample Output

These subcommands are provided for backward compatibility.

5.17. "LOG"/"NOLOG"

Default: "LOG"; sticky

The "LOG" option causes the compiler to display file names, pages, and procedure names (for "FORWARD" procedures) during compilation.

"NOLOG" turns off "LOG".

5.18. "MODTIME"

Sticky

"MODTIME" specifies that the objmod be configured so that it can monitor the total (deep and shallow) time spent in the module. The module must be run under the MAINSAIL performance monitor (MAINPM) to obtain the monitored value (see the "MAINPM User's Guide").

"NOMONITOR" turns off all monitor options.

5.19. "MONITOR"/"NOMONITOR"

Default: "NOMONITOR"; sticky

"MONITOR" is a quick way to specify all of the performance monitoring subcommands: "PERMOD", "PERPROC", "PERSTMT", "MODTIME", and "PROCTIME".

"NOMONITOR" turns off all monitor options.

5.20. "OPTIMIZE"/"NOOPTIMIZE", "OPTIMIZE proci"/"NOOPTIMZE proci", and "OPTIMIZEALL"/"NOOPTIMIZEALL"

Default: "NOOPTIMIZE"; sticky

"OPTIMIZE" tells the compiler to produce code that executes faster in many situations. The exact optimizations performed are machine-dependent, and in some cases, optimized code may be identical to unoptimized code.

"OPTIMIZE" may take an argument list of procedure names, i.e., "OPTIMIZE proc1 proc2 ... procn". In this form, the specified procedures are optimized whether or not the rest of the module is optimized. This form is non-sticky.

"NOOPTIMIZE" turns off the "OPTIMIZE" option. "NOOPTIMIZE" may also take a list of procedures that are not to be optimized, whether or not the rest of the module is optimized. If a procedure is specified in both "OPTIMIZE" and "NOOPTIMIZE" subcommands, the last reference takes precedence.

"OPTIMIZEALL" and "NOOPTIMIZEALL" override "OPTIMIZE" and "NOOPTIMIZE" directives in the source text of a module. The rules for optimization, and the subcommands and directives used to control it, are described in more detail in the "MAINSAIL Language Manual".

5.21. "OUTINTFILE f"

Not sticky

"OUTINTFILE f" specifies that the intmod is to be stored as the file f. The intmod is by default stored in a file the name of which is derived from the module name, unless an "OUTINTLIB" subcommand is specified, in which case the intmod is stored in the specified intmod library. "OUTINTFILE" overrides "NOOUTINTLIB" for the current compilation.

5.22. "OUTINTLIB f"/"NOOUTINTLIB"

Sticky

"OUTINTLIB f" specifies that the intmod is to be stored in the intmod library f. The intlib subsequently remains open.

"NOOUTINTLIB" turns off "OUTINTLIB".

5.23. "OUTOBJFILE f"

Not sticky

"OUTOBJFILE f" specifies that the objmod is to be stored as the file f. The objmod is by default stored in a file with a name derived from the module name, unless an "OUTOBJLIB" subcommand is specified, in which case the objmod is stored in the specified objmod library.

5.24. "OUTOBJLIB f"/"NOOUTOBJLIB"

Sticky

"OUTOBJLIB f" specifies that the objmod is to be stored in the objmod library f.

If for any reason the module cannot be added to the specified library, the compiler prompts for an output file name.

The compiler uses the current "TARGET" subcommand value to determine the format of the target module library.

"NOOUTOBJLIB" turns off "OUTOBJLIB".

5.25. "OUTPUT"/"OUTPUT f"/"NOOUTPUT"

Default: "OUTPUT"; sticky

"OUTPUT" is the same as "GENCODE", "OUTPUT f" is the same as "GENCODE" followed by "OUTOBJFILE f", and "NOOUTPUT" is the same as "NOGENCODE". These subcommands are provided for backward compatibility.

5.26. "PERMOD"

Sticky

"PERMOD" specifies that the objmod be configured so that it can monitor the total number of statements executed in the module. The module must be run under the MAINSAIL performance monitor (MAINPM) to obtain the monitored value (see the "MAINPM User's Guide").

"NOMONITOR" turns off all monitor options.

5.27. "PERPROC"

Sticky

"PERPROC" specifies that the objmod be configured so that it can monitor the total number of statements executed in each procedure. The module must be run under the MAINSAIL performance monitor (MAINPM) to obtain the monitored values (see the "MAINPM User's Guide").

"NOMONITOR" turns off all monitor options.

5.28. "PERSTMT"

Sticky

"PERSTMT" specifies that the objmod be configured so that it can monitor the total number of times each statement is executed. The module must be run under the MAINSAIL performance monitor (MAINPM) to obtain the monitored values (see the "MAINPM User's Guide").

"NOMONITOR" turns off all monitor options.

5.29. "PROCS"/"NOPROCS"

Default: "NOPROCS"; sticky

"PROCS" causes the names of procedures to be printed when their bodies are compiled. The procedure names are normally printed twice: once during the first pass, and once during code generation (unless the procedure has no body, as is typically the case for an inline procedure). During the first pass, the message "Compiling procedure <procName>" is written to logFile; during code generation, the message "Generating code for procedure <procName>" is written.

If the "PROCS" option is not given, there is normally no output to logFile from the compiler during the code generation phase. This option allows the user to monitor the progress of the code generator, which is useful when the compiler gets a fatal error (such as running out of memory) while compiling a module. The module can be compiled with the "PROCS" option to determine which procedure caused the problem.

"NOPROCS" turns off "PROCS".

5.30. "PROCTIME"

Sticky

"PROCTIME" specifies that the objmod be configured so that it can monitor the total (deep and shallow) time spent in each procedure. The module must be run under the MAINSAIL performance monitor (MAINPM) to obtain the monitored values (see the "MAINPM User's Guide").

"NOMONITOR" turns off all monitor options.

5.31. "RECOMPILE proc1 proc2 ... procn"

Not sticky

"RECOMPILE proc1 proc2 ... procn" indicates that an incremental recompilation is to be done, and that the procedures proc1, proc2, ..., procn are to be recompiled. "initialProc" and "finalProc" are the names of unnamed initial and final procedures. The effect is undefined if changes have been made to any of the module's source text outside any of these procedures.

The compiler finds the intmod for the module being recompiled, and finds within it the file and starting position of each of the procedures pi. It then compiles them in the order in which their bodies were originally encountered, automatically compensating for a change in the size of each procedure if it affects the location of another procedure later in the same file. The compiler complains if the expected procedure heading(s) are not found at the expected position(s).

If code is being generated, it is generated just for the recompiled procedures. The old objmod must be located. The newly generated code is appended to the end of the old objmod, and the replaced code becomes wasted space. Similarly, the new intmod information is appended to the end of the old intmod, and the replaced information becomes wasted space. Thus, the intmod and objmod are larger than if a full compilation were done, though the compilation takes less time.

As described more fully in Chapter 4, many options are restored from the old intmod, and the presently specified settings ignored. For example, it is not possible to compile a module with the check option in effect, and then recompile it with the nocheck option, since this would require that all code be regenerated, which is possible only if a full compilation takes place. Options that do not affect global attributes of the intmod or objmod can be alterd, e.g., "LOG"/"NOLOG".

5.32. "REDEFINE id def"/"NOREDEFINE"/"NOREDEFINE defi"

Default: "NOREDEFINE"

"REDEFINE x y" is equivalent to compiling the following global definition at the start of the compilation:

```
$GLOBALREDEFINE x = [y];
```

"y" is the text that remains (if any) after removing blank space after "x" and from the end of the command line. "$GLOBALREDEFINE" is described in the "MAINSAIL Language Manual". It defines identifers that can be referenced across all compilations in a given invocation of the compiler.

"NOREDEFINE" deletes all global definitions. "NOREDEFINE x1 x2 ... xn" deletes the global definitions for the identifiers x1, x2, ..., xn.

The nonsticky indicator ("*") has no effect on these subcommands.

5.33. "RESPONSE"/"NORESPONSE"

Default: "RESPONSE"; sticky

The "NORESPONSE" option means "Do not get a user response to error messages". The message is reported and compilation continues (unless the error is fatal). This is especially useful for running the compiler in "batch" mode, i.e., when no user is present to respond to any error messages. If a fatal error occurs while "NORESPONSE" is in effect, a stack dump is written to logFile and MAINSAIL exits.

"NORESPONSE" turns off "RESPONSE". "RESPONSE" is equivalent to "SUBCOMMAND RESPONSE", "NORESPONSE" to "SUBCOMMAND NORESPONSE".

These options apply to all errors, not just those generated by the compiler, and remain in effect after exiting the compiler. Use the "MAINEX" subcommands "RESPONSE" and "NORESPONSE" outside the compiler to restore the previous option, if desired.

5.34. "SAVEON"/"SAVEON f"/"NOSAVEON"

Default: "NOSAVEON"; sticky

"SAVEON" directs the compiler to put the information into the intmod that is needed to support the ability to open the module during a later compilation, e.g., using the "RESTOREFROM" directive as described in the "MAINSAIL Language Manual". Opening a module makes the symbols and procedure bodies declared in the module available.

"SAVEON f" is equivalent to "SAVEON" followed by "OUTINTFILE f".

"NOSAVEON" turns off "SAVEON".

5.35. "SLIST"/"SLIST f"/"NOSLIST"

Default: "NOSLIST"; sticky unless argument given

"SLIST f" causes the compiler to generate a source listing on the file f in which macros are expanded and source text not compiled due to conditional compilation logic is omitted. "f" may be omitted, in which case the output file is formed from the module name and the extension "lst".

"NOSLIST" turns off "SLIST".

5.36. "SUBCOMMAND s"

"SUBCOMMAND s" causes s to be executed as a MAINEX subcommand. For example, to open an intlib "foo" during a compilation, issue the subcommand "SUBCOMMAND OPENINTLIB foo".

5.37. "TARGET"/"NOTARGET"

Default: "NOTARGET"; sticky

"TARGET s" specifies that code is to be generated for the target system s, where s is a target system abbreviation (case is ignored). "TARGET ?" displays a summary of the system abbreviations. For example, to compile for Aegis (while running on some other operating system), use "TARGET aeg".

"NOTARGET" is the same as "TARGET <hostSystem>"; i.e., further compilations generate code for the host system.

Code generators for cross-compilation are separate products not shipped with a standard development system. Attempting to cross-compile to a system for which no code generator is available generates an error message saying that the required code generator cannot be found. Code generators for any system on which MAINSAIL is supported may be purchased from XIDAK.

5.38. "UNBOUND"/"NOUNBOUND"

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

Default: "NOUNBOUND"

"UNBOUND" specifies that subsequently compiled modules are nonbound-invocation modules, i.e., always allocated as unbound data sections by $invokeModule. Nonbound-invocation modules are further described as temporary features in the "MAINSAIL Language Manual".

5.39. "# s"

The "#" character in compiler subcommand mode introduces a comment. It is especially useful in command files.

6. DISASM, the MAINSAIL Disassembler

DISASM, the MAINSAIL disassembler, is used to examine the code generated by the MAINSAIL compiler.

When the disassembler is run, it first prompts for the name of an input file. At this prompt, you can specify an object file name or a module name if the object file name can be constructed from the module name in the standard way. If the name specified to the input file prompt is a valid module name, then DISASM assumes that a module of that name is to be disassembled, and builds the object file name the same way that the runtime system does when it looks for the object file containing a module. Otherwise, DISASM uses the name as the name of the input object file. If the module to be disassembled is in an objmod library, type <eol> to the input file prompt, and DISASM then prompts for the name of the library and the module name.

DISASM then prompts for the output (disassembly) file name.

Whenever a new input object file or object library name is specified, DISASM prompts for the target system abbreviation. The default is the target used in disassembling the previous module in this DISASM session, or the host system if this is the first module disassembled.

If the intmod file is not the default intmod file for the module being disassembled, the "IF <intmod file name>" command may be used to specify it. "IF" with no arguments sets the intmod file to the default file for the current module.

To disassemble another module, use the "F" command. "F" with no arguments causes DISASM to enter its initial dialogue again. "F <module or file name>" allows the input module or file name to be specified.

The "IL <library name>" command specifies the name of an input intmod library. This command remains in effect until a new "IL" command is specified.

"OF <output file name>" specifies a new output file.

The "D" command disassembles the entire input module, placing the result in the output file; "D <procedure name>" disassembles a single procedure. "D <proc1> <proc2> ... <procn>" disassembles the named procedures.

The "Q" command exits the disassembler.

The "?" command displays a list of valid disassembler commands. Commands not documented here are for XIDAK's internal use.

A sample disassembler session is shown in Table 6-1.

```
MAINSAIL (R) Disassembler
Copyright (c) 1984... by XIDAK, Inc., Menlo Park,
 California, USA.

Input file or module name (eol if in library): erode<eol>
Target system (eol for um20): <eol>
Opening intmod for $SYS...
Output file (eol for logFile): <eol>

Type ? for help

DISASM: d initialproc<eol>
                DISASSEMBLY OF OBJECT FILE erode-um2.obj
                ----------------------------------------

                        SOURCE FILE: erode.msl
                        ----------------------

INITIAL PROCEDURE;
BEGIN
LONG INTEGER ii,jj,i,j;
STRING s;
POINTER(dataFile) f;

504     LINK    FP,-'H18
508     MOVEL   DB,-4(FP)
50C     MOVEL   #'H3808,-8(FP)
514     MOVEQ   #6,D7
516     CLRL    -(SP)
518     DBRA    D7,'H516

ran := new($ranMod); new(elev);

51C     MOVEL   #'H9FAA8580,-(SP)
522     CLRW    -(SP)

... rest of disassembly of initial procedure...

Disassembly on file TTY
DISASM: q<eol>
```

Table 6-1. DISASM Example

7. XRFMRG, the Interface Cross-Reference Merger

The module XRFMRG merges several cross-reference files, as produced by the "ITFXREF" and "FLDXREF" subcommands of the MAINSAIL compiler. Such a consolidated listing provides a cross-reference of field accesses for more than one module.

XRFMRG prompts for the file name of each cross-reference file and then prompts for an output file for the consolidated listing. The consolidated listing has a format similar to a cross-reference file produced by the compiler, and may itself be used as input to XRFMRG. XRFMRG accepts a file containing a list of input file names (one per line) in response to the "Next input file" prompt if it is preceded by the character "@".

Example 7-1 shows a sample session with XRFMRG. Assume that the modules LINCOM and SAMPLE have been compiled with the "ITFXREF" subcommand.

```
MAINSAIL (R) Version 12.10 (? for help)
Copyright (c) 1984, 1985, 1986, 1987, 1988, and 1989 by
 XIDAK, Inc., Menlo Park, California, USA.
*xrfmrg<eol>
Next input file (@indirect, eol to stop): lincom.xrf<eol>
Next input file (@indirect, eol to stop): sample.xrf<eol>
Next input file (@indirect, eol to stop): <eol>
Output file: linsmp.xrf<eol>
```

Example 7-1. XRFMRG Example

Figure 7-2 is a sample output file from XRFMRG. Sections 5.15 and 5.7 describe the output format.

Example 7-3 shows how to specify an indirect command file as input to XRFMRG. File "indirectCmdFile" must contain the names of each input file to XRFMRG, one per line.

For a description of a simple keyword cross-reference utility, see the description of "XREF" in the "MAINSAIL Utilities User's Guide".

```
Combined Cross Reference for Module(s) LINCOM SAMPLE

$ARYMOD.NEWARRAY1
     (m:lincom) LINCOM.INITIALPROC 8.37
$DPYEXE.$DPYSTTYWRITE
     (m:tops20) LINCOM.STTYWRITE 4.9
$DPYEXE.$DPYTTYREAD
     (m:syslib) SAMPLE.TTYREAD 10.18
$DSPMOD.ARYDISPOSE
     (m:lincom) LINCOM.INITIALPROC 8.43
$DSPMOD.NEWUPPERBOUND
     (m:lincom) LINCOM.SYNCHRONIZE 7.43 7.44
$ERRMOD.$FILEERROR
     (m:syslib) LINCOM.SFWRITE 13.9 LINCOM.CFREAD 16.10
     LINCOM.CFWRITE 17.8 LINCOM.GETPOS 20.8
     LINCOM.SETPOS 21.7
$ERRMOD.ERRMSG
     (m:syslib) LINCOM.SETPOS 21.10
```

Figure 7-2. XRFMRG Sample Output

```
*xrfmrg<eol>
Next input file ... : @indirectCmdFile<eol>
Next input file ... : <eol>
Output file: merged.xrf<eol>
```

Example 7-3. XRFMRG Indirect Command File Example

8. The Foreign Language Interface

8.1. Introduction

This chapter describes how procedures written in other languages may be called from MAINSAIL and vice versa. The MAINSAIL component that provides this capability is called the Foreign Language Interface (FLI).

An FLI code generator generates assembly language code that acts as an interface between MAINSAIL and a foreign language. There are two kinds of interfaces and therefore two kinds of FLI code generators. An FLI code generator that generates an interface that allows a MAINSAIL module to call a foreign procedure is called a "Foreign Call Compiler" (FCC); an FLI code generator that generates an interface that allows a foreign procedure to call a MAINSAIL module is called a "MAINSAIL Entry Compiler" (MEC).

On different operating systems, different sets of FLI compilers may be available. On some operating systems, no FLI may be supported; on others, both FCC and MEC may be supported for several different languages. Refer to the appropriate operating-system-specific MAINSAIL user's guide for information on the FLI code generators available for your system.

The FLI-generated assembly language interface between MAINSAIL and a foreign procedure must be assembled with the host assembler and then linked with the foreign code and with a MAINSAIL bootstrap. For each FCC-generated interface with which it is linked, the bootstrap must contain the name of the corresponding FLI module in an internal table. These names are placed in the bootstrap by means of the utility module CONF's "FOREIGNMODULES" command. The names are used to construct assembly language labels that permit linkage to be established to the foreign code.

No "FOREIGNMODULES" entry is made for an MEC-generated interface.

If execution begins in foreign code, the bootstrap must have the $foreignCodeStartsExecution configuration bit set. Use the CONF utility "CONFIGURATIONBITS" command to set this bit. It makes no sense to set this bit unless at least one MEC-generated interface is linked with the bootstrap.

The $foreignCodeStartsExecution bit may not be supported by all MEC's; consult the appropriate system-specific documentation for information.

The "MAINSAIL Utilities User's Guide" describes the use of CONF to make a MAINSAIL bootstrap.

Figure 8.1-1 shows a MAINSAIL bootstrap that has been linked to allow MAINSAIL to interface to and from foreign code. The arrows on the left show how MAINSAIL calls a foreign procedure and those on the right show how a foreign procedure calls MAINSAIL. In a real bootstrap, there may be any number of FCC-generated and MEC-generated interfaces.

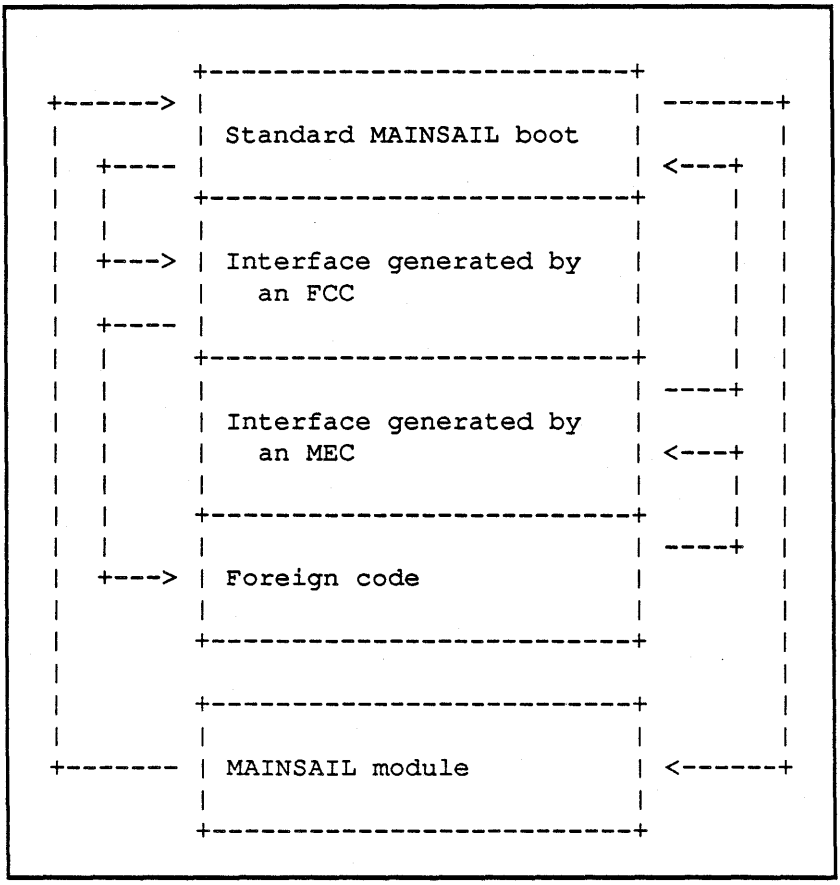


Figure 8.1-1. Interfacing MAINSAIL to Other Languages

8.2. Foreign Call Compiler

The bodies of foreign procedures are written in languages other than MAINSAIL, such as FORTRAN, Pascal, C, or assembler. Each foreign procedure to be called from MAINSAIL must be declared to MAINSAIL with a MAINSAIL procedure header; i.e., the procedure result and parameter data types and passing mechanisms must be mapped into those provided by MAINSAIL. Each such MAINSAIL procedure header is then declared as part of the interface of a "dummy" MAINSAIL module called a foreign module (bodies for the fake interface procedures must also be provided (to make the module syntactically correct) but the bodies are ignored). Each foreign module is compiled with the FCC to generate an assembly interface to each procedure in the foreign module.

When the MAINSAIL intermodule call mechanism detects that the target module is a foreign module, it transfers control to the interface code generated for the foreign module. This code sets up the parameters and environment for the foreign procedure and then invokes it. When the foreign procedure returns, the interface code sets up the parameters for MAINSAIL and the intermodule return mechanism handles the transition back to the MAINSAIL module.

Parameter data type mappings vary from language to language and operating system to operating system; consult the system-specific MAINSAIL documentation for details. Produces parameters passed from MAINSAIL to a foreign language procedure are not necessarily cleared, and should not be assumed to be Zero upon entry to the foreign procedure.

Foreign procedures can be grouped arbitrarily into one or more MAINSAIL foreign modules. Since, at the point of call, MAINSAIL does not know that it is calling a foreign module, substitution of a MAINSAIL module for a foreign module is transparent. For example, if a FORTRAN package is rewritten into a MAINSAIL module, the MAINSAIL module can be substituted for the foreign module without the need to recompile other MAINSAIL modules that call it.

A garbage collection can occur when a foreign module is bound (by an explicit call to bind, or implicitly on the first call to the module). The binding of the FLI module causes the MAINSAIL runtime system to allocate some supporting data structures. Subsequent calls to an FLI module cannot cause a garbage collection (unless the foreign language code calls back into MAINSAIL).

8.3. Foreign Call Compiler Example

Suppose that the programmer wishes to call some procedures in a hypothetical FORTRAN graphics package, the headers of which are shown in Figure 8.3-1. One (or more) foreign modules that describe the foreign procedures must be provided. First, determine the the

MAINSAIL procedure declaration for each FORTRAN subroutine. Then declare each MAINSAIL procedure to be an interface procedure of a foreign module. Figure 8.3-2 shows the module declaration for a MAINSAIL foreign module GRPHCS that contains three foreign procedures corresponding to the FORTRAN procedures.

```
SUBROUTINE LINE (IX1,IY1,IX2,IY2)
INTEGER*2 IX1,IY1,IX2,IY2

REAL FUNCTION CIRCLE (IX1,IY1,IX2,IY2,IX3,IY3)
INTEGER*2 IX1,IY1,IX2,IY2,IX3,IY3

SUBROUTINE POINTS (IX,IY,N)
DIMENSION IX,IY(N)
INTEGER*2 IX,IY,N
```

Figure 8.3-1. Sample FORTRAN Graphics Procedure Headers

```
MODULE grphcs (
    PROCEDURE line (
        INTEGER ix1,iy1,ix2,iy2);
    REAL PROCEDURE circle (
        INTEGER ix1,iy1,ix2,iy2,ix3,iy3);
    PROCEDURE points (
        INTEGER ARRAY(1 TO *) ix2,iy2; INTEGER n)
    );
```

Figure 8.3-2. MAINSAIL Interface for FORTRAN Graphics Procedures

Next write the module GRPHCS. GRPHCS is compiled with the FCC and must be a syntactically correct MAINSAIL module complete with a module declaration and all interface procedures. The procedure bodies may be empty, since the FCC ignores procedure bodies. In addition, all MAINSAIL modules that call any of the foreign procedures in the module GRPHCS must contain the module declaration for GRPHCS.

It is suggested that the module declaration for GRPHCS reside in a file that is sourcefiled by or an intmod that is restored from by the module GRPHCS and by each MAINSAIL module that

calls one or more of its interface procedures. Figure 8.3-3 shows how to write the module GRPHCS. It assumes that the module declaration for GRPHCS (shown in Figure 8.3-2) resides in the file "gHdr". The "SOURCEFILE" directive pulls in the module declaration for GRPHCS and is followed by interface procedures with empty bodies (the second ";" after each procedure header terminates the null procedure body).

```
BEGIN "grphcs"

SOURCEFILE "gHdr"; # include grphcs module declaration

PROCEDURE line (
    INTEGER ix1,iy1,ix2,iy2);;
REAL PROCEDURE circle (
    INTEGER ix1,iy1,ix2,iy2,ix3,iy3);;
PROCEDURE points (
    INTEGER ARRAY(1 TO *) ix2,iy2; INTEGER n);;

END "grphcs"
```

Figure 8.3-3. MAINSAIL Foreign Module for FORTRAN Graphics Procedures

Compile GRPHCS with the FCC to generate the interface between MAINSAIL and each FORTRAN procedure, as shown in Example 8.3-4. This interface is assembly code that translates the parameters from the MAINSAIL calling convention to the FORTRAN calling convention, invokes the FORTRAN procedure, and upon return from the FORTRAN procedure, converts the parameters from the FORTRAN calling convention back into the MAINSAIL calling convention and then returns to the MAINSAIL module.

Use the CONF command "FOREIGNMODULES" to build a bootstrap that indicates to MAINSAIL that module GRPHCS is a foreign module. Assemble this new bootstrap and the output of the FCC and compile the FORTRAN procedures. Then link all of the above object files together to create a new executable MAINSAIL bootstrap. Consult the appropriate operating-system-specific MAINSAIL user's guide to find out how to assemble and link foreign code with MAINSAIL on your system.

```
MAINSAIL (R) Compiler
Copyright (c) 1984, 1985, 1986, 1987, 1988, and 1989 by
 XIDAK, Inc., Menlo Park, California, USA.

compile (? for help): grphcs.msl,<eol>
> fli tf<eol>
> <eol>
Opening intmod for $SYS...

grphcs.msl 1
Opening intmod for $SYS...
Output for GRPHCS stored on grphcs.<assembly suffix>
Intmod for GRPHCS not stored

compile (? for help):
```

Example 8.3-4. Invocation of an FCC

8.4. MAINSAIL Entry Compiler

Normal MAINSAIL modules can be compiled with an MEC to generate an assembly interface that allows any foreign procedure to call any of the interface procedures in the MAINSAIL module. The foreign code is linked with the output of the MEC and with a MAINSAIL bootstrap to produce an executable file.

The foreign procedure can invoke the MAINSAIL procedure using its normal external call mechanism; i.e., the foreign procedure does not know that it is calling a MAINSAIL module. Control is passed to the assembly interface code for the MAINSAIL procedure. This code sets up the MAINSAIL environment and parameters and invokes MAINSAIL to execute the procedure. When the MAINSAIL procedure completes execution, it returns control to the interface code that sets up the parameters for and returns control to the foreign procedure.

Since the return address to the foreign code is always available when foreign code calls into MAINSAIL, it is not necessary to include a table of foreign module names (addresses) in the bootstrap, as is the case when using the FCC. There is, however, a configuration bit that must be set if execution begins in foreign code rather than in MAINSAIL. When execution begins in foreign code, MAINSAIL must initialize itself the first time the foreign code calls a MAINSAIL procedure. The "foreign code starts execution" bit tells MAINSAIL not to print its

identification banner or prompt for the next module to execute when it initializes itself. Use the CONF "CONFIGURATIONBITS" command to set this bit and make a new MAINSAIL bootstrap if execution begins in foreign code.

8.5. MAINSAIL Entry Compiler Example

Figure 8.5-1 shows a FORTRAN procedure that calls an external procedure "mean". Suppose that the procedure mean is written in MAINSAIL, as shown in Figure 8.5-2. The MEC is used to generate the interface that allows mean to be called from the FORTRAN procedure.

```
REAL*8 A,B,C,MEAN
. . .
A = MEAN(B,C)
. . .
STOP
```

Figure 8.5-1. FORTRAN Procedure Calling MAINSAIL Procedure

```
BEGIN "foo"

MODULE foo (LONG REAL PROCEDURE mean (LONG REAL a,b));

LONG REAL PROCEDURE mean (LONG REAL a,b);
RETURN((a + b) / 2.0L);

END "foo"
```

Figure 8.5-2. MAINSAIL Module FOO

Compile the module FOO with the MEC to produce the assembly interface code for procedure mean and with the normal MAINSAIL compiler to produce the executable MAINSAIL object module for FOO, as shown in Example 8.5-3.

If execution begins in the FORTRAN procedure, use the CONF "CONFIGURATIONBITS" command to set the "foreign code starts execution" configuration bit and make a new

```
MAINSAIL (R) Compiler
Copyright (c) 1984, 1985, 1986, 1987, 1988, and 1989 by
 XIDAK, Inc., Menlo Park, California, USA.

compile (? for help): foo.msl<eol>
Opening intmod for $SYS...

foo.msl 1
Objmod for FOO on foo-xxx.obj
Intmod for FOO not stored

compile (? for help): foo.msl,<eol>
> fli ff<eol>
> <eol>
Opening intmod for $SYS...

grphcs.msl 1

Output for GRPHCS stored on grphcs.<assembly suffix>
Intmod for GRPHCS not stored

compile (? for help):
```

Example 8.5-3. Use of an MEC

MAINSAIL bootstrap. Link the FORTRAN code with the MAINSAIL bootstrap and with the interface code generated by the MEC. Refer to the appropriate operating-system-specific MAINSAIL user's guide for information on how to assemble and link foreign code with MAINSAIL on your system.

8.6. Foreign Labels and the "ENCODE" Directive

By default, the FCC assumes that the labels for foreign procedures are derived by some transformation of the corresponding MAINSAIL procedure name, and the MEC generates labels based on the same transformation of the corresponding MAINSAIL procedure name. The default transformation is operating-system-dependent and is described in each operating-system-specific MAINSAIL user's guide.

The default transformation may be overridden by means of the "ENCODE" directive, which may appear anywhere within the foreign module. The form of the "ENCODE" directive is

```
ENCODE p1 s1, p2 s2, ..., pn sn;
```

where the pi are interface procedure identifiers and the si are string constants. The strings si may be arbitrary string constants and are used as the labels for their respective pi when pi are compiled with the FLI compiler. The user is responsible for ensuring that the si conform to the operating-system-dependent assembler's and linker's rules for valid labels.

If the module BAR of Example 8.6-1 is compiled with the FCC compiler, the FCC compiler assumes that the foreign procedure names corresponding to abc and def are "_AbC" and "D$xxx", respectively. If BAR is compiled with the MEC, the labels generated for the interface procedures corresponding to abc and def are "_AbC" and "D$xxx", respectively. If "_AbC" and "D$xxx" are not valid labels on the system for which BAR is compiled, assembly or linkage errors may result.

```
BEGIN "bar"

MODULE bar (
    PROCEDURE abc (REAL r);
    INTEGER PROCEDURE def;
);

ENCODE abc "_AbC", def "D$xxx";

PROCEDURE abc (REAL r);
<body for abc>;

INTEGER PROCEDURE def;
<body for def>;

END "bar"
```

Example 8.6-1. Use of "ENCODE" in a Foreign Module

8.7. Matching Parameters

MAINSAIL parameter data types and passing mechanisms (uses, produces and modifies) must be mapped onto the foreign language data types, and vice versa. For example, "value" parameters are passed as MAINSAIL uses parameters, and "reference" parameters are passed as MAINSAIL produces or modifies parameters. The exact mapping of data types is operating-system- and language-dependent, and is documented in the operating-system-specific MAINSAIL user's guide for your system.

The mapping of a MAINSAIL array or record into a foreign language requires a knowledge of the layout in memory of both the MAINSAIL data structure and the corresponding foreign data structure. XIDAK does not document memory usage conventions for foreign languages; consult the documentation provided by the manufacturer of the target language. The memory layout of arrays and records is described in the "MAINSAIL Language Manual".

8.8. Foreign Code and Garbage Collection

On operating systems that support both the FCC and the MEC, it is possible for MAINSAIL to call foreign code that then calls back into MAINSAIL, and for foreign code to call a MAINSAIL procedure that then calls out to foreign code. MAINSAIL must lock out collections if any collectable parameters are passed between MAINSAIL and the foreign code; if MAINSAIL did a garbage collection while there were pending returns to foreign code, collectables located in the data areas for the foreign code would not be updated and might no longer be valid. Collections are not locked out if no collectables are passed between MAINSAIL and the foreign code.

8.9. Foreign Code and Exceptions

If MAINSAIL traps operating-system-specific exceptions and such an exception occurs in foreign code, then the normal MAINSAIL error handling mechanism is given control. The appropriate MAINSAIL exception is raised and if not handled, an error message that indicates that the exception occurred in foreign code and the name and offset of the most recently executing MAINSAIL module are printed.

9. Arguments on the Command Line

File(s) to be compiled can be specified on the command line. They are compiled in the order specified. If a file name ends in a comma, the compiler enters subcommand mode when it compiles that file (sticky subcommands are sticky, as usual). For example, the command line:

```
compil msl:foo, msl:bar msl:baz
```

causes the compiler to enter subcommand mode before compiling "msl:foo". It does not enter subcommand mode before compiling the remaining files. The compiler returns to its caller after all three files have been compiled.

XRFMRG also accepts one or more file names on the command line. A file name may begin with an "@" to indicate an indirect file name. XRFMRG always prompts for the output file.

Command line syntaxes are subject to change.

10. Invoking the Compiler from a Program

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

To invoke the compiler from a program, allocate a new data section for the module "COMPIL" (of interface class $compil, with $programInterface set) and call the procedure $compile in the allocated data section (see Table 10-1); e.g.:

```
POINTER($compil) p;
...
p := new("COMPIL",$programInterface);
...
IF p.$compile(...) THEN ...
```

The data section may be disposed when all compilations have been completed:

```
dispose(p);
```

```
BOOLEAN
PROCEDURE   $compile     (OPTIONAL STRING cmds;
                          OPTIONAL POINTER(textFile) f);
```

Table 10-1. $compile

cmds is a string that contains exactly what would be typed to the compiler had it been invoked by the user from the terminal (cmds may contain embedded eol's). If cmds is the null string, or when cmds is exhausted, subsequent lines of input are read from f if f is not nullPointer. If cmds is empty and f is nullPointer the compiler enters into its user dialogue (on cmdFile and logFile) as usual. The result is true if no errors were detected (in the last compilation), else false. The compiler prompts are suppressed when the compiler is invoked from "$compile". However, logging information (as shown by the "LOG" subcommand) and error messages are still written to logFile unless redirected or suppressed by the appropriate compiler subcommands in cmds. Interactive define responses are read from cmdFile, not from cmds or f (except when the subcommand "CMDLINE" is in effect).

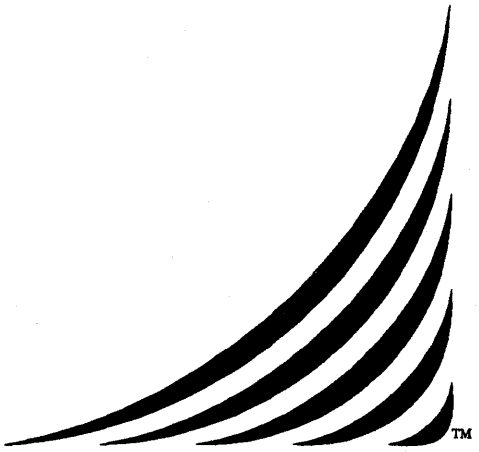
When cmds and f are exhausted, $compile acts as if it had read an eol from the input. If the commands in cmds and f terminate the compilation session before cmds and f are exhausted, the unprocessed input is ignored.

Subcommands should be spelled in full to promote upwards compatibility (to avoid ambiguities with newly introduced subcommands in future releases; no guarantee is made, however, that individual subcommands will be supported from release to release).

For example, to compile the files "foo.msl" and "baz.msl" nocheck and add them to the library "s.lib", use the code shown in Example 10-2.

```
p.$compile(
    "foo.msl," & eol &    # "," means enter subcommand mode
        "library s.lib" & eol &
        "nocheck" & eol &
        eol &             # exit subcommand mode
    "baz.msl" & eol);
```

Example 10-2. Compiling "foo.msl" and "baz.msl" from a Program



MAINDEBUG™

User's Guide

24 March 1989

11. Introduction

The MAINSAIL debugger, MAINDEBUG, provides an interactive tool for observing and controlling the execution of MAINSAIL modules. The debugger coordinates the execution of a module with the concurrent display of the module's source text. When a runtime error such as a nullPointer access or an invalid array subscript occurs, the debugger can be used to display the source text and put the cursor at the statement that caused the error. Relevant text is also interactively displayed when controlling module execution by means of breakpoints and single stepping.

The debugger is a high-level "MAINSAIL debugger"; it works on MAINSAIL language constructs rather than on the underlying hardware's instruction set. Commands controlling module execution operate on MAINSAIL statements and not on native operation codes. Commands for examination and modification of data elements operate on MAINSAIL data types and not on processor-dependent "bytes" or "words".

The debugger uses the MAINSAIL compiler to parse the expressions, statements, definitions, and declarations contained in debugger commands. The MAINSAIL compiler must therefore be present on any system on which MAINDEBUG is to be used.

XIDAK reserves the right to create debugger commands for internal use only. Such commands are not documented here and are subject to change and/or removal without notice. Debugger commands may change from release to release of MAINSAIL.

11.1. Version

This version of the "MAINDEBUG User's Guide" is current as of Version 12.10 of MAINSAIL. It incorporates the "Debugger Version 5.10 Release Note" of October, 1982; the "Debugger Version 7.4 Release Note" of May, 1983; the "MAINDEBUG Release Note, Version 8" of January, 1984; the "MAINDEBUG Release Note, Version 9" of February, 1985; the "MAINDEBUG Release Note, Version 10" of March, 1986; and the "MAINDEBUG Release Note, Version 11" of July, 1987.

12. General Operation

A program may be composed of both debuggable and non-debuggable modules. A debuggable module is slightly larger and slower, since the generated code is affected by the debug option.

12.1. Compilation

All modules that are to take part in debugging must be compiled with the "DEBUG" compiler option. Refer to the "MAINSAIL Compiler User's Guide" for details.

The "DEBUG" compiler option causes an intmod (intermediate module) to be generated, in addition to the objmod (object module). The intmod includes debugging information such as symbol tables and tables that relate code offsets with source file positions for each MAINSAIL statement. An intmod can be stored in a separate file, or in an intlib (intmod library). Intmods are described in the "MAINSAIL Compiler User's Guide". Intlibs are manipulated with the utility module INTLIB as described in the "MAINSAIL Utilities User's Guide".

12.2. Invocation

The debugger can be invoked either from the MAINSAIL executive or in response to the standard MAINSAIL error routine prompt. To invoke it from the MAINSAIL executive, type "DEBUG" in response to the MAINSAIL "*" prompt (i.e., run the module DEBUG). To invoke the debugger after an error has occurred, type "DEBUG" (or enough to make this response unique) to the "Error Response:" prompt.

Invoking the debugger as the module DEBUG before executing the program to be debugged allows you to prepare for the debug session, e.g., to set breakpoints, and then invoke the program from the debugger (it is said to "run under the debugger"). Invoking the debugger in response to an error message gives you control at the point of the error. Continuing execution from a fatal error is not allowed.

12.3. Finding Intmods

The debugger must be able to find the intmod for each module that is to take part in a debug session. When the debugger needs to find an intmod for some module M, it first searches all open intlibs, then tries to open the file named "xxx-int:m", where xxx is the first three letters of

the operating-system name abbreviation, e.g., "um6-int:m" on an M68000 UNIX system (since "um68" is the abbreviation for M68000 UNIX).

A searchpath is customarily specified during MAINSAIL initialization for file names of the form "xxx-int:*", where xxx is the first three letters of the current operating system abbreviation. The searchpath distributed by XIDAK for M68000 UNIX is:

```
SEARCHPATH um6-int:* *-um6.int
```

Thus, on M68000 UNIX the debugger tries to obtain the intmod for M from the file "m-um6.int" if it cannot find it in an open intlib.

The debugger provides the "OI" (Open Intlib) command for opening intlibs ("OI" is described in Section 14.23). Also, the MAINEX subcommand "OPENINTLIB" can be used to open an intlib (MAINEX subcommands can be given at any time by running module SUBCMD). Each intlib that contains an intmod needed during the debug session should be open before the debugger tries to find the intmod. The MAINEX subcommands "INTFILE", "INTLIB", and "INTDEFAULT" also provide control over intmod searches, as described in the "MAINSAIL Utilities User's Guide".

If you put a needed intmod in an intlib and forget to open the intlib before the debugger needs to access it, the debugger fails to find the intmod in the default file, so it issues an error message and prompts for a new file name or library in which to look.

12.4. Context

At any point during a debug session the current context may include a file and/or a module. The debugger provides facilities for independently viewing files regardless of the file's relationship to the module; i.e., there need be no relationship between the current file and the current module. Usually, however, the file contains part or all of the source text for a module being debugged.

Whenever the debugger obtains control, the module context is set to the most recently entered, but not exited, debuggable module, if any. The file context is set to the source file and position that corresponds to the module context. If the corresponding source file cannot be located, the debugger is able to continue, but the debugging of the current module is hampered.

12.5. Positions and iUnits

The module context includes a position within the module. This position is usually specified by positioning the cursor to the appropriate place in the source text, but it can also be specified

numerically as an offset in the object module. Such a position is called an "iUnit" (for "instruction unit"). iUnits are displayed when a stack dump is given, as by the "CALLS" error response or the "CALLS" utility module, and by certain debugger commands. Remembering the iUnit of a location of interest in a module can save you the effort of positioning the cursor to that spot in the source text if you wish to issue a command referring to that location. However, iUnits become invalid whenever a module is recompiled, since the offsets of the instructions corresponding to source statements may change.

12.6. Breakpoints and Single Stepping

MAINDEBUG provides four commands for directly controlling and displaying the execution of MAINSAIL statements: the "B", "T", "S", and "J" commands. Each command causes program execution to "break" under specified conditions. When execution breaks, MAINDEBUG displays a prompt (in the line-oriented interface) or an editor message (in the display-oriented interface) and waits for further commands.

The "B" command sets a permanent breakpoint (one at which execution halts each time the breakpoint is reached); the "T" command sets a temporary breakpoint (one that is removed the next time a break occurs). The "S" command executes the next MAINSAIL statement, without following execution into procedures; the "J" command executes the next MAINSAIL statement, breaking on the first statement of any debuggable procedure invoked. The "S" and "J" commands therefore act identically if the executed statements do not invoke a debuggable procedure. See Chapter 14 for the descriptions of these commands.

The "C" and ".C" commands are used to continue execution from a breakpoint; see Section 14.5.

12.7. Command Interface

The debugger supports two different user interfaces: line-oriented and display-oriented. The display-oriented interface makes available to the user all the features of MAINEDIT during a debugging session. The line-oriented interface runs even on terminal types for which no MAINEDIT display module exists.

12.7.1. Line-Oriented Command Interface

The line-oriented command interface provides debugging commands and a simple set of read-only line editor commands, communicating through the file "TTY". It displays short pieces of the current file in response to each debugger command. The debugger's cursor is indicated by a right curly bracket, "}", immediately preceding the current character. If any command in a

command line changes the cursor position, the new line and cursor position are displayed prior to the prompt for the next command line.

A command line given by the user in response to the debugger's prompt is a sequence of debugging commands. All commands on a line are executed from left to right unless an error occurs, in which case commands after the one in error are not executed.

The debugger's prompt indicates the current context and has the default format (use the "OP" command to change the prompt format):

```
moduleName.procedureName:
```

The procedure name is omitted if there is no procedure context; if there is no module context, the prompt is:

```
MAINDEBUG:
```

When a breakpoint is taken, or when the debugger is entered in response to an error message, the debugger displays the appropriate line of text followed by the debugger prompt, described above, and awaits a debugger command. The command line must be terminated with <eol> before any command on the line is executed.

12.7.2. Display-Oriented Command Interface

The display-oriented command interface uses MAINEDIT, the MAINSAIL Text Editor. This interface provides debugging commands and full access to the command structures offered by the various editor front ends (which include MAINVI, an emulation of the UNIX editor vi, MEDT, an emulation of DEC's EDT, and MAINED, XIDAK's own front end). The output of the debugger and the debugged program is captured in a buffer, so that the user can refer to earlier output at any time. The "MAINEDIT User's Guide" describes MAINEDIT in detail.

Files are displayed on the screen just as they are by MAINEDIT. The editor cursor is used to indicate the current position in a file, rather than the curly bracket cursor of the line-oriented interface. When a command requiring the debugger to position to a particular location is issued, the file containing the specified module is displayed in a window. If the file was not previously in a buffer, a new window is created and the cursor positioned at the beginning of the file. If there was a buffer for the file, then that buffer is used, and the cursor is positioned where it was when last in that buffer.

The display-oriented interface provides immediate feedback when a command is typed. For example, when the debugger command "M" is issued, the debugger immediately prompts for the module name on the editor message line. MAINEDIT's <abort> key can be used to abort a

debugger command before the user has finished typing it in, or to abort a command that is prompting for further input.

In the display interface, when the debugger is given a character or sequence of characters it cannot interpret as a debugger command, it passes the character(s) to the editor. The editor executes them and returns to the debugger. In particular, editor macros defined for control keys and keypad keys can be used when interacting with the debugger.

When the debugger reads a file into a MAINEDIT buffer or breaks at a statement in a MAINEDIT buffer, the buffer is made read-only, so that it cannot be accidentally altered, and the mode is set to escape mode, as indicated by the "E" in the status line. In this mode commands are processed by the debugger rather than the editor. To talk to the editor, type the <ecm> (Enter Command Mode) key. In command mode, all editor commands are available, allowing you to examine files, position the cursor, etc. Use MAINED's "E" command (or the corresponding commands in the other editor front ends) to enter escape mode and talk to the debugger again. A common mistake is to forget whether you are talking to the debugger or editor, and use a command intended for one mode while in the other. Be careful that you are talking to the editor (as shown by the buffer mode letter) when giving an editor command, and talking to the debugger ("E" mode) when giving a debugger command.

The file for the current buffer is the debugger's "current file". When the user changes the current buffer by means of a MAINEDIT command, the debugger's current file is automatically set to be the file for the new buffer. Thus, the use of editor commands to change buffers is nearly equivalent to using the debugger's "F" command to change files.

A file is brought into a buffer and displayed when a breakpoint is reached or when the "DEBUG" command is entered in response to an error message. At this point, the debugger is awaiting a command.

To edit a buffer used by the debugger, remember that read-only mode must first be turned off for that buffer (use MAINED's "-.Oreadonly<eol>" command to turn off read-only mode for the current buffer). After a buffer has been modified, the debugger positions in the buffer may no longer correspond to the appropriate text, so it is recommended not to continue a debugging session after changing a buffer containing a part of a module being debugged.

12.8. Command Syntax

MAINDEBUG commands are divided into three groups. General and debugging commands are used to control debugger operations and are available from both the line-oriented and display-oriented interfaces. Editing commands are provided with the line-oriented interface to allow files to be examined. The editing commands are not available with the display interface, since the MAINEDIT commands are used in this case.

Each command consists of one or more base characters, possibly preceded by an integer count (represented in the descriptions as "n"), dot ("."), plus ("+"), or minus ("-"), and possibly followed by arguments. If the count is omitted, 1 is assumed. If both count and minus or plus are permitted for a command, it does not matter which comes first. Dot must usually occur immediately before the command character, although sometimes ".n" introduces a subcount n. The debugger ignores upper and lower case distinctions in command letters. Spaces and tabs are usually ignored except where they play a role as delimiters as described below.

Arguments can usually occur immediately after the command base, e.g., both "V x" and "Vx" display the value of "x".

Optional parts of commands are indicated in curly brackets; e.g., "{n}D" means the "D" command can be preceded by a count.

12.8.1. Command Arguments

The different kinds of debugger arguments are shown in Table 12.8.1-1.

```
count       integer
id          MAINSAIL identifier
name        text to first space, tab, or semicolon
expr        MAINSAIL expression
dcl         MAINSAIL declarations and definitions
stmt        MAINSAIL statements
```

Table 12.8.1-1. Kinds of Arguments

An id argument is used for identifiers such as module and field names. A name argument is used for file names and coroutine names, which need not be identifiers. For commands that can take either a module name or a file name argument, the debugger scans a name argument rather than an id argument since it does not know whether the argument is a file name or a module name.

The end of count, id, and name arguments can be determined simply by scanning the corresponding object from the argument text. The ends of expr, dcl, and stmt arguments are determined by a more complicated scan carried out by the MAINSAIL compiler as it compiles the argument text. When it encounters a "terminating token", i.e., a piece of text that cannot be the continuation of an expression (for an expr argument), the start of a declaration or definition

(for a dcl argument), or the start of a statement (for a stmt argument), it discards the token and returns to the debugger.

expr and stmt arguments are restricted as described in Appendix A.

If a command can have more than one argument, the arguments are usually separated from one another by either commas or semicolons as shown in the individual command descriptions.

In order to allow several commands on the same command line, a terminator is usually required to separate (the last argument of) a command from the next command. The rules for terminators are as follows:

- The last command on a line need not be followed by a terminator.
- If no form of a command base has arguments, then it needs no terminator. For example, the "D" command (move cursor down) has no form that takes arguments, and hence can be immediately followed by the next command; e.g., "DDD" moves the cursor down three lines (as does "3D").
- A sequence of one or more statements or declarations is terminated with "END". Alternately, the entire sequence can be enclosed in brackets, in which case no terminator is required.
- A sequence of one or more expressions is terminated with a semicolon.
- All other arguments may be terminated with a space, tab, or semicolon. If such an argument contains a space, tab, or semicolon, or starts with a left bracket, enclose it in brackets.

A backslash ("\") at the end of a command line serves as a continuation character to indicate that another command line is to be read and concatenated to the end of the first one. The backslash and <eol> are discarded, so precede the backslash with a space if you need to separate the first line from the second. Any number of continuation lines may be specified in this manner.

12.8.2. Direct Arguments

Most command arguments can be specified either directly or indirectly. In the case of a direct argument, the argument itself appears on the command line; e.g., "V i" displays the value of "i".

A direct argument may be enclosed in brackets to indicate explicitly where it ends. For example, if a file name argument has an embedded space, tab, semicolon, or brackets, it must be enclosed in brackets; use:

```
F[(my file)]
```

rather than:

```
F(my file)
```

since otherwise "(my" would be used as the file name. The debugger is "smart" about finding a matching right bracket; i.e., it ignores brackets in string constants and immediately after single quotes ('[' and ']' are the MAINSAIL integer constants for the character codes of left and irght bracket, respectively). An unmatched bracket may not otherwise occur in a bracketed argument.

12.8.3. Indirect Arguments

An indirect argument is a kind of subcommand that indicates how the text of the actual argument is to be obtained. For example "V @w" displays the value of the expression given by the current "word". The indirect arguments shown in Table 12.8.3-1 are provided.

argument	get argument text from:
@f<s>	contents of file named <s>
@l	rest of current line
@n	name (from cursor to space, tab, or ';')
@t	token (MAINSAIL identifier or constant at cursor)
@w	word (from cursor to space or tab)
@'<c>	text (from cursor to first occurrence of character <c>)
@<n>	<n> lines

Table 12.8.3-1. Indirect Arguments

The command letter after the "@" can be uppercase or lowercase; e.g., "@W" is treated the same as "@w". <s> is a file name, <c> is any character, and <n> is an integer count. "@t" allows MAINSAIL identifiers and constants of type boolean, integer, real, and bits. The identifier or constant starts at the cursor and continues as long as characters are encountered

that form a valid identifier or constant. "@l" takes the remainder of the current line from the current cursor position.

In "@f<s>", <s> may be a direct or indirect argument; e.g., "@f@l" uses the contents of the file whose name is given by the remainder of the current line. Enclose <s> in brackets if it is a direct name that contains a space, tab, or semicolon, e.g., "XS@f[foo.msl;2]" to execute statements in a file named "foo.msl;2".

The current line is the line at which the debugger cursor is currently positioned. This line can be in any file, since the "F" command allows a file to be specified that becomes the current file. In the display interface, the current line is the one on which the editor's cursor is positioned. Thus argument text can be built up using the editor in an arbitrary buffer, then a command issued with an indirect argument that refers to the text. It is often convenient to put the cursor on some output generated by a previous command such as "V" or ".V" and use the "@w" argument to reference that text.

Most arguments can be omitted, in which case an indirect argument is used by default, as shown in Table 12.8.3-2.

type of omitted argument	default argument
expression	@t (id or constant)
declarations or statements	@l (current line)
all others	@l (rest of current line)

Table 12.8.3-2. Defaults for Omitted Arguments

An indirect argument specifies the argument text, but the actual argument may be just the initial part of the argument text. However, an indirect name argument uses all of the argument text rather than just the part up to a space, tab, or semicolon as for a direct name argument.

An argument is assumed to be omitted if the end of the command line, or a comma or semicolon, appears before any argument text. For example, if the command list consists of just "V" with no arguments, the effect is the same as "V@t", which displays the value of the current identifier or constant. Similarly, "V ,j,k" is the same as "V @t,j,k", which displays the value of the current identifier or constant, then j and k.

12.9. Miscellaneous

(Long) bits, address, charadr, and pointer values are displayed in most cases in the preferred radix of the processor, as given by the system macro $preferredRadix (see the "MAINSAIL Language Manual"). The "H" and ".H" commands may be used to force these values to be displayed in hexadecimal.

The debugger accepts "{" in place of "[" and "}" in place of "]" for all occurrences of "[" and "]". No distinction is made; i.e., "{" may match "]" and "[" may match "}".

13. General Commands

13.1. Changing to/from Display-Oriented Interface: "{-}@"

In the line-oriented interface, the "@" command causes the debugger to switch to the display-oriented interface. The user is prompted for the display module (unless the "eparms" file contains the command "DONOTPROMPTFORDISPLAYMODULE") and baud rate (unless the "eparms" file contains the command "DONOTPROMPTFORBAUDRATE"). The current file is displayed with the cursor positioned to the same place as the "}" cursor was in the line-oriented debugger, and the current editor mode is set to command mode. If there is no current file, "CMDLOG" is used. In the display interface, the "@" command acts like <ecm>.

In the display interface, the "-@" command returns to the line interface, provided that the display interface was entered by means of the MAINDEBUG "@" command.

13.2. Quitting: "Q"

"Q" stands for "quit".

If a module is currently being executed under the debugger, it is aborted and control is returned to the debugger. Otherwise, execution of the debugger is terminated.

The debugger asks whether you really want to quit (in case you typed "Q" by mistake, perhaps thinking you were talking to the editor). Typing "Y<eol>" at this point raises the exception $abortProgramExcpt, whereas typing "N<eol>" aborts the quit command.

If any breakpoints are set, the "Q" command prompts whether to remove them before returning. If they are not removed, the debugger automatically gains control if a breakpoint is encountered during susequent execution.

13.3. Quitting Unconditionally: "+Q"

"+Q" unconditionally exits the debugger without prompting for any information.

13.4. Counts: {n}

In the line-oriented interface, a command line with no commands (i.e., no letters, optional count only, followed by an <eol>) re-executes the most recently executed command, except that the old count and subcount, if any, are replaced by the new ones, if any. Any arguments that follow the command are re-used.

Commands executed as a part of a breakpoint command are ignored when determining the most recently executed command for the purposes of the {n} command. For example, if a breakpoint is set with the command:

```
B:V foo<eol>
```

and then the "C" command is issued, and then the breakpoint is reached, typing "<eol>" to the debugger prompt causes the "C" command to be executed rather than "V foo".

13.5. Help: "?"

The user is prompted for which command summary to display. A response of "D" displays the general and debugging commands; "A" displays the debugger indirect arguments (see Section 12.8.1); "E" displays the editing commands.

13.6. Defining Macros: "/c<s>/"

Define a macro with name "c" to be the string <s>.

"c" is any alphabetic character (case is not distinguished). Thus, there are 26 available macros with the names "a" through "z". Each macro is initially defined as the empty string.

<s> is any command sequence, except that a slash within <s> can occur only within string quotes, and in the line interface, the final slash must appear on the same line as the initial one. Recursive definitions, such as "/c...=c.../", are allowed, though they could cause the debugger to loop indefinitely.

Use the "I" (Info) command to display all macro definitions. There is currently no way to save and restore macros between debugging sessions.

13.7. Invoking Macros: "{n}{-}=c"

The "=c" command invokes the macro "c", where "c" is any alphabetic character.

The effect is to replace "=c" with the string defined for "c". {n} and {-} apply only to the first command in the string that replaces "=c".

14. Debugging Commands

14.1. Most Recently Used Field Bases: $p1 and $p2

The debugger maintains the pointer variable $p1 as the value of the most recent non-Zero pointer p used in ".F p.f", "V p.f", ".V p", "H p.f", or ".H p".

It maintains $p2 as the previous different value of $p1.

Both $p1 and $p2 are classified according to the records or data sections that they reference, so they may be used in expressions such as "$p1.f", where f is the name of a field in the record currently referenced by $p1.

The commands in Table 14.1-1 show the field "f" of successively linked records of class c.

```
v p.f           # sets $p1 to p
v $p1.link.f    # sets $p1 to p.link, $p2 to p
v $p1.link.f    # sets $p1 to p.link.link, $p2 to p.link
v $p1.link.f    # sets $p1 to p.link.link.link,
                    $p2 to p.link.link
...
v $p1.link.f    # sets $p1 to p.link.link....link,
                    $p2 to previous $p1

In each case $p2 is set to the previous value of $p1.
```

Table 14.1-1. Examining a Linked List of Records with "$p1"

14.2. Displaying Array Slices: "A ary{,l1,u1{,l2,u2{,l3,u3}}}"

The "A" command displays elements of the array ary. In a one-dimensional array, l2 and subsequent arguments are omitted, and the elements ary[i], i ranging from l1 to u1, are displayed. In a two-dimensional array, l3 and u3 are omitted, and the elements ary[i,j], i ranging from l1 to u1, j ranging from l2 to u2, are displayed. In a three-dimensional array, the elements ary[i,j,k], i ranging from l1 to u1, j ranging from l2 to u2, and k ranging from l3 to u3,

are displayed. Later subscripts vary faster than earlier ones; e.g., k varies faster than j, and j faster than i.

The output shows the subscript(s) of each element and its value. For example, "A ary,0,2" might display the following, assuming ary is a one-dimensional integer array:

```
ary,0,2 =
[0] = 6
[1] = 113
[2] = -9
```

An asterisk ("*") may be used for any li or ui to stand for the corresponding bound declared for the array. Trailing asterisks may be omitted, so that when no bounds are specified, all elements of the array are displayed.

```
"A" with no explicit arguments displays all elements of
the array named by the current word.

"A ,1,5" displays elements 1 through 5 of the array named
by the current word.
```

Example 14.2-1. Uses of the "A" Command

ary may be a (long) integer or (long) bits constant that the user has determined to be the address of an array; e.g., "A 'H123456,1,10" is valid if 'H123456 is the address of an array. Arrays may be moved during memory management, so that constant addresses may become invalid.

14.2.1. Array Display Example

Suppose an array ary is declared and initialized as shown in Example 14.2.1-1. Some debugger "A" commands that might be used and their results are shown in Example 14.2.1-2.

```
INTEGER ARRAY(1 TO 3,1 TO 3) ary;
new(ary); INIT ary (1,2,3,4,5,6,7,8,9);
```

Example 14.2.1-1. Declaration and Initialization of ary

```
Debugger Command      Display        Same as
A ary,1,1,1,1         [1,1] = 1      V ary[1,1]

A ary,1,1,*,*         [1,1] = 1      A ary,1,1
                      [1,2] = 2
                      [1,3] = 3

A ary,1,*,1,1         [1,1] = 1
                      [2,1] = 4
                      [3,1] = 7

A ary,1,1,2,3         [1,2] = 2      A ary,1,1,2
                      [1,3] = 3

A ary,1,2,*,*         [1,1] = 1      A ary,1,2
                      [1,2] = 2
                      [1,3] = 3
                      [2,1] = 4
                      [2,2] = 5
                      [2,3] = 6

A ary,1,3,1,3         [1,1] = 1      A ary
                      [1,2] = 2
                      [1,3] = 3
                      [2,1] = 4
                      [2,2] = 5
                      [2,3] = 6
                      [3,1] = 7
                      [3,2] = 8
                      [3,3] = 9
```

Example 14.2.1-2. Sample Debugger "A" Commands for ary

14.3. Setting Breakpoints: "{+}B{[condition]}{:commands}"

This form of the "B" command sets a breakpoint in the current module. Control is given to the debugger when the breakpoint is encountered during execution.

The breakpoint is set at the iUnit (see Section 12.5) that corresponds to the current cursor position. The cursor must be positioned at the first character of the statement on which the breakpoint is to be set. "[condition]" and ":commands" are optional.

"condition" is an expression that is evaluated whenever execution reaches the breakpoint. If it is false no break occurs. "condition" is any MAINSAIL expression valid in the break context. "condition" is compiled just the first time the breakpoint occurs, so an invalid condition is not recognized until then. If an invalid condition is recognized, a break occurs, an error message is issued, and any commands associated with the breakpoint are not executed.

If "+" is specified, the breakpoint is an ignore-count breakpoint; i.e., it is taken even if the count specified to the "C" command has not yet been reached.

Each time the breakpoint occurs, "commands" is executed as if it were typed in response to the debugger's prompt. If "commands" contains the "C" (Continue) command, the debugger automatically continues from the break (commands after "C" are ignored).

```
"B[i = 5]" breaks when control reaches the breakpoint and
the variable "i" is equal to 5.

"B:Vs;C" breaks, shows the value of "s", then continues.
```

Example 14.3-1. Uses of the "B" Command

"commands" must be enclosed in brackets if another command is to follow the "B" command on the same command line. Otherwise, the debugger has no way of knowing where "commands" ends and the next command begins. If "commands" is present but not enclosed in brackets, then the rest of the command line starting after the ":" is used as "commands".

When a breakpoint is set, the debugger issues a message to this effect, indicating the module and iUnit at the break.

14.3.1. Where Breakpoints May Be Set

In the figures in this section, the caret ("^") appears immediately below the points at which MAINDEBUG breakpoints may be set. The carets do not appear in MAINSAIL source text and are not displayed by MAINDEBUG.

Breakpoints may be placed on any MAINSAIL statement (except a Begin Statement) or on an "END" (see Figure 14.3.1-1). When a breakpoint is placed after positioning the cursor in the source text, the cursor must be on the first visible (non-space, non-tab) character of the statement, or on the "E" in "END". The rules for placing breakpoints on something other than a non-empty statement are as follows:

```
IF foo THENB
^
    WHILE i > j DO IF bar(s,i,j) THEN RETURN;
    ^              ^                 ^
    getNextI(i,j) END;
    ^             ^
```

Figure 14.3.1-1. Breakpoints on MAINSAIL Statements and END's

- A breakpoint can be put at the final "END" of a procedure. See Figure 14.3.1-2.

```
PROCEDURE foo;
BEGIN
...
END;
^
```

Figure 14.3.1-2. Breakpoint at Final "END" of a Procedure

- When a single step returns from a procedure, it breaks at the caller's final "END" if the call was the last thing executed in the procedure. The final "END" of a procedure is not reached if the procedure executes a Return Statement or is aborted by an exception.

- Breakpoints can be set at empty Iterative Statement bodies, and the single step command stays on the loop body until the terminating condition is met. See Figure 14.3.1-3.

- If a Case Statement selects an Empty Statement, a single step breaks at the semicolon for the Empty Statement. See Figure 14.3.1-4.

```
DO UNTIL NOT p := p.link;
^  ^
```

Figure 14.3.1-3. Breakpoint on an Empty Iterative Statement Body

```
[ ] ;
    ^
```

Figure 14.3.1-4. Breakpoint on an Empty Case

- If a procedure has no body, a break can still be set for it on the semicolon terminating the null body of the procedure. See Figure 14.3.1-5.

```
PROCEDURE foo;;
              ^
```

Figure 14.3.1-5. Breakpoint on a Procedure with No Body

- A break can be set at any "END" except the terminating "END" of a Case Statement. See Figure 14.3.1-6.

```
BEGIN IF foo THEN bar ELSE baz END
      ^           ^         ^   ^
```

Figure 14.3.1-6. Breakpoint on an "END"

- A breakpoint placed on an "END" is reached only if control flows through the "END". For example, the END's in Figure 14.3.1-7 are not reached, since a "DONE" is always executed first.

```
DOB IF foo THENB bar; DONE END # not reached
^   ^             ^    ^    ^
    EB baz; DONE END; # not reached
       ^    ^    ^
    END; # not reached
    ^
```

Figure 14.3.1-7. END's Not Reached

14.3.2. Setting a Breakpoint in a Specific Module Instance or Coroutine

Breakpoints are associated with a module's control section, i.e., with the code that implements its procedures. Since all instances (data sections) of a module in all coroutines share the same control section, confusing interactions can occur if you mean for a breakpoint to be associated with just one instance of a module or just one coroutine when the module has several instances or is running in several coroutines.

A conditional breakpoint can be used in conjunction with a debugger variable (see Section 14.6) to cause the breakpoint to be taken in only one data section or coroutine. First declare a debugger pointer variable, say "p":

```
.D POINTER p;
```

Then (if the data section or coroutine desired is the current one) assign p the value of the desired data section or coroutine with:

```
XS p := thisDataSection
```

or:

```
XS p := $thisCoroutine
```

(if the target data section or coroutine is not the current one, you must find some other way of assigning its value to p). Then set the breakpoint with the appropriate condition, i.e.:

```
B[thisDataSection = p]
```

or:

```
B[$thisCoroutine = p]
```

14.4. Setting a Breakpoint at a Specified iUnit: "{+}B@"

The full syntax for this command is:

```
{+}B@ mod1.iUnit1[cond1]:cmd1, ..., modn.iUnitn[condn]:cmdn
```

Breakpoints are set in the specified modules. Control is given to the debugger when any of the breakpoints is encountered during execution.

The "@" must immediately follow the "B". modi, iUniti, condi, and cmdi are direct arguments.

A breakpoint is set at iUniti of the module modi for each "modi.iUniti" specified. iUniti must correspond to the start of a statement in modi. If "modi." is omitted, the current module is used (an error occurs if there is no current module). "[condi]" and ":cmdi" are optional, and have the same form and function as the "[condition]" and ":commands" fields of the "B" command. "cmdi" must be enclosed in brackets if there is another argument for "B@" or another command after the "B@" on the same command line.

If "+" is specified, the breakpoint is an ignore-count breakpoint; i.e., it is taken even if the count specified to the "C" command has not yet been reached.

Use of this command when the iUnit of a desired breakpoint is known (e.g., from previous debugging activity) eliminates the need to bring in a file and then use MAINDEBUG's or MAINEDIT's editing commands to position to the statement at which the break is to be set.

14.5. Continuing: "{n}{.{i}}C"

The "C" command continues execution until the nth breakpoint is encountered, or until an ignore-count breakpoint is encountered.

For the purposes of the "C" command, a conditional breakpoint is considered to be "encountered" only if its associated condition is true.

The ":commands" part of any breakpoints skipped over by the "C" command is ignored; i.e., the commands are not executed.

Ignoring the optional count {n}, there are three valid forms of the "C" command:

- "C": the basic form, which continues execution from the breakpoint.

- ".iC": continue execution from iUnit i, rather than from the breakpoint. i must correspond to the start of a statement in the current procedure. To determine the iUnit of a given position, put the debugger cursor there and issue the "1I" debugger command. This form allows one to back up or skip over statements. The effect is undefined if the iUnit i is not on the same "statement level" as the breakpoint, e.g., if i is in the body of an Iterative Statement that starts after the breakpoint.

- ".C": continue execution from the current cursor position. The cursor must be at a point where a breakpoint could be set. This is a quick way of doing ".iC" where i corresponds to the current cursor position, and hence the same rules apply about the derived iUnit i.

14.6. Declaring Debugger Variables: ".D d1;...;dn {END}"

The di are MAINSAIL variable declarations. The declarations are compiled in the current context, and the declared "debugger variables" can then be used in later debugger commands in any context. MAINSAIL macro definitions (i.e., using "DEFINE" or "REDEFINE") may also occur among the di.

The terminating "END" is required if additional debugger commands are to be given on the same line as the ".D" command; the "END" is a terminating token for the compiler. Alternatively, the di may be enclosed in brackets: ".D[d1;...;dn]".

".D" with no arguments uses "@1" as the default argument; i.e., the current line supplies the argument text and is expected to consist of valid MAINSAIL definitions and declarations. In the display interface, the editor can be used to prepare n lines of definitions and declarations in a separate buffer, and then the ".D@n" command can be issued with the cursor anywhere on the first line.

To avoid confusion, the declared identifiers should not be the same as an identifier in the program being debugged.

A debugger variable may be used as an iterative variable in the FOR-clause of an Iterative Statement (the language normally requires that iterative variables be local variables).

14.7. Executing Modules: "E {moduleOrFileName}{arguments}"

The "E" command is used to execute a specified module, or to invoke the MAINSAIL executive.

```
      .D INTEGER ddi1; STRING dds1; POINTER(c) ddp1;

declares three debugger variables.

           .D DEFINE dm1 = "abc"; STRING dds1;

defines the debugger macro dm1, and declares the variable
dds1.

                              .D@5

reads five lines starting at the current line.  These
lines are assumed to consist of definitions and
declarations.

                          .D@ffoo.msl

reads the definitions and declarations from file f.  Thus,
a file of generally useful definitions and declarations
may be prepared and processed during any debug session.
```

Example 14.6-1. Uses of the ".D" Command

This command reads the remainder of the current line. It interprets the line as an optional module or file name, followed by optional arguments. If a module name is specified, the module is executed with the specified arguments. If a file name is specified, the file is assumed to contain a MAINSAIL object module, which is executed with the specified arguments. Control returns to the debugger when the module execution is complete.

If moduleOrFileName is omitted, the MAINSAIL executive is invoked (see the "MAINSAIL Utilities User's Guide" for a description of the MAINSAIL executive). The MAINSAIL executive prints an identifying banner, then prompts with "*" and waits for input. Control returns to the debugger when just <eol> is typed to the "*" prompt.

The debugger context remains unchanged.

Use this command to execute utility modules, or to start execution of the program to be debugged by specifying the initial module of that program.

```
The commands:

        b@xyz.1340
        exyz

set a breakpoint at offset 1340 of XYZ, then execute XYZ.
```

Example 14.7-1. Use of the "E" Command

14.8. Displaying Individual Fields of Unclassified Pointers: ".F p,f1,f2,..."

".F p,f1,f2,..." is the same as "V p.f1,p.f2,...", except that p can be an unclassified pointer.

For "V p.f" the compiler must know p's class c, and f must be a field of c; otherwise, the compiler issues an error message. However, for ".F p,f", the debugger looks in the class descriptor of the record pointed to by p and sees whether there is a field named "f". If so, the class descriptor indicates its type and displacement from the start of the record, so that the debugger has enough information to print its value. Since "fi" is not compiled, it must be the actual name of a field rather than, say, a macro name that expands to a field name.

"p" may be a (long) integer or (long) bits constant that the user has determined to be the address of a record or data section; e.g.:

```
.F 'H123456,f
```

is valid if 'H123456 is the address of a record or data section with a data field "f". Records and data sections may be moved during memory management, so that constant addresses may become invalid.

The ".F" command is particularly useful if the address of a record is known, say 'H123456, and one or more of its fields are to be displayed: ".F 'H123456,f1,f2,...". The command "V 'H123456.f1" would not compile, since "'H123456.f1" is not a valid MAINSAIL expression.

14.9. Displaying Hexadecimal Values: "H {expr1, ..., exprn}"

The "H" command is identical to the "V" command, except that (long) integer, (long) bits, (long) real, address, charadr, and pointer values are displayed in hexadecimal, even if the preferred radix for the processor is not hexadecimal.

14.10. Displaying Hexadecimal Field Values: ".H {p1, ..., pn}"

The ".H" command is identical to the ".V" command, except that (long) integer, (long) bits, (long) real, address, charadr, and pointer values are displayed in hexadecimal, even if the preferred radix for the processor is not hexadecimal.

14.11. Information: "I"

The "I" command displays debugger information.

The following information is displayed:

- Current file name and position, if using the line-oriented interface.
- The coroutine of the current context, if it is not the root coroutine "MAINSAIL".
- Current module, procedure, and iUnit.
- iUnit of the cursor position.
- Current breakpoint file name and position, if any.
- The coroutine of the breakpoint context, if it is not the root coroutine "MAINSAIL".
- Current breakpoint module, procedure, and iUnit, if any.
- The debuggable procedure count (the number of debuggable procedures that have been entered during the current MAINSAIL session).
- Count break value, if set (see the description of the "K" command).
- Module, iUnit, condition, and command for each breakpoint. "T" is shown for temporary breakpoints and "I" for ignore-count breakpoints.

- Any macro definitions.

14.12. Brief Information: "1I"

The "1I" command is a short form of the "I" command which displays the following information:

- Current file name and position, if using the line interface.
- The coroutine of the current context, if it is not the root coroutine "MAINSAIL".
- Current module, procedure, and iUnit.
- iUnit of the cursor position.

In the display-oriented interface, all this information is shown on the message line.

14.13. Jumping into Procedures: "{n}J"

Execute the next n statements, "jumping" into any debuggable procedures invoked; then break. The "J" command is just like the "S" command, except that a break also occurs on the first statement of any debuggable procedure invoked, and at a call to $resumeCoroutine, "J" steps into the resumed coroutine.

Statements executed within non-debuggable procedures are not counted among the n statements to be executed.

Commands on the same command line after a "J" command are ignored.

14.14. Count Breaks: "K n"

The "K" command sets the count break value to n.

Execution stops when the debuggable procedure count is incremented to n. Each time a procedure in a debuggable module is entered, the count is incremented so that upon the nth entry to a debuggable procedure, the count break value is n.

Procedures compiled inline or non-debuggable do not increment the count.

The current count break value is displayed by the "I" command.

Count breaks provide a means of isolating bugs. If the count value is noted when an error occurs (using the "I" command), program behavior can be observed by setting a count break for some value less than that at which the error occurred, letting the program run until the count break is reached, and then stepping (with the "S" and/or "J" commands) to the error. If a debug session is "repeated", except that some modules are debuggable that previously were not, or vice versa, debug counts noted from the first session may not be applicable to the second.

"K1" sets the count break value to 1, so that a break occurs the first time a debuggable procedure is entered. A convenient way to start a debugging session is to issue "K1Em" to the debugger, which sets the count break to 1, then executes the module M. If M is a debuggable module, execution breaks at the start of its initial procedure.

14.15. Setting Context to Current Breakpoint: "M"

This form of the "M" command (no argument) sets the context to the current breakpoint. This is a convenient way to get the cursor back to the current breakpoint after it has been moved around during debugging commands. If invoked by means of the "DEBUG" response to the "Error response:" prompt, the context is set to the point of the error.

14.16. Setting Context to a Module: "M s"

"M s" sets the debugger context to the module specified by s.

s is either a module name or a file name. If a module name is specified, the module context is set to the bound instance for that module, if it exists, and the file context is set to the start of the file that the compiler first encountered when the module was compiled. If a file name is specified, then that file is assumed to contain some object module m, and the effect is the same as "M m".

See Section 14.17 for setting the context to a particular instance of a module.

Enclose the argument to the "M" command in brackets if it contains a space, tab, semicolon, or brackets.

14.17. Setting Context to a Particular Module Instance: ".M p"

"p" must point to a data section for some module m. The debugger context is set to that instance of m.

"p" may be a (long) integer or (long) bits constant that the user has determined to be the address of a data section; e.g.:

```
.M 'H123456
```

is valid if 'H123456 is the address of a data section. Data sections may be moved during memory management, so that constant addresses may become invalid.

14.18. Releasing a Module: "-M m"

"-M m", where m is a module name, disposes the module m (if it is present) and closes the intmod for m (if it is open). If the current context is set to m, the current context is cleared. The argument m is an id argument.

This command must be carried out for each module m that has taken part in a debug session, and then been recompiled without exiting the debugger, and is then to take part in further debugging. If this command is not given for such a module m, then further debugging will use the old control section for m if it is still in memory, and the old intmod for m if it is still open. In other words, the old version of m will be used even though it has been recompiled, which can be quite confusing.

If you have been debugging and recompiling without exiting the debugger, and you start having problems such as the debugger claiming that the cursor is not at the start of a statement when you try to set a breakpoint, or a recompiled module seems to behave as if it had not been recompiled, then you have probably forgotten to do "-M m" on one or modules. A way to ensure that there are no old objmods or intmods in memory is to exit the MAINSAIL session and start a new one.

If at any time during a MAINSAIL session you recompile a module m that is in memory, and then continue the session in such a way that m will be used again, you must first dispose of m in order to use the newly compiled control section. As described above, you can use the "-M" command to do this if you are in the debugger. If you are not in the debugger, invoke the utility module DISPSE, which prompts for modules to be disposed. DISPSE does nothing about intmods, but they do not affect execution unless you are debugging, and if you reinvoke the debugger it will start out with no open intmods. If you recompile many modules, then rather than try to dispose all of them using DISPSE, it is probably simplest to exit MAINSAIL and start a new MAINSAIL session.

14.19. Walking the Call Stack: "{n}{-}N"

"N" sets the debugger procedure context to the nth debuggable caller or callee.

If "-" is not present, the context is set to the nth debuggable caller (calling procedure) in relation to the current procedure. If "-" is present, the context is set to the nth debuggable callee (called procedure) in relation to the current procedure.

The "N" command allows the user to walk up and down the procedure call stack.

14.20. Walking the Exception Stack: "{n}{-}.N"

If an exception is active in the current context, ".N" sets the debugger context to the debuggable statement at which it was raised (i.e., in the raiser coroutine if it differs from the raisee coroutine). If there is no current debugger context, the debugger sets its context to the debuggable statement in which the current exception was raised. The exception is not considered active in the new context.

"-.N" undoes the most recent ".N" command; i.e., the debugger context is put back to where it was before the most recent ".N" command, and the exception is considered active in the new context. ".N" commands are not remembered across program continuation.

"n.N" performs the ".N" command n times, and "n-.N" performs the "-.N" command n times.

".N" complains if there is no active exception; "-.N" complains if there is no "-.N" command to undo.

14.21. Setting the Current iUnit: "O n"

"O" ("Offset") sets the debugger position context.

The cursor is positioned to the file position corresponding to iUnit n in the current module.

This command is useful for positioning to the offset given for a debuggable module when an error occurs. The offset given in an error message for a non-debuggable module differs from that given for the same module compiled with the debug option, since additional code is generated for debuggable modules. This means that non-debuggable modules must be recompiled debuggable and the program rerun before the "O" command can be used with the offset given in an error message.

14.22. Opening a Coroutine: "OC s"

"OC s" changes the debugger context to the coroutine with name s. The procedure in the new debugger context is the first debuggable procedure in the call chain starting at the resume point

of coroutine s (an error message is given if there is no such procedure). The cursor indicates the point at which control last resided in that procedure. This procedure is now the current debugger context, so that, for example, its local variables can be examined, and the "N" and "-N" commands walk the new coroutine's stack.

The strings shown in Table 14.22-1 may be used in place of s to give a way to move around the coroutine tree, or follow the dynamic resume list, by following the indicated link of the coroutine for the current debugger context. For example, "OC #UP" moves the context to the current coroutine's parent.

#UP	#DOWN
#LEFT	#RIGHT
#PREV	#NEXT

Table 14.22-1. Symbolic Names for Coroutine Fields

Enclose s in brackets if it contains a space, tab, semicolon, or brackets.

14.23. Opening an Intmod Library: "OI s"

Open the intmod library with name s, and put it at the head of the list of intmod libraries searched by the debugger. The debugger can now find intmods in s. Intmod searches are also governed by the MAINEX subcommands "OPENINTLIB", "INTFILE", "INTLIB", and "INTDEFAULT".

Enclose s in brackets if it contains a space, tab, semicolon, or brackets.

14.24. Opening an Objmod Library: "OL s"

Open the module library with name s, making its objmods available for execution. Objmod searches are also governed by the MAINEX subcommands "OPENEXELIB", "EXEFILE", "EXELIB", and "EXEDEFAULT".

This command must be issued before the "Em" command if the module m resides in library s (and s is not already open).

Enclose s in brackets if it contains a space, tab, semicolon, or brackets.

14.25. Debugger Options: "{-}OP s"

Set or clear debugger options. If "-" is present, the options are cleared; otherwise, they are set.

s is a sequence of options. The options are direct arguments which may be separated by spaces and tabs.

There is currently only one standard option, "V".

The "V" option indicates verbose prompt mode. If set, the default prompt:

```
<module name>.<procedure name>:
```

is used in the line-oriented interface; if turned off, the line-oriented prompt is:

```
MAINDEBUG:
```

14.25.1. recursiveDebug and nonRecursiveDebug Modes

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

The temporary option "R" sets recursiveDebug mode; "-OP R" sets nonRecursiveDebug mode. In nonRecursiveDebug mode, breakpoints are temporarily removed from the code, and count breaks are disabled, while the debugger is in control.

In recursiveDebug mode, the debugger leaves break points in the code at all times, and leaves the count break value as set with the "K" command. Thus a break point can occur even while the debugger itself is carrying out a user command if any code executed by the debugger has been compiled debuggable; this is called a recursive invocation of the debugger.

For example, if the user has set a break point in some procedure foo, and then executes the debugger command "V foo", the break point occurs during the "V" command. This is often quite useful. However, if any of the MAINSAIL runtime modules used by the debugger (such as the MAINSAIL compiler or text editor) are compiled debuggable, an unexpected count break could occur in one of those modules, leading to confusing output and possibly causing MAINSAIL to crash. Thus recursive invocation is useful if no system modules are compiled debuggable, but is otherwise dangerous.

In standard releases, the MAINSAIL runtime system is not provided to customers with any module compiled debuggable, so that recursive debugger invocation does not occur due to system modules. However, there are times (e.g., during customer testing of new releases) when debuggable modules are delivered. If the debugger detects that a supporting system module is compiled debuggable, it clears recursiveDebug mode (i.e., it sets nonRecursiveDebug mode) to avoid the potential confusion caused recursive debugger invocations.

In nonRecursiveDebug mode, breakpoints are temporarily removed from the code, and count breaks are disabled, while the debugger is in control. They are restored when the debugger relinquishes control because of commands such as "C", "S", "J", and during invocation of a module with the "E" command. However, this means that user breaks do not occur while in nonRecursiveDebug mode (e.g., "V foo" does not encounter a break point since it has been temporarily removed). Though this is really just a minor inconvenience, it is mentioned to avoid confusion should the user notice that recursive breaks are not occurring. The "I" command indicates whether the debugger is currently in recursiveDebug mode or nonRecursiveDebug mode. The debugger prints a message whenever it clears recursiveDebug mode due to detecting a debuggable runtime module.

In order to allow debugging of runtime modules such as the compiler or editor, it is necessary to be able to set recursiveDebug mode so as to override the debugger's good intentions. For this reason the mode can be explicitly set and cleared with the "OP" command: "OP r" puts the debugger in recursiveDebug mode, and "-OP r" puts the debugger in nonRecursiveDebug mode. The debugger remembers whether either of these commands has been given, and if so does not thereafter alter the mode implicitly.

14.26. Removing a Breakpoint: "R"

Remove a breakpoint. The cursor must be positioned at the breakpoint.

14.27. Removing Breakpoints at Specified iUnits: "R@ module1.iUnit1, ..., modulen.iUnitn"

Remove the specified breakpoints.

The "@" must immediately follow the "R". If "modulei." is omitted, the current module is used.

14.28. Removing All Breakpoints: "R@@"

Remove all breakpoints.

"@@" must immediately follow "R".

14.29. Stepping into Procedures: "{n}S"

Execute the next n statements, "stepping" over procedures; then break. The "S" command does not break within any invoked procedure (use the "J" command for this). Statements executed within invoked procedures are not counted among the n statements to be executed.

Beware of breaking or single stepping on statements that involve low-level manipulations written with the assumption that memory management does not occur during execution of the statements, since execution of the debugger between such statements could lead to memory management. This precludes the use of single stepping on many "sensitive" areas of the MAINSAIL runtime system involved with storage allocation. For this reason, the user should avoid stepping through any procedure bodies distributed as part of the MAINSAIL system.

When single stepping, the debugger breaks on each place it reaches where a breakpoint could be set (see Section 14.3.1); furthermore, the debugger breaks on each expansion call generated by the use of repeatable parameters. For example, since the statement:

```
write(logFile,"Count: ",count,eol)
```

is compiled as if it were:

```
write(logFile,"Count: ");
write(logFile,count);
write(logFile,eol)
```

the "S" command stops three times at "write".

If, while executing "nS", a breakpoint is encountered before the nth statement has been executed, the debugger nevertheless breaks; i.e., breakpoints have precedence over the step count n.

At a call to $resumeCoroutine, the "S" command does not step into the resumed coroutine, but instead breaks when control next returns to the statement after the $resumeCoroutine call. Usually this occurs when the coroutine is next resumed; however, the breakpoint for the "S" command is associated with the object module, not with a coroutine, so that it could next be

encountered in some coroutine other than the one in which the "S" command was given. The "J" command does step into coroutines.

Commands on the same command line after an "S" command are ignored.

14.30. Setting Temporary Breakpoints: "{+}T{[condition]}{:command}"

Same as the "B" command, except that it sets a temporary breakpoint. All temporary breakpoints are removed whenever a break occurs.

14.31. Setting a Temporary Breakpoint at a Specified iUnit: "{+}T@"

The full syntax for this command is:

```
T@ module1.iUnit1[cond1]:cmd1, ..., modulen.iUnitn[condn]:cmdn
```

Same as the "B@" command, except that it sets temporary breakpoints. All temporary breakpoints are removed whenever a break occurs.

14.32. Displaying Values: "V expr1, ..., exprn"

Display the values of the specified expressions. The expri must be MAINSAIL expressions valid in the current context.

In the display debugger, if the output from the "V" command is only one line long, it is by default displayed on the MAINEDIT message line; otherwise, the output is written to the buffer "CMDLOG". Any expri followed by a comma is written to buffer "CMDLOG"; thus, "Vi," writes i to CMDLOG even if there is only one line of output, and "V," writes the value of the current word to "CMDLOG".

If the current line is:

```
IF nextEst := (x / est + est) / 2. THEN ...
   a b        cd   e     f      g       (cursor positions)
```

then the indicated cursor positions result in the following expressions being compiled for "V@1" (argument text starts at cursor and extends to end of current line):

```
cursor at    compiled expression
    a        nextEst := (x / est + est) / 2.
    b        xtEst := (x / est + est) / 2.     (probably an error)
    c        (x / est + est) / 2.
    d        x / est + est
    e        est + est
    f        est
    g        2.
```

Even though the argument text extends to the end of the line, the compiler stops once it finds the end of an expression (text after the terminating token is ignored).

14.33. Displaying Fields: ".V {p1, ..., pn}"

Display objects pointed to by the pointers pi.

".V p", where p points to a record or data section, displays all the (interface) data fields of p's record or data section. If p points to an array, the declaration of the array is displayed, i.e., the type, name, and bounds of the array. Use "A" to display array elements.

p may be a (long) integer or (long) bits constant that the user has determined to be the address of an array, record, or data section; e.g.:

```
.V 'H123456
```

is valid if 'H123456 is the address of an array, record, or data section. Arrays, records, and data sections may be moved during memory management, so that constant addresses may become invalid.

p may be a classified pointer or address of the form "q:c", where q is a pointer or address expression and c the name of a class. The memory at q is displayed as if it were a record of the class. Alignment of fields in the class must comply with the processor's requirements, if any. $p1 and $p2 are not updated by this form of p. A (long) integer or (long) bits constant cannot be used as q, since the compiler would complain; instead of ".V 'H1234:c", use ".V cva('H1234L):c".

When using the display-oriented interface there is an especially convenient way to examine the records in a linked list. Suppose the records are linked through a pointer field "link", and the first record is pointed to by a pointer variable "p". Start with ".Vp", which prints all the fields of the first record. To view the next record, move the cursor to the value displayed for "link", e.g., 'H123456L, and give the command ".V<eol>". This uses the current word ('H123456L) as the pointer to the next record. Any number of additional records can be viewed this way.

14.34. Examining Memory: "XM expr"

Examine consecutive memory locations.

"expr" is a pointer, address, or charadr expression, or a (long) integer or (long) bits constant. Memory locations are displayed starting at the memory address given by "expr", and initially proceeding towards higher addresses. If expr is a charadr, it is rounded down to the next lower data-type-aligned address before use.

The command enters a subcommand mode with a ":" prompt. Responses may include any of the subcommands listed in Table 14.34-1.

```
<t>      display type <t> = bo,i,li,r,lr,b,lb,s,a,c,p
t        display characters
-        reverse direction
<eol>    display same type in same direction
?        show this list
<x>      (<x> is any text that does not match another
          subcommand)
         exit subcommand mode and execute <x> as debugger
         commands. Use <x> = <space><eol> to exit
         subcommand mode and not execute any debugger
         commands.
```

Table 14.34-1. "XM" Subcommands

Each time a value is displayed, the memory address is automatically incremented (default) or decremented by the size of the examined value. The subcommand "t" causes memory to be examined as characters. Five integers' worth of text is displayed each time (typically 10 to 25 characters, depending on the machine).

14.35. Executing Statements: "XS s1;...;sn {END}"

The si are statements that are compiled and executed. The si must be MAINSAIL statements valid in the current context.

The "END" is required if additional debugger commands are to be given on the same line as the "XS" command. It is a terminating token for the compiler. Alternatively, the si may be enclosed in brackets: "XS [s1;...;sn]".

"XS" with no arguments uses "@1" as the default argument; i.e., the current line supplies the argument text.

```
                        XS i := 0

assigns 0 to i.

               XS cRead(s) END v s[1 TO 5]

removes the first char of s, then displays its first
five characters.

       XS FOR i := 1 UPTO 10 DO ttyWrite(a[i],eol)

displays elements 1 through 10 of a.  The "A" command
could be used instead.

                       XS@ffoo.msl

compiles and executes statements in the file "foo.msl".
```

Example 14.35-1. Use of the "XS" Command

15. Editing Commands

Except as noted, these commands are available only in the line-oriented interface.

For the purposes of this chapter, "words" in source text are composed of visible characters and are separated by spaces and tabs. Commands that usually display a cursor do not do so if there are no words on a line, i.e., if the line is blank.

15.1. Moving Down: "{n}D"

Move the cursor down n lines.

The cursor is positioned at the beginning of the first word on the line.

15.2. Setting File Context: "F s"

Set the debugger file context to the file named s.

The current source file, if any, is closed, and the file context is set to s. The debugger context is otherwise unaffected. The cursor is positioned at the first word in the file. The file s need have no particular relationship to any module being debugged, though it usually contains (some of) the source text of a debuggable module. This command may also be used with the display debugger interface.

In the line-oriented interface, if s is not specified the current file is closed and the file context is cleared.

15.3. Moving to Page and Line: "{p}{.n}G", "+{n}G", and "-{n}G"

Move the cursor to the specified page and line.

The cursor is positioned to the first word of the specified line and/or page. There are several forms as shown in Table 15.3-1.

Form	Position cursor at
G	Top of next page
+n G	Forward n pages, top line
-n G	Backward n pages, top line
p G	Top of page p
.n G	Line n of current page
. G	Top of current page
p.n G	Line n of page p

Table 15.3-1. Cursor Positioning with the "G" Command

15.4. Listing Lines: "{-}{n}L"

If "-" is not present, list n lines starting at the current line; if "-" is present, list the n lines before the current line and the current line.

15.5. Moving to a File Position: "P n"

Put the cursor at position n in the current file.

n is a long integer constant (no trailing "L").

15.6. Moving Up: "{n}U"

Move the cursor up n lines.

The cursor is positioned at the beginning of the first word on the line.

15.7. Displaying a Window of Lines: "{n}W"

Display a window of n lines centered about the current line.

If n is omitted, a default amount is displayed.

15.8. Moving Left: "{n}<"

Move the cursor left n characters on the current line.

15.9. Moving Left by Words: "{n}("

Move the cursor left n words on the current line.

15.10. Moving Right: "{n}>"

Move the cursor right n characters on the current line.

15.11. Moving Right by Words: "{n})"

Move the cursor right n words on the current line.

15.12. Searching for a Character: "{n}{-}'c"

Search the current line for the nth occurrence of the character c.

The search starts at the current cursor position. If {-} is present, search backward (to the left); otherwise search forward (to the right).

Case is not distinguished in the search character.

15.13. Searching for a String: "{n}{-}"{s"}"

Search the current file for nth occurrence of the string s.

If "-" is not present, the remainder of the current file is searched, starting one character position past the current cursor position, for the nth occurrence of string s. A double quote character in s must be doubled, as in a MAINSAIL string constant. The terminating quote may be omitted if no additional commands are given on the line.

Case is not distinguished in the search string.

If "-" is present, search backwards towards the beginning of the file, starting at one character before the cursor.

If s (and the trailing double quote) is not specified, the previous search string is used. For example, a command line consisting of a quote all by itself searches for the most recently used search string. To continue the search for the next occurrence of the string, it is sufficient to type <eol>, since that repeats the previous command.

16. Line-Oriented Interface Sample Session

In this example, several debugger commands are used on a small MAINSAIL module called SAMPLE. The complete text of SAMPLE appears in Appendix B. This example is intended to be as machine-independent as possible; however, the particular numbers and source file names that appear in the examples may not be the same on your operating system. Such differences as may exist are not important to the substance of the discussion.

Before it can be used with the MAINSAIL debugger, a module must be compiled with the compiler's "DEBUG" subcommand. Attempting to use the debugger on a module compiled without the "DEBUG" option results in a message like the following from the debugger:

```
Not a debuggable module: SAMPLE
```

Compiling SAMPLE with the "DEBUG" option is accomplished as shown in Example 16-1.

```
compile (? for help): sample,<eol>
> debug<eol>
> <eol>

Opening module $SYS from intlib...

sample 1
Objmod for SAMPLE stored on sample-<os>.obj
Intmod for SAMPLE stored on sample-<os>.int

compile (? for help):
```

Example 16-1. Compiling the Module SAMPLE with the "DEBUG" Option

When SAMPLE's source file name is typed in response to the "compile (? for help):" prompt, it is followed by a comma. The compiler interprets this as a request to enter compiler subcommand mode. It therefore gives the subcommand prompt, a greater-than sign (">"). You could type a question mark here to see a list of available compiler subcommands. Instead the word "DEBUG" is typed, meaning that all subsequent modules should be compiled debuggable. Typing just <eol> to the ">" prompt means to exit compiler subcommand mode and proceed with the compilation. The new compiler output files replace any files with the

same name that may have existed previously; if SAMPLE were run now, the version run would be the debuggable one (unless SAMPLE's object module is also in a objmod library open for execution; see the description of MAINEX in the "MAINSAIL Language Manual" for the rules governing the search for an object module). The intmod generated for SAMPLE contains the information needed by the debugger to determine the correspondence between source text and object code so that it can correctly display the execution of statements. The execution of debuggable object modules is nearly identical to that of non-debuggable modules, except that the debuggable modules are slightly larger and slower. In the case of a tiny, I/O-bound module like SAMPLE, the difference in execution speed is too small to observe.

To start the debugging session, run the MAINSAIL debugger, MAINDEBUG, by typing its module name ("DEBUG") to the MAINSAIL executive.

Next give the name of the module to be debugged with the debugger's "M" command. It is possible to type a question mark to the "MAINDEBUG:" prompt to see a list of commands available in the debugger.

The debugger displays the first line of SAMPLE's source file, along with the names of the current module and procedure (there is no current procedure in Example 16-2).

```
*debug<eol>

MAINDEBUG (tm) (type ? for help)
Copyright (c) 1984, 1985, 1986, 1987, 1988, and 1989 by
 XIDAK, Inc., Menlo Park, California, USA.

MAINDEBUG: m sample<eol>
Opening intmod for SAMPLE from sample-<os>.int
Opening intmod for $SYS...

}BEGIN "sample"
SAMPLE: d<eol>
```

Example 16-2. Debugging the Module SAMPLE

Now repeatedly give the debugger's "D" command, which tells the debugger to position Down one line in the file, until you come to a line of code where it is possible to place a breakpoint. Breakpoints may be placed only on executable MAINSAIL statements (except as noted in Section 14.3.1). After each debugger command, the debugger displays the current source line with a curly-bracket cursor ("}") pointing to the current position on the line, as shown in Example 16-3 (no cursor is shown when the current position is on a blank line).

```
SAMPLE: d<eol>

}INTEGER bucketSize;  # size of hash bucket
SAMPLE: d<eol>

SAMPLE: d<eol>

SAMPLE: d<eol>

}INTEGER PROCEDURE hash (STRING s);
SAMPLE: d<eol>

}BEGIN
SAMPLE: d<eol>

}INTEGER        h,i,j;
SAMPLE: d<eol>

}h := length(s); i := h MIN 4; j := 1;
SAMPLE: b<eol>
Break set at SAMPLE.11
```

Example 16-3. Using The "D" and "B" Commands

The debugger's "B" command was used to set a breakpoint on the first statement in the file. This is not the first statement executed by the program, but it is a useful place to break. The "B" command with no arguments sets a breakpoint at the current cursor position, which must be on the first character of a statement or the "E" of an "END".

The "E" command now invokes SAMPLE. Refer to Example 16-4.

```
SAMPLE.: esample<eol>

Size of hash bucket: 131<eol>
Next key to be hashed (eol to stop): A<eol>
```

Example 16-4. Using The "E" Command

SAMPLE starts running until it hits the breakpoint, as shown in Example 16-5.

```
}h := length(s); i := h MIN 4; j := 1;
SAMPLE.HASH: v s<eol>
s = "A"
```

Example 16-5. Hitting the Breakpoint

The first command given at the breakpoint is the debugger's "V" command, which prints out the value of a variable. In this case the variable examined is the string "s", which is what was typed in response to SAMPLE's "Next key to be hashed (<eol> to stop):" prompt.

Now use the debugger's "S" command to step over the first statement in the procedure where the breakpoint was set. See Example 16-6.

```
SAMPLE.HASH: s<eol>

h := length(s); }i := h MIN 4; j := 1;
SAMPLE.HASH:
```

Example 16-6. Single Stepping

Notice how the debugger's cursor is now positioned at the next statement to be executed. The statement "h := length(s)" has just been executed. Example 16-7 shows subsequent uses of the single step ("S") and examine value ("V") commands.

The "V" command can be used with more than one variable at a time if the variables are in a list separated by commas.

The "C" command continues execution until the next breakpoint. Here SAMPLE finished the procedure in which it originally broke, then returned to the main program and asked for some more input. After the input is typed to SAMPLE, a break is reached again at the original breakpoint, which has not been removed. The debugger displays the source line on which it broke just as it did the first time it broke there. See Example 16-8.

```
SAMPLE.HASH: s<eol>

h := length(s); i := h MIN 4; }j := 1;
SAMPLE.HASH: s<eol>

}WHILE i .- 1 GEQ 0 DOB j .+ 2; h .+ cRead(s) * j END;
SAMPLE.HASH: v h,i,j,s<eol>
h = 1
i = 1
j = 1
s = "A"

SAMPLE.HASH:
```

Example 16-7. Single Stepping and Examining Variables

```
SAMPLE.HASH: c<eol>
65

Next key to be hashed (eol to stop): xyz<eol>

}h := length(s); i := h MIN 4; j := 1;
SAMPLE.HASH: s<eol>

h := length(s); }i := h MIN 4; j := 1;
SAMPLE.HASH: v h<eol>
h = 3

SAMPLE.HASH: c<eol>
119

Next key to be hashed (eol to stop): <eol>

Debug: q<eol>
Want to return from debugger (Yes or No): y<eol>
*
```

Example 16-8. Continuing and Quitting

The first <eol> terminates execution of SAMPLE. Control is now returned to the debugger, and the "Q" (Quit) command is given. The debugger's "Q" command requires confirmation to guard against accidental exit from the debugger.

A complete MAINDEBUG sample session without commentary appears in Example 16-9. Instead of using the "D" command eight times to position to the right line, a count has been given. "8D" tells the debugger to perform the "D" command eight times. Similarly, a count given before an "S" command means to step that many times; certain other debugger commands also take counts. More complete descriptions of the debugger command syntax appear in Chapter 14.

In Example 16-9, the "E" command is followed by the name of the module to execute. This form of the "E" command is more convenient when no subcommands need to be given with the name of the module invoked. It avoids the recursive invocation of the MAINSAIL executive which occurs when the "E" command is used without a module name.

```
MAINSAIL (R) Version 12.10 (? for help)
Copyright (c) 1984, 1985, 1986, 1987, 1988, and 1989 by
 XIDAK, Inc., Menlo Park, California, USA.
*compil<eol>

MAINSAIL (R) Compiler
Copyright (c) 1984, 1985, 1986, 1987, 1988, and 1989 by
 XIDAK, Inc., Menlo Park, California, USA.

compile (? for help): sample,<eol>
> debug<eol>
> <eol>
Opening intmod for $SYS...

sample 1
Objmod for SAMPLE stored on sample-<os>.obj
Intmod for SAMPLE stored on sample-<os>.int

compile (? for help): <eol>
*debug<eol>
```

Example 16-9. Line-Interface MAINDEBUG Session (continued)

```
MAINDEBUG (tm) (type ? for help)
Copyright (c) 1984, 1985, 1986, 1987, 1988, and 1989 by
 XIDAK, Inc., Menlo Park, California, USA.

Opening intmod for SAMPLE from sample-<os>.int
Opening intmod for $SYS...

MAINDEBUG: m sample<eol>

}BEGIN "sample"
SAMPLE: 8d<eol>

}h := length(s); i := h MIN 4; j := 1;
SAMPLE: b<eol>
Break set at SAMPLE.11

SAMPLE: e sample<eol>

Size of hash bucket: 131<eol>
Next key to be hashed (eol to stop): A<eol>
}h := length(s); i := h MIN 4; j := 1;
SAMPLE.HASH: v s<eol>
s = "A"

SAMPLE.HASH: s<eol>
h := length(s); }i := h MIN 4; j := 1;

SAMPLE.HASH: s<eol>
h := length(s); i := h MIN 4; }j := 1;

SAMPLE.HASH: s<eol>

}WHILE i .- 1 GEQ 0 DOB j .+ 2; h .+ cRead(s) * j END;
SAMPLE.HASH: v h,i,j,s<eol>
h = 1
i = 1
j = 1
s = "A"
```

Example 16-9. Line-Interface MAINDEBUG Session (continued)

```
SAMPLE.HASH: c<eol>
65

Next key to be hashed (eol to stop): xyz<eol>

}h := length(s); i := h MIN 4; j := 1;
SAMPLE.HASH: s<eol>

h := length(s); }i := h MIN 4; j := 1;
SAMPLE.HASH: v h<eol>
h = 3

SAMPLE.HASH: c<eol>
119

Next key to be hashed (eol to stop): <eol>
MAINDEBUG: q<eol>
Want to return from debugger (Yes or No): y<eol>
*
```

Example 16-9. Line-Interface MAINDEBUG Session (end)

17. Display-Oriented Interface Sample Session

This chapter contains a sample debugging session using the MAINEDIT-based debugger interface. The module debugged in this session is SAMPLE, as in the line-oriented session of Chapter 16. Familiarity with MAINEDIT's front end MAINED is assumed; if you are not familiar with MAINED, consult the "MAINEDIT User's Guide". It is also recommended that you read Chapter 16. Of course, you may use a front other than MAINED if you use the proper commands to invoke and return to the debugger; see the "MAINEDIT User's Guide" for details.

It is not, of course, possible to show the response to individual editor keystrokes on a piece of paper. Nor is it possible to fit a "snapshot" of a standard-size terminal screen on the page. What this sample session shows are snapshots on a very small terminal screen (fifty-eight characters wide by eight lines high). The cursor is marked by an underline. The snapshots are of selected moments in the course of the debugging session. The commands typed by the user to get from one snapshot to the next are described in the text. To be sure, there is no substitute for actually doing something; it is suggested that you follow along by performing the steps described below on a terminal while reading the script.

To begin with, you must enter the editor by following the directions found in the appropriate operating-system-specific MAINSAIL user's guide. You must of course use the display module appropriate to the type of terminal you have. It is assumed here that you have already invoked MAINEDIT, with MAINED as the default front end, and that your screen presently contains a single window with a buffer called "FOO" in it, as in Example 17-1. The debugger uses the display interface if it is invoked from the editor (unless you have a special configuration bit set in the MAINSAIL bootstrap; see the description of CONF in the "MAINSAIL Utilities User's Guide").

It is assumed that you have compiled the module SAMPLE debuggable as shown in the line-interface sample session.

Now invoke the MAINSAIL debugger with the MAINED command "QE". The "QE" command writes the prompt "Module or file name:" on the message line at the top of the screen. Type "debug<eol>". The buffer CMDLOG is then created if it doesn't already exist. In this example it is assumed that CMDLOG does not exist prior to the "QE" command.

The debugger now prints out a herald and waits for input. See Example 17-2.

Notice that the buffer CMDLOG is now in escape mode, indicating that the debugger program is expecting input. Since the module SAMPLE has already been compiled debugged, you can

```
----P.1-----L.1-----C----FOO-------------------------------
:This is a buffer named foo.  It does not have very much :
:of interest in it, but it is a starting place for the   :
:display debugger example._                               :
E
E
E
```

Example 17-1. A Buffer Named "FOO"

```
Debug command (? for help)
----P.1-----L.1-----E----CMDLOG--------------------------
:                                                        :
:MAINDEBUG (tm) (type ? for help)                        :
:_                                                       :
----P.1-----L.1----------FOO-------------------------------
:This is a buffer named foo.  It does not have very much :
:of interest in it, but it is a starting place for the   :
```

Example 17-2. The Debugger, Starting Up

now give the debugger's "M" command. As soon as you type the "M" key, the debugger writes a prompt on the top line of the screen; see Example 17-3.

```
Module or file: _
----P.1-----L.1-----E----CMDLOG--------------------------
:                                                        :
:MAINDEBUG (tm) (type ? for help)                        :
:                                                        :
----P.1-----L.1----------FOO-------------------------------
:This is a buffer named foo.  It does not have very much :
:of interest in it, but it is a starting place for the   :
```

Example 17-3. The Debugger Prompt for the "M" Command

Were you to decide at this point that you did not want to debug a module after all, but rather go do something else, you could type the <abort> key (as described in the "MAINEDIT User's Guide"). The debugger would beep and return the cursor to its original position in CMDLOG. You could then hit the <ecm> key and give any number of editor commands. You could create files, edit a document, or perform any other operations permitted by MAINEDIT. The debugger would still be waiting in the background for you to give the MAINED "E" command. When you did so, you would again be talking to the debugger, and could type in any debugger command. The debugger would be in precisely the same state as when you left it; that is, it would have exactly the same context.

For the sake of practice, type "<abort><ecm>" right here. The mode character for CMDLOG becomes "C". Give the "QY" command to MAINED, which makes the current buffer fill the entire screen (that is, it gets rid of the window into buffer FOO). See Example 17-4.

```
Module or file:
----P.1-----L.1-----C----CMDLOG--------------------------
:                                                        :
:MAINDEBUG (tm)  (type ? for help)                       :
:_                                                       :
E
E
E
```

Example 17-4. The Screen after the "QY" Command

Now you type MAINED's "E" command, and you are talking to the debugger again (notice how the mode character changes from "C" to "E"). Once again type the debugger's "M" command, but this time type the module name in response to the prompt: "sample<eol>". See Example 17-5. The end-of-line after the file name echoes as a backslash ("\") character.

Now position the cursor to the line where you set the breakpoint, as in the line-oriented interface example. You cannot use the debugger's "D" command since the editing commands used by the line-oriented interface debugger are not available with the editor interface. If you type "D" to the debugger in the display interface it prompts:

```
nothing or P or 'V type id1,...':
```

meaning that you can type here <eol> to use the debugger's "D" command, "P" to use the "DP" command, or "V" to use the "DV" command. Using the "D" or "DP" command, however, does nothing but print out the message:

```
Module or file: sample\
----P.1-----L.1-----E----SAMPLE--------------------------
:BEGIN "sample"                                          :
:                                                        :
:INTEGER bucketSize;  # size of hash bucket              :
----P.1-----L.1----------CMDLOG--------------------------
:                                                        :
:MAINDEBUG (tm) (type ? for help)                        :
```

Example 17-5. After Giving the "M SAMPLE" Command

```
Not available in the display debugger
```

All cursor motion is performed by using the appropriate MAINED commands instead of the line-interface debugger commands. Therefore, in this case, type <ecm> to get back to MAINED's command mode, then use the down arrow key (or the "\" command) to go down to the line (eight lines below the current line) where you wish to set a break. You can, of course, use any other MAINED command to position to the desired place, including the "T", "G", and "W" commands.

On terminals with arrow keys, the arrow keys are usually defined by default in such a way that it is not necessary first to type <ecm> and use them in MAINED's command mode. They may be used while in escape mode; the debugger passes the arrow key codes automatically to the editor. The same is not true of other editor commands, e.g., the "T", "G", and "W" commands.

After going down eight lines, the screen looks as in Example 17-6.

```
Module or file: sample\
----P.1-----L.7-----C----SAMPLE--------------------------
:BEGIN                                                   :
:INTEGER h,i,j;                                          :
:h := length(s); i := h MIN 4; j := 1;                   :
----P.1-----L.1----------CMDLOG--------------------------
:                                                        :
:MAINDEBUG (tm) (type ? for help)                        :
```

Example 17-6. Positioned to the Breakpoint Place

Now to return to escape mode so that you are talking to the debugger instead of the editor, give MAINED's "E" command. If you forget to do this, and go on to the next step and type the "B" command, MAINED beeps instead of performing the "Break line" command. This is because the debugger has set the buffer containing SAMPLE to be read-only to prevent inadvertent modification of the buffer. If you change a buffer while it is being debugged, the positions that the debugger uses to keep track of source text are no longer correct for the module you are debugging, and the debugger gets lost. Therefore, do not modify a buffer during a debugging session if you intend to continue debugging a module with a source file in that buffer.

MAINED's "E" command ("E" is for "Escape") takes you back to the debugger. Now give the debugger's "B" command. When you type the "B" key, the debugger responds immediately by prompting:

```
{@mod.iUnit}[cond]:cmd,...:
```

In the usual case you just want to set an unconditional break at the current cursor position, which you do by typing <eol> to this prompt. The debugger then prints out the location where the break was set (see Example 17-7).

```
Break set at SAMPLE.11
----P.1-----L.7-----C----SAMPLE---------------------------
:BEGIN                                                    :
:INTEGER h,i,j;                                           :
:h := length(s); i := h MIN 4; j := 1;                    :
----P.1-----L.1----------CMDLOG---------------------------
:                                                         :
:MAINDEBUG (tm) (type ? for help)                         :
```

Example 17-7. The Break Is Set

Now that the breakpoint has been set in SAMPLE, you are ready to execute the module. To do this, use the debugger's "E" command. This command executes another module. If the module name is not given, it recursively invokes MAINSAIL. The "E" command prompts:

```
module or file name or <eol>:
```

to which you type just <eol>. The result of this maneuver is shown in Example 17-8.

You are now in insert mode, talking to the MAINSAIL executive. This executive was invoked from the debugger, which was invoked from the editor, which was invoked from yet another

```
module or file name or <eol>: \
----P.1-----L.7----------SAMPLE--------------------------
:BEGIN                                                    :
:INTEGER h,i,j;                                           :
:h := length(s); i := h MIN 4; j := 1;                    :
----P.1-----L.3-----I----CMDLOG--------------------------
:MAINSAIL (tm) version 11.99 (? for help)                 :
:*_                                                       :
```

Example 17-8. After the Debugger's "E" Command

executive in the dim and distant past. You can see why it is important to remember the program to which you are giving commands at any given time!

Now widen the CMDLOG window a little bit, so as to have some breathing room. Type "<ecm>Q+1Y", which increases the size of the current window by one line. Now type the "I" command to get back into insert mode, and then type "sample<eol>", which begins the execution of that module. See Example 17-9.

```
module or file name or <eol>: \
----P.1-----L.7----------SAMPLE--------------------------
:BEGIN                                                    :
:INTEGER h,i,j;                                           :
----P.1-----L.3-----I----CMDLOG--------------------------
:MAINSAIL (tm) version 11.99 (? for help)                 :
:*sample                                                  :
:Size of hash bucket: _                                   :
```

Example 17-9. Invoking SAMPLE

Next type in the number 131, then end-of-line, just as in the non-display debugger example. The program then prompts:

```
Next key to be hashed:
```

You can type in any string here. Try the string "Hello", then type end-of-line. What happens now is that the execution of SAMPLE reaches the breakpoint you set in the procedure "hash", since this is the first call to that procedure. The debugger positions the cursor to the place in the SAMPLE buffer where the break was set, and the resulting display is shown in Example 17-10.

```
Break at SAMPLE.HASH.11
----P.1-----L.7-----E----SAMPLE--------------------------
:INTEGER h,i,j;                                          :
:h := length(s); i := h MIN 4; j := 1;                   :
----P.1-----L.4----------CMDLOG---------------------------
:*sample                                                 :
:Size of hash bucket: 131                                :
:Next key to be hashed: Hello                            :
```

Example 17-10. At the First Breakpoint

When you reach the breakpoint, the mode character is automatically set to "E". This means you can now give debugger commands, since you are talking to the debugger again. Give the debugger's "S" (single Step) command. The cursor jumps from the beginning of the statement "h := length(s)" to the beginning of the statement "i := h MIN 4". Give the "S" command again, and the cursor jumps to the beginning of the next statement to be executed. On each step, the statement that was jumped over has just been executed.

If your screen is as small as the one in the examples, you do not have the next statement visible on the screen at this moment. Therefore, if you give the "S" command again, the screen scrolls in order to display the point at which execution stops. In general, when the debugger needs to position the cursor on a statement, it scrolls the screen, creates new windows on the screen, or even creates a new buffer automatically if necessary in order to position the cursor correctly.

Now you have given the "S" command three times since the breakpoint at the beginning of the procedure, and you are sitting on the WHILE-clause of the Iterative Statement (see Example 17-11).

At this point the procedure's three local variables have been initialized, so you can usefully examine their values. Type the debugger's "V" (Value) command. The debugger immediately prompts at the top of the screen:

```
e1,e2,...:
```

```
Break at SAMPLE.HASH.11
----P.1-----L.8-----E----SAMPLE--------------------------
:h := length(s); i := h MIN 4; j := 1;                    :
:WHILE i .- 1 GEQ 0 DOB j .+ 2; h .+ cRead(s) * j END;    :
----P.1-----L.4----------CMDLOG--------------------------
:*sample                                                  :
:Size of hash bucket: 131                                 :
:Next key to be hashed: Hello                             :
```

Example 17-11. At the Start of the WHILE-Clause

You may now give a list of MAINSAIL expressions (variable names in this case), and the debugger prints out the value of each variable. If the information to be printed out is more than one line long, it is printed at the end of the CMDLOG buffer; otherwise it is printed at the top of the screen. For example, suppose you just want to see the value of the variable "h". Then you respond "h<eol>" to the "V" command's prompt. The result appears in Example 17-12.

```
h = 5
----P.1-----L.8-----E----SAMPLE--------------------------
:h := length(s); i := h MIN 4; j := 1;                    :
:WHILE i .- 1 GEQ 0 DOB j .+ 2; h .+ cRead(s) * j END;    :
----P.1-----L.4----------CMDLOG--------------------------
:*sample                                                  :
:Size of hash bucket: 131                                 :
:Next key to be hashed: Hello                             :
```

Example 17-12. The Value of One Variable

If you want to see the values of "h", "i", "j", and "s", then you can type "h,i,j,s<eol>" in response to the "V" command prompt. The result is shown in Example 17-13.

Notice that the CMDLOG buffer has been scrolled to display the values of all the variables; in fact, the values of "h" and "i" have scrolled right off the screen. You can go back and look at these values simply by using MAINED commands to scroll backwards through the CMDLOG buffer. Type <ecm> to talk to MAINED again, then give MAINED's ".Y" (go to other window) command. You may use the up-arrow key (or MAINED's "^" command) or

```
e1,e2,...: h,i,j,s\
----P.1-----L.8-----E----SAMPLE--------------------------
:h := length(s); i := h MIN 4; j := 1;                    :
:WHILE i .- 1 GEQ 0 DOB j .+ 2; h .+ cRead(s) * j END;    :
----P.1-----L.9----------CMDLOG--------------------------
:j = 1                                                    :
:s = "Hello"                                              :
:                                                         :
```

Example 17-13. The Values of Several Variables

MAINED's "-W" command to move backwards through CMDLOG and see the values of the variables that were scrolled off the screen. After doing this the situation might look as in Example 17-14.

```
e1,e2,...: h,i,j,s\
----P.1-----L.8----------SAMPLE--------------------------
:h := length(s); i := h MIN 4; j := 1;                    :
:WHILE i .- 1 GEQ 0 DOB j .+ 2; h .+ cRead(s) * j END;    :
----P.1-----L.7-----C----CMDLOG--------------------------
:h = 5                                                    :
:i = 4                                                    :
:j = 1                                                    :
```

Example 17-14. After Scrolling Back through CMDLOG

You can now type the "E" command to get back to the debugger. The cursor does not have to be in the SAMPLE buffer, in which you were positioned when last talking to the debugger; in fact, the SAMPLE buffer need not even be visible. You can still give any debugger command that does not depend on the cursor position. If you give a command that needs to position the cursor in the SAMPLE buffer, then that buffer is made visible if necessary, and the cursor is positioned there automatically.

Now type the debugger's "C" command to continue the execution of SAMPLE. The result is shown in Example 17-15. The program has printed out "96" (the value of the hash code for the

string "Hello") and is prompting for the next string to hash. The "I" mode character indicates that the program is waiting for input in insert mode.

```
e1,e2,...: h,i,j,s\
----P.1-----L.8----------SAMPLE--------------------------
:h := length(s); i := h MIN 4; j := 1;                    :
:WHILE i .- 1 GEQ 0 DOB j .+ 2; h .+ cRead(s) * j END;    :
----P.1-----L.12----I----CMDLOG-------------------------
:96                                                       :
:                                                         :
:Next key to be hashed (eol to stop): _                   :
```

Example 17-15. After the Debugger's "C" Command

You could, of course, type in another string. You would break again at the breakpoint set earlier in the procedure HASH. You could then step some more, set or clear breakpoints, examine variables, or even execute other modules and then return to debugging SAMPLE. However, these projects are left to the initiative and imagination of the reader. You are now going to terminate execution of SAMPLE by typing <eol> to the program's prompt. See Example 17-16.

```
e1,e2,...: h,i,j,s\
----P.1-----L.8----------SAMPLE--------------------------
:h := length(s); i := h MIN 4; j := 1;                    :
:WHILE i .- 1 GEQ 0 DOB j .+ 2; h .+ cRead(s) * j END;    :
----P.1-----L.13----I----CMDLOG-------------------------
:                                                         :
:Next key to be hashed (eol to stop):                     :
:*_                                                       :
```

Example 17-16. Exiting from SAMPLE

You are returned to the MAINSAIL executive's asterisk prompt. This executive is the one created by the debugger. You could here type the name of another module, but instead type <eol> again and return to the debugger. See Example 17-17. Note how the mode letter changes from "I" to "E"; it is no longer possible to type program input into CMDLOG, since

there is no program waiting for input from CMDLOG. You can type only debugger commands, except that if you type <ecm> you may then type editor commands.

```
e1,e2,...: h,i,j,s\
----P.1-----L.8----------SAMPLE--------------------------
:h := length(s); i := h MIN 4; j := 1;                    :
:WHILE i .- 1 GEQ 0 DOB j .+ 2; h .+ cRead(s) * j END;    :
----P.1-----L.14----E----CMDLOG-------------------------
:Next key to be hashed (eol to stop):                     :
:*                                                        :
:_                                                        :
```

Example 17-17. Returning to the Debugger

Now give the debugger's "Q" (Quit) command. The debugger prompts:

```
Want to return from debugger (Yes/No<eol>):
```

Type "y<eol>". You may now continue whatever you were doing in the editing session before starting to debug SAMPLE, or you may exit from the editor altogether with the MAINED's "QF" command. The situation after exiting the debugger is shown in Example 17-18. Note how the mode character changes again from "E" to "C"; it is no longer possible to give debugger commands, since the debugger has exited. If you attempt to use MAINED's "E" command now, it complains "Nothing to escape to". Of course you may always restart the debugger by using the "QE" command as you did at the beginning of this example.

```
Want to return from debugger (Yes/No<eol>): Y\
----P.1-----L.8----------SAMPLE--------------------------
:h := length(s); i := h MIN 4; j := 1;                    :
:WHILE i .- 1 GEQ 0 DOB j .+ 2; h .+ cRead(s) * j END;    :
----P.1-----L.14----C----CMDLOG-------------------------
:Next key to be hashed (eol to stop):                     :
:*                                                        :
:_                                                        :
```

Example 17-18. After Exiting from the Debugger

It should be noted that it is possible to give the debugger's "Q" command BEFORE the module (in this case, a MAINSAIL executive) invoked by the debugger has finished execution, provided you are at a breakpoint. If this is done, the debugger prompts:

```
At a break in SAMPLE -- want to quit (Yes/No<eol>):
```

Responding affirmatively to this prompt results in the termination of the execution of SAMPLE and a return to control by the debugger.

If you give the "+Q" command, you exit the debugger unconditionally, i.e., with no prompting.

Alternatively, you may give MAINED's "QF" command during the debugger's execution at any time if you first get into command mode (the one with the "C" mode letter). If you give this command, the editor recognizes that the debugger is still running and prompts:

```
Invoked from debug: want to abort it (Yes No)?
```

If you answer affirmatively in this case, you are prompted to save altered buffers before returing to the operating system.

18. $debugExec

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

```
PROCEDURE   $debugExec  (OPTIONAL STRING cmdStr);
```

Table 18-1. $debugExec

A call to $debugExec invokes MAINDEBUG. If cmdStr is specified, the commands in cmdStr are executed before commands are read from "TTY". The commands in cmdStr are written as they would be typed to the line-oriented MAINDEBUG executive.

Appendix A. Debugger Restrictions

Expressions and statements interpreted by the debugger may not contain modules or classes passed as procedure arguments. An error occurs, for example, if the expression:

```
new(c)
```

where c is a class identifier, or:

```
bind(m)
```

where m is a module identifier, is used. Instead, to create a record of the same class as an existing pointer p, use:

```
$createRecord(p)
```

and use the string form of bind, e.g.:

```
bind("m")
```

These restrictions may be lifted in a future release.

Appendix B. The Module SAMPLE

```
BEGIN "sample"

INTEGER bucketSize;  # size of hash bucket

INTEGER PROCEDURE hash (STRING s);
BEGIN
INTEGER h,i,j;
h := length(s); i := h MIN 4; j := 1;
WHILE i .- 1 GEQ 0 DOB j .+ 2; h .+ cRead(s) * j END;
RETURN(h MOD bucketSize) END;


INITIAL PROCEDURE;
BEGIN
STRING s;
ttyWrite("Size of hash bucket: "); bucketSize := cvi(ttyRead);
DOB ttyWrite("Next key to be hashed (eol to stop): ");
    IF NOT s := ttyRead THEN DONE;
    ttyWrite(hash(s),eol & eol) END END;

END "sample"
```

Appendix C. Command Summary

C.1. General Command Summary

?	display command summary
{n}	repeat last debugger command with count n
Q	quit (exit the debugger)
/c<s>/	define char c to be <s>
n{-} =c	replace with string for macro char c
<ecm>	enter MAINEDIT command mode (in display interface)
@	change to display interface
-@	change back to line interface

Table C.1-1. General Command Summary

C.2. Debugging Command Summary

See Table C.2-1 for a summary of the debugging commands.

C.3. Editing Command Summary

See Table C.3-1 for a summary of the editing commands. With the exception of the "F" command, these commands are available only in the line interface.

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

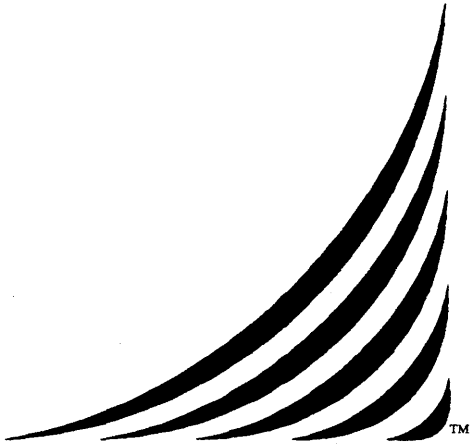
Table C.2-1. Debugging Command Summary (continued)

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

Table C.2-1. Debugging Command Summary (end)

```
    {n}D            down n lines, then first word on line
      F s           open source file s
{p}{.l}G            go to page p line l
   -{p}G            go backward p pages (then first line)
   +{p}G            go forward p pages (then first line)
 {-}{n}L            list n lines starting {before} at
                      current line
      P n           go to position n
      Q             quit (exit the program)
    {n}U            up n lines, then first word on line
    {n}W            window of size n about current line
    {n}<            left n chars on current line
    {n}(            left n words on current line
    {n}>            right n chars on current line
    {n})            right n words on current line
    {n}'c           search line forward for nth char c
   {n}-'c           search line backward for nth char c
    {n}""s""        search file forward for nth string s
   {n}-""s""        search file backward for nth string s
      /c.../        define char c to be ...
      =c            replace with string for macro char c
      @             enter MAINEDIT command mode
```

Table C.3-1. Editing Command Summary



MAINPM™

User's Guide

24 March 1989

19. Introduction

Medium- to large-scale software products are usually developed in several stages. All but the last stage are concerned with the software's correctness. The last stage involves improving the software's performance once the software appears to be correct. A typical program spends most of its time in a relatively small part of its code, but the program's author(s) may have little or no idea where that small amount of code is. Thus, they run the risk of making performance improvements in the "wrong" areas of the program, so that the improvements have little effect on the performance of the program as a whole. MAINPM helps focus attention on those parts of the program in which performance improvements would be most effective.

MAINPM reports to the user several aspects of a MAINSAIL program's performance:

- The amount of resource usage while a module or procedure is active ("deep usage"). Deep usage includes the resources used in invoked modules or procedures.
- The amount of resource usage spent in each module or procedure ("shallow usage"). Shallow usage does not include the resouces used in other modules or procedures.
- The number of statements executed in each module or procedure.
- The number of times each statement in a module is executed.

MAINPM can also list the procedures or statements that were not executed when a program was run.

Resouces used may be CPU time, wall clock time, chunk space usage, string space usage, or a user-defined quantity. CPU time information is available only on systems that provide access to a system clock. User-defined resouce measurement is described in detail in Chapter 23.

19.1. Version

This version of the "MAINPM User's Guide" is current as of Version 12.10 of MAINSAIL. It incorporates the "MAINPM Version 7.4 Release Note" of May, 1983; the "MAINPM Release Note, Version 8" of January, 1984; the "MAINPM Release Note, Version 9" of February, 1985; the "MAINPM Release Note, Version 10" of March, 1986; and the "MAINPM Release Note, Version 11" of July, 1987.

20. General Operation

A module of which the performance is to be monitored in some way may need to be specially compiled, depending on the kind of monitoring to be done. If the module is specially compiled, the resulting object module is larger and slower than its unmonitored counterpart.

To use MAINPM, first compile the modules you wish to monitor. If the kinds of monitoring you want require special subcommands, specify the kinds of monitoring you want done by means of the compiler subcommands listed in Table 20.1-1. After compiling all the modules to be monitored, execute MAINPM. Using commands to MAINPM:

- Specify what is to be monitored (necessary only if the default monitoring is unsatisfactory).

- Execute the program of which the modules are a part. As the program executes, the MAINSAIL runtime system gathers statistics about the monitored modules and writes the information to a file. When the program finishes, control reverts to MAINPM.

- Specify the statistics that you wish to see reported. MAINPM transforms the statistics into a more readable form and writes them to a text file.

More than one kind of monitoring may be done at a time. Different kinds of monitoring may be done for different modules during the same execution of a program.

MAINSAIL gathers statistics about all specially compiled modules that are active while MAINPM is executing a program.

20.1. Compiling Monitored Modules

A module to be monitored for statement counts or procedure or module CPU times must be specially compiled to include additional information in its object module. This is done by specifying one or more of the compiler subcommands listed in Table 20.1-1, according to the kinds of statistics you wish to have gathered when the module is executed. Each subcommand pertains to the current and subsequent modules compiled using that invocation of the compiler. See the "MAINSAIL Compiler User's Guide" for more information on the use of compiler subcommands.

PERMOD	Count the number of statements executed in this module.
PERPROC	Count the number of statements executed in each procedure in this module.
PERSTMT	Count the number of times each statement in this module is executed.
MODTIME	Measure this module's deep CPU time (time the module is active) and shallow CPU time (time spent in this module only).
PROCTIME	Measure each procedure's deep CPU time (time the procedure is active) and shallow CPU time (time spent in this procedure only).
MONITOR	Combine "PERMOD", "PERPROC", "PERSTMT", "MODTIME", and "PROCTIME".
NOMONITOR	Gather no statistics for this or later modules.

Table 20.1-1. Compiler Subcommands

20.2. Executing MAINPM

After you have compiled all the modules you wish to monitor, execute the module MAINPM to direct the gathering and reporting of statistics.

When you execute your program while running MAINPM, MAINSAIL gathers statistics and writes them to a data file created by MAINPM. The logical name of the statistics file is "(pm statistics file)". This logical name is ordinarily entered by means of a subcommand to the MAINSAIL executive in the configuration file (see the "MAINSAIL Utilities User's Guide"). If you wish to use a different file name from the usual one for the statistics file, specify it by means of the "ENTER" subcommand when you invoke MAINPM, as shown in Example 20.2-1.

```
*mainpm,<eol>
>enter (pm statistics file) foo.stat<eol>
><eol>
MAINPM (type ? for help)

MAINPM:
```

Example 20.2-1. Overriding the Statistics File Name

20.3. MAINPM Commands

After you invoke MAINPM, it prompts you for commands. Commands to MAINPM control:

- the entities to be monitored,
- the kinds of monitoring to be done,
- the invocation of modules,
- the kinds of statistics reported, and
- the level of detail to which the statistics are broken down (i.e., module, procedure, or statement).

Enter commands one per line, each followed by <eol>. Upper and lower case are not distinguished in commands. You need type only as many characters of a command as are needed to distinguish it from other commands.

Table 20.3-1 contains a quick summary of MAINPM commands.

20.3.1. Specifying What to Monitor

The basic types of monitoring available are:

- Statement counts: counting the number of statements executed and the number of times a given statement is executed. Statement count monitoring requires some

Command	Meaning
?	Display MAINPM commands
COUNTS	Report statement counts
EXECUTE m	Invoke module m (m may be a file name)
INFO	Display current MAINPM status
LIST f	Write statistics to report file f
LIST	Write statistics to current report file
MODULE	Break statistics down to module level
MONITORAREA	Undo previous {NO}MONITORAREA commands
MONITORAREA t	Monitor areas with title = t
MONITORAREA ?	Show non-null titles of current areas
MONITORLIB	Undo previous {NO}MONITORLIB commands
MONITORLIB f	Monitor modules from lib files f = f1 f2 ...
MONITORLIB ?	Show file names of currently open libraries
MONITORMODULE	Undo previous {NO}MONITORMODULES commands
MONITORMODULE m	Monitor modules m = m1 m2 ...
NOCOUNTS	Do not report statement counts
NOMONITORAREA t	Do not monitor areas with title = t
NOMONITORLIB f	Do not monitor modules from lib files f = f1 f2 ...
NOMONITORMODULE m	Do not monitor modules m = m1 m2 ...
NOTIMING	Do not report timing information
NOUNEXECUTED	Do not report unexecuted entities
PC {t}	PC monitor {with virtual/real time interval t}
PC VIRTUAL {t}	PC monitor {with virtual time interval t}
PC REAL {t}	PC monitor {with real time interval t}
PROCEDURE	Break statistics down to procedure level
QUIT	Exit MAINPM
SPACE {DEEP}	Monitor shallow {and deep} chunk and string space
STATEMENT	Break statistics down to statement level
TIMING x	Report timing information, kind of time = x
UNEXECUTED	Report unexecuted entities

Table 20.3-1. Summary of MAINPM Commands

compiler subcommands described in Table 20.1-1, but no MAINPM command need be given before the monitored program is executed.

- Module and procedure CPU times: counting the CPU usage of every specially compiled module and/or procedure. This kind of monitoring requires some compiler subcommands described in Table 20.1-1, but no MAINPM command need be given before the monitored program is executed.

- Module and procedure user-defined times: counting the incrementing and decrementing of a user-specified variable. This kind of monitoring requires some code in the program to manipulate the variable and some compiler subcommands described in Table 20.1-1, and the "TIMING USER" command needs to be given before the monitored program is executed; see Chapter 23.

- Chunk and string space monitoring: counting the use of chunk space (for records, arrays, and data sections) and string space. No compiler subcommands need be given for the monitored modules, but the MAINPM "SPACE" command must be given before the monitored program is executed; see Chapter 21.

- PC-sampled times: periodically interrupting the program and seeing which procedure is executing. No compiler subcommands need be given for the monitored modules, but the MAINPM "PC" command must be given before the monitored program is executed; see Chapter 22.

The types of monitoring are summarized in Table 20.3.1-1.

More than one kind of monitoring can take place on the same program execution. In the above list, only module and procedure CPU timing and module and procedure user-defined timing are mutually exclusive; either one can be performed simultaneously with any combination of the other three kinds of monitoring, depending on the compiler subcommands and MAINPM commands given.

For chunk and string space monitoring, the "MONITOR" and "NOMONITOR" commands provide additional control of what is to be monitored by allowing monitoring only of specified areas, modules, or objmod libraries. The "MONITOR" and "NOMONITOR" commands must be given before the monitored program is executed; see Chapter 21 for details.

20.3.2. Invoking Modules from MAINPM

Use the "EXECUTE" command to invoke a module. MAINPM uses the first word following the word "EXECUTE" as either the name of the module to be invoked or the name of the file containing the module; subsequent words are treated as arguments to the executed module.

Type of Monitoring	Compiler Subcommands	Pre-Execution MAINPM Commands	Other Requirements
statement counts	PERMOD PERPROC PERSTMT (MONITOR)	none	none
module CPU time	MODTIME (MONITOR)	none	can't monitor user time simultaneously
procedure CPU time	PROCTIME (MONITOR)	none	can't monitor user time simultaneously
module user time	MODTIME (MONITOR)	TIMING USER	special code in program
procedure user time	PROCTIME (MONITOR)	TIMING USER	special code in program
chunk and string space	none	SPACE {DEEP} (MONITORxxx) (NOMONITORxxx)	{NO}MONITOR only for extra control
PC-sampled time	none	PC {xxx}	none

Table 20.3.1-1. Types of MAINPM Monitoring

While that module is running, MAINSAIL gathers statistics about the execution of all modules that were compiled to be monitored. MAINPM also obtains the invoked module's total execution time, regardless of whether or not it was compiled to be monitored. The total execution time includes the time spent in other modules.

20.3.3. Specifying Kinds of Statistics

The commands in Table 20.3.3-1 control the kinds of statistics reported for statement count and module and procedure CPU and user-defined time monitoring. These commands can be given

either before or after the monitored program's execution; they do not affect the gathering of the statistics, just the way they are presented in the report.

Command	Meaning
COUNTS	Report statement counts.
NOCOUNTS	Do not report statement counts.
NOTIMING	Do not report timing information.
NOUNEXECUTED	Do not report unexecuted entities.
TIMING	Report timing information.
UNEXECUTED	Report unexecuted entities.

Table 20.3.3-1. Commands Controlling Kinds of Statistics

MAINPM reports timing information unless you specify the "NOTIMING" command. It reports statement counts unless you specify the "NOCOUNTS" command. It reports unexecuted procedures or statements only if you specify the "UNEXECUTED" command.

MAINPM reports unexecuted procedures or statements only for modules in which statement counts at the procedure or statement level were gathered, so if you want this information reported, be sure to specify the appropriate subcommand when you compile your modules (see Section 20.1).

20.3.4. Specifying Level of Detail

The commands in Table 20.3.4-1 control the level of detail to which the statistics are broken down. These commands can be given either before or after the monitored program's execution; they do not affect the gathering of the statistics, just the way they are presented in the report.

Command	Meaning
MODULE	Break statistics down to the module level.
PROCEDURE	Break statistics down to the procedure level.
STATEMENT	Break statistics down to the statement level.

Table 20.3.4-1. Commands Controlling Level of Detail

If none of these commands is specified, MAINPM breaks the statistics down to the lowest level for which statistics are available, based on the compiler subcommands with which the modules of the monitored program were compiled. Otherwise it breaks them down to the level corresponding to the last command specified. If no statistics of a given kind are available at that level for a given module, MAINPM uses the statistics at the lowest level available. In particular, since timing information is not available at the statement level, the lowest level to which timing information is ever broken down is the procedure level.

20.3.5. Writing to the Report File

Once the kinds of statistics and level of detail are specified to your liking, use the "LIST" command to generate the report.

If a parameter follows the word "LIST", MAINPM uses it as the name of the report file to which it writes the report. If there is already a report file, MAINPM closes it before creating the new one. The report file remains open until either a new one is specified in a subsequent "LIST" command or MAINPM terminates.

If no parameter follows the word "LIST", MAINPM writes the report to the report file currently open. If there is no open report file, MAINPM prompts you for the name of a file that it creates and subsequently uses as the current report file.

If you want to separate the report into two (or more) parts, then for each part:

- Specify the kinds of statistics and level of detail you wish to see.
- Generate that part of the report by means of the "LIST" command.

MAINPM remembers the kinds of statistics and level of detail to report from one part of a report to the next.

Example 20.3.5-1 illustrates a dialogue between a user and MAINPM that results in a report consisting of two parts. The first part shows each procedure's shallow and deep time, and the second part shows the number of statements executed in each procedure.

20.3.6. Displaying Current MAINPM Status

Use the "INFO" command to find out what would be reported if a report were generated right now. MAINPM displays the kinds of statistics that would be reported, the level of detail to which they would be reported, and the name of the report file.

```
MAINPM: procedure<eol>
MAINPM: nocounts<eol>
MAINPM: list report<eol>
MAINPM: counts<eol>
MAINPM: notiming<eol>
MAINPM: list<eol>
```

Example 20.3.5-1. Timing Followed by Statement Counts

20.3.7. Exit from MAINPM

Use the "QUIT" command to exit MAINPM when you're finished. If your previous command was not a "LIST" command, MAINPM asks you if you are sure that you want to quit. If your response is not "Y" (or "y"), it ignores the "QUIT" command.

20.4. Details on the Statistics File

This section describes exactly what data are written to the statistics file and when. Data for a monitored module m are written to the file:

- when m is disposed by means of the string or module form of dispose (but not the pointer form of dispose) during the execution of a module from MAINPM.
- when m is allocated and a report file is generated by the MAINPM "LIST" command.
- when m is allocated and MAINPM exits after running a module but without having generated a report file.

Once a module's statistics are written to the statistics file, the state of its statistics is reset to what it was when the module was first allocated. This means that if several modules are run in turn during a single MAINPM session, and a report is generated after each run, the statistics for any modules that were present for more than one run will reflect only their activities during the run associated with each report file.

A module's statistics reflect all instances of the module that may have been active. The statistics are not broken down across different data sections of the same module.

21. Monitoring Chunk and String Space

The MAINPM command that monitors chunk and string space usage is "SPACE". It must be given before the monitored program is executed. The "{NO}MONITOR" commands determine which modules and areas are monitored for chunk and string space usage.

If just "SPACE" is specified, shallow data are gathered; if "SPACE DEEP" is specified, shallow and deep data are gathered. There are two reasons to leave out "DEEP": 1) the program may run slower since additional work is required to gather deep statistics, and 2) the deep data may not be of interest.

The various forms of the "MONITOR" command are used to indicate which areas and modules are to be monitored, and have no effect unless "SPACE {DEEP}" is also specified.

The "{NO}MONITORAREA" commands are used to indicate which areas are to be monitored. Two lists are maintained: one of areas to be monitored (monitoredAreas) and one of areas not to be monitored (unmonitoredAreas). "MONITORAREA t" adds the area with title t to monitoredAreas, and removes it from unmonitoredAreas if present. "NOMONITORAREA t" adds the area with title t to unmonitoredAreas, and removes it from monitoredAreas if present. If an area is on monitoredAreas it is monitored; if it is on unmonitoredAreas it is not monitored; if it is on neither list it is monitored if monitoredAreas is empty. The command "MONITORAREA" (no title) clears both lists, in which case all areas are monitored (this is the default).

The "{NO}MONITORMODULE" commands are implemented similarly to the "{NO}MONITORAREA" commands in that two lists of module names are maintained, monitoredModules and unmonitoredModules; also, the "{NO}MONITORLIB" commands maintain two lists of library file names, monitoredLibraries and unmonitoredLibraries. If a module is on monitoredModules it is monitored; if it is on unmonitoredModules it is not monitored; if it is on neither list then if it is from a library in monitoredLibraries it is monitored; if it is from a library in unmonitoredLibraries it is not monitored; finally, if it is on none of the lists then it is monitored if unmonitoredLibraries is not empty or monitoredModules is empty. Logical file names and searchpaths are not applied to the file name argument to "{NO}MONITORLIB"; you must specify the actual file name.

The command "MONITORMODULE" (no module names) clears monitoredModules and unmonitoredModules. The command "MONITORLIB" (no library file names) clears monitoredLibraries and unmonitoredLibraries. If both lists are empty, all modules are monitored (this is the default). The user will almost certainly not want the MAINSAIL runtime modules monitored, and thus in the absence of any other "{NO}MONITORMODULE" or

"{NO}MONITORLIB" commands, the user should give a "NOMONITORLIB f" command where f is the name of the file containing the MAINSAIL system library. Use "MONITORLIB ?" to see the name of all currently open libraries, one of which is presumably the MAINSAIL system library.

The commands "{NO}MONITORAREA @f", "{NO}MONITORMODULE @f", and "{NO}MONITORLIB @f" open a file named f and read arguments, one per line, until a blank line or end-of-file is encountered. This allows a file of area titles, module names, or library names to be formed for use with MAINPM.

Whenever a chunk is allocated in a monitored area, the runtime system "walks" the procedure stack looking for the first caller in the call chain for a procedure in a monitored module (the procedures searched include only those in the current coroutine not suspended by an exception). If there is such a procedure, the size of the chunk is added to the shallow and deep chunk array elements for the procedure. If "SPACE DEEP" was specified, the stack walk continues until all monitored procedures in the call chain have been credited with the space usage (only the deep space is incremented). If the stack is large, e.g., due to many recursive calls, the deep data gathering may take much longer than the shallow data gathering; the overall effect on program execution time may or may not be noticeable.

The deep space for a procedure P thus includes all space allocated by calls in P. Shallow space for P includes all space allocated by calls in P as long as there is no other monitored procedure "between" the call and the actual space allocation. For example, if "SPACE DEEP" is in effect, and P calls "new(c)" to allocate a record of the class c in the default area, and the default area is monitored, then P's shallow and deep chunk space are credited with the size of the record. If P calls a procedure Q, and Q is in a monitored module, and then Q calls "new(c)", Q is credited with shallow and deep space, and P is credited with just deep space. If Q is not in a monitored module, then P is credited with both shallow and deep space.

Space can be credited to P's shallow space even if P does not directly cause the space to be allocated; it is only necessary that space be allocated and there be no other monitored procedure invoked (directly or indirectly) from P in the calling chain. Data allocated by the MAINSAIL runtime system is also credited to monitored callers. For example, if a procedure calls sin, and this call causes the allocation of the bound data section for the runtime module that contains sin (i.e., it is the first time a procedure has been called in that module), then any data allocated (including the bound data section) are credited to P. Since such behavior is somewhat unpredictable, it is not always possible to understand where all the space shown by MAINPM was allocated just by examining the user program, and there may be differences from one run to the next if the entire MAINSAIL session is not duplicated exactly.

The chunks and strings brought into memory by a call to $structureRead are included in the chunk and string space.

Some procedures in the MAINSAIL runtime system deallocate strings in a way that occasionally shows up as a negative value for string space.

If "SPACE" is specified, two listings of shallow chunk and string space are generated, one sorted by chunk space and the other by string space. If "SPACE DEEP" is specified, two listings are generated, one showing the shallow and deep chunk space sorted by shallow chunk space, and the other showing the shallow and deep string space sorted by shallow string space. Lines for which all space values are zero are omitted from the listings.

Sample output from the "SPACE" command is shown in Example 24.6-4. The two tables differ only by the column on which the data is sorted. As can be seen, the execution time is about 7% longer for "SPACE DEEP" than for just "SPACE". The totals at the end of each listing indicate the total chunk and string space allocated from the "EXECUTE" command given to MAINPM until the return to MAINPM. These totals are generally larger than the corresponding totals at the bottom of the columns since some allocation can occur when the monitored modules have no active procedures, and the allocation of the monitoring arrays are not credited to the shallow or deep time of any procedures.

22. PC Sampling

The MAINPM command "PC" is available on some systems and causes MAINPM to monitor the program with PC (program counter) sampling. The "PC" command has three forms:

```
PC {t}              PC monitor {with virtual/real time
                      interval t}
PC VIRTUAL {t}      PC monitor {with virtual time interval t}
PC REAL {t}         PC monitor {with real time interval t}
```

Only an unambiguous prefix of each keyword needs to be given; case is ignored.

The "PC" command must be given before the monitored program is executed.

In each form of the "PC" command, MAINPM causes an interrupt to occur every t microseconds (or as close to t microseconds as the operating system's timer permits). At each such interrupt MAINSAIL examines the PC to determine which procedure is currently being executed, and increments its PC count by 1. At the end of program execution the total PC counts for each procedure are printed out (omitting procedures with no sampled PC's). If the program runs sufficiently long with respect to the number of PC interrupts, the result is a profile of where the program is spending its time. Sampled PC's not in any MAINSAIL procedure (e.g., PC's in the MAINSAIL bootstrap or foreign or operating system code) are credited in the listing to "foreignPcs".

If t is omitted, 10000 is used by default; i.e., a PC interrupt occurs every 10000 microseconds, or 1/100th of a second. The actual frequency of PC interrupts is system-dependent, and the value of t is really just a hint; many systems may be able to interrupt far less frequently than every microsecond.

This timing option requires no compiler subcommands. The user program is slowed down, but not very much (in particular, the slowdown is much less than that caused by the "MODTIME" or "PROCTIME" compiler subcommands). Only "shallow" time is gathered; i.e., a procedure is credited with a PC only if the PC was actually in that procedure at a PC interrupt. Deep time is not implementable since that would require a traversal of the MAINSAIL stack, but the stack frames can be in an inconsistent state when a PC interrupt occurs.

22.1. "PC VIRTUAL {t}"

"PC VIRTUAL {t}" monitors PC's with the time interval t applied to "virtual" time; i.e., the timer interrupts occur every t microseconds of "virtual" time. The exact definition of virtual time is system-dependent, but it is roughly the time spent executing the process (CPU time, plus possibly some operating system overhead on behalf of the process), as opposed to real or wall clock time. In particular, where possible, it does not include time spent waiting on I/O or other external events since during such time the operating system is probably executing some other process.

22.2. "PC REAL {t}"

"PC REAL {t}" monitors PC's with the time interval t applied to "real" time; i.e., the timer interrupts occur every t microseconds of "real" time. Real time is meant to be "wall clock time", i.e., time as measured by a stopwatch. Typically the foreignPcs listing is higher for this option, since any time spent waiting on I/O is attributed to foreignPcs. For example, if a program prompts for user input, and the user waits a minute before responding, then one minute's worth of PC sampling is credited to foreignPcs (since the code that waits for terminal input is in the MAINSAIL bootstrap, which is considered a foreign PC).

This form of monitoring can help you see how much time your program spends waiting for external events such as I/O, though since all foreign PC's are summarized in one foreignPcs value, you cannot distinguish PC's sampled for different types of I/O.

22.3. "PC {t}"

"PC {t}" is the same as "PC VIRTUAL {t}" if virtual PC monitoring is available; otherwise, the same as "PC REAL {t}" if real PC monitoring is available; otherwise, an error message is given.

A given system may support one, both, or neither of the PC monitoring concepts. An error message is given if an unsupported PC monitoring concept is chosen.

23. User-Specified Resource Monitoring

The MAINPM command "TIMING" takes an optional argument that specifies what quantity is to be regarded as time. If no "TIMING" command is given, CPU time is the default. The available arguments are:

"TIMING" Argument	Effect
TIME	monitor CPU time (the default)
USER	monitor a user-specified time

This form of the "TIMING" command must be given before the monitored program is executed.

Only a unique prefix of an argument need be specified (at present the first letter is sufficient), and case does not matter. An invalid argument (e.g., "?") causes the list of valid arguments to be displayed.

"TIMING USER" indicates that the user program has defined a kind of resource that is to be utilized (called the "user time", although it can represent any resource, not just time). There must be a long integer variable that is updated by the program as appropriate to show the passage of user time. The MAINPM timing mechanism must have access to this variable throughout execution of the monitored program. The program tells MAINPM where the variable is by initializing two variables:

- $timerPtr, a pointer "base"
- $timerDspl, a long integer displacement from $timerPtr

The user time variable must be at the address given by "displace($timerPtr,$timerDspl)". $timerPtr and $timerDspl must be initialized before the monitored program starts executing (this means they cannot be initialized in a monitored module's initial procedure; a separate module must be invoked to initialize them before the monitoring of the program starts).

For example, the program can allocate a record with a long integer field, then initialize $timerPtr to point to the record and $timerDspl to be the displacement to that field:

```
CLASS timerClass (LONG INTEGER userTime);
...
$timerPtr := new(timerClass);
$timerDspl := cvli(DSP(timerClass.userTime));
```

To increment the time, the program does:

```
$timerPtr:timerClass.userTime .+ <amount to increment>;
```

The user can also decrement the time, to indicate that a resource has been "unconsumed". In this case, the MAINPM printout shows the net time (which may be negative) used by each procedure or module.

Both shallow and deep times are maintained. If the user time is updated in the body of a procedure (or module), then the shallow time (and deep time) of that procedure (or module) is affected. If the user time is updated by a procedure called inline, the effect is as if the user time were incremented in the caller.

23.1. User Time Example

The module FOO in Example 23.1-1 uses some resource of which the programmer wishes to keep track. consumeResource is called to allocate the resource and releaseResource to deallocate it.

When run from MAINPM, the module shown in Example 23.1-2 must be run before timing starts in order to initialize $timerPtr. See Example 23.1-3.

```
BEGIN "preFoo"

CLASS timerCls (LONG INTEGER userTime);

INITIAL PROCEDURE;
BEGIN
$timerPtr := new(timerCls);
$timerDspl := cvli(DSP(timerClass.userTime));
END;

END "preFoo"
```

Example 23.1-2. Module to Run before FOO

```
BEGIN "foo"

CLASS timerCls (LONG INTEGER userTime);

...

INLINE PROCEDURE consumeResource;
BEGIN
...
IF $timerPtr THEN $timerPtr:timerCls.userTime .+ 1L END;

INLINE PROCEDURE releaseResource;
BEGIN
...
IF $timerPtr THEN $timerPtr:timerCls.userTime .- 1L END;

...

END "foo"
```

Example 23.1-1. A User-Timed Module

```
MAINPM: execute prefoo<eol>
MAINPM: timing user<eol>
MAINPM: execute foo<eol>

... foo executes...

MAINPM: list foo.mpm<eol>
```

Example 23.1-3. Timing FOO

24. Report File

MAINPM creates report files consisting of five parts, all of which are optional.

The first part is a table containing statistics gathered about modules or procedures as a whole.

The second part is the program's total execution time. This part is omitted if CPU time is specified and the operating system does not provide access to a system clock.

The third part is a listing of the program's source text, with the number of times each statement was executed listed in the left margin of the line containing the start of that statement. This part of the report file is present only if you asked to see statement counts at the statement level, and at least one monitored module was compiled with the "PERSTMT" subcommand.

The fourth part of the report file is a table listing the procedures that were unexecuted. This part is present only if you asked to see unexecuted procedures or statements, and at least one monitored module was compiled with the "PERPROC" or "PERSTMT" subcommand.

The fifth part of the report file is a listing of all the unexecuted statements in the monitored modules of the program. This part is present only if you asked to see unexecuted statements, and at least one monitored module was compiled with the "PERSTMT" subcommand.

24.1. Counts Table

The counts table contains one row per monitored module or procedure. Each row consists of the module or procedure's name, its shallow and (if applicable) deep resource usage (if resource usage information is being reported), and the number of statements executed within the module or procedure (if statement counts are being reported).

The modules and procedures for which resource usage information is available are listed in decreasing order of their usage. Any remaining modules and procedures are listed in decreasing order of their statement counts.

Each procedure name is preceded by the name of the module in which the procedure occurs. A period separates the two names. Names are not truncated. If the module name, period, and procedure name would come too close to or overlap with the next field in the row, then the rest of the row appears on the following line.

In the case where the resource represents time, each shallow or deep time is reported both in seconds and as a percentage of the total execution time for the program. Where the resource is chunk or string space, both storage or character units and percentages are reported.

24.2. Total Execution Time

This part of the report file consists of the program's total execution time (in seconds, if CPU time). A program's total CPU time (as measured by MAINPM) includes the time it takes for MAINSAIL to load the program's first module into memory. This time is not included in either the shallow time or deep time of the first module's initial procedure, so the deep CPU time for that module or procedure is usually less than the program's total CPU time.

The total execution time also includes the time spent in unmonitored modules (of which the MAINSAIL runtime system is a particular example). This time is not included in any of the shallow times reported, so the sum of the shallow times for all monitored modules and procedures may be less than the total execution time.

24.3. Source Text with Statement Counts

This part of the report file contains the source text for monitored procedures, with statement counts listed in the left margin. Large portions of text (e.g., entire pages) that contain no statements are not listed.

The statement count in the left margin of a line indicates the number of times each statement on that line was executed. If a line of source text contains several statements that were not executed the same number of times, then the line is broken into several lines, each with its own statement count that applies only to the statements shown on that line, as shown in Example 24.3-1.

24.4. Unexecuted Procedures

This part of the report file is an alphabetized list of the monitored procedures that were not executed, one procedure per line. MAINPM is unaware of monitored modules that were not executed, so it cannot report as unexecuted the procedures in such modules.

```
Line in original source text:

n := 254; i := 0;
WHILE n DOB IF n MOD 2 THEN i .+ 1; n .DIV 2 END;

As shown in report file:

1 n := 254; i := 0;
1 WHILE n DOB
8               IF n MOD 2 THEN
7                                  i .+ 1;
8                                          n .DIV 2 END;
```

Example 24.3-1. Lines Containing More Than One Statement

24.5. Unexecuted Statements

This part of the report file is a listing of the unexecuted statements in the monitored modules of the program. MAINPM lists a portion of text surrounding each unexecuted statement (to establish context), and precedes each portion with the page and line number of the line containing the start of the unexecuted statement and the name of the procedure containing that statement. If a portion of text is in a different file from that of the previous portion, MAINPM lists the name of the file. If the portions that would be listed for two unexecuted statements overlap, MAINPM combines them into a single portion.

Each unexecuted statement is indicated by means of a "0" in the left margin of the line in which it is contained; there are no numbers in the left margins of the lines listed only to establish context.

24.6. Report File Examples

Example 24.6-1 shows the source text of a monitored module compiled with the "MONITOR" subcommand, Example 24.6-2 shows the timing and statement counts table generated by MAINPM after executing that module, and Example 24.6-3 shows the source text with statement counts for the monitored module. No special MAINPM commands were given to specify the type of monitoring to be done or statistics to be gathered.

```
BEGIN "number"

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

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

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

INITIAL PROCEDURE;
BEGIN
ttyWrite("10 factorial computed recursively is ",
    f(10),eol);
ttyWrite("10 factorial computed iteratively is ",
    iFactorial(10),eol);
ttyWrite("The 20th Fibonacci number is ",
    fibonacci(20),eol)
END;

END "number"
```

Example 24.6-1. Source Text of Monitored Module

Example 24.6-4 and Table 24.6-5 show outputs for a module compiled with no monitoring subcommands when the "SPACE" and "SPACE DEEP" MAINPM commands were given before execution, respectively.

```
NAME                    SHALLOW TIME      DEEP TIME      STMT COUNT
(mod or mod.proc)     (seconds)    (%)  (seconds)    (%)
-----------------     ---------    ---  ---------    ---  ----------
NUMBER.FIBONACCI         55.667  99.23     55.667  99.23       43782
NUMBER.F                   .050    .09       .050    .09          22
NUMBER.INITIALPROC         .033    .06     56.050  99.91           9
NUMBER.IFACTORIAL          .000    .00       .000    .00          12

Total execution time: 56.100 seconds
```

Example 24.6-2. Timing and Statement Counts Table

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

                  SOURCE FILE: number.msl
                  -----------------------

      BEGIN "number"

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

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

      LONG INTEGER PROCEDURE fibonacci (INTEGER n);
21891 IF n LEQ 1 THEN
10946                   RETURN(cvli(n))
10945 ELSE RETURN(fibonacci(n - 2) + fibonacci(n - 1));

      INITIAL PROCEDURE;
      BEGIN
    1 ttyWrite("10 factorial computed recursively is ",
          f(10),eol);
    1 ttyWrite("10 factorial computed iteratively is ",
          iFactorial(10),eol);
    1 ttyWrite("The 20th Fibonacci number is ",
          fibonacci(20),eol)
      END;

      END "number"
```

Example 24.6-3. Source Text with Statement Counts

(sorted by shallow chunks)

NAME (mod or mod.proc)	SHALLOW CHUNKS (strUnits)	(%)	SHALLOW STRINGS (chars)	(%)
MAINED.MOVECURSORTOPAGEANDLINE	248424	72.88	383952	95.75
MAINED.TRANSFERPAGES	11856	3.48	133	.03
MAINED.INITYOURSELF	3056	.90	1624	.40
MAINED.DELETELINESATADBELOW	936	.27	0	.00
MAINED.FORMYOURIMAGE	912	.27	626	.16
MAINED.DOLETTERF	448	.13	776	.19
MAINED.FORMSTATUSLINE	0	.00	426	.11
MAINED.DOPERIOD	0	.00	117	.03
MAINED.SETTABSTOPS	0	.00	22	.01
MAINED.EXECUTECOMMANDS	0	.00	18	.00
MAINED.SLASH	0	.00	10	.00
MAINED.OBJECT	0	.00	5	.00
	265632	77.93	387709	96.68

(sorted by shallow strings)

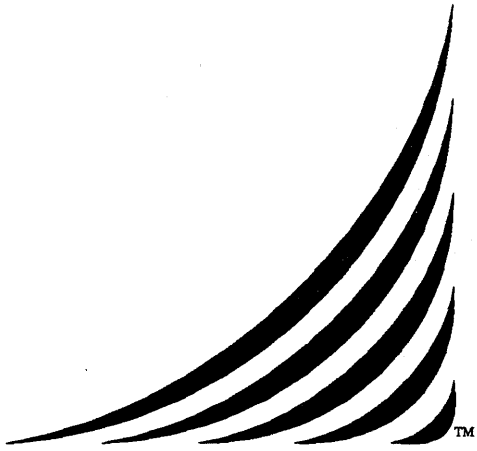
NAME (mod or mod.proc)	SHALLOW CHUNKS (strUnits)	(%)	SHALLOW STRINGS (chars)	(%)
MAINED.MOVECURSORTOPAGEANDLINE	248424	72.88	383952	95.75
MAINED.INITYOURSELF	3056	.90	1624	.40
MAINED.DOLETTERF	448	.13	776	.19
MAINED.FORMYOURIMAGE	912	.27	626	.16
MAINED.FORMSTATUSLINE	0	.00	426	.11
MAINED.TRANSFERPAGES	11856	3.48	133	.03
MAINED.DOPERIOD	0	.00	117	.03
MAINED.SETTABSTOPS	0	.00	22	.01
MAINED.EXECUTECOMMANDS	0	.00	18	.00
MAINED.SLASH	0	.00	10	.00
MAINED.OBJECT	0	.00	5	.00
MAINED.DELETELINESATANDBELOW	936	.27	0	.00
	265632	77.93	387709	96.68

Total execution time: 15.300 seconds
Total chunk space usage: 340856
Total string space usage: 401007

Example 24.6-4. "SPACE" Command Output

```
(sorted by shallow chunks)
NAME                                SHALLOW CHUNKS      DEEP CHUNKS
(mod or mod.proc)          (strUnits)    (%) (strUnits)    (%)
MAINED.MOVECURSORTOPAGEANDLINE
                                248424  72.88    248424  72.88
MAINED.TRANSFERPAGES             11856   3.48     11856   3.48
MAINED.INITYOURSELF               3056    .90      3056    .90
MAINED.DELETELINESATANDBELOW       936    .27       936    .27
MAINED.FORMYOURIMAGE               912    .27       912    .27
MAINED.DOLETTERF                   448    .13       448    .13
MAINED.EXECUTECOMMANDS               0    .00     14216   4.17
MAINED.LETTERD                       0    .00     12792   3.75
MAINED.DELETEPAGESATANDBELOW         0    .00     11856   3.48
MAINED.LETTERG                       0    .00        64    .02
                                265632  77.93

(sorted by shallow strings)
NAME                              SHALLOW STRINGS     DEEP STRINGS
(mod or mod.proc)                  (chars)    (%)     (chars)    (%)
MAINED.MOVECURSORTOPAGEANDLINE
                                  383954  95.75     383954  95.75
MAINED.INITYOURSELF                 1624    .40       1624    .40
MAINED.DOLETTERF                     776    .19        776    .19
MAINED.FORMYOURIMAGE                 626    .16       1052    .26
MAINED.FORMSTATUSLINE                426    .11        426    .11
MAINED.TRANSFERPAGES                 133    .03        133    .03
MAINED.DOPERIOD                      117    .03        117    .03
MAINED.SETTABSTOPS                    22    .01         22    .01
MAINED.EXECUTECOMMANDS                18    .00       2133    .53
MAINED.SLASH                          10    .00         10    .00
MAINED.OBJECT                          5    .00          5    .00
MAINED.LETTERD                         0    .00        138    .03
MAINED.DELETEPAGESATANDBELOW           0    .00        133    .03
MAINED.HORIZONTALTAB                   0    .00         22    .01
                                  387711  96.69

Total execution time: 16.380 seconds
Total chunk space usage: 340856
Total string space usage: 401002
```

Table 24.6-5. "SPACE DEEP" Command Output



MAINSAIL® Structure Blaster™

User's Guide

24 March 1989

25. Introduction

25.1. General Description

The MAINSAIL Structure Blaster is a set of procedures that manipulate entire MAINSAIL data structures as a unit. Structure Blaster procedures can read from or write to a file structures composed of records, arrays, and nonbound data sections with a single procedure call. Given a pointer to a record, array, or nonbound data section (the "root" of the structure), all records, arrays, nonbound data sections, and string text accessible from the root pointer may be written to a data file as a "data image", to a text file as a "text form" or "compressed text form", or to either a data or text file as a "PDF image" (Portable Data Format image). The image or text form can be read back into memory and the structure re-created. The Structure Blaster can also be used to dispose of a structure, make a copy of a structure, compare two structures, obtain information about a data or PDF image, or convert a data image into a text form or a text form into a data image.

It is the user's responsibility to port his or her data image files from operating system to operating system and MAINSAIL version to MAINSAIL version; see Section 26.4.

25.2. Version

This version of the "MAINSAIL Structure Blaster User's Guide" is current as of Version 12.10 of MAINSAIL. It incorporates the "MAINSAIL Structure Blaster Release Note, Version 9" of February, 1985; the "MAINSAIL Structure Blaster Release Note, Version 10" of March, 1986; and the "MAINSAIL Structure Blaster Release Note, Version 11" of July, 1987.

The Structure Blaster version is 8 as of MAINSAIL Version 12.10.

26. Structure Blaster Function and Performance

Emphasis has been placed on fast input and re-creation of structures from data images. The Structure Blaster can read a data image and re-create the structure significantly faster than if each object in the structure were allocated with the procedure "new" and the field values and elements individually read from a file. The data image is read into contiguous memory in one I/O operation if possible. It contains an exact image of the final structure, along with the string text. The image is quickly processed to make relative pointers and string descriptors absolute, and to create a "string space" from the stored string text (if the text is small, it is copied to an existing string space). On output the data image is written out as it is created rather than created in memory and then written out, so that there need not be space in memory for both the structure and the data image.

PDF images are processed somewhat more slowly than data images, but faster than text images. They are portable among the different operating systems on which MAINSAIL runs and among different versions of MAINSAIL; i.e., they can be written on one machine or release of MAINSAIL and written on another, unlike data images.

Text forms are processed more slowly than data or PDF images. Text forms, like PDF images, are portable among operating systems and different versions of MAINSAIL. The non-compressed text form is a human-readable format that can be used to view or modify a structure with an ordinary text editor.

When input/output performance is important, structures should be stored as data or PDF images rather than text forms; where compactness and portability among systems are required, PDF images are preferred. Table 26-1 ranks the different structure formats based on requirements for speed, portability, image size, and human readability or editability.

Compared to reading a host data image, our benchmarks show that reading a PDF structure ranges from as fast or faster to as much as 4 times slower, depending on the structure and on the machine. Typically, reading a PDF structure is less than a factor of 2 times slower on machines where the host format matches PDF, and less than 3.5 times slower on machines where the host format does not match PDF.

Our benchmarks also show that a PDF image is typically 1/2 to 3/4 the size of a data image, depending on the structure and on the machine.

Criterion	PDF	data	text	compressed text
portability	1	4	2	3
speed	2	1	4	3
image size	1 (*)	2	4	3
readability/ editability	N/A	N/A	1	3

1 = best, 4 = worst
* smallest

Table 26-1. Relative Advantages and Disadvantages of Different Structure Image Formats

26.1. Alignment and Positioning of Structure Images in Files

A data image is stored by default as an integral number of pages in a file to facilitate fast I/O; however, the user may specify non-page-alignment, which saves space if a file contains numerous structures smaller than a page. Input and output of page-aligned structures are usually faster than of non-page-aligned structures.

The user can specify the starting page (for page-aligned structures) or starting position (for non-page-aligned structures) of a structure image in a file, so that more than one structure image can reside in the same file.

PDF images and text forms are never specially page-aligned; they always start at the current or specified file position.

26.2. Portability Considerations in PDF Images

Although the PDF image format is portable, the actual values of the data must also be portable if the image is to be shared among different kinds of machines. PDF images can be written for

a data structure containing only portable values to a file on one machine and then read into memory on a different kind of machine.

A portable MAINSAIL data structure has the following characteristics:

- All (long) integer, (long) real and (long) bits data in the structure are in the MAINSAIL guaranteed portable range (refer to the description of data types in the "MAINSAIL Language Manual" for more information on guaranteed ranges).
- Strings consist only of characters that have an ASCII equivalent (see the PDF character set translation tables in the "MAINSAIL Language Manual").
- The structure does not contain any data sections.
- The structure does not contain addresses or charadrs.

No check for any of the above conditions is actually made when a PDF image is written.

All strings in the structure are translated between host characters and PDF (ASCII) characters when the structure is read or written as a PDF image. Hence, strings in a structure to be written in the PDF representation cannot be used to encode arbitrary binary values that should not be treated as translatable characters.

26.3. Opening Structure Blaster Files

Structure Blaster data files are usually opened with the $unBuffered bit set in the call to "open". This results in a substantial performance increase on many systems. Consult the description of the procedure "open" in the "MAINSAIL Language Manual" for details on the use of the $unBuffered bit. The $unBuffered bit must not be set for text files to which a structure is written. Structure Blaster data files must be opened with the random bit set in the call to "open"; however, the random bit need not be set for text files.

A file may opened for PDF I/O with the $pdf openBits bit. All structures written to such a file are PDF images. Alternatively, $pdf may be specified to $structureWrite to cause a PDF image to be written to a file not opened with the $pdf bit ($structureRead automatically figures out whether an image is a PDF image rather than a data image based on information stored in the image).

If a structure is to be written to a file (data or text), the "output" bit must be set when the file is opened; if a structure is to be read from the file, the "input" bit must be set.

26.4. Portability of Data Images and Text Forms

Data images are not directly portable from one operating system to another. To move a data image from one operating system to another, it may be translated to a PDF or text form (or compressed text form for increased efficiency) on the host operating system, shipped to the target system, and translated on the target system back into a data image.

Data images may not be portable from one version of MAINSAIL to another, or from one version of the Structure Blaster to another. Compressed text forms are not guaranteed to be portable from one version of MAINSAIL to another (although as of Version 12.10 of MAINSAIL, all old text forms are readable by the current version of MAINSAIL), but are portable among systems running the same version of MAINSAIL and the Structure Blaster. If a data image created by an old version of the Structure Blaster must be used by a new version of the Structure Blaster, use the old version to translate the structure into a text form or PDF image, then use the new version to translate the text form or PDF image into a data image. The procedure $structureInfo may be used to determine the Structure Blaster version number of a structure.

Text forms that do not contain data sections are portable from one operating system to another and from one version of the Structure Blaster to another, provided that no data value in the structure exceeds the range of its data type on the target system.

Data images may change in size when translated from one operating system or version to another. Therefore, if a data image is to contain data representing or depending on file positions or structure sizes (e.g., the file position of some other data image), the user must write a utility to adjust such positions and sizes in data images as they are translated. It is recommended that information dependent on data image sizes not be stored in data images; an alternative is to maintain in each file a directory (which is not itself a data image) of data images in the file.

It is the user's responsibility to write any utility necessary to translate files containing data images from one operating system or version to another. $structureDataToText and $structureTextToData are primitives from which such a utility may be constructed.

Data sections (in data or PDF images or text forms) are not portable from one operating system to another or from one version of the corresponding object module to another; i.e., a data section should not be assumed to remain valid if the corresponding module is recompiled. This is because changes in other parts of MAINSAIL can affect what is stored in a data section. For example, if an interface procedure is moved from one module to another, the arrangement of implicit module pointers may be altered in the data sections of all modules that call the interface procedure, thereby invalidating all data sections stored in data or PDF images or text forms before the change.

The Structure Blaster attempts to detect data sections inconsistent with the corresponding object modules. However, this consistency check may fail. If an inconsistent data section is not detected, the effect is undefined.

In addition to the constraints imposed by the Structure Blaster itself, structures ported from one machine or version of MAINSAIL to another should not depend on values that are not meaningful one both machines or versions. In particular, a structure or the arrangement of structures in a file should not depend on values that may vary from machine to machine or version to version of MAINSAIL, including data type sizes, file name formats, and such system-dependent values as $pageSize, $charSet, and $maxInteger.

27. Text Forms

Text forms serve four main purposes:

1. A data image may be translated to a text form and shipped to another operating system (although if this is the only purpose of the text form, a PDF image should be used instead).

2. The values in a structure may be examined by a human reader (non-compressed text forms only).

3. A data image may be translated to a text form and converted to be acceptable to another version of the Structure Blaster (a PDF image is also satisfactory for this purpose).

4. A data structure may be created or edited by hand; the text form serves as a language for specifying MAINSAIL data structures (non-compressed text forms only).

Except as noted below, descriptions of the contents of a text form apply only to non-compressed text forms. Compressed text forms, although portable among operating systems, are not intended for human readers, and their format is subject to change.

A detailed understanding of text forms is necessary only if they are to be edited by a human user or parsed by a program; otherwise, the contents of a text form should be self-explanatory.

Except within string constants, the case of alphabetic characters in a text form is irrelevant. The hash mark ("#") may be used as in MAINSAIL source text to indicate that the remainder of a line in a text form is to be ignored.

A text form consists of a sequence of "attributes" followed by a sequence of "units".

27.1. Constants

Constants are written in text forms as if they appeared in MAINSAIL source, except:

- The trailing "L" need not be present for long bits, long integer, and long real values.

- String variables are written as a sequence of MAINSAIL string constants, one per line, which are automatically concatenated when the constant is read. Each constant can be preceeded by any amount of white space. A double quote in the string constant may be specified as two double quotes, as in MAINSAIL source text, or with a "\" (backslash) escape sequence. "\" escape sequences are shown in Table 27.1-1. The empty string is "".

The "\" is used in string constants in a text form as an escape character according to the following table:

Escape Sequence	Corresponding String
\d \dd \ddd	the character with code d, dd, or ddd, where each d is a decimal digit. If the character after this escape sequence is a decimal digit, then the \ddd form must be used (with leading zero characters if necessary)
\n	eol (end-of-line)
\p	eop (end-of-page)
\t	tab (horizontal tab)
\"	double quote (same as doubled double quote)
\\	backslash ("\")

"\" escape sequences are not used in string constants in MAINSAIL source text.

Table 27.1-1. "\" Escape Sequences in Text Form String Constants

- An address constant can be written either as "NULLADDRESS" or as a long bits constant that represents the value of the address.

- A charadr constant can be written either as "NULLCHARADR" or as a long bits constant that represents the base address of the charadr, optionally followed by a

string of the form "+ n character{s}", where n is the number of characters past the base address, e.g.:

```
'HD6D40
'HD6D40 + 1 character
'HD6D40 + 2 characters
```

- A pointer or array constant can be written either as "NULLPOINTER" or as the internal name of the referenced chunk unit.

27.2. Attributes

An attribute is a line in a text form that describes the contents of the text form. Attributes may be separated from one another and from the first unit by one or more blank lines.

There is currently just one valid attribute. It has the form:

```
systemWrittenOn <operating system name abbreviation>
```

A list of possible operating system name abbreviations may be found in the "MAINSAIL Language Manual". The "systemWrittenOn" attribute identifies the operating system on which the text form was written, which is used when a text form written on one system is read on another (e.g., to translate escape sequences in string constants). If this attribute is not present, it is assumed that the text form was written on the current host system.

27.3. Units

A unit is a series of lines in a text form representing an object stored in the text form. A unit is either a "class unit" or a "chunk unit". A class unit declares a class, and a chunk unit gives the values for a record ("record unit"), array ("array unit"), vector ("vector unit"), or data section ("dataSec unit"). Units are separated by blank lines, and hence may not themselves contain blank lines. A text form is terminated either by a page mark or by end-of-file. If several text forms are to be stored in the same file, it is the user's responsibility to write the separating page marks into the file (e.g., by means of "write(f,eop)"). The first chunk unit in the text form is assumed to be the root of the data structure.

Each unit starts with a "header line" that "declares" the unit, followed by a "dash line", which is a row of hyphens to underline the header line. Typically the dash line is the same length as the header line, but this is not necessary. The dash line must be present.

Each header line starts with an "internal name" for the unit, followed by a colon and a space. Internal names consist of a valid MAINSAIL identifier or integer constant preceded by the charcter "<" and followed by the character ">". For example, "<123>" and "<foo>" are valid internal names. Internal names give a unique name to each unit so that units may be referenced unambiguously within the text form.

27.3.1. Class Units

All class units must occur before any chunk units. The general form of a class unit is:

```
<internalName>: CLASS className
-------------------------------
... field declarations...
```

className is the name of the class. The field declarations are specified one per line. Each has the form:

```
dataType fieldName
```

where dataType is a data type name, e.g., "BOOLEAN" or "LONG INTEGER", and fieldName is the name of the field. Address and pointer fields cannot be classified. An array field is declared as a pointer.

An example of a class unit is shown in Example 27.3.1-1.

27.3.1.1. Obsolete Class Units

Text forms generated from data images created by Structure Blaster Versions 1, 2, and 3 employ an alternate header line for classes that describe the data interface fields of a module:

```
interface CLASS for MODULE moduleName
-------------------------------------
```

In this case no internal name is used. The user should not create such class units since they may not be supported in future releases.

27.3.2. Record Units

A record unit specifies the values for the fields of a record, and has the form:

A class unit corresponding to a class declared as:

```
CLASS strListCls (
    STRING val,key;
    POINTER(strListCls) next;
);
```

would look like:

```
<51>: CLASS strlistcls
----------------------
STRING val
STRING key
POINTER next
```

Example 27.3.1-1. A Sample Class Unit

```
<recordInternalName>: record of CLASS <classInternalName>
-----------------------------------------------------------
... field specifications...
```

<classInternalName> is the internal name of the class that describes the record fields. Field specifications are specified one per line, except that string constants may occupy more than one line. Each specification has the form:

```
fieldName = constantValue
```

constantValue must be of the same data type as declared for fieldName. An example of a record unit is shown in Example 27.3.2-1.

27.3.3. Array Units

An array unit specifies the values for the elements of an array, and has the format:

```
<internalName>: typeName {LONG} ARRAY(<bounds>) {aryName}
---------------------------------------------------------
... element specifications...
```

```
A record of the class shown in Example 27.3.1-1 with
field values "This is a string" for val, "THIS" for key,
and nullPointer for next would look like:

    <52>: record of CLASS <51> # strlistcls
    ---------------------------------------
    val = "This is a string"
    key = "THIS"
    next = NULLPOINTER
```

Example 27.3.2-1. A Sample Record Unit

"LONG" is optional, and is ignored in any case, since the internal format of long and short arrays is the same. The bounds are a series of (long) integer pairs, separated by the word "TO", as in MAINSAIL source text. aryName is the name of the array; it may be omitted since it is possible for an array name to be the null string. The element specifications are written one per line, except that string constants may use more than one line. Each specification has the form:

```
elementSpecifier {FOR n} = value
```

where the "FOR n" part is optional. elementSpecifier is the bracketed subscript part of a MAINSAIL subscripted variable, i.e., "[i]", "[i,j]", or "[i,j,k]" for a one-, two-, or three-dimensional array, respectively, where i, j, and k are (long) integer constants. The elementSpecifiers need not be in any particular order. If "FOR n" is present, then n elements are initialized to the value, starting at the one specified; otherwise, just one element is initialized. The elementSpecifiers are read and corresponding initializations performed from the first elementSpecifier to the last, so that an element receives the last specified value if initialized more than once. Uninitialized elements have Zero values.

An example of an array unit is shown in Example 27.3.3-1.

27.3.4. Vector Units

A vector is a data structure used internally by MAINSAIL. A user cannot declare or manipulate a vector. A vector unit has the form:

```
<internalName>: typeName VECTOR(0 TO n)
---------------------------------------
... element specifications...
```

```
An array unit for an array declared as:

    CHARADR ARRAY(*) cAry;

and allocated with:

    new(cAry,-2,40);

with elements 3 through 6 given non-Zero values
might look like:

    <88>: CHARADR ARRAY(-2 TO 40) cary
    ----------------------------------
    [-2] FOR 5 = NULLCHARADR
    [3] = 'HC4120
    [4] FOR 2 = 'HD48C0 + 1 character
    [6] = 'HACC48 + 1 character
    [7] FOR 34 = NULLCHARADR
```

Example 27.3.3-1. A Sample Array Unit

A vector unit appearing in a text form must not be altered.

27.3.5. DataSec Units

A dataSec unit specifies the values in a data section, and has the form:

```
<iName>: dataSection for MODULE modName {of CLASS <cName>}
-----------------------------------------------------------
... specifications for interface data fields...
... own data specifications...
```

"of CLASS <cName>" must be present if the module has any interface data fields, in which case "<cName>" is the internal name for the class that declares the fields.

A data section consists of the interface data fields followed by the own pointers, the own strings and then any other own variables. Own variables are the "outer" variables for the module (those declared outside of any procedure, but local to the module), variables declared local to a

procedure but qualified with "OWN", and some implicit, undeclared variables maintained by the MAINSAIL runtime system.

The specifications for the interface data fields have the form of the field specifications for a record.

The own data specifications have the form:

```
typeName {variableName} {FOR n} = value
```

variableName is not currently written out by the Structure Blaster since MAINSAIL does not know at runtime the names of the own variables. "FOR n" gives a repetition factor just as in an array unit (the repetition factor cannot be used if variableName is present). The own specifications must appear in the same order in which they occur in the data section; the user should not edit them.

An example of a dataSec unit is shown in Example 27.3.5-1.

27.4. Editing a Text Form

A text form can be created or edited, and then read into memory, as long as it is syntactically and semantically correct. Values can be changed, fields and elements can be added or deleted, and records, arrays, and data sections can be added or deleted. A text form can be created "from scratch", and then read into memory as a structure or translated into a data image.

Two kinds of discrepancies are permitted to facilitate editing and subsequent reading of existing text forms:

1. If a field name is used in a record or dataSec unit but is not declared in the corresponding class unit, it is assumed that the user has deleted the declaration for the field in the class unit, and hence the field specification is ignored (since the field does not exist in the record being initialized).

2. If a field name is declared in a class, but is not initialized in a referencing record or dataSec unit, it is assumed that the user has inserted the declaration for the field into the class unit. Such fields are initialized to Zero.

Thus, the user can add or delete class fields without having to update every referencing record or dataSec unit, thereby facilitating the editing of text forms.

A module WINMOD with an interface declaration like:

```
MODULE winMod (
    INTEGER curCharNum,charNumBefore;
    INTEGER windowNumLines,windowNumChars;
    INTEGER windowFirstCharNum;
    LONG INTEGER curLineNum,lineNumBefore,
        windowFirstLineNum;
    LONG BITS attributes;
    STRING bufferName,fileName;
    POINTER(lineManagerCls) lineMgr;
    POINTER(bufferManagerCls) nextBfrMgr,
        bfrMgrForWindowAbove,bfrMgrForWindowBelow;
);
```

might generate a dataSec unit like:

Example 27.3.5-1. A Sample DataSec Unit (continued)

```
<94>: dataSection for MODULE winmod of CLASS <93>
-------------------------------------------------
curcharnum = 1
charnumbefore = 0
windownumlines = 29
windownumchars = 72
windowfirstcharnum = 1
curlinenum = 12L
linenumbefore = 0L
windowfirstlinenum = 1L
attributes = 'H0L
buffername = "BUFFER ONE"
filename = ""
linemgr = <45>
nextbfrmgr = <66>
bfrmgrforwindowabove = NULLPOINTER
bfrmgrforwindowbelow = <66>
POINTER FOR 19 = NULLPOINTER
POINTER = <70>
POINTER = <92>
POINTER FOR 2 = NULLPOINTER
STRING = "Type <eol> to continue, A to abort, N for next,"
         " ? for help"
BOOLEAN = FALSE
INTEGER = 4
INTEGER FOR 2 = 0
INTEGER = 4
LONG INTEGER = 0L
LONG INTEGER FOR 2 = 1L
INTEGER = -1

WINMOD has 23 own pointers, one own string, and 9 other
own variables.  The names of the own variables are not
provided; some of these variables may be implicit
variables that have no names.
```

Example 27.3.5-1. A Sample DataSec Unit (end)

The fields of a record unit need not appear in the same order as the fields of the corresponding class unit. When the record is assembled in memory, the order of the memory fields is the order of the fields in the corresponding class unit.

If the "warning" bit is set in the ctrlBits argument of $structureTextToData or the textFile form of $structureRead, a warning message is given whenever the first kind of discrepancy occurs.

27.5. Compressed Text Forms

The procedures $structureWrite and $structureDataToText accept the ctrlBits bit $compressed, which indicates that a compressed text form is to be read or written. The use of the $compressed bit results in a text form that is more compact (and is processed faster) than a non-compressed text form, but is not human-readable. Compressed text forms are useful for structure files that must be transferred among different systems or MAINSAIL versions, but are not examined by human readers.

Compressed text forms are detected automatically on input.

28. Structure Blaster Procedures

28.1. $structureCompare

```
BOOLEAN
PROCEDURE   $structureCompare
                        (POINTER root1,root2;
                         OPTIONAL BITS ctrlBits);
```

Table 28.1-1. $structureCompare

$structureCompare can be used to compare two structures. It returns true if and only if root1 and root2 point to "equal" structures, where structures are recursively defined to be equal as follows:

- Two records are equal if they have the same class (class names and all field names are the same, and fields are in the same order and have the same data type) and if all corresponding fields are equal or point to equal structures (for non-array and non-pointer data types, x and y are equal if the MAINSAIL expression "x = y" is true).

- Two arrays are equal if the have the same data type, dimension, and bounds, and if all corresponding elements are equal or point to equal structures.

- $structureCompare of data sections is not currently fully implemented. $structureCompare will consider two data sections unequal if they are for different modules or any interface, outer, or own variables are different, but may also consider two data sections unequal when no information visible to the ordinary programmer is different. At present, do not include data sections in any structures to be compared with $structureCompare.

$structureCompare may need to allocate additional data structures to do the comparison. The only valid ctrlBits bit is errorOK, which suppresses error messages.

The results of comparing structures containing any MAINSAIL runtime data structures (e.g., file pointers or runtime data sections) are undefined.

28.2. $structureCopy

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

Table 28.2-1. $structureCopy

$structureCopy returns a copy of the structure to which root points. NullPointer is returned if and only if an error occurs. If area is specified, the chunks of the data structure are copied into area; otherwise, they are copied into $defaultArea.

The only valid ctrlBits bits are delete, errorOK, $charsInArea, and $shareStrings. If errorOK is set, error messages are suppressed. If delete is set, the structure pointed at by root is disposed after being copied, as if "$structureDispose(root)" were performed immediately after the call to $structureCopy. If $charsInArea is set and area is specified, string text is copied into area along with chunks; otherwise, string text is copied into $defaultArea. If $shareStrings is set, just one copy of each unique string is stored in the copied structure. It can take longer to copy a structure with $shareStrings set, but can save considerable space if many strings in the structure share the same text.

The structure to be copied must not contain any MAINSAIL runtime data structures (e.g., file pointers or runtime data sections), since copies of such structures may be invalid; their use has undefined effects.

28.3. $structureDataToText

```
BOOLEAN
PROCEDURE    $structureDataToText
                          (POINTER(dataFile) inFile;
                           POINTER(textFile)
                               outFile;
                           OPTIONAL LONG INTEGER
                               inputStartPageOrPos;
                           PRODUCES OPTIONAL
                               LONG INTEGER
                               inputNumPagesOrSize;
                           OPTIONAL BITS ctrlBits);
```

Table 28.3-1. $structureDataToText

$structureDataToText translates a data image in inFile to a text form in outFile, starting at the current position in outFile. The data image is located at inFile file page (if $nonPaged is not set in ctrlBits) or inFile file position (if $nonPaged is set in ctrlBits) inputStartPageOrPos. The size of the data image in pages (if $nonPaged is not set in ctrlBits) or storage units (if $nonPaged is set in ctrlBits) is returned in inputNumPagesOrSize. False is returned if and only if an error occurs.

The only valid ctrlBits bits are errorOK, $nonPaged, and $compressed. If errorOK is specified, error messages are suppressed when an error condition occurs. If $nonPaged is set, the structure is not page-aligned. $nonPaged must be set if and only if the structure was written with $nonPaged set; otherwise, the effects are undefined. If $compressed is set, a compressed text form is produced instead of a non-compressed text form.

28.4. $structureDispose

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

Table 28.4-1. $structureDispose

$structureDispose disposes of the structure to which root points. The root pointer is modified to be nullPointer.

All records, nonbound data sections, and arrays in the data structure are disposed. This is a convenient way of disposing of an arbitrarily linked data structure.

The only valid ctrlBits bit is errorOK. If specified, error messages are suppressed when an error condition occurs.

The effects of disposing a structure containing pointers to file records or other data structures used by the MAINSAIL runtime system are undefined. Use $structureDispose (or the delete option to $structureWrite) only if the structure to be disposed is known not to contain any pointers to MAINSAIL runtime data structures.

The results are undefined of calling $structureDispose on a structure containing a data section of which the final procedure disposes of any component of the data structure. The final procedure associated with the data section is executed during the call to $structureDispose; if some record, array, or data section disposed by the final procedure is also contained in the structure, then that record, array, or data section is effectively disposed twice. Any own variables contained in the data section (whether interface variables or not) are considered to be contained in the structure.

The results are undefined of calling $structureDispose on a structure containing a bound data section.

If a structure can be easily disposed by repeated calls to the procedure dispose (e.g., a singly-linked list), or if the entire structure is in a single area than can be disposed with $disposeArea, it is more efficient to do so than to call $structureDispose. Call $structureDispose only for structures that are difficult to traverse and in an area that is not to be disposed.

28.5. $structureInfo

```
CLASS $structureInfoCls (
    INTEGER          $imageType;
    LONG INTEGER     $version,
                     $date,
                     $time,
                     $numPagesOrSize;
);

POINTER($structureInfoCls)
PROCEDURE   $structureInfo
                          (POINTER(file) f;
                           OPTIONAL LONG INTEGER
                              startPageOrPos;
                           OPTIONAL BITS ctrlBits);
```

Table 28.5-1. $structureInfo

$structureInfo returns information about a data or PDF image in f. startPageOrPos indicates where the image starts in the file. It is interpreted as the number of the first page of the image, where 0L is the first page in the file, if the $nonPaged bit is not set in ctrlBits and the image in the file is a data image; otherwise, it is interpreted as a file position. NullPointer is returned if f is a text file and the image is not a PDF image or if f is a data file and the image is neither a PDF nor a data image.

$imageType is $dataImage if the image is a data image or $pdfImage if the image is a PDF image. $version is the version of the image in f. The version numbers for different image formats are not related to each other or to the MAINSAIL version number. $date and $time specify the date and time, respectively, at which the image was written. $numPagesOrSize is the image size (number of pages if the image is a paged data image, number of storage units if the image is a non-paged data image, or number of characters if the image is a PDF image).

errorOK and $nonPaged are valid in ctrlBits. If errorOK is set, error messages are suppressed. The $nonPaged bit should be set in ctrlBits if and only if it was set when the structure was written; otherwise the effects are undefined.

XIDAK reserves the right to add new fields to $structureInfoCls.

28.6. $structureRead

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

POINTER
PROCEDURE    $structureRead
                          (POINTER(textFile) f;
                           OPTIONAL BITS ctrlBits;
                           OPTIONAL POINTER($area) area);
```

Table 28.6-1. $structureRead

$structureRead reads a structure from f and returns a pointer to the structure. NullPointer is returned if and only if an error occurs. The image in a data file can be either a data image or a PDF image and the image in a text file can be either a text form or a PDF image.

In both the dataFile and textFile forms, $pdf is valid in ctrlBits. If this bit is set or if the file is open for PDF I/O, then $structureRead reads a PDF image from the file. If $pdf is not set in ctrlBits and if the file is not open for PDF I/O, $structureRead automatically determines the format of the image in the file and reads it accordingly. This automatic detection introduces some overhead during $structureRead. Also, the heuristic used to detect what type of image resides in the file is not foolproof and could give the wrong result. For these reasons, the $pdf bit should be specified if the image is known to be a PDF image.

If area is specified, the structure is allocated in the specified area; otherwise it is allocated in $defaultArea.

In the dataFile form, startPageOrPos indicates where the image starts in the file. It is interpreted as the number of the first page of the image, where 0L is the first page in the file, if

the $nonPaged bit is not set in ctrlBits and the image in the file is a data image; otherwise, it is interpreted as a file position. In the textFile form, the structure is assumed to start at the current file position.

In the dataFile form, numPagesOrSize is used if the image in the file is a data image. It is ignored if the image in the file is a PDF image. numPagesOrSize specifies the image size. It is interpreted as the number of storage units in the image if the $nonPaged bit is set in ctrlBits; otherwise, it is interpreted as the number of pages in the image. If numPagesOrSize is 0L, then $structureRead obtains this information from the image in the file. An error occurs if numPagesOrSize is non-zero and does not agree with the value stored in the image. It is never necessary to specify numPagesOrSize, since $structureRead can obtain the image size from the image in the file. However, $structureRead is more efficient if numPagesOrSize is specified, since it then does not have to read the first part of the image into memory just to obtain the image size.

In the dataFile form, actualNumPagesOrSize is the image size. It is the number of character units if the image is a PDF image, the number of storage units if the image is a data image and the $nonPaged bit is set in ctrlBits, or the number of pages if the image is a data image and the $nonPaged bit is not set in ctrlBits.

In the textFile form, the image is assumed to start at the current file position. A text image must be terminated by a page mark (eop character) or end-of-file. When $structureRead returns from reading a text image, the current file position is immediately after the terminating page mark or at end-of-file.

errorOK is valid in ctrlBits in both the dataFile and textFile forms. If errorOK is set, error messages are suppressed.

In the dataFile form, $nonPaged and $charsInArea are valid in ctrlBits. $nonPaged indicates that the structure is not page-aligned and should be set in ctrlBits if and only if it was set when the structure was written; otherwise the effects are undefined. If $charsInArea is set and area is specified, string text is put in the specified area; otherwise string text is put in $defaultArea.

In the textFile form, warning and keepNul are valid in ctrlBits. They are used if the image in the file is a text image and are ignored if the image is a PDF image. If warning is set, a warning is given for any fields encountered in a record unit but not declared in the corresponding class unit. The values specified for such fields are not used. If keepNul is set, character codes in string constants expressed as "\" followed by one or more digits are not translated to the host character set; instead they retain the decimal values specified (high-order bits are discarded if host characters have more bits than target characters). If keepNul is not set, character code translation is performed on "\"-digit escape sequences. Character code translation is always performed on characters not expressed as "\"-digit escape sequences; if translation is not desired, such characters must be expressed as "\"-digit escape sequences.

$compressed should not be set in ctrlBits since $structureRead can detect a compressed text image.

When a data section for a module m is read in with $structureRead, the Structure Blaster carries out the same logic as done for "new(m)", except that:

- The data section read in is used instead of creating a new data section.
- m's initial procedure, if any, is not invoked because it has already been invoked to initialize the data section when it was first created.
- m's own and outer variables are not reset to Zero; they have the same value as they did on the call to $structureWrite.

m gets linkage to and interface consistency checking is performed on any existing bound modules that it references.

The result of $structureRead is undefined if the structure in the file contains a MAINSAIL runtime data structure (e.g., a file pointer or runtime module data section).

28.7. $structureSetup

```
CLASS $strucInfo (
    LONG INTEGER    $totalPagesOrSize;
    INTEGER         $imageType);

POINTER($strucInfo)
PROCEDURE   $structureSetup
                        (POINTER root;
                         OPTIONAL BITS ctrlBits;
                         OPTIONAL POINTER(file) f);
```

Table 28.7-1. $structureSetup

$structureSetup does preliminary processing on the structure pointed to by root. It returns nullPointer if and only if an error occurs.

The most common use of $structureSetup is to determine the size of a data or PDF image. An application program may need this information, for example, to reserve space in a file for an image. In order to determine the image size, $structureSetup must process the entire structure. The information gathered while processing the structure is returned so that $structureWrite need not process the structure again. If the size of the image need not be known prior to writing the structure then $structureSetup need not be called ($structureWrite automatically calls $structureSetup if it is not provided with information about the structure).

$structureSetup does preliminary processing for either a data image or a PDF image. Preprocessing is done for a PDF image if the $pdf bit is set in ctrlBits or if f points to a file open for PDF I/O; otherwise, preprocessing is done for a data image.

errorOK and $nonPaged are valid in ctrlBits. If errorOK is specified, error messages are suppressed. $nonPaged is used only when setting up for a data image; it is ignored when setting up for a PDF image. If the $nonPaged bit is set in ctrlBits, preprocessing is done for a non-paged data image; if the $nonPaged bit is not set in ctrlBits, preprocessing is done for a paged data image. If the $nonPaged bit is set in ctrlBits, then it must also be specified in all subsequent calls to $structureWrite, $structureRead, and $structureInfo for root.

The class $strucInfo describes only part of the information returned by $structureSetup; the other information is useful only to $structureWrite or $structureUnSetup. The field $totalPagesOrSize indicates the image size. It is the number of character units for a PDF image, the number of storage units for a non-paged data image, or the number of pages for a paged data image. The result of a subsequent call to $structureWrite or $structureUnSetup is undefined if the information in the record returned by $structureSetup is altered.

The memory used to store the information returned by $structureSetup is automatically released during a subsequent call to $structureWrite (if passed to $structureWrite in the strucInfo argument). If $structureWrite is not subsequently called, then $structureUnSetup should be called to reclaim the memory.

28.8. $structureTextToData

```
BOOLEAN
PROCEDURE   $structureTextToData
                          (POINTER(textFile) inFile;
                           POINTER(dataFile)
                               outFile;
                           OPTIONAL LONG INTEGER
                              outputStartPageOrPos;
                           PRODUCES OPTIONAL
                               LONG INTEGER
                               outputNumPagesOrSize;
                           OPTIONAL BITS ctrlBits);
```

Table 28.8-1. $structureTextToData

$structureTextToData translates a text form in inFile to a data image in outFile. The text form is assumed to start at the current position in inFile, and its end is delimited by a page mark or end-of-file, whichever occurs first. The data image is written to outFile file page (if $nonPaged is not set in ctrlBits) or outFile file position (if $nonPaged is set in ctrlBits) outputStartPageOrPos. The size of the data image in pages (if $nonPaged is not set in ctrlBits) or storage units (if $nonPaged is set in ctrlBits) is returned in outputPagesOrSize. False is returned if and only if an error occurs.

Valid ctrlBits are errorOK, $nonPaged, warning, and keepNul. warning and keepNul have the same effect as for $structureTextRead. If errorOK is set, error messages are suppressed if an error condition occurs. If $nonPaged is set, the data image is not page-aligned in outFile. If set in a call to $structureTextToData, $nonPaged must be set in all subsequent calls to $structureRead and $structureInfo for the same structure.

28.9. $structureUnSetUp

```
PROCEDURE    $structureUnSetUp
                          (MODIFIES POINTER($strucInfo) p;
                           OPTIONAL BITS ctrlBits);
```

Table 28.9-1. $structureUnSetUp

$structureUnSetup disposes of p and all associated data structures and sets p to nullPointer. p must have been returned by a previous call to $structureSetup. $structureUnSetup has no effect if p is nullPointer.

$structureUnSetup should be called to reclaim memory if $structureSetup was called and the structure was not written to a file by a subsequent call to $structureWrite.

There are currently no valid ctrlBits bits.

28.10. $structureWrite

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

Table 28.10-1. $structureWrite (Generic) (continued)

```
BOOLEAN
PROCEDURE    $structureWrite
                          (POINTER(textFile) f;
                           POINTER root;
                           OPTIONAL BITS ctrlBits;
                           OPTIONAL POINTER($strucInfo)
                              strucInfo);
```

Table 28.10-1. $structureWrite (Generic) (end)

$structureWrite writes the structure pointed to by root to f. The format of the image written depends on the arguments. 0L is returned if and only if an error occurs.

For the dataFile form, the format of the image written to f is determined as follows:

- If strucInfo is specified, then the format of the image written is determined from the information in the record pointed to by strucInfo;
- otherwise, if the $pdf bit is set in ctrlBits or if f is open for PDF I/O, then a PDF image is written;
- otherwise, a data image is written.

For the textFile form, the format of the image written to f is determined as follows:

- If strucInfo is specified, if the $pdf bit is set in ctrlBits, or if f is open for PDF I/O, then a PDF image is written;
- otherwise, a text form is written.

f must be open for random access if a data image or a PDF image is written.

In the dataFile form, startPageOrPos indicates the starting position of the image in the file. It is interpreted as a page number, where 0L is the first page in the file, if the $nonPaged bit is not set in ctrlBits and a data image is being written to the file. Otherwise it is interpreted as a file position. In the textFile form, the structure is written beginning at the current file position.

If strucInfo is specified, it must be the value returned by a previous call to $structureSetup for root. If strucInfo is nullPointer, $structureWrite automatically calls $structureSetup to process

the structure. Before $structureWrite returns, it automatically calls $structureUnSetup to reclaim memory.

In the dataFile form, the value returned by $structureWrite indicates the size of the image written to the file. It is the number of character units if the image written is a PDF image, the number of storage units if the image written is a non-paged data image, and the number of pages if the image written is a paged data image. In the textFile form, the value returned is true if the image was successfully written and false if an error occurred.

errorOK, delete, and $pdf are valid in ctrlBits for both the textFile and the dataFile forms. If errorOK is set, error messages are suppressed. If delete is set, $structureWrite deletes the structure after it is written to the file. It may be more efficient to set the delete bit than to call $structureWrite followed by $structureDispose. If the $pdf bit is set in ctrlBits, then a PDF image is written to f.

In the dataFile form, $nonPaged and $shareStrings are valid in ctrlBits. Both the $nonPaged and $shareStrings bits are used when writing a data image; they are ignored when writing a PDF image (the $shareStrings bit may someday be implemented for PDF, but is not at present). A data image is page-aligned in the file unless the $nonPaged bit is set in ctrlBits. $nonPaged must be set in ctrlBits if it was specified in a previous call to $structureSetup for root. If $nonPaged is set in ctrlBits, then it must also be specified in all subsequent calls to $structureRead and $structureInfo for root. If the $shareStrings bit is set in ctrlBits then only one copy of the characters for each unique string is stored in the image. By unique string it is meant all strings whose lengths are different or whose characters differ; i.e., two strings with string descriptors pointing to two different memory locations are not unique if the strings are the same length and contain exactly the same characters. Setting the $shareStrings bit in ctrlBits is an optimization which may save space in the stored image but which may increase the time it takes to write the image.

In the textFile form, $compressed is valid in ctrlBits. The $compressed bit is used when writing a text form; it is ignored when writing a PDF image. If the $compressed bit is set in ctrlBits, then a compressed text form is written.

Accessible nonbound data sections (those created by a call to "new") are written out in their entirety, including own and outer variables.

Pointers that point to free chunks or bound data sections are written as nullPointer. Address and charadr values are written "as is"; such values are not portable, but this allows a program to write a structure that contains address or charadr values and then read it back into memory during the same MAINSAIL session.

The structure pointed to by root cannot contain any MAINSAIL runtime data structures, such as file pointers or runtime data sections. $structureWrite may not fail if root contains such data structures, but the results of a subsequent $structureRead are undefined.

28.10.1. Structures Written Only from Specified Areas

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

The Structure Blaster can write a structure such that only those parts of a structure in a specified area (or areas) are written. Pointers into unspecified areas are written out as nullPointer. This is useful, for example, if the user builds up (or reads in) a structure in a particular area, alters various fields including making some pointers point to chunks outside the area, and then wants to blast out the original chunks (with possibly altered field values).

This is accomplished by marking the areas of interest, as described below, and then specifying the $markedAreas bit in the ctrlBits parameter of the $structureWrite call. Marking the areas has no effect unless $markedAreas is set in the $structureWrite call.

An area pointed to by "POINTER($area) ap" is marked as follows:

```
ap.$areaAttr .IOR $markedArea;
```

Note that $markedArea is a long bits constant to be set in $areaAttr, whereas $markedAreas is a bits constant to be set in the ctrlBits parameter to $structureWrite.

Before setting $markedArea in the desired areas as indicated above, the user should first call the parameterless proceure $unmarkAllAreas to turn off the $markedArea bit in all existing areas, since this bit may still be set from previous operations (it is sometimes set by the MAINSAIL runtime system).

For example, suppose root points to a data structure that is primarily in the area pointed to by ap, and you want to write out just that part that is inside ap. Do as follows:

```
$unmarkAllAreas; myArea.$areaAttr .IOR $markedArea;
sp := NULLPOINTER; # MODIFIES POINTER($strucInfo) strucInfoPtr
$structureWrite(f,root,0L,sp,$markedAreas);
```

$structureWrite locates all chunks accessible from root, and then all those accessible from these "secondary" chunks, and so forth. If the $markedAreas bit is set in the ctrlBits argument, this locating does not follow into unmarked areas. Thus chunks are located only if they are

accessible from a path from root that lies entirely within marked areas. A chunk that is part of the structure when considering all areas, but which is referenced only from an unmarked area, is not be found; indeed, there would be no way to "hook up" such a chunk to that part of the structure that is only in marked areas.

29. Structure Blaster Examples

Suppose that the pointer p is the root of a structure that is to be stored in the data file f starting at the beginning of the file (page zero), and that f has previously been opened for random input and output. Example 29-1 shows the simplest way to write the structure out and read it in again.

```
POINTER(dataFile)    f;
POINTER(...)         p;
...
$structureWrite(f,p);
...
p := $structureRead(f);
```

Example 29-1. Writing and Reading a Structure

To avoid the extra page read required by $structureRead to get the size of the image, do as in Example 29-2.

```
LONG INTEGER numPages;
...
numPages := $structureWrite(f,p);
...
p := $structureRead(f,0L,numPages);
```

Example 29-2. Use of $structureRead NumPages Parameter

To get the size of the image, then write it out, do as in Example 29-3, where findStartPage is a user procedure that finds a big enough hole in f for the image.

```
LONG INTEGER         startPage,numPages;
POINTER($strucInfo) strucInfo;
...
strucInfo := $structureSetup(p);
startPage := findStartPage(f,strucInfo.$totalPagesOrSize);
numPages := $structureWrite(f,p,startPage,strucInfo);
...
p := $structureRead(f,startPage,numPages);
```

Example 29-3. Use of $structureSetup

30. Structure Blaster Utility Modules

30.1. STRTXT

The module STRTXT translates a data image in a file to a text file containing a text form, or vice versa. It prompts for all necessary information. See Example 30.1-1.

30.2. STRCHK

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

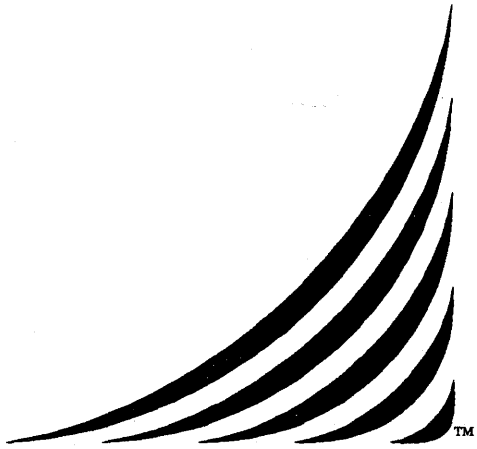
The utility STRCHK is used to print summary information about the contents of a Structure Blaster image stored in a file. It prompts for the name of the file and information about the position of the structure within the file. The output from STRCHK is self-explanatory.

```
If the file "foo.txt" contains:

    <1>: CLASS foo
    STRING s
    POINTER next

    <2>: record of class <1>
    s = "Hello, there."
    next = <3>

    <3>: record of class <1>
    s = "Bye now!"
    next = NULLPOINTER
    <end-of-file or end-of-page character>
```

Example 30.1-1. STRTXT Example (continued)

```
then a STRTXT session might look as follows on M68000
UNIX:

  *strtxt<eol>

  DT (data => text), TD (text => data), <eol>: td<eol>
  Input file: foo.txt<eol>
  Output file: struct.dat<eol>
  paged output (Yes or No): y<eol>
  Data image size = 512 storage units

  DT (data => text), TD (text => data), <eol>: dt<eol>
  Input file: struct.dat<eol>
  paged input (Yes or No): y<eol>
  startPage for input (just <eol> for 0): <eol>
  Output file: tty<eol>
  compressed output (Yes or No): n<eol>
  systemWrittenOn um68

  <120>: CLASS foo
  STRING s
  POINTER next

  <88>: record of CLASS <120> # foo
  s = "Hello, there."
  next = <168>

  <168>: record of CLASS <120> # foo
  s = "Bye now!"
  next = NULLPOINTER
  Data image size = 512 storage units

  DT (data => text), TD (text => data), <eol>: <eol>
```

Example 30.1-1. STRTXT Example (end)



MAINSAIL® Utilities User's Guide

24 March 1989

31. Introduction

This guide contains documentation for a number of MAINSAIL utility modules. These modules are included as part of the runtime system on every MAINSAIL release.

31.1. Version

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

31.2. Changes to Utility Programs

XIDAK reserves the right to change the syntax of commands and arguments to its utility programs and the format of output from these programs. An effort will be made to keep command and output formats reasonably compatible with previous versions; however, if such compatibility would prevent XIDAK from implementing an increase in functionality, command and/or output formats will be altered.

XIDAK may eliminate some utility programs if they are obsolete or if their functionality is subsumed by some other program.

XIDAK reserves the right to create programs and commands for XIDAK internal use only. Such programs and commands are not documented in this manual and are subject to change or removal without notice.

31.3. Command Line Arguments

XIDAK is experimenting with command line syntax and the format of the arguments is likely to change in future releases. Only those utilities documented as accepting command line arguments do so; command line arguments are ignored by other programs.

In all cases, if no arguments are specified, the utilities prompt for any information needed. If arguments are specified, the program executes with prompting, then exits, unless documented otherwise.

In description of command line syntax, arguments enclosed in curly brackets ("{" and "}") are optional. Optional arguments followed by an asterisk ("*") may be repeated zero or more times.

When arguments are specified to MAINEX, the comma used to enter subcommand mode must follow the module name, not the arguments. Thus, to enter subcommand mode following an invocation of DIR, do:

```
*dir, foo bar<eol>
```

not:

```
*dir foo bar,<eol>
```

32. CALLS, Call Chain Examiner

Command line syntax: calls {coroutineName}

The utility module CALLS prints out the call chain of the specified coroutine (the current coroutine if none specified). PRNTCO may be used to display the existing coroutines. The output is similar to that produced by the "CALLS" response to the "Error response:" prompt.

```
FRAMEHDR ADR MODULE  DECIMAL OFFSET  PROCEDURE
      'H7A87E CALLS             52     INITIALPROC
      'H7A8B6 KERMOD         52542     $NEWDATASEC
      'H7A8FC MAINEX         12616     $INVOKEMODULE
      'H7A922 EDIT           34014     ESCAPEFROMEDITOR
      'H7A942 MAINED          6750     DOLETTERE
      'H7A97C MAINED         19436     EXECUTECOMMANDS
      'H7A998 EDIT           14532     DOEXECUTECOMMANDS
      'H7A9D8 UCREX          42416     ECMD
      'H7AA0E UCREX         178922     $INVOKEPROC
      'H7AA32 CMDPRC           974     CALLPROCS
      'H7AA58 UCREX         107260     PROCESSCOMMAND
      'H7AA8A UCREX         107524     COMMANDLOOP
      'H7AAAE UCREX         108490     RUNUCREX
      'H7AABE UCREX         108542     INITIALPROC
      'H7AAF6 KERMOD         52542     $NEWDATASEC
      'H7AB3C MAINEX         12616     $INVOKEMODULE
      'H7AB70 MAINEX         12954     EXECUTEMODULE
      'H7ABA0 MAINEX         14056     $MAINSAILEXEC
      'H7ABB8 KERMOD         76634     RUNMAINSAIL
      'H7ABEC KERMOD         14848     INITIALPROC
```

Example 32-1: Sample CALLS Output

33. CLOSEF

Command line syntax: closef {fileName}*

On the command line, several file names (which may not contain spaces) may be specified. The named files are closed if they are open. If the file names are omitted, CLOSEF prompts for the name of a file to close. Responding with "?" causes it to ask, for each file on the open file list, whether or not the file should be closed. Typing "y" (or "Y") followed by <eol> causes it to close the file; any other response causes it to leave the file open.

Responding to CLOSEF's file name prompt with a non-blank string other than "?" causes it to close the files on the open file list with "name" fields equal to the response. On systems where file names are not case-sensitive, case distinctions are ignored. Then CLOSEF prompts for the name of another file to close.

CLOSEF should be used only as a last resort, when a file must be closed and there is no other way to close it. CLOSEF does no checking to prevent "important" files from being closed; e.g., it allows you to close "TTY" or the MAINSAIL system module library, though this usually leads to errors later on. Use care in closing files by means of CLOSEF, since in most cases the programs that opened the files do not expect the files to be closed suddenly.

The MAINSAIL runtime system normally closes all open files upon exit.

34. CONCHK, Memory Consistency Checker

The module CONCHK is a memory consistency checker. This utility checks the consistency of memory by going through steps similar to a garbage collection, but without actually freeing any storage. It prints an error message if it finds that memory is inconsistent; otherwise it prints "Memory is consistent".

Memory is inconsistent when a pointer points to non-existent data. There are several ways in which a user program can make memory inconsistent. If this has happened and a garbage collection occurs, the garbage collector prints a runtime error message indicating that there is a problem with memory.

The consistency checker can be used to isolate code that first makes memory inconsistent. To do this, use debugger count breaks (see the "MAINSAIL Debugger User's Guide") to do a binary search (re-run the program each time) to determine the count after which memory is first inconsistent (invoke the consistency checker each time with the debugger command "ECONCHK"). Once the procedure call is isolated, single step, each time invoking the consistency checker, to determine the statement that made memory inconsistent.

Some kinds of errors detected by the garbage collector may avoid detection by CONCHK.

35. CONF, the MAINSAIL Configurator

MAINSAIL gains control from the host operating system through execution of a "bootstrap". The bootstrap is an executable file that contains the code needed to start MAINSAIL execution and certain configuration parameters to be used for that execution. It is the only part of MAINSAIL that must be processed by the host operating system linkage editor.

The MAINSAIL configurator module, CONF, lets the user create custom bootstraps. It provides an interactive means of restoring configuration parameters from a file, altering the configuration parameters, saving the configuration parameters in a file, and writing a bootstrap file. The bootstrap file is usually an assembly language source file that must be assembled and linked or a link editor file that must be linked.

Custom bootstraps make it possible for each MAINSAIL execution to have a custom tailored execution environment. For example, the maximum amount of memory used by MAINSAIL is one configuration parameter that can be controlled. Custom bootstraps also make it possible to execute a module directly without entering a dialogue with the MAINSAIL executive, MAINEX.

The configuration is composed of a target-independent set of parameters and, on some operating systems, operating-system-dependent parameters. The target-independent configuration parameters are described herein; the other parameters are described in the appropriate operating-system-specific MAINSAIL user's guides.

35.1. Configuration Files

CONF initializes default values for all configuration parameters from a file called a "configuration file". A configuration file is a text file that contains commands to CONF. All commands in this file must appear as they would if interactively typed to CONF.

A systemwide configuration file (accessed by MAINSAIL with the logical name "cnf:xxx", where "xxx" is the platform abbreviation for the host system for which a bootstrap is built) that specifies default configuration parameters is supplied with MAINSAIL (the configuration file for the host system is always supplied; those for other systems only if the cross-compilers for those systems have been purchased). Systems staff may modify the default configuration file(s) to reflect site-specific information. The standard MAINSAIL bootstrap shipped with a MAINSAIL system causes values for configuration parameters to be restored automatically

from the host systemwide configuration file when CONF is invoked, or from the appropriate file for another system when the "PLATFORM" command is given.

Default parameters can be changed interactively with CONF commands or by modifying a configuration file and then restoring values from the file.

The interactive commands are convenient when changing only a few configuration parameter values. In this case, invoke CONF. Configuration values are automatically restored from the default configuration file; if that file cannot be opened, no initial "RESTORE" is performed. Use the appropriate CONF commands to alter the default configuration values, then save the bootstrap.

If there are many configuration parameter values to be changed, make a copy of the standard configuration file and modify it appropriately (see Section 35.2 for a description of the commands that appear in this file). Run CONF, "RESTORE" from the new configuration file, and save the bootstrap.

Example 35.1-1 shows how to restore configuration values from a configuration file. The "RESTORE" command causes new configuration values to be restored from the file "newconfig". The new bootstrap is written in the file "mainsa.asm", assuming "mainsa.asm" is the file name specified in the "BOOTFILENAME" command in the configuration file used to make the current bootstrap. This new configuration file can then be assembled and linked to create an executable bootstrap file. The details of assembling and linking a bootstrap are described in the appropriate operating-system-dependent documentation.

```
MAINSAIL (R) Version 12.10 (? for help)
Copyright (c) 1984, 1985, 1986, 1987, 1988, and 1989 by
 XIDAK, Inc., Menlo Park, California, USA.
*conf<eol>
MAINSAIL (R) Bootstrap Configurator
Restoring configuration values from file
    <systemwide configuration file>
CONF: RESTORE newConfig<eol>
Configuration values restored from file newConfig
CONF: <eol>
Bootstrap written in file mainsa.asm
```

Example 35.1-1. CONF Example

35.2. Configuration Commands

Table 35.2.1-1 shows the set of target-independent commands accepted by CONF.

35.2.1. Command Formats

Most CONF commands are entered on a single input line. The parameters "COMMANDSTRING", "SUBCOMMANDS", and "FOREIGNMODULES" may also have multiline values. These values are specified in commands by following the command with <eol>, then typing in the lines of the parameter value, terminated with a blank line. In order to specify a Zero value for a parameter that may have multiline values, use a pair of double quotes, e.g.:

```
COMMANDSTRING ""
SUBCOMMANDS ""
FOREIGNMODULES ""
```

If a line containing only an equals sign ("=") appears in a multiline CONF parameter value, then the current value of the CONF parameter is inserted at that point. For example, if the current value of the "FOREIGNMODULES" parameter is:

```
UNISYS
BSDITF
SUNGRF
```

and you issue the command:

```
FOREIGNMODULES<eol>
=<eol>
MYMOD<eol>
<eol>
```

then the "FOREIGNMODULES" value becomes:

```
UNISYS
BSDITF
SUNGRF
MYMOD
```

"=" can appear anywhere in the command and can occur any number of times.

For single-line conf commands, "=" is also accepted and means leave the current value, though this is less useful.

Command	Use
?	Print help information.
<eol>	Write bootstrap and quit.
BOOTFILENAME s	Make the bootstrap file name s.
COLLECTMEMORYPERCENT n	Perform memory management when predict n% of memory reclaimed.
COMMANDSTRING	Enter initial command string.
COMMANDSTRING<eol>	Enter multiline string; end with blank line.
CONFIGURATIONBITS b	Set the configuration bits to b.
FOREIGNMODULES s	Declare foreign module s.
FOREIGNMODULES<eol>	Enter multiple foreign modules, one per line; end with blank line.
INITIALSTATICPOOLSIZE n	Set initial size of static pool
KERMODNAME s	Set KERMOD's file name to s.
MAXMEMORYSIZE n	Set max storage units for MAINSAIL's use to n.
MINSIZETOALLOCATE n	Set min storage units allocated per os call.
OSMEMORYPOOLSIZE n	Set size of memory pool to be used by os.
PLATFORM s	Set target platform to s.
QUIT	Quit without writing a bootstrap.
RESTORE s	Restore configuration values from file s.
SAVE s	Save current configuration values in file s.
SHOW	Show current configuration values.
STACKSIZE n	Set size of MAINSAIL stacks.
SUBCOMMANDS s	Enter one-line initial subcommand.
SUBCOMMANDS<eol>	Enter multiline initial subcommands; end with blank line.
SYSTEMLIBNAME s	Set the system library file name to s.

Table 35.2.1-1. CONF Command Summary

35.2.2. <eol>

<eol> causes CONF to write a bootstrap file, using the current values of the configuration parameters. It then exits to the MAINSAIL executive or the host operating system so that the new bootstrap can be assembled and linked by the host operating system assembler and linkage editor.

35.2.3. "BOOTFILENAME s"

"BOOTFILENAME" sets the name of the bootstrap file to s. When CONF is exited with <eol>, the bootstrap is written to this file. Some operating systems may perform a transformation on the specified boot file name so that it conforms to operating system conventions for assembly language or object files.

35.2.4. "COLLECTMEMORYPERCENT n"

The parameter n is an integer between 1 and 99 inclusive. MAINSAIL decides to do memory management if it predicts that n percent of memory would be reclaimed by doing a garbage collection. Thus a higher value for "COLLECTMEMORYPERCENT", such as 10%, means that MAINSAIL collects garbage less often since more garbage has to build up before a garbage collection occurs. A smaller value, such as 1%, means more garbage collections since only 1% of memory is allowed to be garbage. The heuristic depends on an estimate that MAINSAIL makes of how much of memory is garbage; this estimate changes as garbage collections occur. MAINSAIL may garbage collect regardless of the value of "COLLECTMEMORYPERCENT" when it cannot get more memory from the operating system.

Large values for "COLLECTMEMORYPERCENT" may lead to unnecessary paging if a process's virtual size is larger than the physical memory available. Values for "COLLECTMEMORYPERCENT" larger than about 10% have not been shown to be useful. A value of about 1% may be reasonable for a large batch job.

35.2.5. "COMMANDSTRING"

"COMMANDSTRING" permits the user to enter a string that, when the resulting bootstrap is executed, is treated exactly as if it had been typed to the MAINSAIL executive. It can contain the name of a module to be executed, in which case the MAINSAIL executive dialogue is eliminated and the specified module is executed directly. Subcommands may be specified to the executive by ending the executive command line with a comma. Follow the module name

with a comma, list subcommands one per line, then terminate with an extra <eol>. See the description of MAINEX in this manual.

If the command string is a single command, it can be specified on the same line with the "COMMANDSTRING" command. If it consists of multiple lines it must start on the line immediately following the "COMMANDSTRING" line ("COMMANDSTRING" must appear alone on a line). In this case, the command string is terminated with a blank line (all lines up to the blank line are treated as part of the command string).

35.2.6. "CONFIGURATIONBITS b"

"CONFIGURATIONBITS" specifies that the bits set in the value "b" are the new configuration bits. Table 35.2.6-1 lists the possible configuration bits and their meanings.

Bit Position		
Octal	Hexadecimal	Name
'10	'H8	$foreignCodeStartsExecution
'100	'H40	$noAutoCmdFileSwitching
'200	'H80	$echoIfRedirected
'400	'H100	$echoCmdFile

Other bits are reserved and should not be set unless mentioned in the appropriate operating-system-specific MAINSAIL user's guide.

Table 35.2.6-1. CONF Configuration Bits

Bit '10 ('H8), $foreignCodeStartsExecution, is for use in conjunction with the Foreign Language Interface ("FLI"). It specifies that foreign language code will gain control before MAINSAIL, and that MAINSAIL is not to initialize itself until the foreign code makes a call into MAINSAIL through the FLI. See the "MAINSAIL Compiler User's Guide" for more information on the FLI.

Bit '100 ('H40), $noAutoCmdFileSwitching, indicates that the exception $cmdFileEofExcpt is not to be raised when end-of-file on cmdFile is reached (as described in the "MAINSAIL Language Manual"); instead, normal end-of-file indicators are returned.

Bit '200 ('H80), $echoIfRedirected, 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".

Bit '400 ('H100), $echoCmdFile, always echoes cmdFile to logFile.

Example 35.2.6-2 shows how to use the "CONFIGURATIONBITS" command. The value '100 specifies that $cmdFileEofExcpt is disabled, and '200 specifies that cmdFile and logFile are to be echoed to "TTY" when they are redirected to files other than "TTY". The new bootstrap is written to the file "mainsa.mac". The file "mainsa.mac" can then be assembled and linked to create the new executable bootstrap.

```
MAINSAIL (R) Version 12.10 (? for help)
Copyright (c) 1984, 1985, 1986, 1987, 1988, and 1989 by
 XIDAK, Inc., Menlo Park, California, USA.
*conf<eol>
MAINSAIL (R) Bootstrap Configurator
Restoring configuration values from file
    <systemwide configuration file>
CONF: BOOTFILENAME mainsa.mac<eol>
CONF: CONFIGURATIONBITS '300<eol>
CONF: <eol>
Bootstrap written in file mainsa.mac
```

Example 35.2.6-2. Using the CONF Command "CONFIGURATIONBITS"

35.2.7. "FOREIGNMODULES"

The names of foreign modules, as produced by the MAINSAIL foreign call FLI compiler, are listed one per line after this subcommand. The list of foreign modules is terminated by a blank line. MAINSAIL does not automatically know about foreign modules linked with the bootstrap; the name of a foreign module must appear in the list in order to be callable from MAINSAIL.

If there is only one foreign module in the list, it may be specified after the keyword "FOREIGNMODULES"; e.g., "FOREIGNMODULES FOO" may be used if FOO is the only foreign module.

The case of the foreign module names is automatically converted to the case used for labels generated by the FLI compiler.

35.2.8. "INITIALSTATICPOOLSIZE n"

The parameter to "INITIALSTATICPOOLSIZE" is the initial size (in storage units) of the static page pool (the pages from which static memory is allocated).

35.2.9. "KERMODNAME s"

"KERMODNAME" specifies that the file s contains the MAINSAIL kernel object module, KERMOD. This name must be changed if the kernel file is moved or renamed. The kernel module cannot reside in the system module library since it must be given control before any module library can be opened.

35.2.10. "MAXMEMORYSIZE n"

The parameter to "MAXMEMORYSIZE" is the largest number of storage units MAINSAIL is allowed to ask for from the operating system. On some systems this limit may be approximate.

35.2.11. "MINSIZETOALLOCATE n"

The parameter to "MINSIZETOALLOCATE" is the minimum number of storage units to request from the operating system whenever MAINSAIL needs more memory.

35.2.12. "OSMEMORYPOOLSIZE n"

The parameter to "OSMEMORYPOOLSIZE" is the number of storage units to reserve for allocation by system or FLI routines. The memory is requested from the OS, then released, when MAINSAIL initializes itself. The purpose is to avoid the appearance of static pages allocated by system or FLI routines in the middle of the MAINSAIL address space, acting as "firewalls" that prevent the efficient use of memory. This parameter has no effect on many operating systems; the operating-system-specific MAINSAIL documentation describes its effect on those systems where it is implemented.

35.2.13. "PLATFORM s"

"PLATFORM s" writes a bootstrap for the named target platform. s is an platform name abbreviation. "PLATFORM ?" shows a list of possible target platform name abbreviations. The default target is the host platform.

If "PLATFORM" is not the first command specified in a CONF session, previously specified configuration values may be lost.

If a platform is specified for which no cross-CONF module is available, MAINSAIL complains that it is unable to find a target-dependent CONF module. Cross-CONF modules are shipped with cross-compilers, which must be purchased separately from XIDAK.

35.2.14. "QUIT"

"QUIT" causes CONF to exit without writing a bootstrap file.

35.2.15. "RESTORE s"

"RESTORE" causes CONF to read commands from configuration file s. s must contain only valid CONF command lines. The configuration parameter values specified in s supersede the current values. All other parameters retain their previous values.

35.2.16. "SAVE s"

"SAVE" causes CONF to create a configuration file s and then save the current set of configuration parameters in it. The parameters are saved as commands to CONF, exactly as they are entered from cmdFile.

35.2.17. "SHOW"

"SHOW" causes CONF to display the current values of all configuration parameters. String parameters with null value are displayed as "".

35.2.18. "STACKSIZE n"

The "STACKSIZE" command specifies the number of storage units to allocate for the stack of each new coroutine. On some operating systems, the stack of the initial coroutine ("MAINSAIL") is not affected by the "STACKSIZE" parameter; if this is the case, the system-specific MAINSAIL documentation explains in further detail.

35.2.19. "SUBCOMMANDS"

"SUBCOMMANDS" is followed by a list of MAINEX subcommands that are executed prior to any commands or subcommands specified to the MAINEX "*" prompt (either typed in directly or specified in the "COMMANDSTRING" command). All valid MAINEX subcommands are allowed. Subcommands are given one per line, the last one followed by a blank line.

This command provides an alternative to "COMMANDSTRING" for entering subcommands. The subcommands specified in the standard MAINSAIL configuration and subcommands files are needed by MAINSAIL and must not be deleted. Subcommands can, however, be added to the "SUBCOMMANDS" section of the system configuration file.

35.2.20. "SYSTEMLIBNAME s"

"SYSTEMLIBNAME" specifies that the file s is the objmod library that contains the MAINSAIL runtime system modules. This name must be changed from the default if the system module library is moved or renamed.

35.2.21. System-Specific CONF Commands

Various operating systems provide target-specific commands. These commands include "CMSBITS" and "UNIXBITS". Consult the appropriate operating-system-specific MAINSAIL user's guide for descriptions of these commands.

36. COPIER, File Copier

Command line syntax: copier {srcFile {dstFile}}*

The module COPIER copies a text file. It prompts for the names of the input and output files, or uses srcFile as the input file name and dstFile as the output file name, if given. To exit, type <eol> in response to the input file prompt.

The most common use of COPIER is to convert between file formats. This is done by using the appropriate device module syntax in the file names specified. The actual device modules available depend on the host operating system. For example, under some operating systems, the device prefix "VAR>" indicates a variable-length-record-oriented text file and the device prefix "BS>" indicates a MAINSAIL byte stream text file. Example 36-1 copies a text file, changing its format from byte stream (device prefix "BS>") to record format (device prefix "VAR>"). User responses are underlined. COPIER can be used to convert between all text file formats for which device modules exist.

```
MAINSAIL (R) Version 12.10 (? for help)
Copyright (c) 1984, 1985, 1986, 1987, 1988, and 1989 by
 XIDAK, Inc., Menlo Park, California, USA.
*copier<eol>
Text File Copier
Input file: bs>test<eol>
Output file: var>test<eol>
Input file: <eol>
```

Example 36-1. COPIER Example

37. Coroutine Utilities

Command line syntax for KILLCO: killco {coroutineName}*

Command line syntax for RSMCO: rsmco {coroutineName}

Several utility modules are provided to interactively manipulate coroutines.

The utility PRNTCO prints the coroutine tree using an indented listing with the name of each coroutine in place of the node. All the children of a node are printed at the same indented level on the lines following their parent as shown in Example 37-1. Each child is recursively followed by its subtree.

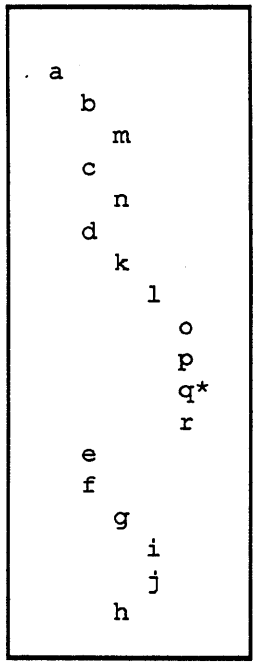
```
parent
  child[1]
  child[2]
  ...
  child[n]
```

Example 37-1. PRNTCO Parent and Children

PRNTCO writes an asterisk ("*") after the current coroutine ($thisCoroutine). For example, a coroutine tree might be printed as in Example 37-2, where "q" is the current coroutine.

The utility KILLCO prompts for coroutine names (or accepts their names on the command line) and kills each specified coroutine, so that the user can interactively prune the coroutine tree. If the coroutine names are specified on the command line, they are always killed recursively; otherwise, KILLCO prompts for whether to kill the named coroutines' children or to promote them in the coroutine tree (kill with the $nonRecursive bit set).

The utility RSMCO accepts a coroutine name on the command line or prompts for a coroutine name and resumes the specified coroutine.



Example 37-2. PRNTCO Output

38. DELFIL, File Deleter

Command line syntax: delfil {fileName}*

The module DELFIL allows the user to delete one or more files from within MAINSAIL. It prompts for the name of each file to be deleted, or accepts the file names on the command line. A message is written if a file cannot be deleted.

Example 38-1 shows how to use DELFIL to delete files. The files "foo", "bar", and "baz" are deleted.

```
MAINSAIL (R) Version 12.10 (? for help)
Copyright (c) 1984, 1985, 1986, 1987, 1988, and 1989 by
 XIDAK, Inc., Menlo Park, California, USA.
*delfil<eol>
Next file to be deleted (just eol to stop): foo<eol>
Next file to be deleted (just eol to stop): bar<eol>
Next file to be deleted (just eol to stop): baz<eol>
Next file to be deleted (just eol to stop): <eol>
*

                Alternatively:

*delfil foo bar baz<eol>
*
```

Example 38-1. DELFIL Example

39. DIR

Command line syntax: dir {{fileSpec}* {switch}*}*

DIR displays a listing of files according to command line arguments. DIR accepts any number of file names or file name patterns containing wildcards ("*" stands for any sequence of 0 or more characters; "?" stands for any single character) intermixed with any number of switches beginning with "-"; the switches are keywords, some optionally followed by a colon and an argument. The available switches are:

- -?: print a list of available switches.
- -cbefore:<date>: show only files created before <date>.
- -csince:<date>: show only files created since <date>.
- -directories: list directories only, not files
- -fast: show only file names, not additional information.
- -files: list files only, not directories
- -larger:<size>: show only files larger than <size> (<size> is in the units of $fileInfoCls.$OSDSize).
- -mbefore:<date>: show only files modified before <date>.
- -msince:<date>: show only files modified since <date>.
- -noreverse: do not reverse the ordering of the file listing.
- -reverse: reverse the ordering of the file listing.
- -smaller:<size>: show only files smaller than <size>.
- -sort:modify: sort by modify date/time.
- -sort:create: sort by create date/time.
- -sort:name: sort alphabetically by file names.

- -sort:size: sort by file size.

Dates may be in any format accepted by $strToDate that contains no spaces, e.g., "7-oct-86" but not "7 October 1986".

Switches may be abbreviated to a unique prefix; e.g., "-so:c" may be substituted for "-sort:create". Examples:

Command	Effect
`dir`	show contents of current directory
`dir -fast`	same, but show only file names
`dir foo bar`	show the files "foo" and "bar"
`dir *.msl`	show all files on current directory ending in ".msl"
`dir /foo/*.msl*`	show all files on /foo containing ".msl" (assuming UNIX-like directory syntax)
`dir -larger:50`	show all files on current directory larger than 50
`dir -ms:11-oct-85`	show files on current directory modified since 11 October 1985

If more than one file specification appears, separate listings are shown for each.

40. DISPSE, Module Disposer

Command line syntax: dispse {modName}*

The module DISPSE allows the user to dispose of a module's data and control sections. It accepts the names of modules on the command line or prompts for them if none is specified. It calls the system procedure dispose with the string argument(s) given.

Example 40-1 shows how to use DISPSE to dispose of the data and control sections for the modules FOO and BAR.

```
MAINSAIL (R) Version 12.10 (? for help)
Copyright (c) 1984, 1985, 1986, 1987, 1988, and 1989 by
 XIDAK, Inc., Menlo Park, California, USA.
*dispse<eol>
Next module to be disposed (just eol to stop): foo<eol>
Next module to be disposed (just eol to stop): bar<eol>
Next module to be disposed (just eol to stop): <eol>
*

                Alternatively:

*dispse foo bar<eol>
*
```

Example 40-1. DISPSE Example

41. DVIEW, Data File Viewer

The module DVIEW displays the contents of a data file. DVIEW prompts for the name of the file to be examined. Once the input file is successfully opened, the integer at position 0 is displayed and the user is prompted for a command. DVIEW always displays the current position in the file followed by "/" followed by the integer at that position in decimal, octal, and then in hexadecimal. The octal value is preceded by "'" and the hexadecimal value by "'H". Available commands are displayed when "?" is typed in response to the DVIEW command prompt, and are listed in Table 41-1.

```
n         Examine position n
eol       Step forward through file
^         Step backward through file
+n        Position forward n integers
-n        Position backward n integers
S xxx     Search for integer xxx
W xxx     Write integer xxx
F xxx     Examine file xxx
Q         Quit
```

Table 41-1. DVIEW Commands

41.1. n, <eol>, "^", "+n", and "-n"

These commands tell DVIEW what position in the input file to display. If an integer n is typed, the integer at position n is displayed. Typing <eol> causes DVIEW to display the next integer in the file. For example, if the integer at position 10 has just been displayed and the <eol> command is issued, DVIEW displays the integer at position 12, assuming that an integer takes two positions. "^" displays the previous integer, "+n" displays the nth next integer, and "-n" displays the nth previous integer in the file.

41.2. "S xxx"

The "S" command searches forward for the integer xxx. If found, the integer is displayed. If this integer is not found in the file, a message is printed and the integer at the current position is redisplayed.

41.3. "W xxx"

The "W" command writes the integer xxx to the current position of the input file. The contents of the current position are then redisplayed.

The file being examined is initially opened for input only. When the "W" command is issued, the file is closed and reopened for input and output access. After the value has been written, the file is again closed and then reopened for input only. Since DVIEW opens the file for random access, and random output is not supported for record-oriented files, the "W" command is not allowed for such files.

41.4. "F xxx"

The "F" command closes the current input file and opens xxx as the new input file. The integer at position 0 is automatically displayed.

41.5. "Q"

The "Q" command exits DVIEW.

41.6. DVIEW Example

Example 41.6-1 shows how to use DVIEW to examine a data file. The file "data" is opened and the integer at location 0 is displayed. <eol> displays the integer at location 1 (this example assumes that an integer occupies one file position) and "100<eol>" displays the integer at location 100. The search command searches forward from the current position (100) for the integer 1 and finds it at position 362.

```
MAINSAIL (R) Version 12.10 (? for help)
Copyright (c) 1984, 1985, 1986, 1987, 1988, and 1989 by
 XIDAK, Inc., Menlo Park, California, USA.
*dview<eol>
Data file viewer (? for help)
File to view: data<eol>
       0/   562  '1062  'H232
DVIEW><eol>
       1/   23088332800  '254013000000  'H5602C0000
DVIEW>100<eol>
     100/   17200840693  '200117777765  'H4013FFFF5
DVIEW>s 1<eol>
     362/   1  '1  'H1
DVIEW><eol>
     363/   3  '3  'H3
DVIEW>q<eol>
*
```

Example 41.6-1. DVIEW Example

42. GCCHP, Global File Cache Parameter Utility

42.1. Overview

The file cache (described in detail in the "MAINSAIL Language Manual") is an optimization for random access device modules. Its purpose is to improve I/O performance for files opened for random access. The file cache maintains a buffer cache that is a linked list of buffers. All files opened for random access share the file cache; i.e., there is only one file cache the buffers of which service all random access files.

When a device module makes a request for a buffer, the file cache is searched. If the buffer is found, it is returned; otherwise a new buffer is read. Thus, an I/O operation is eliminated for every buffer request satisfied by the file cache. This significantly improves performance on I/O-bound machines. For example, the MAINSAIL compiler running under VM/SP CMS on an IBM 3081 showed a 5-to-1 improvement in wall clock performance with the caching disk module.

Maintenance of the cache is governed by three parameters: the minimum number of buffers in the file cache (requestedMinSize), the maximum number of buffers in the file cache (requestedMaxSize), and the file cache hit percentage to be maintained (requestedHitPercent). Buffers are added to the file cache as necessary until requestedMinSize buffers exist, after which the hit percentage governs further addition of buffers. Each time a request is made for a non-resident buffer, the actual hit percentage is compared with requestedHitPercent. If the actual hit percentage is less than requestedHitPercent (minus five percent) and the number of buffers is less than requestedMaxSize, a new buffer is created. The current buffer is added to the cache, making it larger. If the number of buffers is between requestedMinSize and requestedMaxSize and the actual hit percentage is greater than requestedHitPercent (minus five percent), the least recently accessed buffer is removed from the cache, written if necessary, and returned as empty. Again, the current buffer is added to the list, but since one has been removed, the file cache size remains constant.

XIDAK reserves the right to change the implementation of the file cache without notice.

42.2. How to Use GCCHP

The file cache is automatically used for all random access files. The default values for requestedMinSize, requestedMaxSize, and requestedHitPercent are operating-system-dependent.

GCCHP (Global CaCHe Parameters) allows the user to set the global file cache parameters from the MAINSAIL executive. When invoked, it displays the current file cache parameters, the actual hit percentage, and the maximum number of buffers that have been allocated for the file cache, and then prompts for new values.

Example 42.2-1 illustrates how GCCHP may be used. The global file cache parameters are first changed so that the minimum number of buffers is 30, the maximum number of buffers is 80, and the hit percentage is 100. The compiler, which opens random access files, is then run and GCCHP is invoked again to display the actual hit percentage and maximum number of buffers allocated for the file cache. The current global file cache parameters remain unchanged.

```
MAINSAIL (R) Version 12.10 (? for help)
Copyright (c) 1984, 1985, 1986, 1987, 1988, and 1989 by
 XIDAK, Inc., Menlo Park, California, USA.
*gcchp<eol>
Current global cache parameter values:
    requestedMinSize: 8
    requestedMaxSize: 32
    size: 0
    requestedHitPercent: 95
    hit percent: 6

Want to specify new parms (Yes or No): y<eol>
New requestedMinSize (8): 30<eol>
New requestedMaxSize (32): 80<eol>
New hit percent (95): 100<eol>
*compil<eol>

MAINSAIL (R) Compiler
Copyright (c) 1984, 1985, 1986, 1987, 1988, and 1989 by
 XIDAK, Inc., Menlo Park, California, USA.

compile (? for help): sample<eol>
Opening intmod for $SYS...

sample 1 ...
Objmod for SAMPLE on sample-<os>.obj
Intmod for SAMPLE not stored

compile (? for help): <eol>
*gcchp<eol>
Current global cache parameter values:
    requestedMinSize: 30
    requestedMaxSize: 80
    size: 30
    requestedHitPercent: 100
    hit percent: 80

Want to specify new parms (Yes or No): n<eol>
```

Example 42.2-1. GCCHP Example

43. FILMRG, File Merging Utility

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

Command line syntax: filmrg {baseFile {file1 {file2 {outputFile}}}}*

FILMRG takes as input an original file (the "base file") and two files (the "edited files") that may have differences from the base file. FILMRG produces as output a merged file composed of the base file modified to incorporate the differences from both of the edited files, and also a difference file displaying the differences among the files. FILMRG is useful for developing a composite version of a file from two files that have undergone separate editing paths from a common starting point.

User interaction with FILMRG is similar to user interaction with LINCOM (see the description of LINCOM in the "MAINSAIL Utilities User's Guide").

FILMRG prompts for the names of the base file, the first edited file, and the (optional) second edited file. FILMRG then prompts with "Subcommands? (Yes or eol):". If "Y" (or "y") is typed in response to this prompt, FILMRG allows the user to set various options that control the comparison of the input files. The prompts displayed when the options are set are shown in Table 43-1. FILMRG then determines whether there are any differences among the files. If there are any, then FILMRG asks for the name of the merged (output) file; otherwise, FILMRG reports that the files are identical.

```
Name of output file for differences (eol for logFile):
Automatic continuation? (Yes or eol):
Max display lines (eol for 23):
Lines to look ahead (eol for 200):
Nonblank lines to match (eol for 2):
```

Table 43-1. FILMRG

The "Name of output file for differences", "Automatic continuation", "Max display lines", "Lines to look ahead", and "Nonblank lines to match" subcommands have the same meaning as for LINCOM.

If the output file for differences is the default (logFile), the interactive display of differences is somewhat different from LINCOM's. If the number of lines for one of the differences exceeds the maximum allowed, FILMRG prompts whether to display more lines for the same file before displaying lines from the corresponding difference in the next file.

If a difference consists of an insertion in the base file, the line "<insertion>" is shown for the difference in the base file; if a difference consists of a deletion from the base file, the line "<deletion>" is shown for the difference in the base file. If both edited files' lines are the same in a given difference, the second edited file's difference is shown as the line "<same>".

The merged file contains the differences from both of the edited files. If the differences from the base file are the same in both edited files, the affected section of the base file is replaced in the merged file by the corresponding section of the edited files. Wherever the two edited files do not contain the same difference from the base file, the affected sections from each of the three files are included in the merged file in the format:

```
********** <base file name> **********
<section from base file>
********** <first edited file name> **********
<section from first edited file>
********** <second edited file name> **********
<section from second edited file>
********** end **********
```

Each such conflicting section requires the user to edit the merged file in order to produce a version of the file that resolves the conflict. FILMRG reports the number of conflicting sections at the end of each merge.

An example of FILMRG execution is shown in Example 43-2. One of the conflicting sections from the merged file "newUnix.msl" is shown in Example 43-3.

```
Original file: /oldv9src/unix.msl<eol>
First edited file: /v9src/unix.msl<eol>
Second edited file (<eol> if none): /v11src/unix.msl<eol>
Subcommands? (Yes or eol): y<eol>
Show differences? (eol or No): n<eol>
Nonblank lines to match (eol for 5): <eol>
Name for merged file (<eol> for none): newUnix.msl<eol>
12 conflicting changes need editing in output file

Merge more files? (Yes or eol): <eol>
```

Example 43-2. FILMRG Example

```
****************** /oldv9src/unix.msl ******************
"Desired module OSDTAP, UNISYS, BSDITF, S5ITF "
    & "(in quotes)" & eol &
    "(UNISYS, BSDITF, and S5ITF are foreign modules): ";
****************** /v9src/unix.msl ******************
"Desired module: OSDTAP, UNISYS, BSDITF, S5ITF, MVUX "
    & "(in quotes)" & eol &
    "(UNISYS, BSDITF, S5ITF, and MVUX " &
    "are foreign modules): ";
****************** /v11src/unix.msl ******************
"Desired module (in quotes).  Choices are:" & eol &
"BSDITF, BSDTCP, MVUX, OSDTAP, S5ITF, UNISYS" & eol &
"(BSDITF, BSDTCP, MVUX, S5ITF, and UNISYS " &
    "are foreign modules)" & eol;
************************ end ************************
```

Example 43-3. A Sample Conflicting Section from the Merged File

44. HSHMOD, Hash Lookup Utility

44.1. Overview

The module HSHMOD (HaSH MODule) is not a utility that is invoked from the MAINEX executive. Rather, it is a module that can be invoked from user modules.

HSHMOD maintains a symbol table implemented with a hash code retrieval algorithm for fast lookup. The symbol table consists of an array, the "collision array", declared as "POINTER(hashedRecord) ARRAY". Each element of the collision array is the head of a linked list of all records that hash to the element's index. The class "hashedRecord" is declared as shown in Figure 44.1-1.

```
CLASS hashedRecord (
    STRING key;
    POINTER(hashedRecord) link;
);
```

Figure 44.1-1. Declaration of hashedRecord

The records in the symbol table are allocated by the user, and must begin with "key" and "link" fields as declared in Figure 44.1-1. The user's record may have additional fields by using hashedRecord as a prefix class. This is shown in Example 44.1-2. The class "person" provides records with a "key", "link", "homeAddress", and "age" to be entered into HSHMOD's symbol table. Such records may be passed to the HSHMOD procedures described below since they extend the hashedRecord class that is known to HSHMOD.

HSHMOD need not know anything about the fields beyond the "key" and "link", which are all it uses to maintain the symbol table. HSHMOD's interface is shown in Figure 44.1-3. XIDAK reserves the right to change this interface by adding new fields and new optional parameters to the procedures.

```
CLASS(hashedRecord) person (
    STRING homeAddress;
    INTEGER age;
);
```

Example 44.1-2. Using HashedRecord as a Prefix Class

```
CLASS hshCls (
    PROCEDURE hashInit
        (OPTIONAL INTEGER tableSize);
    PROCEDURE hashEnter
        (POINTER(hashedRecord) p);
    POINTER(hashedRecord) PROCEDURE hashLookup
        (STRING key);
    POINTER(hashedRecord) PROCEDURE hashRemove
        (STRING key);
    POINTER(hashedRecord) PROCEDURE hashNext
        (POINTER(hashedRecord) p);
    PROCEDURE hashLookupNextInit
        (STRING key);
    POINTER(hashedRecord) PROCEDURE hashLookupNext;
    BOOLEAN PROCEDURE hashRemoveRecord
        (POINTER(hashedRecord) p);
    LONG INTEGER PROCEDURE hashStore
        (POINTER(dataFile) f;
         OPTIONAL LONG INTEGER startPageOrPos;
         OPTIONAL BITS ctrlBits);
    LONG INTEGER PROCEDURE hashLoad
        (POINTER(dataFile) f;
         OPTIONAL LONG INTEGER startPageOrPos;
         OPTIONAL BITS ctrlBits);
);

MODULE(hshCls) hshMod;
```

Figure 44.1-3. HSHMOD Interface

44.2. HSHMOD Header

The file with logical name "(system library)" contains the class declarations for hashedRecord and hshCls and the declaration of HSHMOD. To use HSHMOD, include the directives shown in Figure 44.2-1 at the beginning of each module that uses HSHMOD. HSHMOD is included as a standard module in the MAINSAIL system runtime library.

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

Figure 44.2-1. Including the HSHMOD Declarations

44.3. The Procedure hashInit

hashInit is meant to be used at most once, before any of the other procedures have been called, to specify the size of the hash table. The hash algorithm computes a number between 0 and tableSize - 1 from key, and uses this number as an index into the collision array. Any number greater than zero can be specified, though prime numbers are recommended. The larger tableSize, the more efficient the hash lookup. If hashInit is not called by the user, HSHMOD automatically invokes "hashInit(131)" the first time one of its procedures is invoked, since 131 is a good table size for applications that store moderate numbers of identifiers. Very large applications may wish to use a larger table size.

44.4. The Procedure hashEnter

"hashEnter(p)" enters the record pointed to by p in the symbol table. "p.key" is used as the hash key. Many records with the same hash value may be entered into the symbol table. Each record is added to the front of its collision list. More than one record with the same key may be entered in the symbol table.

44.5. The Procedure hashLookup

"hashLookup(s)" searches for the record with key s, and if found, returns a pointer to the record stored in the symbol table. If not found, nullPointer is returned. If more than one record with

key s is in the symbol table, it is not specified which one is returned by hashLookup. The code shown in Example 44.5-1 ensures that a symbol is entered just once.

```
IF NOT hashLookup(name) THENB # not already entered
    p := new(person); p.key := name;
    p.homeAddress := homeAddress; p.age := age;
    hashEnter(p) END;
```

Example 44.5-1. Using the Procedure hashLookup

44.6. The Procedure hashRemove

"hashRemove(s)" searches for the record with key s, and if found, removes it from the symbol table and returns a pointer to the removed record. If more than one record with key s is in the symbol table, it is not specified which one is removed by hashRemove. If not found, nullPointer is returned.

44.7. The Procedure hashNext

"hashNext(p)" provides a means of sequencing through the records in the hash table. hashNext returns a pointer to the record in the symbol table that is "after" record p, where "after" means the record at p.link, if there is one, else the record at the next non-empty collision array element after the one containing p. If there are no more records in the symbol table, hashNext returns nullPointer. If p is nullPointer, the first record in the collision array is returned. hashNext is typically used as shown in Example 44.7-1.

```
p := NULLPOINTER;
WHILE p := hashNext(p) DOB <process record p> END
```

Example 44.7-1. Using the Procedure hashNext

Example 44.7-2 shows how to use hashNext to dispose of all records in a symbol table.

```
WHILE p := hashNext(NULLPOINTER) DOB
    p := hashRemove(p.key); dispose(p) END
```

Example 44.7-2. Disposing of All Records in a Symbol Table

44.8. The Procedures hashLookupNextInit and hashLookupNext

hashLookupNextInit prepares HSHMOD to find all records with the key "key". Subsequent calls to hashLookupNext return each record with the specified key until all such records have been returned; hashLookupNext then returns nullPointer.

44.9. The Procedure hashRemoveRecord

hashRemoveRecord removes the record to which p points from the hash table. It returns true if it found p's record in the table, false otherwise. If there are several records with the same key in the table, hashRemoveRecord may be used to remove a specified record. hashRemove removes an unspecified record with the given key; all records with a key s can be removed by:

```
DO UNTIL NOT hashRemove(s);
```

44.10. The Procedures hashLoad and hashStore

hashStore and hashLoad store and load a hash table into or from f at the position startPageOrPos. At present, hashStore and hashLoad use the Structure Blaster; ctrlBits may specify the valid ctrlBits to $structureWrite or $structureRead, respectively (this is subject to change). hashStore and hashLoad return the size of the structure stored or loaded, in pages or storage units, depending on whether or not $nonPaged is set in ctrlBits.

Since hashStore and hashLoad use the Structure Blaster, they do not work if the Structure Blaster is not present, and hash tables may not be stored between versions of MAINSAIL; all other Structure Blaster caveats also apply.

44.11. Creating Instances of HSHMOD

The user should create an instance of HSHMOD for each symbol table rather than use the bound instance, since other modules in the program may use HSHMOD for other symbol tables. Different HSHMOD instances must be created to keep the symbol tables separate. To do this, declare a pointer for the instance, and use that pointer to qualify all calls to the interface procedures, as shown in Figure 44.11-1.

```
POINTER(hshCls) mySymbolTable;
...
mySymbolTable := new(hshMod); # create the instance
...
# Use mySymbolTable for each call to a HSHMOD interface
# procedure
IF p := mySymbolTable.hashLookup(s) THEN ...
...
# When finished with the symbol table, dispose of each
# record as illustrated earlier, and then:
dispose(mySymbolTable); # dispose of the instance
```

Figure 44.11-1. HSHMOD Pointers

45. IFX, ELSEX, and ENDX: MAINEX Script Conditional Execution Modules

The module IFX reads lines from cmdFile until it encounters a line reading "THENX". It evaluates the condition specified by the lines; if it is true, it exits, but if it is false, it reads (and discards) lines until it encounters a line reading "ELSEX" or "ENDX" (it also skips nested "IFX"-"ENDX" sequences).

The module "ELSEX" skips lines until it reaches a line reading "ENDX" (it also skips nested "IFX"-"ENDX" sequences).

The module "ENDX" does nothing.

The result of these behaviors is that IFX, ELSEX, and ENDX implement conditional commands when invoked from MAINEX. The syntax is:

```
ifx
<lines specifying a conditional expression>
thenx
<modules to invoke if conditional expression is true>
endx
```

or:

```
ifx
<lines specifying a conditional expression>
thenx
<modules to invoke if conditional expression is true>
elsex
<modules to invoke if conditional expression is false>
endx
```

where the modules to invoke may also contain properly nested "IFX"-"ENDX" sequences.

The conditional expression has the syntax:

```
<conditional expression> ::=
        <term> { OR <conditional expression> }
<term> ::= <factor> { AND <term> }
<factor> ::= { NOT } <primary>
<primary> ::=
         ( <conditional expression> ) |
         BINDABLE ( <module name> ) |
         FILEEXISTS ( <file name> )
```

where "::=" means "is defined as", the vertical bar separates alternative definitions, and curly brackets indicate an optional part of an expression.

The primary "BINDABLE(m)", where m is a module name, is true if m can be bound. Otherwise it is false. m is not quoted.

The primary "FILEEXISTS(f)", where f is a file name, is true if f can be opened for input. Otherwise it is false. f is a quoted string constant, in which double quotes must be doubled.

Primaries are combined with "AND", "OR", "NOT", and parentheses with the same precedence as those operators have in MAINSAIL.

The case of the letters in module names and keywords is unimportant.

The following MAINEX script:

```
ifx
not fileexists("data.file")
thenx
ifx
not bindable(mkfile)
thenx
compil
mkfile.msl,
debug


endx
mkfile
data.file
endx
```

checks whether the file "data.file" exists. If not, it checks whether the module MKFILE exists; if not, the module is compiled. MKFILE is then invoked to create "data.file".

The MAINEX script:

```
ifx
bindable(foo) and bindable(bar)
and bindable(baz)
thenx
foo
elsex

endx
```

runs the module FOO if FOO, BAR, and BAZ are bindable; otherwise, it exits MAINEX (since a blank line causes MAINEX to exit).

Example 45-1. Use of IFX, ELSEX, and ENDX

46. INTCOM

INTCOM compares two intmods to see if they are the same except for their compilation dates. The module can be invoked interactively or it can be called from a program using the procedure $compareIntmods.

INTCOM has a strict notion of whether or not two intmods match; except for the compilation date and time fields embedded in the modules, each pair of corresponding characters must be the same. This means, e.g., that if a module is compiled and then one of its procedures is recompiled without having been changed, INTCOM would report that the original version and the subsequent version of the intmod do not match, because the information about the recompiled procedure is not in the same place.

46.1. Interactive Use of INTCOM

If INTCOM is run interactively, it first prompts for the target abbreviation of the operating system for which the two modules to be compared were compiled. Responding with <eol> means that the modules were compiled to run on the host system.

Next, INTCOM prompts for the names of the files containing the two intmods. The first prompt asks for a file name. The user can type the name of the file containing the intmod, or if MAINSAIL can derive the intmod file name from the module name alone, the user can type just the module name. If the object module is in a library, the user can respond to the first prompt with <eol>, in which case INTCOM prompts for the name of the library file containing the intmod, and then the name of the module. INTCOM goes through the same sequence to get the names of both intmods.

If the two intmods match, INTCOM reports that they matched after comparing them. Otherwise, it issues a short message indicating why they did not match. In any event, INTCOM then prompts for the names of two more intmods to compare. To exit INTCOM, the user should type <eol> to the prompt for another file name and <eol> to the prompt for a library name.

If one of the intmods examined in the previous comparison was in a library, then when INTCOM prompts for another pair of intmod names, it assumes that the corresponding one of the intmods to be compared this time is in the same library, and so it prompts for a module name without first prompting for a file name and then a library name. For example, if during the previous comparison, the first intmod was in a library "foo" and the second was not in a

library, INTCOM assumes that the first intmod it prompts for during the next iteration is also in the library "foo", and prompts for the name of another module in "foo". It makes no such assumptions about the second intmod for which it prompts. If the user types <eol> in response to the module name prompt, INTCOM prompts for the name of a file, as it did when it was first run.

46.2. Calling INTCOM from a Program

The interface to $compareIntmods is shown in Table 46.2-1.

```
BOOLEAN
PROCEDURE   $compareIntmods
                          (STRING file1,lib1,mod1,file2;
                           OPTIONAL STRING lib2,mod2,target;
                           PRODUCES OPTIONAL STRING msg);
```

Table 46.2-1. $compareIntmods

$compareIntmods returns true if and only if the two object modules specified by its input parameters are the same except for their compilation dates.

file1, lib1, and mod1 specify the first intmod to be compared, and file2, lib2, and mod2 the second intmod.

file1 can specify the name of the intmod file, or just the name of the module if MAINSAIL can determine the name of the intmod file from the module name. If file1 is non-Zero, lib1 and mod1 are ignored. If the intmod is in a library, lib1 can specify the library's file name and mod1 can specify the module name within the library, in which case file1 should be the null string. file2, lib2, and mod2 specify the second module in the same way that file1, lib1, and mod1 specify the first.

target specifies the target abbreviation of the operating system for which the two modules were compiled. If target is omitted or is the null string, the host operating system is assumed.

msg is set only if $compareIntmods returns false. It contains a brief description of why the procedure did not return true, such as that it could not find one of the modules, or that one of them was not really an intmod, or that the modules did not match.

47. INTLIB

Command line syntax: intlib {one INTLIB command}

INTLIB, the intmod librarian, is used to create and maintain intmod libraries. Putting several intmods into an intmod library eliminates the need for individual intmod files, reducing clutter in the file system.

Many INTLIB commands are similar to those of MODLIB, the objmod librarian.

"Open" intmod libraries are not open files; i.e., they do not consume an operating system file handle. The files are opened when information is added to them or extracted from them and then immediately closed; however, they are considered to be "open" libraries because the MAINSAIL runtime system maintains a global list of libraries. All MAINSAIL programs that use intmods may search for them in the same global intmod library list, and all program commands that open an intmod library add it to the global list. For example, opening an intmod library with the MAINEX subcommand "OPENINTLIB" opens it for the purposes of the compiler, opening an intmod library in the compiler opens it for the purposes of the debugger, etc. Intmod libraries remain open until explicitly closed.

Modifying an intmod library on the global open intmod library list has undefined effects.

As described in the "MAINSAIL Language Manual", intmods are searched for in the order:

1. A file name, if the intmod directive specifies a file name (or a syntactically invalid module name).

2. All open intmod libraries, most recently opened first.

3. The default file name for the module.

4. The name specified, treated as a file name.

5. A remembered file name, if opening a supporting intmod.

There are no restrictions on the way modules are organized in intmod libraries. An intmod library might consist of all modules related to a given program, or it might consist of several unrelated "utility" modules. An intmod library may also contain intmods for several different targets (this is not true of objmod libraries, which contain only objmods for the same target).

MAINSAIL does not impose a limit on the number of intmod libraries that may be open during execution, although the underlying operating system may have a limit on the number of open files.

An intmod library can be opened in the following ways:

- With the MAINEX "OPENINTLIB" subcommand. The subcommand may be specified interactively or in a MAINSAIL bootstrap.
- With the compiler's "OPENINTLIB" or "ININTLIB" subcommand.
- With the debugger's "OI" subcommand.
- By using an INTLIB device prefix.
- Implicitly by the compiler or debugger, when looking for supporting intmods, if a remembered intmod file name contains an INTLIB device prefix.
- Performing an operation on it with INTLIB.

47.1. How INTLIB Opens Library Files

When INTLIB operates on a module library file, the library remains open for the remainder of the MAINSAIL session (i.e., the library is added to the global open intmod library list). INTLIB uses an already opened library file for a specified library if the specified library name is exactly equal to the name used when the library file was first opened.

47.2. Library Size

MAINSAIL imposes no limit on the number of modules in an intmod library.

INTLIB automatically extends the size of a library file to accommodate new modules. However, a library file never shrinks; if many deletions are done without corresponding adds that reuse the space freed by the deletions, it may be desirable to use the "COPY" command to compact the library into a new file.

47.3. INTLIB Commands

Table 47.3-2 lists the available INTLIB commands. The following conventions are used in INTLIB command descriptions:

- "modList" is a list of module specifications separated by blanks. A module may be specified differently depending on the command, as summarized in Table 47.3-1.

- <sys. abbrev.> is a system name abbreviation, as returned by $systemNameAbbreviation. Since intmods for more than one system may be stored in an intmod library, it may be important to distinguish between them. For example, if an intmod for a module FOO is stored in a library for both Aegis and M68000 UNIX, the two intmods can be distinguished as "foo-aeg" and "foo-um68". It is not necessary to specify "-<sys. abbrev.>" when all modules operated on are for the same target system; a single "TARGET" command to specify the default target system suffices in this case.

- "libName" is a library file name. "sLibName" is a source library name (the name of a library from which modules are removed or copied) and "dLibName" is a destination library name (the name of a library to which modules are added).

- Anything enclosed in curly brackets ("{" and "}") is optional.

Form	May Be Used in Commands
<module name>{-<sys. abbrev.>}	all
<file name>	DIRECTORY, LEGALNOTICE
<module name>{-<sys. abbrev.>}=<file name>	ADD, EXTRACT
<source module name>{-<sys. abbrev.>}= <destination module name>{-<sys. abbrev.>}	COPY, MOVE

Table 47.3-1. Forms of a "modList" Element

A very long INTLIB command may be broken across several lines by terminating each line with a backslash ("\"). Thus, the command lines:

```
add foo.lib\
 abc\
 def\
 ghi
```

are equivalent to the single command line:

```
add foo.lib abc def ghi
```

Command	Description
<eol>	Exit INTLIB
ADD dLibName modList	Add module(s) to dLibName
COPY sLibName dLibName{ modList}	Copy modules from sLibName to dLibName
CREATE libName	Create an empty library named libName
DELETE sLibName modList	Delete module(s) from sLibName
DIRECTORY libName{=fileName}{ modList}	Examine the modules in library libName, writing info to file fileName
EXTRACT sLibName{ modList}	Create new file(s) containing copies of modules in sLibName
LOG	Show informational messages
MOVE sLibName dLibName{ modList}	Move module(s) from sLibName to dLibName
NOLOG	Suppress info messages
NOUPDATE	Suppress updates
QDIRECTORY libName={fileName}{ modList}	Print quick directory of library libName, writing info to file fileName
Quit	Exit INTLIB
READ fileName	Read INTLIB commands from file fileName
TARGET{ targetSystem}	Specify target operating system for subsequently opened libraries
UPDATE	Turn off NOUPDATE

Table 47.3-2. INTLIB Commands

47.3.1. <eol>, "QUIT"

To exit, type <eol> or "QUIT" when you are finished using the intmod librarian.

47.3.2. "ADD dLibName{,sysAbbrev} modList"

Add the modules specified by modList to the intmod library dLibName. modList is a list of module specifications, separated by blanks. Valid module specifications for this command are given in Table 47.3.2-1. Module specifications can be mixed within modList.

If dLibName does not exist, it is created. If free pages constitute more than a certain fraction of dLibName and dLibBame contains no hole large enough to accommodate a module in modList, the library is compacted.

If sysAbbrev is specified, it is an operating system abbreviation (as given by $systemNameAbbreviation; e.g., "cms", "uibm", "vms"). The specified system becomes the default target system for the purposes of the current command. The initial default target is the host system. The target for individual modules may be overridden by specifying a dash followed by a system abbreviation for individual modules as shown in Table 47.3.2-1.

An error message is issued if the specified target for an "ADD" does not match the target for which the intmod was actually compiled.

Form	Meaning
<module name>{-<system abbrev.>}	Add the module found in the default intmod file.
<module name>{-<system abbrev.>}=<file name>	Add the module found in the file <file name> as the module <module name>.

Table 47.3.2-1. INTLIB "ADD" Command modList Forms

47.3.3. "COPY sLibName{,sysAbbrev} dLibName{,sysAbbrev}{ modList}"

Copy the modules specified by modList in sLibName to dLibName. If modList is absent, copy all modules in sLibName. If dLibName does not exist, it is created.

If sysAbbrev is specified for either library, it is the default system abbreviation for the current command (if it is specified for both and the two targets differ, an error message is issued). If modList contains system specifications for individual modules, they override the default. An

error message is issued if a module name is used without a system abbreviation and no module exists for the default target system.

If sysAbbrev is specified for either library, but no modList is given, only modules for the specified target are copied from sLibName to dLibName. If no sysAbbrev and no modList are given, then all modules (for all systems) are copied from sLibName to dLibName.

If dLibName does not exist before the "COPY" command, then it contains no holes after the command is completed. Thus, the "COPY" command may be used to compact a library with unused space.

47.3.4. "CREATE libName"

Create an intmod library named libName. The library is initially empty.

47.3.5. "DELETE sLibName{,sysAbbrev} modList"

Delete the modules specified in modList from the intmod library libName. modList is a list of module names separated by blanks. If sysAbbrev is specified, the modules in modList are deleted for the specified system; otherwise, they are deleted for the current default target. A system abbreviation in modList overrides the default target.

47.3.6. "DIRECTORY libName{,sysAbbrev}{=fileName}{ modList}"

List information about the modules contained in the intmod library libName. libName may be the character "*" if the intmods reside in individual intmod files instead of in an intmod library. fileName is the name of the file to which the list is to be written. If fileName is omitted, the list is written to logFile. modList is a list of module names to be included in the directory list, separated by blanks. Valid module specifications for "DIRECTORY" are <module name> and <file name>. If the module resides in a library, only <module name> is permitted. If the module does not reside in a library ("*" was specified for the library name), then if <module name> is specified, the default intmod file name is formed. If <file name> is specified (the given string is not a possible module name), the given file name is used. The object module in the named file is examined.

If modList is omitted, all modules in the intmod library are listed if sysAbbrev is omitted; if sysAbbrev is present, only modules for the specified target are listed.

The information listed includes the module's directory name, the module name (if it differs from the directory name), the module's target system, the number of pages it occupies, the date

and time at which it was compiled, the version for which it was compiled, and the options with which it was compiled. Example 47.3.6-1 is a sample session with MAINSAIL showing execution of the INTLIB directory command and the resulting listing.

```
MAINSAIL (R) Version 12.10 (? for help)
Copyright (c) 1984, 1985, 1986, 1987, 1988, and 1989 by
 XIDAK, Inc., Menlo Park, California, USA.
*intlib<eol>
MAINSAIL (R) Intmod Librarian (? for help)
Copyright (c) 1986, 1987 by XIDAK, Inc., Menlo Park,
  California, USA.

INTLIB: dir qform-um6.ilb<eol>

Directory of qform-um6.ilb

DirName Target TextPages   Date      Time   Version Options

dumb      um68       83    13-Mar-87  13:53:28    11.11    C S
dumbo     um68       76    13-Mar-87  13:53:45    11.11    C S
ps        um68      145    13-Mar-87  13:54:22    11.11    C S
ptronx    um68       82    13-Mar-87  13:54:02    11.11    C S
qcmds     um68      285    13-Mar-87  13:52:35    11.11    C S
qfhdr     um68       46    13-Mar-87  13:51:33    11.11    C S
qform     um68       85    13-Mar-87  13:54:35    11.11    C S
qindex    um68       91    13-Mar-87  13:53:09    11.11    C S
qpass1    um68      136    13-Mar-87  13:51:53    11.11    C S
qpass2    um68       46    13-Mar-87  13:52:04    11.11    C S
qpass3    um68      131    13-Mar-87  13:52:54    11.11    C S

11 intmods using 1206 text pages.
1 directory textPage. 0 free textPages.
1207 total textPages.

INTLIB: <eol>
```

Example 47.3.6-1. Using the INTLIB "DIRECTORY" Command

The meanings of the abbreviations shown in the options column of the directory listing, separated by spaces, are shown in Table 47.3.6-2. For example, "C D L" indicates that the module was compiled with checking set, debuggable, and with a legal notice. "Cmp", "Cms",

"Cps", or "Cmps" is used to display multiple counting options. "Tmp" is displayed to indicate both "MODTIME" and "PROCTIME" options.

Option Abbreviation	The Module Was Compiled with:
C	CHECK subcommand
AC	ACHECK subcommand
D	DEBUG subcommand
O	OPTIMIZE subcommand
A	ALIST subcommand
U	UNBOUND subcommand
IB	inline procedures have bodies
I	INCREMENTAL subcommand
S	SAVEON subcommand
Cm	PERMODULE subcommand
Cp	PERPROC subcommand
Cs	PERSTMT subcommand
Tm	MODTIME subcommand
Tp	PROCTIME subcommand

Table 47.3.6-2. Option Letters Displayed by the "DIRECTORY" Command

47.3.7. "EXTRACT sLibName{,sysAbbrev}{ modList}"

Extract the specified module(s) from the intmod library libName. sysAbbrev, if specified, is the default target system abbrevation for the command; otherwise, the current default target is used. modList is a list of module specifications, separated by blanks. Valid module specifications for this command are given in Table 47.3.7-1. Different types of module specifications can be mixed within modList.

If modList is omitted, all modules are extracted from the library and written to their default files.

47.3.8. "LOG"

The informational messages written by INTLIB when operating on libraries are enabled.

Form	Meaning
<module name>{-<sys. abbrev.>}	Extract the module and write it to the default intmod file.
<module name>{-<sys. abbrev.>}=<file name>	Extract the module with the name <module name> and write it to a file named <file name>.

Table 47.3.7-1. INTLIB "EXTRACT" Command modList Forms

47.3.9. "MOVE sLibName{,sysAbbrev} dLibName{,sysAbbrev} modList"

The arguments to "MOVE" have the same format and effect as the arguments to "COPY", except that modules copied from sLibName are also deleted from dLibName.

47.3.10. "NOLOG"

The informational messages written by INTLIB when operating on libraries are suppressed.

47.3.11. "QDIRECTORY libName{,sysAbbrev}{=fileName}{ modList}"

The arguments to "QDIRECTORY" have the same format and effect as the arguments to "DIRECTORY", except that less information is given about each module.

47.3.12. "READ fileName"

"READ" causes INTLIB commands to be read from the named file. The "READ" command can be used to do nested readings to an arbitrary depth.

47.3.13. "TARGET{ sysAbbrev}"

"TARGET" specifies the default target system for subsequent commands. sysAbbrev is a system abbrevation, as given by $systemNameAbbreviation. If the target system is omitted, the default target is set to the host system. The default target is initially the host system.

47.3.14. "UPDATE"/"NOUPDATE"

Default: "UPDATE"

Operations modifying large libraries (i.e., adding or deleting intmods) take longer than operations on small libraries. After each INTLIB command, the updated directory list for every modified library is written to the corresponding library file, and all library files open by INTLIB are closed. This helps keep the library files in a consistent state should a system crash or other abort occur between commands, and keeps the number of open files to a minimum. As the directory gets large, it takes longer and longer to store and read it each time.

Since the storing of the directory occurs only after each command, the single command "COPY srcLib dstLib" is a much faster way to copy all modules from srcLib to dstLib than a separate command for each module. If not all modules are to be copied, then the command:

```
COPY srcLib dstLib mod1 mod2 ... modn
```

is faster than n separate commands since just one library update occurs (after all the modules have been copied).

However, there are times when it is more convenient to use a separate command for each module, e.g., in a script generated by a program, or when the commands are being sent to INTLIB through calls to $executeIntlibCommands. The "UPDATE" and "NOUPDATE" commands may be used to handle such situations.

The default is "UPDATE". Once the "NOUPDATE" command is given, all libraries referenced are kept open, and directories are not written to each altered library after each command. The "UPDATE" command turns off this effect of "NOUPDATE", and also immediately closes all libraries and updates the directories of all altered libraries. An altered library is updated when it is closed, even if "NOUPDATE" is in effect (all libraries are closed by INTLIB's final procedure, so all libraries are updated when MAINSAIL exits).

The commands:

```
NOUPDATE
COPY srcLib dstLib mod1
COPY srcLib dstLib mod2
...
COPY srcLib dstLib modn
UPDATE
```

execute faster than the same commands without the surrounding "NOUPDATE" and "UPDATE" commands. The speed-up may be significant when many modules are involved.

If the system should crash, or if the MAINSAIL execution is aborted, between "NOUPDATE" and "UPDATE", the libraries opened for output access may be in an unrecoverable state.

47.4. INTLIB as a Device Prefix

An intmod in a library can be treated as a file by specifying an INTLIB device prefix. This is often useful if, e.g., the debugger cannot find an intmod, and is prompting for an intmod file name; if the intmod resides in a library, a file name prefixed with an INTLIB device prefix is the appropriate response.

The syntax of an INTLIB device prefix for a library named libName and a module named modName is given by:

```
"INTLIB(" & libName & ")" & $devModBrkStr & modName
```

The case of the letters in "INTLIB" and modName is not important.

For example, on systems where $devModBrk is the character ">", a module FOO in a library "proj-xxx.ilb" would be specified by:

```
intlib(proj-xxx.ilb)>foo
```

The target system of an intmod file may be specified either by following the library name with a comma and the target system abbreviation or by following the module name with a dash and the target system abbreviation; e.g.:

```
intlib(proj-xxx.ilb,vms)>foo
```

or:

```
intlib(proj-xxx.ilb)>foo-vms
```

If the target is not specified, the host system is assumed.

If the library is omitted in an INTLIB file specification, i.e., if the syntax:

```
"INTLIB" & $devModBrkStr & modName
```

is used, all open intmod libraries are searched for modName.

When an INTLIB file f is opened, f.name is modified to include the library name and target if these were not specified in the call to "open".

INTLIB files cannot be opened for output access; i.e., they must be opened read-only.

47.5. INTLIB Program Interface

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

INTLIB can be controlled from a user program by calling the procedure $executeIntlibCommands, declared as shown in Figure 47.5-1.

```
BOOLEAN PROCEDURE $executeIntlibCommands
                        (OPTIONAL STRING cmds;
                         OPTIONAL POINTER(textFile) f;
                         OPTIONAL BITS ctrlBits)
```

Figure 47.5-1. Declaration of $executeIntlibCommands

cmds is a string that contains exactly what would be typed by a user giving commmands to INTLIB. In addition, f can indicate a file from which commands are read once cmds becomes empty. If f is omitted, cmdFile is used. To force INTLIB to return rather than read from f when cmds becomes empty, specify the "QUIT" command as the last command in cmds.

A return value of false indicates that an error occurred during processing of a command. errMsg is called to report the error; the invoking program can intercept the error using the MAINSAIL exception mechanism. In this case, $exceptionStringArg1 starts with the substring "INTLIB:".

48. LIB and LIBEX, File Library Device Module and Librarian

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

48.1. Introduction

48.1.1. Likely Changes

Some changes to LIB are likely in the future. In the descriptions of LIB and LIBEX, text between square brackets, "[" and "]", denotes features considered especially likely to change (but note that other features may change as well).

48.1.2. Motivation

LIB is an "object" librarian. It provides a portable, tree-structured name space for accessing files and directories. Each object can be stored within its library's base file, or stored in a separate host file, but is accessed through the library.

LIB can be used to overcome the open file limit (maximum number of active "file handles" or "file descriptors") imposed by many operating systems, e.g., UNIX. For this purpose, the library files are stored within a library's base file. When these library files are opened, only one host operating system file handle is used: the file handle for the library's base file.

LIB can be used to supplement the flat file name space provided by certain operating systems, e.g., CMS, with a more flexible tree-structured name space. For this purpose, the files can be stored either in a library's base file or separately in individual host files.

LIB provides file version numbers even when the underlying operating system does not support file version numbers.

LIB is not a database; it does not provide any capability for multi-user access and does not provide journaling or rollback capabilities normally associated with a database.

[The results of more than one process simultaneously accessing a single library are undefined. In a future release this restriction may be partially lifted to allow multiple read-only accesses to a library.]

LIB uses the MAINSAIL Structure Blaster. The Structure Blaster must be installed for LIB to function. Since LIB uses the Structure Blaster, the same compatibility restrictions apply to libraries with respect to MAINSAIL versions as apply to structure images stored in data files (see the "MAINSAIL Structure Blaster User's Guide" for details).

[While XIDAK will strive to make the format of libraries forward compatible, XIDAK reserves the right to change this format if XIDAK deems it necessary.]

48.2. General Concepts

48.2.1. Basic Definitions

- A file is "accessed" when it is opened. The name of the file is interpreted either by the host operating system or by LIB.
- Each file must be "stored"; disk pages are required to store the information associated with each file. These pages may be either in a separate host file or in a subset of the pages of the base file of the library that contains the file.
- A "host file" is a file stored in, and accessed through, the file system of the underlying operating system.
- The "base file" of a library is a host file that, at a minimum, contains the directory structure for the library. The base file may also be used to store library files.
- A "library file" is a file that is accessed through a library and may be stored either as a host file or within the base file of the library.
- A library file is said to be "contained" in a library if it is accessed through the library, regardless of whether it is stored as a host file or stored within the library's base file.

48.2.2. The Base File

Each library is defined by a single host file, the base file for the library. This file always contains and stores the directory structure for the library and can optionally store files contained in the library.

Each base file contains a structure that implements a tree-structured name space for accessing objects, much like the tree-structured name space implemented by the UNIX file system for accessing files.

[The directory structure for a library is loaded from the library's base file into memory when the first library file contained in the library is opened, and kept in memory as long as any library file contained in the library is open. When all library files contained in the library are closed, the directory structure is rewritten in the base file if any changes have been made to the library.]

The base file for a library is kept open as long as any library file contained in the library is open and is closed when no library files contained in the library are open. Thus, the base file consumes only a single file handle.

If the library files are stored within the base file, then no additional file handles are required to access these files. One file handle is required to access each library file that is stored in a host file.

48.2.3. The Directory Structure

A directory consists of a set of objects. Each object has a name and is either a library file or another directory. Each object may have multiple versions, but all versions of the same object must be of the same type. The top-level directory is known as the "root directory" for the library.

Directories form a tree-structured name space. Objects are accessed by means of path names similar to those used in the UNIX operating system. A LIB path name consists of a sequence of directory names followed by an element name. The path name is interpreted by starting in the root directory and successively locating the directories named in the path.

48.2.4. Library File Name Syntax

Library files are accessed by means of a fully qualified name, as shown in Figure 48.2.4-1.

The base file name must be enclosed in matching parentheses. It must be a MAINSAIL file name, i.e., a string that could be passed to the system procedure "open".

The closing parenthesis following the base file name must be immediately followed by the $devModBrk character (usually ">"; see the "MAINSAIL Language Manual") and a path name.

A path name always starts with a "/" and ends with a "/". A path name is composed of a sequence of elements, each of which names a directory. Each directory name may optionally be followed by a ";" followed by a version number. Directories are separated by "/".

A fully qualified name may contain a file name. The file name may optionally be followed by a version separated by a ";".

Directory names and file names are case-insensitive, and are composed of any alphanumeric character followed by a sequence consisting of zero or more alphanumeric or punctuation characters (the allowed punctuation characters are ".", "+", and "-").

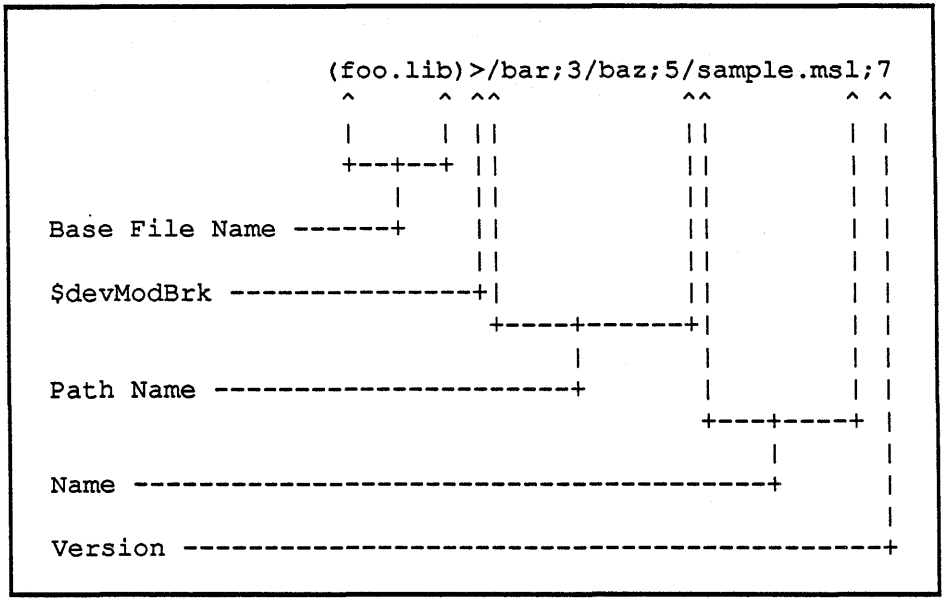


Figure 48.2.4-1. Components of a Fully Qualified Name

The special name ".." names the parent directory of the currently connected directory and "/" names the root directory of the currently connected library (see Section 48.5.3).

Some commands also accept "*" to mean all objects in a specified directory, although "*" has not been uniformly implemented in this version of LIB.

48.2.5. Versions

Each object in a library, including directories, can have multiple versions, just as files can have multiple versions under the VAX/VMS and TOPS-20 operating systems. A version number is a positive integer that specifies a distinct copy of the object (different versions do not share the same contents in any way).

A new version of a library file is created whenever the library file already exists and is opened for create, or is the destination for a "ADD", "COPY", or "RENAME" command, or whenever a file is opened with the LIB device module and the create open bit is specified.

A new version of a directory is created whenever the directory already exists and is the destination of a "MAKE", "COPY", or "RENAME" command.

The number of old versions to keep may be specified on a per-directory basis by means of the "CREATE", "MAKE", or "KEEP" commands.

For objects open for input, if the version number is omitted, the highest existing version number is used.

48.2.6. Hard Delete Attribute

Associated with each directory is the "hard delete attribute". If this attribute is set, deletions occur immediately, as under the UNIX operating system. If this attribute is clear, then deletions are deferred until the object is expunged, as under the TOPS-20 operating system.

48.2.7. Storage of Objects

Each library directory can have associated with it a host directory. If a host directory is associated with a library directory, then all library files contained in the library directory are actually stored in separate host files contained in the host directory. If a library directory has no host directory associated with it, then the library files contained in the library directory are stored within the library's base file.

If library files are stored within the base file they are protected from independent manipulation by host operating system commands (such as the host operating system's delete or rename commands), but this may lead to very large base files since the base file must be big enough to hold the library files and the image of the directory structure.

If the library files are stored as separate host files, they are not protected from independent manipulation by host operating system commands. This situation can lead to corruption of the library, but yields a small base file since the base file need be only large enough to hold the image of the directory structure.

[Currently, library files stored as separate host files are stored in the host directory using a host file name of the form "z<xxx>.lfi", where <xxx> is a unique pattern generated by LIB. Host files with names of this form must not be independently manipulated or library corruption will result. The format of the host file name is subject to change.]

48.2.8. The "lparms" File

LIB maintains a parameters file, called "lparms". This file stores definitions and other information from run to run.

If the host directory to which the user is connected when running LIB contains no "lparms" file, then LIB looks for "lparms" on the user's home directory (as returned by the system procedure $homeDirectory). Then, if no "lparms" file exists on either of these directories, LIB creates one on the host directory to which the user is connected when running LIB.

LIB maintains definitions made with the "DEFINE" command in "lparms". LIB updates the "lparms" file whenever the "SAVE", "EXPUNGE", or "QUIT" commands are executed, or whenever LIB is disposed, or MAINSAIL exited.

The user may explicitly edit "lparms" to add LIB commands to be executed whenever LIB is initialized.

48.3. Device Module Interface

Since LIB is a device module, the files stored in a library can be directly accessed by any MAINSAIL program through MAINSAIL's standard open procedure. If a library file is opened for create and the library's base file does not exist and no host file exists with the same name as the name of the new library's base file, LIB automatically creates the base file.

[LIB does not currently support automatic creation of directories within a library; only the root directory is automatically created when the library's base file is created. All other directories must be explicitly created with the "MAKE" command. A future version of LIB may provide for automatic creation of directories.]

To open a library file from a program, use the fully qualified name of the library file prefixed by "LIB", as shown in Figure 48.3-1.

```
open(f,"lib(foo.lib)>/bar;3/baz;5/sample.msl;7",input);
```

Figure 48.3-1. How to Open a Library File from a Program

To specify that an existing application, e.g., the MAINSAIL compiler, use a library file, use the standard device module syntax as shown in Figure 48.3-2.

```
compile (? for help):
    lib(foo.lib)>/bar;3/baz;5/sample.msl;7<eol>
```

Figure 48.3-2. How to Specify a Library File to an Application

For example, Figure 48.3-2 shows a library file used as input to the MAINSAIL compiler. There is no need to "EXTRACT" the source file from the library before compiling it. The output of the compiler could also be directed back into the library. This same syntax may be used to execute modules contained in a library.

[In Version 12 of MAINSAIL, if you edit a library file with MAINEDIT while running on an operating system that does not support version numbers, MAINEDIT issues a bogus query:

```
"Replace existing file foo.msl;1 (Yes or No): "
```

You should normally answer yes to this query; LIB automatically creates a new version of the library file and does not replace the existing version. A future version of MAINEDIT may eliminate this bogus query.]

48.4. Program Interface

48.4.1. doCommandsInFile and doCommandsInString

```
BOOLEAN
PROCEDURE   doCommandsInFile
                        (POINTER(textFile) userCmdFile;
                         OPTIONAL POINTER(textFile)
                             userLogFile;
                         OPTIONAL LONG BITS ctrlBits);

BOOLEAN
PROCEDURE   doCommandsInString
                        (STRING userCmdString;
                         PRODUCES OPTIONAL STRING
                             userLogString;
                         OPTIONAL LONG BITS ctrlBits);
```

Table 48.4.1-1. doCommandsInFile and doCommandsInString

[A module that calls doCommandsInFile or doCommandsInString must include their declarations with the MAINSAIL directive shown in Figure 48.4.1-2. This directive may change or be eliminated in subsequent versions of LIB.]

```
SOURCEFILE "msl:libhdr";
```

Figure 48.4.1-2. Accessing LIB's Program Interface

Except for device module functions, all LIB functions are invoked by executing the commands specified in Section 48.5. Commands can be executed by calling either doCommandsInFile or doCommandsInString.

Normally doCommandsInFile and doCommandsInString print LIB's herald each time they are called. If the suppressHerald bit is set in ctrlBits, the printing of LIB's herald is suppressed.

48.4.1.1. doCommandsInFile

doCommandsInFile reads the commands from userCmdFile and writes its output to userLogFile. If these parameters are omitted, cmdFile and logFile are used instead. doCommandsInFile processes commands until it hits end of file on userCmdFile or encounters an error condition when userCmdFile is not cmdFile. It returns true if no errors were encountered, or false if any error was encountered and userCmdFile is not cmdFile.

48.4.1.2. doCommandsInString

doCommandsInString reads the commands from userCmdString and writes its output to userLogString. If userCmdString contains more than one command, each command is separated from the one following it by an eol. doCommandsInString processes commands until it exhausts userCmdString or encounters an error condition. It returns true if no errors were encountered, or false if any error was encountered.

If useMemFiles is set in the ctrlBits parameter, the userCmdString is copied to a MEM file and then doCommandsInFile is called with this MEM file as userCmdFile and another MEM file as userLogFile. This provides a way for doCommandsInString to provide input to modules other than LIB that read input from cmdFile and write output to logFile. Upon return from doCommandsInFile, the contents of the userLogFile MEM file are converted to a string and produced as userLogString.

doCommandsInString normally fails if userLogString approaches the maximum length of a MAINSAIL string. If windowUserLogString is set in the ctrlBits parameter, the userLogString is "windowed"; a message is prepended, then a sliding window approximately 32K characters long is maintained that shows the most recent characters written to userLogString (older characters are discarded).

48.4.2. Exiting MAINSAIL and Disposing of LIB

When the system procedure exit is called or when LIB is disposed, LIB must make sure that the base files of all open modified libraries are updated.

When exiting, LIB waits until all final procedures have been called by MAINSAIL and then disconnects the source and destination directories, forces any open libraries closed, and updates the "lparms" file.

In addition to the processing it does during exit, LIB must force close all library files that are open whenever LIB is disposed. The actual information written to the disk for such files is

undefined. This also means that application programs that use library files may attempt to do I/O on a file that was forced close by LIB. In general, disposing of LIB should be avoided.

48.4.3. Library Integrity

A library's base file is the ultimate repository for information contained in the library. During execution, however, LIB keeps the library's directory in memory. LIB writes file pages to the base file as necessary, but writes the directory to the base file only when the library is closed, when MAINSAIL is exited, LIB is disposed, or the "SAVE", "EXPUNGE", or "QUIT" command is executed.

This means that if a library is modified, but MAINSAIL is aborted before the directory is written to disk, the modifications are lost. The prior contents of the library should be preserved, however, because LIB is careful about how it updates the disk. It writes file pages and new directories only to disk pages that contained no active information in the prior directory, and only after writing the new directory updates the library's header page to point to the new directory disk image.

48.5. LIB Commands

All of LIB's functions, except for device module functions, are implemented through "commands". Commands can be executed by calling either doCommandsInFile or doCommandsInString (see Section 48.4.1).

Whenever a library file is specified in a command, it is specified by means of either a fully qualified name or a relative name that produces a fully qualified name when combined with the current directory connections. In general you should not include the "LIB" device module prefix when specifying library files in commands.

In the following description of command syntax, the notation {s} means that s may either be omitted or specified exactly once. The notation {s}+ means that s must be specified one or more times.

48.5.1. Command Lines

doCommandsInFile and doCommandsInString both process lines of text containing commands, arguments to commands, and switch values. Command lines are broken up into words separated by blanks. The first word of a command line must be an unambiguous specification of a command. In general, only enough of a command need be specified to distinguish it from all other commands, but is recommended that in non-interactive situations the whole command be used so that the future addition of commands does not make the partial command ambiguous.

48.5.2. Switches

Certain commands, e.g., "LS", accept "switches", which are modifiers to govern the detailed operation of the command. Switch values are specified as single characters, or pairs of characters, within a word that must start with the character "-". Case is ignored when interpreting switches. Switch words may appear as any word but the first on a command line and there may be multiple switch words on the same command line. [Switch syntax may change in future versions.]

48.5.2.1. Mode Switches

The switches shown in Figure 48.5.2.1-1 may be applied to any command.

Switch Value	Meaning
C	Set confirmation mode for duration of command
NC	Clear confirmation mode for duration of command
V	Set verbose mode for duration of command
NV	Clear verbose mode for duration of command

Figure 48.5.2.1-1. Mode Switches

The "C" switch sets confirmation mode for the duration of the command regardless of the current confirmation mode setting. The setting of the current confirmation mode is unchanged after the command terminates.

The "NC" switch clears confirmation mode for the duration of the command regardless of the current confirmation mode setting. The setting of the current confirmation mode is unchanged after the command terminates.

The "V" switch sets verbose mode for the duration of the command regardless of the current verbose mode setting. The setting of the current verbose mode is unchanged after the command terminates.

The "NV" switch clears verbose mode for the duration of the command regardless of the current verbose mode setting. The setting of the current verbose mode is unchanged after the command terminates.

48.5.3. Connection Commands

LIB supports the notion of "connecting" to a directory. Connecting to a directory eliminates the need to supply the full path name to access objects contained in the directory. LIB supports the idea of separately connecting the source and destination of operations, so that, for example, you can copy files from one directory (the source) to a different directory (the destination) without having to type the full path name of either directory.

Whenever you connect to a library, that library's base file is opened. You cannot delete a library to which you are currently connected.

The "CONNECT" and "CD" commands perform the same function. They connect both the source and destination directories to the directory s. If the connection succeeds, then the named

```
CONNECT s       Connect both source and destination to
                  directory s.
CD s            Same as CONNECT s.
SRCCONNECT s    Connect source to directory s.
DSTCONNECT s    Connect destination to directory s.
PWD             Print current connections.
```

Figure 48.5.3-1. "CONNECT", "CD", "SRCCONNECT", "DSTCONNECT", and "PWD" Commands

directory is known as the "connected directory" and the base file containing such directory is known as the "connected base file".

If s is omitted, the current connections of both the source and destination directories are disconnected. If s is specified, then it can be either a fully qualified name, or a relative name, of a directory.

Any directory to which a connection is to be made must already exist, and be accessible, or the connection fails. Once connected, the associated library base file is left open until the connection is broken.

The "SRCCONNECT" command operates the same way as the "CONNECT" command, but affects only the source connection.

The "DSTCONNECT" command operates the same way as the "CONNECT" command, but affects only the destination connection.

The "PWD" command prints the current source and destination connections.

48.5.4. Definition Commands

Because fully qualified names can become quite long, LIB supports the notion of a "logical name"; a (typically) short string that can be used in place of another (typically) longer string.

The "DEFINE" command defines a logical name. When two strings are specified, the first string is defined to be a logical name with the value of the second string. Whenever LIB is parsing a library file name and finds that the initial portion of such a name is a logical name, followed by the character ":", the logical name and the ":" are stripped off and replaced by the logical name's definition.

```
DEFINE s t      Use t whenever s: is used as a prefix.
UNDEFINE {s}+   Cease applying s: as a prefix.
```

Figure 48.5.4-1. "DEFINE" and "UNDEFINE" Commands

If only one string is supplied, LIB prints the current definition (if any) of the logical name specified by the string. If no strings are supplied, LIB prints all logical name definitions.

For example, if the command "DEFINE fl (foo.lib)>/" were given, then the name "fl:baz" would reference "(foo.lib)>/baz". Such definitions are saved in the "lparms" file.

The "UNDEFINE" command undefines the logical name specified by the argument string(s).

48.5.5. Directory Information Commands

[The format and contents of the directory listing will likely change in future versions.]

```
DIRECTORY {s}+ Print directory information for s.
LS {s}+        Same as DIRECTORY {s}+.
```

Figure 48.5.5-1. "DIRECTORY" and "LS" Commands

The commands "DIRECTORY" and "LS" operate identically. They list directory information about the file or directory s.

48.5.5.1. Directory Information Command Switches

In addition to the switches specified in Figure 48.5.2.1-1, the "DIRECTORY" and "LS" commands accept the switches shown in Figure 48.5.5.1-1.

Normally "DIRECTORY" and "LS" produce a listing that contains only non-deleted objects. The "D" switch instead produces a listing that contains only deleted objects.

Switch Value	Meaning
D	Select only deleted objects.
S	Produce short listing.
L	Produce long listing.

Figure 48.5.5.1-1. "DIRECTORY" and "LS" Switches

Normally "DIRECTORY" and "LS" produce a listing, e.g., Figure 48.5.5.1-2, that contains a single line of information per object. The "S" switch produces a short listing that gives only the name and version of each object. The "L" switch produces a long listing that contains virtually all of the information LIB has about each object.

[All of the formats and information content of "LS" commands are subject to change.]

```
ROOT;1              5:44:10 19-Oct-86 greg      DSL         3
  bar;1             5:00:59 19-Oct-86 greg      DSL         1
  sample.msl;1      4:32:30 19-Oct-86 greg      FTL      3072
```

Figure 48.5.5.1-2. Normal Listing Produced by "DIRECTORY" and "LS"

The normal listing produced by "DIRECTORY" and "LS" contains for each object:

- The name and version of the object.
- The date, time, and user of the last modification of the object.
- Attributes of the object. Figures 48.5.5.1-3 and 48.5.5.1-4 show how to interpret the object attributes listed.

The short listing produced by "DIRECTORY" and "LS", shown in Figure 48.5.5.1-5, lists the name and version of each selected object.

The long listing shown in Figure 48.5.5.1-6 produced by "DIRECTORY" or "LS" includes the line produced by a normal "DIRECTORY" or "LS" plus a line showing the time, date, and user when the object was created, plus either a host directory name, for a directory object, or a host

```
123     Meaning
D       Indicates that this is a directory object.
 H      Indicates that the hard delete attribute is
          set in this directory.
 S      Indicates that the hard delete attribute is
          cleared in this directory.
  L     Indicates that files created in this directory
          will be stored within the library's base file.
  H     Indicates that files created in this directory
          will be stored as separate host files.

The directory's size is indicated in terms of the number
of files contained in the directory.
```

Figure 48.5.5.1-3. Directory Object Attributes

```
123     Meaning
F       Indicates that this is a file object.
 D      Indicates that this is a data file.
 T      Indicates that this is a text file.
  L     Indicates that this file is stored within
          the library's base file.
  H     Indicates that this file is stored in a
          separate host file.

File sizes are listed in terms of storage units for
data files, and characters for text files.
```

Figure 48.5.5.1-4. File Object Attributes

```
ROOT;1
  bar;1
  sample.msl;1
```

Figure 48.5.5.1-5. Short Listing Produced by "DIRECTORY" and "LS"

file name, for a file object. For directories, "INF" is printed if the directory will keep infinite versions of objects, or a number indicating how many versions of objects are kept.

For directories, the host directory name tells where host files are created whenever a library file is created in the library directory. If omitted, library files created within the directory are stored as part of the library's base file.

For files, the host file name specifies where the library file is stored. If omitted, the library file is stored as part of the library's base file.

```
ROOT;1              5:44:10 19-Oct-86 greg      DSL         3
                INF 8:43:25 18-Oct-86 greg
  bar;1             5:00:59 19-Oct-86 greg      DSL         1
                  1 4:32:15 19-Oct-86 greg
  sample.msl;1      4:32:30 19-Oct-86 greg      FTL      3072
                    4:32:29 19-Oct-86 greg
```

Figure 48.5.5.1-6. Long Listing Produced by "DIRECTORY" and "LS"

48.5.6. Library Creation Command

```
CREATE s {t}   Create a library with file name s.
```

Figure 48.5.6-1. "CREATE" Command

The "CREATE" command creates a new library with host file name s. By default the root directory of the newly created library has the hard delete attribute cleared and will keep infinite versions.

If t is not specified, files created in the root directory of s are stored within the library's base file. If t is specified, then it must be the full path name of a host directory in which LIB is to store files created in the root directory of s.

"CREATE" fails if the library is already open, e.g., you are connected to a directory contained within it.

On operating systems that do not support version numbers, the user is queried for confirmation to overwrite s if s currently exists and confirmation mode is set (see Section 48.5.15.1). If the host file contains a valid object library, all objects contained in the library are expunged before the base file is overwritten.

Switch Value	Meaning
H	Set hard delete attribute in root directory
S	Clear hard delete attribute in root directory
<n>	Keep <n> versions in the root directory
I	Keep infinite versions in new directory

Figure 48.5.6-2. "CREATE" Switches

In addition to the mode switches specified in Figure 48.5.2.1-1, the "CREATE" command accepts the switches shown in Figure 48.5.6-2.

The "H" switch specifies that the root directory of the newly created library should have the hard delete attribute set.

The "S" switch specifies that the root directory of the newly created library should have the soft delete attribute set. Since soft delete is the default, this switch is unnecessary, but is included for completeness and so that "CREATE" is parallel to "MAKE".

A numeric switch value is treated as the number of of versions to keep in the newly created directory, while the "I" switch value specifies that an infinite number of versions should be kept in the newly created directory. If neither a numeric switch nor an "I" switch is specified, the newly created directory inherits the number of versions to keep from its parent.

Different versions of directories are completely separate objects; the sets of files in them are disjoint.

48.5.7. Directory Creation Commands

```
MAKE s {t}   Make a new directory s on host directory t.
MKDIR s {t}  Same as MAKE s {t}.
```

Figure 48.5.7-1. "MAKE" and "MKDIR" Commands

The "MAKE" and "MKDIR" commands operate identically. They create a new directory by the name of s. s may be either the full path name of a directory or a directory name relative to the current destination connection.

If t is specified, then it must be the full path name of a host directory in which LIB is to store files created in s. If t is not specified, then the directory s inherits from its parent where to store files created in s.

Switch Value	Meaning
H	Set hard delete attribute in new directory
S	Clear hard delete attribute in new directory
<n>	Keep <n> versions in new directory
I	Keep infinite versions in new directory

Figure 48.5.7-2. "MAKE" Switches

In addition to the mode switches specified in Figure 48.5.2.1-1, the "MAKE" command accepts the switches shown in Figure 48.5.7-2.

The "H" switch specifies that the newly created directory should have the hard delete attribute set instead of inheriting this attribute from its parent directory.

The "S" switch specifies that the newly created directory should have the hard delete attribute cleared instead of inheriting this attribute from its parent directory.

A numeric switch value is treated as the number of of versions to keep in the newly created directory, while the "I" switch value specifies that an infinite number of versions should be kept

in the newly created directory. If neither a numeric switch or an "I" switch is specified, then the newly created directory inherits the number of versions to keep from its parent.

48.5.8. Host File Manipulation Commands

```
ADDTEXT s t     Add host text file s as t.
ADDDATA s t     Add host data file s as t.
EXTRACT s t     From s, extract host file t.
```

Figure 48.5.8-1. "ADDTEXT", "ADDDATA", and "EXTRACT" Commands

The "ADDTEXT" command adds the host text file s to a library under the name t by copying the host file to the library file.

The "ADDDATA" command adds the host data file s to a library under the name t by copying the host file to the library file.

The "EXTRACT" command extracts the library file s into the host file t. On operating systems that do not support version numbers the user is queried for confirmation to overwrite t if t currently exists and the confirmation option is set (see Section 48.5.15.1).

48.5.9. Object Copying Commands

```
COPY s t        Copy s to t.
CP s t          Same as COPY s t.
```

Figure 48.5.9-1. "COPY" and "CP" Commands

The "COPY" and "CP" commands operate identically. They create a copy of object s with the name t. If s is a directory all objects in s are recursively copied. If an object named t of the same type already exists, a new version is created.

Objects can be copied across directories and libraries. Objects cannot be copied to a host file.

The modify date, time, and user of the copied object are identical to that of the original object, but that the create date, time, and user are those in effect when the copy operation is performed (this means the modify date and time may precede the create date and time).

48.5.10. Object Renaming Commands

```
RENAME s t      Rename s to t.
MV s t          Same as RENAME s t.
```

Figure 48.5.10-1. "RENAME" and "MV" Commands

The "RENAME" and "MV" commands operate identically. They change the name of object s to be object t. If an object name t of the same type already exists, a new version is created.

Objects can be renamed across directories and libraries. Objects cannot be renamed to a host file. When an object is renamed across libraries it is equivalent to a copy followed by a delete. When an object is renamed within the same library, only the directory data structure is modified.

The modify date, time, and user of the renamed object are identical to that of the original object, but that the create date, time, and user are those in effect when the rename operation is performed.

48.5.11. Object Removal Commands

These commands remove objects from a library.

```
DELETE {s}+     Mark s for deletion.
RM {s}+         Same as DELETE {s}+.
DROP {s}+       Mark superfluous versions of s for
                  deletion.
UNDELETE {s}+   Unmark s for deletion.
EXPUNGE {s}+    Expunge s.
```

Figure 48.5.11-1. "DELETE", "RM", "DROP", "UNDELETE", and "EXPUNGE" Commands

The "DELETE" and "RM" commands operate identically. They mark the object s for deletion, and if the directory containing s has the hard delete attribute set, expunge it as well.

The "DROP" command deletes all versions of the object s except the highest-numbered version. [Future enhancements to LIB may make it necessary to retain certain older versions.] "DROP" does not expunge the deleted objects (unless they are contained in a directory with the hard delete attribute).

If a "DROP" or "DELETE" is attempted on a non-empty directory, and the confirmation option is set, then the user is queried for confirmation before dropping or deleting the directory.

The "UNDELETE" command unmarks the object s for deletion. This has an effect only if the directory containing s has the hard delete attribute cleared and if the object has not been expunged.

The "EXPUNGE" command expunges the object s from the library. Expunging an object removes it completely from the library so that it can never be recovered. When a library file stored as a host file is expunged, the host file storing the library file is deleted. Objects are automatically expunged by the "DELETE", "RM", and "DROP" commands for directories which have the hard delete attribute set.

The "DROP", "UNDELETE", and "EXPUNGE" commands accept "*" as an element name indicating "all". For example, the command "DROP foo/*" drops old versions of all objects contained in the directory "foo", whereas the command "DROP foo" drops old versions of the object "foo". [In a future release more general wildcards may be supported by these commands.]

48.5.12. Deletion Control Commands

These commands manipulate the directory attributes to control soft and hard delete.

```
HARDDELETE s    Perform hard deletes within directory s.
SOFTDELETE s    Perform soft deletes within directory s.
```

Figure 48.5.12-1. HARDDELETE and SOFTDELETE Commands

The "HARDDELETE" command sets the hard delete attribute for the directory s and expunges s. Subsequent deletions from s immediately cause an expunge on the deleted object.

The "SOFTDELETE" command clears the hard delete attribute for the directory s. Subsequent deletions from s mark the object for deletion, but not expunge it. An explicit "EXPUNGE" command must be given before the object is actually expunged from the library. Objects marked for delete may be recovered with an "UNDELETE" command, if it is given before the object is expunged.

48.5.13. Version Control Command

```
KEEP n {s}+      Keep n versions in directory s.
```

Figure 48.5.13-1. KEEP Command

The "KEEP" command modifies how many versions are to be kept in directory s. n must be either a number, in which case n versions are kept, or an initial substring of "INFINITE", in which case an inifinite number of versions are kept. If a number is specified, "KEEP" drops any versions in excess of this number.

48.5.14. Save Command

```
SAVE {s}+        Save library s.
```

Figure 48.5.14-1. SAVE Command

The "SAVE" command updates library base files with the current state of their memory-resident directories and updates the "lparms" file. If s is specified, it must be the name of a library to be saved. If s is not specified, then all open modified libraries are saved.

48.5.15. Mode Commands

These commands affect the global operation of LIB; they control its "modes".

48.5.15.1. Confirmation Commands

LIB implements the notion of a "confirmation mode". When in confirmation mode, LIB queries the user for confirmation before performing operations that might be unintended by the user. For example, on operating systems that do not support version numbers for host files, when confirmation mode is set, the user is asked for confirmation before extracting a library into a host file that already exists. When confirmation mode is cleared, such an "EXTRACT" overwrites the existing host file without querying the user for confirmation.

```
CONFIRM         Set confirmation mode.
NOCONFIRM       Clear confirmation mode.
```

Figure 48.5.15.1-1. "CONFIRM" and "NOCONFIRM" Commands

The "CONFIRM" command sets confirmation mode. This is the default.

The "NOCONFIRM" command clears confirmation mode.

48.5.15.2. Verbosity Commands

LIB implements the notion of "verbose mode", which is the default. When verbose mode is set, LIB prints verbose information about what it is doing. This is useful for novice users and when performing potentially dangerous operations. When this mode is cleared, LIB suppresses much of the information it prints when in verbose mode. This is often useful for expert users. [Currently LIB produces the same output regardless of whether verbose mode is set or cleared.]

```
VERBOSE         Set verbose mode.
NOVERBOSE       Clear verbose mode.
```

Figure 48.5.15.2-1. "VERBOSE" and "NOVERBOSE" Commands

The "VERBOSE" command sets verbose mode.

The "NOVERBOSE" command clears verbose mode.

48.5.16. Termination Commands

```
QUIT            Quit Object Librarian.
EXIT            Exit to the operating system.
```

Figure 48.5.16-1. "QUIT" and "EXIT" Commands

The "QUIT" command quits LIBEX, returning to whichever module originally invoked LIBEX, e.g., MAINEX.

The "EXIT" command quits LIBEX and exits to the operating system through the MAINSAIL system procedure.

48.5.17. Input Redirection Commands

The "READ" command tells LIBEX to read its commands from the file s.

```
READ s          Execute LIB commands contained in file s.
```

Figure 48.5.17-1. "READ" Command

48.5.18. Module Execution Command

The "EXECUTE" command executes an arbitrary MAINSAIL module using the system procedure $invokeModule. Accordingly, s can be either the name of a module or the name of a file containing the module, and "," can be appended to the name to enter MAINEX subcommand mode before executing the module.

```
EXECUTE s           Execute MAINSAIL module s.
```

Figure 48.5.18-1. "EXECUTE" Command

If MAINEX subcommand mode is entered, or if the executed module uses cmdFile and logFile, output is directed to logFile and input obtained from cmdFile, even if the "EXECUTE" command is issued from doCmdsInString. If doCmdsInString needs to supply information to be read by cmdFile, or record information written to logFile, then the useMemFiles bit must be set in the call to doCmdsInString.

48.5.19. Maintenance and Diagnostic Commands

These commands are primarily used by maintenance personnel and may change or cease to be supported in future versions without notice.

```
STATUS          Print LIB's status.
PAGEMAP s       Print a page map of library s.
PAGESUMMARY s   Print a page summary of library s.
HEADER {s}+     Print library header page from host file s.
```

Figure 48.5.19-1. "HEADER", "PAGEMAP", "PAGESUMMARY", and "STATUS" Commands

The "STATUS" command prints a summary of LIB's status. Currently this information includes a listing of all open libraries and the number of references to each. "STATUS" also prints the current connections and current mode settings.

The "PAGEMAP" command prints out a page map of the library s. The page map shows free pages, the pages occupied by the directory data structure, and the pages used for objects stored within the library's base file.

The "PAGESUMMARY" command prints a short summary of page usage in the base file of library s, including the total number of pages, total number of free pages, number of trailing free pages, and number of pages occupied by the directory structure. The library s must not be open at the time the "PAGESUMMARY" command is given.

The "HEADER" command prints the library header page from the host file s. This information contains the consistency checking information LIB uses to make sure it has a base file as well as the major and minor MAINSAIL and LIB version numbers.

48.6. LIBEX

LIBEX is a module used to invoke "doCommandsInFile(cmdFile,logFile)" from MAINEX. LIBEX is required because MAINEX unbinds the modules it executes after they finish executing. LIB should not be unbound after returning from doCommandsInFile because files might still be open that are using it as a device module. Thus, always invoke LIBEX from MAINEX; never directly invoke LIB from MAINEX.

48.6.1. Sample LIBEX Execution

Example 48.6.1-1 shows a sample execution of LIBEX, followed by access to files within a library from the compiler. The example is explained in detail using the line numbers in the left-hand columns.

1. LIBEX is invoked from MAINEX.

2. The user creates a library with a base file named "foo.lib". Since a no-confirm switch was specified, if a host file named "foo.lib" that is not a valid library base file already exists, it is overwritten. If a valid library base file named "foo.lib" already exists, all objects it contains are expunged and the library deleted.

3. The user defines a logical name to reference the root directory of "foo.lib".

4. The user connects to the root directory of foo.lib using the logical name "fl:".

5. The user gets a directory listing of the root directory of "foo.lib".

6. The user adds a host text file, "sample.msl", to the root directory of "foo.lib" with the name "sample.msl".

7. The user saves all open modified libraries, at this point only "foo.lib".

8. The user copies the library file, "sample.msl", to itself. Since the root directory is keeping infinite versions, a new copy of "sample.msl" is created and the prior version retained.

9. The user makes a new directory, named "bar", in the root directory of "foo.lib". Since no switches were specified to the "MAKE" command, the new directory, "baz", inherits all attributes of the root directory of "foo.lib".

10. The user connects to the new directory, "bar".

```
 1 *libex
 1 LIB: Version 1.2 Alpha Release (? for Help)
 2 LIB: cr -nc foo.lib<eol>
 2 Created library foo.lib
 3 LIB: def fl (foo.lib)>/<eol>
 3 Defined fl as (foo.lib)>/
 4 LIB: cd fl:<eol>
 4 Src connected to  (foo.lib)>/
 4 Dst connected to  (foo.lib)>/
 5 LIB: ls<eol>
 5 ROOT;1              13:35:15 11-Mar-87 greg      DSL        0

 6 LIB: addt sample.msl sample.msl<eol>
 6 Added text file sample.msl as (foo.lib)>/sample.msl;1
 7 LIB: save<eol>
 7 Saved foo.lib
 8 LIB: cp sample.msl sample.msl<eol>
 8 (foo.lib)>/sample.msl;1 copied to (foo.lib)>/sample.msl;2
 9 LIB: make bar<eol>
 9 Made directory (foo.lib)>/bar;1/
10 LIB: cd bar<eol>
10 Src connected to  (foo.lib)>/bar;1/
10 Dst connected to  (foo.lib)>/bar;1/

11 LIB: ls<eol>
11 bar;1              13:35:17 11-Mar-87 greg      DSL        0
12 LIB: ls ..<eol>
12 ROOT;1             13:35:17 11-Mar-87 greg      DSL        3
12   bar;1            13:35:17 11-Mar-87 greg      DSL        0
12   sample.msl;2     13:35:16 11-Mar-87 greg      FTL     2712
12   sample.msl;1     13:35:16 11-Mar-87 greg      FTL     2712
13 LIB: cp ../sample.msl sample.msl<eol>
13 (foo.lib)>/sample.msl;2 copied to
     (foo.lib)>/bar;1/sample.msl;1
14 LIB: cd ..<eol>
14 Src connected to  (foo.lib)>/
14 Dst connected to  (foo.lib)>/
15 LIB: drop<eol>
15 Marked (foo.lib)>/sample.msl;1 for delete
```

Example 48.6.1-1. LIBEX Execution (continued)

```
16 LIB: ls -ds<eol>
16   sample.msl;1
17 LIB: exp<eol>
18 LIB: ls -l<eol>
18 ROOT;1              13:35:18 11-Mar-87 greg      DSL        3
18                INF 13:35:15 11-Mar-87 greg
18   bar;1             13:35:18 11-Mar-87 greg      DSL        1
18                INF 13:35:17 11-Mar-87 greg
18   sample.msl;2      13:35:16 11-Mar-87 greg      FTL     2712
18                     13:35:17 11-Mar-87 greg
19 LIB: cp bar bar<eol>
19 (foo.lib)>/bar;1/ copied to (foo.lib)>/bar;2/
20 LIB: ls bar<eol>
20 bar;2               13:35:19 11-Mar-87 greg      DSL        1
20   sample.msl;1      13:35:16 11-Mar-87 greg      FTL     2712

21 LIB: make -lh baz tmp<eol>
21 Made directory (foo.lib)>/baz;1/
22 LIB: cp sample.msl baz/sample.msl<eol>
22 (foo.lib)>/sample.msl;2 copied to
     (foo.lib)>/baz;1/sample.msl;1
23 LIB: cp sample.msl baz/sample.msl<eol>
23 Expunged (foo.lib)>/baz;1/sample.msl;1
23 (foo.lib)>/sample.msl;2 copied to
     (foo.lib)>/baz;1/sample.msl;2
24 LIB: ls -l baz<eol>
24 baz;1               13:35:20 11-Mar-87 greg      DHH        1
24                   1 13:35:19 11-Mar-87 greg      /tmp
24   sample.msl;2      13:35:20 11-Mar-87 greg      FTH     2712
24                     13:35:20 11-Mar-87 greg      /tmp/z8327.lfi
25 LIB: pwd<eol>
25 Src connected to  (foo.lib)>/
25 Dst connected to  (foo.lib)>/
```

Example 48.6.1-1. LIBEX Execution (continued)

```
26 LIB: execute compil<eol>
26 MAINSAIL (R) Compiler
26 Copyright (c) 1984, 1985, 1986, 1987, 1988, and 1989 by
26  XIDAK, Inc., Menlo Park, California, USA.
26 compile (? for help): lib>sample.msl,<eol>
26 > output lib>sample.um6<eol>
26 > <eol>
26 Opening intmod for $SYS from intlib(sys-um6.ilb)>sys-um68
26 LIB(foo.lib)>/sample.msl;2 1 2
26 Objmod for SAMPLE stored on LIB(foo.lib)>/sample.um6;1
26 Intmod for SAMPLE not stored
26 compile (? for help): <eol>
26 Back from compil
27 LIB: execute lib>sample.um6<eol>
27 Size of hash bucket: 131<eol>
27 Next key to be hashed (eol to stop): abc<eol>
27 36
   Next key to be hashed (eol to stop): <eol>
27 Back from lib>sample.um6
28 LIB:
```

Example 48.6.1-1. LIBEX Execution (end)

11. The user gets a normal directory listing of the directory "bar".

12. The user gets a normal directory of the parent directory of "bar" (which is the root directory of "foo.lib").

13. The user copies the file "sample.msl" from the parent directory of "bar" to the directory "bar" under the name, "sample.msl".

14. The user connects to the parent of "bar".

15. The user drops old versions of objects contained in the connected directory (which is the root directory of "foo.lib"). This marks the older version of "sample.msl" for deletion.

16. The user gets a long directory listing of files marked for deletion in the connected directory. This shows the older version of "sample.msl" deleted in the prior step.

17. The user expunges all files marked for deletion in the connected directory. This expunges the older version of "sample.msl" marked for deletion in prior steps.

18. The user gets a long directory listing of the connected directory. This shows only one version of the text file "sample.msl" and the directory, "bar".

19. The user copies the directory "bar" to a new version of itself. This copy is recursive, so the new version of "bar" contains a copy of the text file "sample.msl" contained in the prior version of "bar".

20. The user gets a normal directory listing of "bar".

21. The user makes a new directory, named "baz", in the currently connected directory. The user has specified switches to set the hard delete attribute in "baz", and keep only one version of objects contained in "baz". The user has also specified that files contained in "baz" be stored as separate host files in the host directory with relative host name "tmp".

22. The user copies the text file "sample.msl" to the new directory "baz". LIB has actually copied the contents of "sample.msl" to a unique file on the host directory "tmp".

23. The user again copies "sample.msl" to "baz". Since "baz" has the hard delete attribute set and is keeping only one version, the prior version of "sample.msl" is automatically deleted and expunged. As a result, a new unique file on the host directory "tmp" has been created, a copy from the prior host directory made, and then the prior copy deleted from "tmp".

24. The user gets a long directory listing of "baz", which shows that files contained in "baz" are stored in host directory with full path name of /tmp, and that the text file, "sample.msl", contained in "baz" is stored in the host file "/tmp/z8327.lfi".

25. The user asks that the current connections be listed.

26. The user invokes the MAINSAIL compiler using LIB's execute command. The user specifies as the source file the library file "(foo.lib)>/sample.msl" and as the objmod file the library file "(foo.lib)>/sample.um6". After the compilation is done, the user quits the compiler and returns to LIBEX.

27. The user then invokes the module contained in the library file, "(foo.lib)>/sample.um6".

48.7. Known Bugs and Problems in LIB

The following problems exist in LIB:

1. Because MAINSAIL's notion of file versions (as specified by "$attributes TST $hasFileVersions") is tied to the operating system's disk file characteristics, there are spurious "OK to overwrite existing file..." messages generated when LIB is used on operating systems that do not support file versions (e.g., UNIX). In fact, LIB creates a new version of the file rather than overwriting the specified file, regardless of the host system's attributes, except in directories in which the hard delete attribute is set and in which only one version is kept.

2. The MAINSAIL compiler's subcommand "OUTINTFILE" cannot specify the LIB device module because this subcommand uses the MAINSAIL $rename function and the $rename function cannot currently rename across device modules.

3. Caution should be used when exiting MAINSAIL or disposing of LIB when there are open, modified, libraries. LIB attempts to preserve the integrity of libraries, but there may be undetected circumstances that cause library corruption, particularly if LIB is disposed when there are open library files. Try to close explicitly all library files before exiting MAINSAIL or disposing of LIB.

49. LINCOM, Text File Comparer

Command line syntax: lincom {fileName1 {fileName2}}*

LINCOM compares two text files and displays the differences between the files on a per-line basis.

LINCOM accepts file names on the command line or prompts for the names of the two text files to be compared. If command line arguments are given, specify a comma after the second element of a pair to enter subcommand mode. If an odd number of file names are given on the command line, LINCOM prompts for the last fileName2. If no files are specified on the command line, LINCOM prompts with "Subcommands? (Yes or eol):". If "Y" (or "y") is typed in response to this prompt, LINCOM allows the user to set various options that control the comparison of the input files. The prompts displayed when the options are set are shown in Table 49-1.

```
Name of output file for differences (eol for logFile):
Automatic continuation? (Yes or eol):
Max display lines (eol for 23):
Lines to look ahead (eol for 200):
Nonblank lines to match (eol for 2):
Treat all blanks as significant? (Yes or eol):
```

Table 49-1. LINCOM Subcommand Prompts

The differences between the two files are written to an output file. The default output file is logFile.

The default for "Automatic continuation?" is "No". In this case, LINCOM prompts with "Continue? (eol or No):" after each difference it finds in the input files. If "Yes" is answered to the "Automatic continuation?" prompt, LINCOM lists all differences in the input files without prompting.

"Max display lines" is the largest number of lines displayed at a single difference. If the difference is longer than the specified number of lines, LINCOM prompts with "There's more

of this difference...want to see it? (eol or No):". If the user responds with <eol>, the remainder of the difference is shown; otherwise, LINCOM skips to the next difference.

"Lines to look ahead" is the maximum number of lines scanned in order to resynchronize the input files when a difference is found. The default is 200. If LINCOM cannot resynchronize after finding a difference (i.e., the number of lines that differ exceeds the lookahead value), LINCOM prints the message "Lookahead insufficient to synchronize:" and continues. If <eol> is typed in response to the "Continue? (eol or No):" prompt, LINCOM doubles the lookahead value and continues. In this case, a message indicating that the lookahead value has been doubled is displayed.

"Nonblank lines to match" is the number of nonblank lines that have to match before LINCOM considers the input files synchronized after finding a difference. The default is 2. Any blank lines between the matching nonblank lines must also match in order for LINCOM to consider the input files to be resynchronized.

The default for "Treat all blanks as significant?" is "No". In this case, LINCOM ignores leading and trailing blanks and tabs when comparing lines, and treats other sequences of blanks and tabs as a single blank. If "Yes" is answered to the "Treat all blanks as significant?" prompt, two lines being compared must be exactly the same in order for LINCOM to consider them to match.

When differences between two files are found, LINCOM prints an identification line for each file, giving the name of the file and the page and line number of the file followed by the text that differs from the other file. After all the differences between two files have been reported, LINCOM prompts with "Compare more files? (eol or No):".

Example 49-2 shows how to use LINCOM and what its output looks like. Since <eol> was entered in response to the "Subcommands" prompt, default values are used.

```
MAINSAIL (R) Version 12.10 (? for help)
Copyright (c) 1984, 1985, 1986, 1987, 1988, and 1989 by
 XIDAK, Inc., Menlo Park, California, USA.
*lincom<eol>
Input file 1 (just <eol> to stop): file1<eol>
Input file 2: file2<eol>
Subcommands? (Yes or eol): <eol>
-------- file1 page 1 line 37 ---------------
EN (aegis interface) m:aegis
EN (aegis saveon) m:aegis.svn

-------- file2 page 1 line 37 ---------------
EN (Aegis interface) m:aegis
EN (Aegis saveon) m:aegis.svn

Continue? (eol or No): <eol>
-------- file1 page 1 line 41 ---------------
EN (unix68 saveon) m:unix68.svn
EN (unix68 interface) m:unix68

-------- file2 page 1 line 41 ---------------
EN (unix68 interface) m:unix68
EN (unix68 saveon) m:unix68.svn
EN (Sun unix saveon) m:sun.svn
EN (HP unix saveon) m:hpunix.svn

Continue? (eol or No): <eol>
Input file 1 (just <eol> to stop): <eol>
*
```

Example 49-2. LINCOM Example

50. MAINEX, the MAINSAIL Executive

Command line syntax: {modName {args}}

MAINEX is the module that invokes user modules. Subcommands that govern the execution of the user module may be specified to MAINEX when a user module is invoked.

If the MAINSAIL bootstrap does not specify a module to execute (see Section 35.2.5) and no command line is given, MAINEX prints a herald and an asterisk prompt when MAINSAIL is invoked. The user may then specify the name of a MAINSAIL module or the name of a file containing a MAINSAIL object module (or a comment if the line begins with the pound sign ("#") character). The specified module is executed if it is found; otherwise, MAINEX informs the user that the module cannot be found and returns to the asterisk prompt.

If the bootstrap command string or command line is not null, the command line is appended to the command string, and module names and their arguments are read, one module per line of the resulting string, until the string is exhausted. MAINEX examines the command line only when the executive starts execution; it does not examine the command line value resulting from each module's execution, so it is not possible for a module to cause another module to be executed by MAINEX by setting the command line and then returning to MAINEX.

MAINEX executes a module by calling the MAINSAIL system procedure $invokeModule, which binds the module. Binding a module has the side effect of executing its initial procedure, which functions as the "main body" of the program. After execution of the module, MAINEX disposes the module's bound data section (but not the control section). MAINEX then prompts for the next module to be executed. Other module(s) brought into memory as a result of execution of the module specified to MAINEX are not automatically disposed. When MAINSAIL exits to the host operating system, the final procedure of each module that has not been disposed is executed.

If the module to be executed already has a bound data section, and has not been compiled with the subcommand "UNBOUND" (see the "MAINSAIL Compiler Release Note, Version 12"), then MAINSAIL issues a message that the module could not be invoked.

50.1. MAINEX Executive Dialogue

Whenever MAINEX gains control, it identifies itself and prints a "*" prompt to the file logFile (initially directed to the user's terminal) indicating that it is waiting for input. MAINEX reads

its input from the file cmdFile, which is also initially directed to the operating system's primary input medium (usually the user's terminal). Figure 50.1-1 shows possible responses to the prompt (these responses are displayed when "?" is typed).

```
<eol>       To end your dialogue with MAINSAIL
modName     To execute module modName
fileName    To execute module in file fileName

To enter subcommand mode, end the line with a comma.
```

Figure 50.1-1. MAINEX Commands

If <eol> is typed in response to the MAINEX "*" prompt, control is returned to the calling procedure, which is usually the host operating system.

Example 50.1-2 shows how to execute a module MYMOD. "MYMOD" is typed in response to the MAINEX "*" prompt. Case is ignored when a module name (but not a file name) is entered at the "*" prompt. MYMOD is typically contained in an open module library (see Section 50.2.25) or in a file of which the name is derived from the module name in an operating-system-dependent manner. The search rules for object modules are described in the "MAINSAIL Language Manual".

```
MAINSAIL (R) Version 12.10 (? for help)
Copyright (c) 1984, 1985, 1986, 1987, 1988, and 1989 by
 XIDAK, Inc., Menlo Park, California, USA.
*myMod<eol>

(The module myMod executes to completion.)
```

Example 50.1-2. Invoking a Module by Module Name

If the input to MAINEX is not a valid module name, MAINEX assumes that it is a file name. For example, if the user types "mymod.xxx", MAINEX assumes this is a file name since it is not a valid module name (it is more than six characters and contains a period). In this case, MAINEX attempts to execute the object module contained in the file "mymod.xxx", as shown in Example 50.1-3.

```
MAINSAIL (R) Version 12.10 (? for help)
Copyright (c) 1984, 1985, 1986, 1987, 1988, and 1989 by
 XIDAK, Inc., Menlo Park, California, USA.
*mymod.xxx<eol>

(The module in "mymod.xxx" executes to completion.)
```

Example 50.1-3. Invoking a Module by File Name

50.2. MAINEX Subcommands

MAINEX subcommand mode is entered by ending a line with a comma (","). MAINEX prompts with ">" and waits for input. Table 50.2-1 lists the available MAINEX subcommands; a list similar to this is printed when "?" is typed to the ">" prompt.

Except in file names, case is ignored by MAINEX when interpreting subcommand input. In addition, only as much as is necessary to uniquely identify a subcommand need be typed. Subcommand mode is exited by typing <eol> to the ">" prompt.

Mixed case is used in the subcommands to show the minimum permitted abbreviation of MAINEX subcommands. The uppercase portion of each MAINEX subcommand in mixed case is a sufficient abbreviation for that subcommand (as if the upperCase bit were given to cmdMatch).

<eol>	Exit subcommand mode
CHECKCONSISTENCY	Check module interface consistency (default)
CLoseexelib f	Close objlib f for module execution
CLOSEINTLIB f	Close intlib f for intmod access
CLOSEOBJLIB f	Close objlib f for objmod access
CMDFILE f	Open file f as the new cmdFile
CONTROLINFO	Print coroutine, exception info
CSUBCOMMANDS f	Read subcommands from file f, if f exists
DEFINETIMEZONE s n	s is name for time zone with GMT offset n
DSTENDRULE s	Daylight savings time ends at time described by s
DSTNAME s	Set name of local time zone used during daylight savings time
DSTOFFSET n	Set daylight savings time offset from GMT
DSTSTARTRULE s	Daylight savings time starts at time described by s
ECHOCMDFILE	Echo cmdFile, even if not redirected
ECHOifredirected	Echo redirected logFile and cmdFile I/O on tty
ENTER ln tn	Use true name tn for logical name ln
ENTER ln	Undo current ENTER for name ln

Table 50.2-1. MAINEX Subcommands (continued)

EXEDEFAULT m ...	exeSearch: default search for m's objmod
EXEFILE m{=f} ...	exeSearch: get m's objmod from file f
EXEFILE	exeSearch: search files before objlibs
EXELIB m{=f} ...	exeSearch: get m's objmod from objlib f
EXELIB	exeSearch: search objlibs before files
EXESHOW {m ...}	Show entries {for m ...} in exeList
FILEINFO	Print file open, close, delete, rename info
GMTOFFSET n	Set offset of local time zone from GMT
INTDEFAULT m ...	intSearch: default search for m's intmod
INTFILE m{=f} ...	intSearch: get m's intmod from file f
INTFILE	intSearch: search files before intlibs
INTLIB m{=f} ...	intSearch: get m's intmod from intlib f
INTLIB	intSearch: search intlibs before files
INTSHOW {m ...}	Show entries {for m ...} in intList
LOGFILE f	Open file f as the new logFile
LOOKUP ln	List the ENTER association for ln
LOOKUP	List all ENTER associations
MAP n	Print memmap every nth time it's changed
MAP	Print memmap along with MEMINFO
MEMINFO	Print memory management info
OBJDEFAULT m ...	objSearch: default search for m's objmod

Table 50.2-1. MAINEX Subcommands (continued)

OBJFILE m{=f} ...	objSearch: get m's objmod from file f
OBJFILE	objSearch: search files before objlibs
OBJLIB m{=f} ...	objSearch: get m's objmod from objlib f
OBJLIB	objSearch: search objlibs before files
OBJSHOW {m ...}	Show entries {for m ...} in objList
OPenexelib f	Open objmod f for module execution
OPENINTLIB f	Open intlib f for intmod access
OPENLIBRARY f	Same as OPENEXELIB f
OPENOBJLIB f	Open objlib f for objmod access
RESPONSE	Get user response to error messages
SEARCHPATH r s	Set searchpath for pattern r to be s
SEARCHPATH r	Undo current SEARCHPATH for pattern r
SEARCHPATH	List all SEARCHPATHs
SETFILE m f	Get module m from file f
SETFILE m	Undo current SETFILE for module m
SETFILE	List all SETFILE associations
SETMODULE dn tn	Associate dummy module dn with true name tn
SETMODULE dn	Undo current SETMODULE for name dn
SETMODULE	List all SETMODULE associations
STDNAME s	Set name of local time zone used during standard time
SUBCOMMANDS f	Read subcommands from file f
SWAPINFO	Print module swap info
# s	a comment (for command files), s is ignored
NOCHECKCONSISTENCY	Turn off CHECKCONSISTENCY
NOCONTROLINFO	Turn off CONTROLINFO
NOECHOCMDFILE	Turn off ECHOCMDFILE
NOECHOifredirected	Turn off ECHOIFREDIRECTED
NOFILEINFO	Turn off FILEINFO
NOMAP	Turn off MAP
NOMEMINFO	Turn off MEMINFO
NORESPONSE	Turn off RESPONSE
NOSWAPINFO	Turn off SWAPINFO

Table 50.2-1. MAINEX Subcommands (end)

50.2.1. Long Subcommand Lines

MAINEX allows long subcommand lines to be broken by terminating them with the continuation character ("\" on all currently supported systems). If a subcommand line ends with the continuation character, the next line is treated as part of this line and the continuation character is subsequently discarded.

The continuation character and the immediately following eol cancel each other out; they are not replaced with a blank or any or other character. This allows single words to be split across more than one line; e.g.:

```
search\
path
```

would be equivalent to the single line:

```
searchpath
```

Typically, lines are split at word boundaries, in which case users should be careful to include the space that separates the words where the split was made either right before the continuation character in the first line or as the first character of the second line.

50.2.2. Search Rule Manipulation Subcommands

The subcommands listed in Table 50.2.2-1 govern searches for intmods, non-executable objmods, and executable objmods. The terms intSearch, objSearch, exeSearch, intList, objList, and exeList are defined and explained in the "MAINSAIL Language Manual".

Any of the module arguments m shown in Table 50.2.2-1 may consist of a module name optionally followed by a dash and a target system abbreviation (e.g., "FOO-UIBM", "BAR-VMS"). The initial default target for each subcommand is the host system. When a target appears in a subcommand argument list, it changes the default target to that system for the current module and the remainder of the argument list. A null target system abbreviation specifies the host system (e.g., "BAZ-").

In the search rule subcommands, the prefix determines the type of search or list to which the subcommand applies: "INT" subcommands govern intSearches (or the intList), "OBJ" subcommands objSearches (or the objList), and "EXE" subcommands exeSearches (or the exeList).

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 (default)
OBJLIB		objSearch: search objlibs before files (default)
EXELIB		exeSearch: search objlibs before files (default)

Table 50.2.2-1. MAINEX Search List Subcommands Summary

50.2.2.1. "[INT|OBJ|EXE][FILE|LIB] m1{=f1} ... mn{=fn}"

For each mi, delete any entry for mi (for the target in effect) from the specified list, then add a new entry for the module mi, the file fi, for an individual file if the command suffix is "FILE" or a library if the suffix is "LIB", for the target in effect. For the "FILE" forms, if fi is omitted, use the default file name formed from mi and the target system. For the "LIB" forms, if fi is omitted, use the previous file name (f1 must be specified). No spaces are allowed in "mi=fi"; the file names cannot have embedded spaces.

50.2.2.2. "[INT|OBJ|EXE]DEFAULT m1 ... mn"

Remove any entry for the mi for the target in effect from the indicated list, thereby causing the default search to be in effect for the mi.

50.2.2.3. "[INT|OBJ|EXE]SHOW {m1 ... mn}"

Show entries for the mi on the indicated list; if no mi are specified, show all entries on the list.

50.2.2.4. "[INT|OBJ|EXE]FILE"

First try the default file; if that fails, search open libraries. This is not the default; the subcommand must be specified if files are to be searched before libraries.

50.2.2.5. "[INT|OBJ|EXE]LIB"

First search open libraries; if that fails, try the default file. This is the default.

50.2.3. "CHECKCONSISTENCY"/"NOCHECKCONSISTENCY"

The "CHECKCONSISTENCY" subcommand turns on interface consistency checking (as described in the "MAINSAIL Language Manual") when subsequent modules are bound. Interface consistency checking is on by default.

The "NOCHECKCONSISTENCY" subcommand turns off interface consistency checking. Interface consistency checking remains off until the "CHECKCONSISTENCY" subcommand is given. Being able to turn off consistency checking is useful for a system that is "known" to be consistent (i.e., has been run many times) because it saves runtime, especially when modules have been swapped out since the consistency check must swap in all involved modules. If interface consistency checking is turned off, difficult-to-track errors may occur if module interfaces are actually inconsistent.

50.2.4. "CLoseexelib <fn>"

The "CLOSEEXELIB" subcommand closes the object module library file named <fn> for execution access (the list of objlibs open for execution is distinct from the list open for read/write access; both are global to the entire MAINSAIL session). <fn> may have been opened by a previous "OPENLIBRARY" subcommand or by the system procedure openLibrary.

"CL" is a sufficient abbreviation of "CLOSEEXELIB".

50.2.5. "CLOSEINTLIB <fn>"

The "CLOSEINTLIB" subcommand closes the intmod library file named <fn>, so that it no longer plays a role in intmod searches conducted by the compiler, debugger, etc. <fn> may have been opened by a previous "OPENINTLIB" subcommand or by another program; the open intmod library list is global to the entire MAINSAIL session.

50.2.6. "CLOSEOBJLIB <fn>"

The "CLOSEOBJLIB" subcommand closes the object module library file named <fn> for read/write access (the list of objlibs open for execution is distinct from the list open for read/write access; both are global to the entire MAINSAIL session). <fn> may have been opened by a previous "OPENOBJLIB" subcommand or by a MAINSAIL system program (e.g., the compiler).

50.2.7. "CMDFILE <fn>"

The "CMDFILE" subcommand allows the user to redirect cmdFile. <fn> is the name of the file that is to become cmdFile. The "CMDFILE" subcommand takes effect as soon as subcommand mode is exited. Once in effect, a new line is read from <fn> whenever a module (including MAINEX) requests input from cmdFile.

Example 50.2.7-1 shows an example command file and how the "CMDFILE" subcommand is used. MAINSAIL is invoked and cmdFile is redirected to the file "commands". When subcommand mode is exited, input to the compiler is taken from "commands". The module MOD1 is compiled with the "DEBUG" compiler option (consult the "MAINSAIL Compiler User's Guide" for a description of compiler options). The first <blank line> exits the compiler subcommand mode; the second exits the compiler. Note that a third <blank line> may be included to exit MAINSAIL, but this is not necessary. If input is requested from cmdFile but the end of cmdFile has been reached, then (unless the $noAutoCmdFileSwitching configuration bit is set) the message "WARNING: Unexpected <eol> on cmdFile, switching to TTY" is displayed and cmdFile is automatically redirected to primary input.

50.2.8. "CONTROLINFO"/"NOCONTROLINFO"

The subcommand "CONTROLINFO" causes messages to be written to "TTY" whenever a coroutine resumes or kills another, or whenever an exception is raised, propagated, or handled, or whenever a handler is invoked. "NOCONTROLINFO" cancels "CONTROLINFO".

50.2.9. "CSUBCOMMANDS <fn>"

The "CSUBCOMMANDS <fn>" subcommand works just like the "SUBCOMMANDS <fn>" subcommand, except that if the file named named <fn> does not exist, no error message is given. The "C" stands for "conditional"; the subcommand file is read only if it is present.

50.2.10. "DEFINETIMEZONE s n"

Define the time zone abbreviation s as representing n seconds west of Greenwich; in other words, n is the number of seconds to be added to the standard time in the local time zone to get Greenwich Mean Time. If s (which may be either one or two words) is also the name of the local standard or daylight time zone, then the "DEFINETIMEZONE" information is ignored. Use of this subcommand, unlike the other date and time MAINEX subcommands, does not cause $timeSubcommandsSet to become true. The time zone abbreviation is recognized by $dateAndTimeToStr.

```
Contents of a command file "commands":

    compil
    mod1,
    debug
    <blank line>
    <blank line>


MAINSAIL Session:

    MAINSAIL (R) Version 12.10 (? for help)
Copyright (c) 1984, 1985, 1986, 1987, 1988, and 1989 by
 XIDAK, Inc., Menlo Park, California, USA.
    *,<eol>
    >cmdFile commands
    ><eol>

MAINSAIL (R) Compiler
Copyright (c) 1984, 1985, 1986, 1987, 1988, and 1989 by
 XIDAK, Inc., Menlo Park, California, USA.

    compile (? for help): > >
    mod1 1 ...

    Objmod for MOD1 stored on...

    compile (? for help):
```

Example 50.2.7-1. Using the MAINEX Subcommand "CMDFILE"

50.2.11. "DSTENDRULE s"

See Section 50.2.14 for a description of "DSTENDRULE".

50.2.12. "DSTNAME s"

See Section 50.2.32 for a description of "DSTNAME".

50.2.13. "DSTOFFSET n"

"DSTOFFSET n", where n is a long integer (without trailing "L") informs MAINSAIL:

- If n is zero, that daylight savings time does not affect the local time zone. The "DSTSTARTRULE" and "DSTENDRULE" commands are ignored.

- If n is non-zero, that n is the offset, in seconds, to be added to standard time to get daylight savings time when daylight savings time is in effect. In the United States, n is 3600L (one hour) in areas that have daylight savings time.

If no "DSTOFFSET" command is specified, n is assumed to be zero.

50.2.14. "DSTSTARTRULE s" and "DSTENDRULE s"

The start and stop rules handled by the standard algorithm allow the starting and stopping times and dates for daylight savings time to be expressed as:

- A month,

- a day of the month expressed either as a numeric date or as a day of the week and the number of its occurrence,

- and a time of day.

In "DSTSTARTRULE s" and "DSTENDRULE s", s is a string containing the above information in the format:

```
<month name> <weekday name> <occurrence> <time of day>
```

or:

```
<month name> <date of month> <time of day>
```

<month name> is an unambiguous prefix of one of the twelve month names. <weekday name> is an unambiguous prefix of one the names of the seven days of the week. <occurrence> is "1" for the first occurrence of the named weekday in the month, "2" for the second, etc.; "5" or any greater number may be used for the last occurrence, since any given weekday never occurs more than five times in a single month. <date of month> is the day of the month; any number greater than the number of days in the month is treated as the last day of the month. <time of

day> is a time of day in any format acceptable to the MAINSAIL system procedure $strToTime.

Some sample values of s and their interpretations are:

String	Interpretation
`jan mon 5 12:30`	`12:30 P.M. on the last Monday in January`
`JUNE SUN 2 0:00`	`Midnight on the second Sunday in June`
`August 15 7:30 p.m.`	`7:30 P.M. on the 15th of August`
`April Sunday 5 2:00`	`2:00 A.M. on the last Sunday in April`
`April Sunday 1 2:00`	`2:00 A.M. on the first Sunday in April`
`October Sunday 5 2:00`	`2:00 A.M. on the last Sunday in October`

"DSTSTARTRULE s" and "DSTENDRULE s" specify the rule for the starting and ending dates and times, respectively, of daylight savings time. "DSTSTARTRULE" specifies the time in standard time, and "DSTENDRULE" in daylight savings time; i.e., they are specified using the kind of time in effect before the time change. In northern hemisphere locations that adjust standard time in the winter (calling it "winter time"), or southern hemisphere locations adjusting standard time in the summer, the month in "DSTSTARTRULE" will be later in the year than in "DSTENDRULE". The algorithm deals with this situation correctly, applying the adjustment from late in one year through the early part of the next.

If "DSTOFFSET" is zero, "DSTSTARTRULE" and "DSTENDRULE" are ignored; otherwise, if "DSTSTARTRULE" and "DSTENDRULE" are not both specified, daylight savings time is assumed to apply throughout the year.

50.2.15. "ECHOCMDFILE"/"NOECHOCMDFILE"

The subcommand "ECHOCMDFILE" tells MAINSAIL to echo the contents of cmdFile to logFile (regardless of whether either is redirected). "NOECHOCMDFILE" turns off "ECHOCMDFILE".

50.2.16. "ECHOifredirected"/"NOECHOifredirected"

The subcommand "ECHOIFREDIRECTED" echoes logFile to "TTY" if logFile is not "TTY", and cmdFile to logFile if cmdFile or logFile is not "TTY". "ECHOIFREDIRECTED" may be abbreviated to "ECHO". "NOECHOIFREDIRECTED" turns off "ECHOIFREDIRECTED".

50.2.17. "ENTER <ln> <fn>"

The "ENTER" subcommand associates logical name <ln> with file name <fn>. Whenever a file open is performed using the name <ln>, the name <fn> is used instead. If <fn> is not specified, any previous association for <ln> is broken.

In Example 50.2.17-1, whenever "(data file)" or "eparms" is opened, the file name "data1" or "realEparms" is used. Note that <fn> is taken to be the last sequence of characters that does not contain any blanks so that any sequence of characters, including blanks, can be associated with a file name. Here, "(data file)" is associated with file name "data1", and "eparms" is associated with the file name "realEparms".

```
MAINSAIL (R) Version 12.10 (? for help)
Copyright (c) 1984, 1985, 1986, 1987, 1988, and 1989 by
 XIDAK, Inc., Menlo Park, California, USA.
*m,<eol>
>enter (data file) data1<eol>
>enter eparms realEparms<eol>
><eol>
```

Example 50.2.17-1. Using the MAINEX Subcommand "ENTER"

This command allows flexibility in file names without modification of source code. For example, a given module can run on several machines without modification even though file name formats across those machines may differ if a logical name is used in the call to open and an "ENTER" is used to associate that logical name with the actual file name. XIDAK logical names are surrounded by parentheses. It is suggested that users adopt a different convention for logical names, so as to avoid conflict with XIDAK logical names.

A logical file name with embedded blanks cannot be used as arguments of commands to some MAINSAIL utilities (e.g., a library file name in a MODLIB command).

50.2.18. "FILEINFO"/"NOFILEINFO"

The "FILEINFO" subcommand causes a message to be written to "TTY" each time a file is opened, closed, deleted, or renamed. It may be useful for debugging. "NOFILEINFO" turns off "FILEINFO".

50.2.19. "GMTOFFSET n"

"GMTOFFSET n", where n is a long integer (without trailing "L") informs MAINSAIL that n is the number of seconds to be added to the standard time in the local time zone to get Greenwich Mean Time. For example, in the Pacific Time Zone, "GMTOFFSET 28800" would be appropriate, since 28800 = 7 * 3600, and Pacific Standard Time is seven hours behind Greenwich Mean Time. Standard time zones east of Greenwich should express n as a negative number; e.g., in France use "GMTOFFSET -3600".

If no "GMTOFFSET" command is specified, n is assumed to be zero.

50.2.20. "LOGFILE <fn>"

The "LOGFILE" subcommand allows the user to redirect logFile. <fn> is the name of the file that is to become logFile. The "LOGFILE" subcommand takes effect as soon as subcommand mode is exited. Once in effect, all output to logFile is redirected to <fn>.

Example 50.2.20-1 shows how the "LOGFILE" subcommand is used. MAINSAIL is invoked and logFile is redirected to the file "log". When subcommand mode is exited, the compiler writes its prompt to the file "log" and reads its input from cmdFile. If cmdFile is primary input (usually the terminal), as in this example, the user must enter commands, but no prompts are displayed. "sample<eol>" tells the compiler which module to compile, and <eol> exits the compiler. The second <eol> is needed to return to the operating system.

50.2.21. "LOOKUP <ln>"

The "LOOKUP" subcommand causes the association for logical name <ln> to be displayed if there is one. If <ln> is not specified, the values of all logical names are specified.

```
MAINSAIL Session:

MAINSAIL (R) Version 12.10 (? for help)
Copyright (c) 1984, 1985, 1986, 1987, 1988, and 1989 by
 XIDAK, Inc., Menlo Park, California, USA.
*compil,<eol>
>logFile log<eol>
><eol>
sample<eol>
<eol>
<eol>


Contents of logFile "log":


MAINSAIL (R) Compiler
Copyright (c) 1984, 1985, 1986, 1987, 1988, and 1989 by
 XIDAK, Inc., Menlo Park, California, USA.

compile (? for help): Opening intmod for $SYS...

sample 1 ...
Objmod for SAMPLE on sample-<sys. abbrev.>.obj
Intmod for SAMPLE not stored

compile (? for help):
```

Example 50.2.20-1. Using the MAINEX Subcommand "LOGFILE"

50.2.22. "MAP n"/"NOMAP"

The "MAP n" subcommand causes a memory utilization map to be displayed periodically. The map consists of one or more lines, each consisting of a hexadecimal number followed by sequence of characters. Each character corresponds to a MAINSAIL memory page, and the hexadecimal number at the beginning of the line is the first address of the page corresponding to the first map character of the line. The integer "n" governs the frequency with which the map is displayed. The larger "n" the less frequently the memory map is displayed (an "n" of 1 produces the maximum display frequency). Table 50.2.22-1 lists the characters displayed in a memory map and their meaning.

The "MAP" subcommand allows the user to evaluate a program's memory consumption. This is most useful on computers with a small address space where memory is at a premium.

Char	Page Type	Use
.	Free	Available for allocation
C	Control	Contains (part of) a control section
B	Static	I/O buffer or other static structure, or not allocated by MAINSAIL
b	Free Static	Free page in static page pool
S	String	String space
D	Data	MAINSAIL chunks (arrays, records, data sections)

A lowercase character (except "b") indicates a continuation of a data structure or group of data structures. For example, if a control section extends across three pages, it appears in the page map as "Ccc"; a series of contiguously allocated data pages might appear as "Dddddd".

Table 50.2.22-1. Memory Map Display Characters

The "MAP" subcommand with no argument produces a memory map of the same format as produced by the other form of the "MAP" subcommand. However, maps are printed only immediately before and after each event printed out by the "MEMINFO" subcommand. This subcommand automatically turns on the "MEMINFO" subcommand.

"NOMAP" turns off a previous "MAP" command.

50.2.23. "MEMINFO"/"NOMEMINFO"

This subcommand causes a message to be printed when various memory management events occur: garbage collections, asking the operating system for more memory, and so forth. The messages produced are either self-explanatory or of interest only to XIDAK personnel. The information printed may vary from release to release. "NOMEMINFO" turns off "MEMINFO".

50.2.24. "OPenexelib <fn>"

The "OPENEXELIB" subcommand opens the object module library file named <fn> for execution (the list of objlibs open for execution is distinct from the list open for read/write access; both are global to the entire MAINSAIL session).

"OP" is a sufficient abbreviation of "OPENEXELIB".

50.2.25. "OPENLIBRARY <fn>"

"OPENLIBRARY" is equivalent to "OPENEXELIB".

50.2.26. "OPENINTLIB <fn>"

The "OPENINTLIB" subcommand opens the intmod library file named <fn>, so that it plays a role in intmod searches conducted by the compiler, debugger, etc. The open intmod library list is global to the entire MAINSAIL session.

50.2.27. "OPENOBJLIB <fn>"

The "OPENOBJLIB" subcommand opens the object module library file named <fn> for read/write access (the list of objlibs open for execution is distinct from the list open for read/write access; both are global to the entire MAINSAIL session).

50.2.28. "RESPONSE"/"NORESPONSE"

When "RESPONSE" is in effect (default), errMsg prompts for a user response if the raised $systemExcpt is not handled (except when the "warning" bit is set in ctrlBits). When "NORESPONSE" is in effect, errMsg never prompts.

50.2.29. "SEARCHPATH"

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

The "SEARCHPATH" command has the form:

```
SEARCHPATH f f1 f2 ... fn
```

where f is a pattern to be matched to a file name, and the fi are file name specifications which are to be used in place of the original file name. Whenever a file is opened, MAINSAIL looks through the list of current searchpath specifications (from most recently added to least recently added), and if the file name matches the pattern f, then it processes each fi in the sequence given by substituting the wildcard characters in fi and attempting to open an existing file named by the resulting string. If such an open succeeds, the procedure open returns the file so opened; if no such open succeeds, the procedure open returns false (if errorOK is set) or prompts for a new file name (if errorOK is not set).

If a file name matches more than one pattern in the list, the most recently added matching pattern is checked first, then older matching patterns.

If the "f" part of a pattern is the same as a pattern already on the list, then the entry for that pattern is removed from the list, and the new entry added to the head of the list.

f may contain any number of asterisks as wild card characters. "*" matches any sequence of zero or more characters. The fi may contain any number of replacement specifiers of the form "*n", where n is a positive integer, e.g., "*1", "*10". A replacement specifier "*n" is replaced with the string that matched the nth "*" in f (numbered from the left as 1, 2, ...; if n is greater than the number of wildcards in f, "*n" is replaced with the null string). "*" in an fi not followed by a digit is treated as "*1", which is a useful short form since there is often just one "*" in f.

If no fi are specified, the searchpath associated with the pattern f is removed.

The backslash character, "\", may be used in f or an fi as an escape character to indicate that the next character is not to be treated specially (such \'s are discarded). For example, if "*" is to appear as a normal character it must be written as "*"; "*2" followed by "34" must be written as "*2\34" since "*234" means that the 234th wildcard match is to be substituted. "\" must be written as "\\". "\" as the last character in f or an fi is treated as the character "\"; i.e., the backslash character may not be used to escape a space or tab character.

For example, if the command:

```
SEARCHPATH * * *.msl /usr/john/*.msl /x/*.msl
```

is given, open attempts to open every file first with the specified name, then with the specified name followed by ".msl", then with the specified name preceded by "/usr/john/" and followed by ".msl", and finally preceded by "/x/" and followed by ".msl".

If the command:

```
SEARCHPATH *-before-*.* *1-after-*2.*3
```

is given, then open replaces the file name "foo-before-upgrade.txt" with "foo-after-upgrade.txt".

The command:

```
SEARCHPATH \*-\*.txt star-star.txt
```

is equivalent to:

```
ENTER *-*.txt star-star.txt
```

assuming there are no other searchpaths that would match the file name "*-*.txt".

A structured use of searchpaths is to choose prefixes to represent searchpath names; e.g., one might use prefixes of the form "<identifier>:" (anything would do). For example, one could use "source:" as a source searchpath prefix, and "saveon:" for a saveon searchpath prefix. Then, in a portable way, "source:foo" could be used to open a source file "foo", and "saveon:baz" to open a saveon file "baz". The searchpaths must first be set up with MAINEX subcommands:

```
>SEARCHPATH source:* *.msl /12.10/*.msl<eol>
>SEARCHPATH saveon:* *.svn /12.10/*.svn<eol>
```

In this way, MAINSAIL appends the extension, and at the same time tries various directories, so that the source file can use portable file names (i.e., there is no need to put the extension into source code for file name constants).

When the open routine tries to open the various file names formed from the fi's, it uses the errorOK bit and goes on to the next one if it cannot be opened. Thus even if the incoming file name is something like "foobar.lib", it would try "foobar.lib.msl", but that is OK since it will fail (assuming there is no "foobar.lib.msl").

Searchpaths are case-sensitive on systems where "$attributes TST $fileNamesAreCaseSensitive".

The searchpath syntax may change in future releases.

50.2.30. "SETFILE <tmn> <fn>"

The "SETFILE" subcommand associates the name of the actual module <tmn> with the file <fn>. MAINSAIL then accesses the module in the file <fn> when the actual module is

referenced. The association made by the most recent "SETFILE" subcommand is the one that is used.

If <fn> is omitted, any file association for <tmn> is removed. If <tmn> is omitted, all current module-file associations are listed.

"SETFILE m f" is equivalent to "EXEFILE m=f". "SETFILE m" is equivalent to "EXEDEFAULT m".

50.2.31. "SETMODULE <dmn> <tmn>"

The "SETMODULE" subcommand associates the dummy module name <dmn> with the true module name <tmn>. All subsequent references to the dummy module are redirected to the actual module <tmn>. The association made by the most recent "SETMODULE" subcommand is the one that is used.

If <tmn> is omitted, any file association for <dmn> is removed. If <dmn> is omitted, all current module-module associations are listed.

50.2.32. "STDNAME s" and "DSTNAME s"

In "STDNAME s" and "DSTNAME s", s is the commonly used name (usually abbreviated) of the local time zone. The "STDNAME" value is used if standard time is in effect, the "DSTNAME" if daylight savings time is in effect. For example, for the Eastern Time Zone of the United States, specify the commands:

```
STDNAME EST
DSTNAME EDT
```

If "DSTOFFSET" is zero, "DSTNAME" is ignored, and "STDNAME" always used for the name of the current time zone. If s is not specified, the null string is returned by the affected time zone name procedures.

50.2.33. "SUBCOMMANDS <fn>"

This subcommand indicates that further MAINEX subcommands are to be read from the file <fn>. When end-of-file is reached on that file, MAINEX resumes reading from the file containing the "SUBCOMMANDS" subcommand. This subcommand is useful if a series of subcommands is frequently executed or executed by a number of different MAINSAIL bootstraps.

50.2.34. "SWAPINFO"/"NOSWAPINFO"

The "SWAPINFO" subcommand is used to monitor module swapping. When specified, messages are written to "TTY" when a module is brought into memory and when a module is discarded from memory, as described in the "MAINSAIL Language Manual". "NOSWAPINFO" turns off "SWAPINFO".

Modules in an open mapped library are never removed from memory.

50.2.35. "# s"

The "#" character in MAINEX subcommand mode introduces a comment. It is especially useful in command files.

50.3. Default Subcommand File

MAINEX attempts to read one or two subcommand files when MAINSAIL is initialized (but after any "SUBCOMMAND" entries in the bootstrap are processed):

- First, it looks in the home directory (as given by the system procedure $homeDirectory) for a file name formed from "v" followed by the major version number and the extension ".cmd" (in this release, "v12.cmd").
- If the current directory (as returned by $currentDirectory) is not the home directory, it looks in the current directory for a file of the same name. If it finds it, it also executes the subcommands in it.

50.4. $mainsailExec

```
PROCEDURE     $mainsailExec
                          (OPTIONAL STRING s);
```

Table 50.4-1. $mainsailExec

$mainsailExec recursively invokes the MAINSAIL executive. The current command line is appended to s. If the resulting s is non-Zero, each line read from s is interpreted as the name of

a module (possibly followed by arguments) and executed; $mainsailExec returns when the string is exhausted. If s is Zero, the recursively invoked executive functions as MAINEX normally does: it repeatedly prompts with the asterisk prompt for a module name and arguments, then executes the named module, until a bare <eol> is read from cmdFile, at which point it returns to its caller.

50.5. $getSubcommands

```
PROCEDURE  $getSubcommands
                          (MODIFIES OPTIONAL STRING cmdStr;
                           OPTIONAL POINTER(textFile)
                                      theCmdFile);
```

Table 50.5-1. $getSubcommands

The procedure $getSubcommands may be used to enter MAINEX subcommands from a program. The accepted subcommands are the same ones that are valid in the usual MAINEX subcommand mode.

If both cmdStr and theCmdFile are Zero, prompts are written to logFile and subcommands read from cmdFile. If cmdStr is not Zero, the subcommands are read from cmdStr, and no prompts are written to logFile. If cmdStr is Zero and theCmdFile is not Zero, commands are read from theCmdFile, and prompts are written to logFile only if theCmdFile equals cmdFile.

Subcommand names spelled out in full (not abbreviated) are more likely to be upwards-compatible with future releases, since abbreviations may conflict if another subcommand with the same prefix is created. Since all subcommands are subject to change from release to release, however, upward compatibility is never guaranteed. Programmers making use of $getSubcommands may have to change the value of cmdStr or the contents of theCmdFile if MAINEX subcommands change.

51. The Device Modules MEM and NUL

51.1. MEM

The device module MEM maintains a file in memory. A MEM file is distinguished by the name of the file when it is opened. MEM file names are prefixed with the string:

```
"MEM" & $devModBrkStr
```

e.g., "MEM>xxx" if $devModBrk is ">" ("xxx" may be omitted). Case is ignored in the device module prefix.

MEM files are extended as necessary. Extending a MEM file too far may exhaust MAINSAIL's address space. MEM files are always deleted when closed, whether or not the delete bit is set in the call to close.

Two MEM files with the same name but opened with different calls to open are distinct files and do not contain the same data.

51.2. NUL

The device module NUL is a data sink. A NUL file is distinguished by the name of the file when it is opened. NUL file names are prefixed with the string:

```
"NUL" & $devModBrkStr
```

e.g., "NUL>xxx" if $devModBrk is ">" ("xxx" may be omitted). Case is ignored in the device module prefix.

Data written to NUL files are discarded. When a read is performed from a NUL file, the file acts as if end-of-file had been reached.

52. MM

Command line syntax: mm {mm command}*

The MAINSAIL monitor module MM allows the user to invoke various memory management operations or examine various memory management parameters. It takes commands that tell it what to do, e.g., a string garbage collection, a chunk compaction, a memory map. In each case, it tells how many CPU seconds were required to do each operation (it is interesting to see how the various memory management strategies compare timewise). "?" lists the available commands, which are shown in Table 52-1.

Any of the commands that take a file, such as "ai f", can take ">>f" in place of "f" to mean to append to the end of the file if it already exists (in this case a page mark is written to separate the new data from the old).

The command "all {f}" prints all stats to the file f ("TTY" if f is omitted). This currently includes the output from the following commands: "i", "mi", "m", "mai", "ari", "ci", "di". This is a convenient way to get a dump of all the stats available from MM without having to give the individual commands.

The "ari" command shows for each area:

- the area's title
- the attr bits that are set
- the total number of chunk pages allocated for the area
- the number of pages, used characters, and unused characters for each string space
- a summary of the area's free chunks by size

The size of a free chunk includes the chunk overhead and is rounded up according to chunk alignment so that it is the actual amount of space occupied by the free chunk. There is some bookkeeping overhead in a string space.

The "ci" command displays information about currently allocated chunks of memory, i.e., records, data sections, and arrays. The chunks are not necessarily accessible; i.e., if a garbage collection were performed, some of the displayed chunks might be deallocated.

```
?                help (this message)
ai f             write array info to file f
all f            write all stats to file f
ari f            write area info to file f
amp n            set $allowedMemoryPercent to n
aop n            set $allowedOverheadPercent to n
ccs              compact chunk space
cda              collect dscrArea
cfc              coalesce free chunks
ci f c1 ... cn   write chunk info and recs of classes
                   c1 ... to file f
cli f            write class info to file f
cmc              check memory consistency
cmd              compress memory down
cmu              compress memory up
cp               collect pointers
cpmd             compress page map down
cpmu             compress page map up
cps              collect pointers and strings
cs               collect strings
csp n            contract static pool by n pages
dc               decrement $collectLock
di f             write descriptor info to file f
dp               deallocate trailing free pages, if
                   operating system permits
i                show info
ic               increment $collectLock
m f              write memory map to file f
mai f            write module age info to file f
mi f             write module info to file f
opev n           set $overheadPercentExitValue to n
sw s             swap out module s
sw1              swap out one module of MAINSAIL's choice
swa              swap out all modules
xsp n            expand static pool by n pages
<eol>            quit
```

Table 52-1. MM Commands

If the output file f for the "ci" command is omitted, the output file defaults to logFile. After f, "ci" accepts a series of class names. For each class name specified, the command displays the

contents of all chunks of the class. The command always displays a summary enumerating the number, size, type, and class name of each chunk. Examining the contents of all chunks of a class may help determine why they are present in memory.

The "di" command displays how many times allocation, deallocation, and garbage collection have been performed for records, arrays, and modules for which a descriptor exists. The statistics shown indicate how often each event has occurred since the allocation of the descriptor; since descriptors are rarely collected, this is usually since the start of the MAINSAIL execution. If the output file f is omitted, it defaults to "TTY". The output format should be self-explanatory.

The memory manager keeps track of the total size of any gaps in the operating system's allocation of memory to MAINSAIL (i.e., where the address of the memory MAINSAIL last acquired from the operating system did not immediately follow the last address of previously acquired memory). The MM "i" command displays information about any such gaps (presumably memory consumed by the operating system or by foreign code linked with MAINSAIL). The amount of memory consumed by gaps is the amount of memory that should be specified to the CONF "OSMEMORYPOOLSIZE" command. The "i" command also shows the total static pool size actually used so far. This can be of assistance when trying to decide how big to make the initial static pool (use the default setting, run your program, then see how big it got, then make a new boot with approximately that value for the initial static pool size). It also shows file cache parameters and the maximum size that memory has grown to.

The brief descriptions of the other commands are intended to be self-explanatory (if you don't understand the explanation, then you probably don't want to do it). The list of commands is subject to change as memory management strategies evolve.

When invoked with command line arguments, several command names can be separated by spaces (except in the case of "ci", where all subsequent arguments are treated as arguments to "ci"); e.g., "mm m cps m" invokes MM, shows a memory map, does a garbage collection, then shows a memory map again.

53. MODLIB, the MAINSAIL Objmod Librarian

A MAINSAIL program is a collection of modules. The modules are not linked into a single unit as is common for most programming languages. Instead, MAINSAIL dynamically brings modules into memory as needed and takes care of intermodule communication in a machine-independent manner.

MODLIB, the objmod librarian, is used to create and maintain objmod (object module) libraries. Module libraries provide a means of logically organizing MAINSAIL object modules. Some advantages of module libraries over individual object module files are listed below:

- When a system consists of several modules, putting them into a module library eliminates the need for individual object module files, thus reducing clutter in the file system.

- An objmod library is opened once (for execution access) during execution. Each module that does not reside in an object module library is individually opened during execution when it is first accessed. This is generally a time-consuming operation.

- Sometimes it is necessary to remove modules from memory in order to make room for another module. A module that does not reside in memory must first be written to a "swap" file before its space is used. A module that resides in an open library need not be written to the swap file.

As described in the "MAINSAIL Language Manual", module libraries open for execution access are searched when MAINSAIL needs to access a non-resident module. They are searched in order of most recently opened to least recently opened so that if two modules with the same name exist in two libraries, the module in the most recently opened library is found and used.

There are two ways in which an objmod library can be "open": for execution access and for read/write access. The procedure openLibrary and the MAINEX subcommand "OPENLIBRARY" open objmod libraries for execution access. MAINEX and compiler subcommands are used to open libraries for read/write access; such libraries are used in objmod searches, e.g., by the compiler, and are also cached for MODLIB operations. The list of objlibs open for execution is distinct from the list open for read/write access; both are global to the entire MAINSAIL session.

There are no restrictions on the way in which modules are organized in objmod libraries. A module library might consist of all modules related to a given program, or it might consist of several unrelated "utility" modules. However, all objmods in an objmod library must be for the same target system (unlike intmod libraries).

MAINSAIL does not impose a limit on the number of module libraries that may be open during execution, although the underlying operating system may have a limit on the number of open files.

A module library can be opened for execution access in any of the following ways:

1. With the system procedure openLibrary (see the "MAINSAIL Language Manual").

2. With the MAINEX "OPENLIBRARY" subcommand (see Section 50.2.25). This subcommand may be specified interactively or in a MAINSAIL bootstrap.

3. With the MAINDEBUG "OL" command (see the "MAINDEBUG User's Guide").

53.1. How MODLIB Opens Library Files

When MODLIB opens a module library file for execute access, the library file remains open for the remainder of the MODLIB session. MODLIB uses an already opened library file for a specified library if the library name is exactly equal (or equal ignoring case on systems where "$attributes NTST $fileNamesAreCaseSensitive") to the name used when the library file was first opened.

Unlike libraries open for execute access, objmod libraries open for read/write access are not open files; i.e., they do not consume an operating system file handle. The files are opened when information is added to them or extracted from them and then immediately closed; however, they are considered to be "open" libraries because the MAINSAIL runtime system maintains a global list of libraries open for read/write access. Objmod libraries for read/write access remain open until explicitly closed.

53.2. Library Size

MAINSAIL imposes no limit on the number of modules in a library.

MODLIB automatically extends the size of a library file to accommodate new modules. However, a library file never shrinks; if many deletions are done without corresponding adds that reuse the space freed by the deletions, it may be desirable to use the "COPY" command to compact the library into a new file.

53.3. MODLIB Commands

Table 53.3-2 lists the available MODLIB commands. The following conventions are used in MODLIB command descriptions:

- "modList" is a list of module specifications separated by blanks. A module may be specified differently depending on the command, as summarized in Table 53.3-1.
- "libName" is a library file name. "sLibName" is a source library name (the name of a library from which modules are removed or copied) and "dLibName" is a destination library name (the name of a library to which modules are added).
- Anything enclosed in curly brackets ("{" and "}") is optional.

Form	May Be Used in Commands
<module name>	all
<file name>	DIRECTORY, LEGALNOTICE
<module name>=<file name>	ADD, EXTRACT
<source module name>=<destination module name>	COPY, MOVE

Table 53.3-1. Forms of a "modList" Element

A very long MODLIB command may be broken across several lines by terminating each line with a backslash ("\"). Thus, the command lines:

```
add foo.lib\
 abc\
 def\
 ghi
```

are equivalent to the single command line:

```
add foo.lib abc def ghi
```

Command	Description
<eol>	Exit MODLIB
ADD dLibName modList	Add module(s) to dLibName
COPY sLibName dLibName{ modList}	Copy modules from sLibName to dLibName
CREATE libName	Create an empty library named libName
DELETE sLibName modList	Delete module(s) from sLibName
DIRECTORY libName{=fileName}{ modList}	Examine the modules in library libName, writing info to file fileName
EXTRACT sLibName{ modList}	Create new file(s) containing copies of modules in sLibName
LEGALNOTICE libName{ modList}	Show information about "$LEGALNOTICE" directives in modules in library libName
LOG	Show informational messages
MOVE sLibName dLibName{ modList}	Move module(s) from sLibName to dLibName
NOLOG	Suppress info messages
NOUPDATE	Do not update library file directory each time modified
QDIRECTORY libName{=fileName}{ modList}	Quick directory of libName, writing info to fileName
QUIT	Exit MODLIB
READ fileName	Read MODLIB commands from file fileName
TARGET{ targetSystem}	Specify target operating system for subsequently opened libraries
UPDATE	Cancel "NOUPDATE"

Table 53.3-2. MODLIB Commands

53.3.1. <eol>, "QUIT"

To exit, type <eol> or "QUIT" when you are finished using the module librarian.

53.3.2. "ADD dLibName modList"

Add the modules specified by modList to the module library dLibName. modList is a list of module specifications, separated by blanks. Valid module specifications for this command are given in Table 53.3.2-1. Different types of module specifications can be mixed within modList.

If dLibName does not exist, it is created. If free pages constitute more than a certain fraction of dLibName and dLibName contains no hole large enough to accommodate a module in modList, the library is compacted.

Form	Meaning
<module name>	Add the module found in the default file, the name of which is operating-system-dependent.
<module name>=<file name>	Add the module found in the file <file name> as the module <module name>.

Table 53.3.2-1. MODLIB "ADD" Command modList Forms

53.3.3. "COPY sLibName dLibName{ modList}"

Copy the modules specified by modList in sLibName to dLibName. If modList is absent, copy all modules in sLibName. If dLibName does not exist, it is created.

If dLibName does not exist before the "COPY" command, then it contains no holes after the command is completed. Thus, the "COPY" command may be used to compact a library with unused space.

53.3.4. "CREATE libName"

Create a module library named libName. The module library is initially empty.

53.3.5. "DELETE sLibName modList"

Delete the modules specified in modList from the module library libName. modList is a list of module names separated by blanks.

53.3.6. "DIRECTORY libName{=fileName}{ modList}"

List information about the modules contained in the module library libName. libName may be the character "*" if the modules reside in individual object module files instead of in a module library. fileName is the name of the file to which the list is to be written. If fileName is omitted, the list is written to logFile. modList is a list of module names to be included in the directory list, separated by blanks. Valid module specifications for "DIRECTORY" are <module name> and <file name>. If the module resides in a library, only <module name> is permitted. If the module does not reside in a library ("*" was specified for the library name), then if <module name> is specified, a file name is formed in the operating-system-dependent default fashion. If <file name> is specified (the given string is not a possible module name), the given file name is used. The object module in the named file is examined.

If modList is omitted, all modules in the module library are listed.

The information listed includes the module's directory name, the module name, the number of pages it occupies, the date and time at which it was compiled, the version for which it was compiled, and the options with which it was compiled. Example 53.3.6-1 is a sample session with MAINSAIL showing execution of the MODLIB directory command and the resulting listing.

The meanings of the letters shown in the options column of the directory listing are shown in Table 53.3.6-2. For example, "CDL" indicates that the module was compiled with checking initially set, debuggable, and with a legal notice. "Cmp", "Cms", "Cps", or "Cmps" is used to display multiple counting options. "Tmp" is displayed to indicate both "MODTIME" and "PROCTIME" options.

53.3.7. "EXTRACT sLibName{ modList}"

Extract the specified module(s) from the module library libName. modList is a list of module specifications, separated by blanks. Valid module specifications for this command are given in Table 53.3.7-1. Different types of module specification can be mixed within modList.

If modList is omitted, all modules are extracted from the library and written to their default files.

```
MAINSAIL (R) Version 12.10 (? for help)
Copyright (c) 1984, 1985, 1986, 1987, 1988, and 1989 by
 XIDAK, Inc., Menlo Park, California, USA.
*modlib<eol>
MAINSAIL (tm) Module Librarian (? for help)
MODLIB: dir compil.lib<eol>

Directory of compil.lib

DirName Pages     Date        Time       Version    Options

ERRORS     15    10-Nov-84    4:14:53      9.8     C L
ITFXRF     14    10-Nov-84    4:33:50      9.8     C L
PASS1     298    10-Nov-84    5:33:47      9.8     C L
PASS2      73    10-Nov-84    5:36:13      9.8     C L
SYMSEC     26    10-Nov-84    6:28:06      9.8     C L

5 modules using 426 pages.
1 directory page. 0 free pages. 427 total pages.

MODLIB: <eol>
```

Example 53.3.6-1. Using the MODLIB "DIRECTORY" Command

Form	Meaning
<module name>	Extract the module and write it to the default file, the name of which is operating-system-dependent.
<module name>=<file name>	Extract the module with the name <module name> and write it to a file named <file name>.

Table 53.3.7-1. MODLIB "EXTRACT" Command modList Forms

Option Abbreviation	The Module Was Compiled with:
C	CHECK subcommand
AC	ACHECK subcommand
L	a non-null legal notice
D	DEBUG subcommand
O	OPTIMIZE subcommand
U	UNBOUND subcommand
IB	inline procedures have bodies
I	INCREMENTAL subcommand
S	SAVEON subcommand
Cm	PERMODULE subcommand
Cp	PERPROC subcommand
Cs	PERSTMT subcommand
Tm	MODTIME subcommand
Tp	PROCTIME subcommand

Table 53.3.6-2. Option Letters Displayed by the "DIRECTORY" Command

53.3.8. "LEGALNOTICE libName{ modList}"

The library and modules examined are selected in accordance with libName and modList in the same way as by the "DIRECTORY" command. The legal notices of the examined modules are written to logFile.

53.3.9. "LOG"/"NOLOG"

Default: "LOG"

"LOG" enables the informational messages written by MODLIB when operating on libraries. "NOLOG" cancels "LOG".

53.3.10. "MOVE sLibName dLibName modList"

Move module(s) from sLibName to dLibName. modList is a list of module names to be moved, separated by blanks. The specified modules are copied to dLibName and deleted from sLibName. Valid module specifications for this command are given in Table 53.3.10-1. Module specifications can be mixed within modList.

If dLibName does not exist, it is created.

Form	Meaning
<module name>	Move the module named <module name> from sLibName to dLibName.
<source module name>=<destination module name>	Move the module named <source module name> from sLibName to dLibName, changing its directory name to <destination module name>.

Table 53.3.10-1. MODLIB "MOVE" Command modList Forms

53.3.11. "QDIRECTORY libName{=fileName}{ modList}"

Give a quick list (module names only) of the modules contained in the module library libName. The parameters to "QDIRECTORY" have the same meaning as the corresponding parameters to "DIRECTORY".

53.3.12. "READ fileName"

"READ" causes MODLIB commands to be read from the named file. The "READ" command can be used to do nested readings to an arbitrary depth.

53.3.13. "TARGET{ targetSystem}"

The "TARGET" command specifies the operating system for which subsequently opened libraries are formatted. The "TARGET" command does not affect the target of libraries that have already been opened. The specified target is in effect until the end of the MODLIB session or until another "TARGET" command is given. The initial target is the same as the host system.

If the target system is omitted, the target is set to the host system.

targetSystem is the system abbreviation for the target system. "TARGET ?" causes a list of valid target abbreviations to be displayed.

If MODLIB assumes the wrong target when a library is opened, undefined errors result. Therefore, the user must be careful to specify the correct target.

53.3.14. "UPDATE"/"NOUPDATE"

Default: "UPDATE"

Operations modifying large libraries (i.e., adding or deleting modules) take longer than operations on small libraries. After each MODLIB command, the updated directory list for every modified library is written to the corresponding library file, and all library files open by MODLIB are closed. This helps keep the library files in a consistent state should a system crash or other abort occur between commands, and keeps the number of open files to a minimum. As the directory gets large, it takes longer and longer to store and read it each time.

Since the storing of the directory occurs only after each command, the single command "COPY srcLib dstLib" is a much faster way to copy all modules from srcLib to dstLib than a separate command for each module. If not all modules are to be copied, then the command:

```
COPY srcLib dstLib mod1 mod2 ... modn
```

is faster than n separate commands since just one library update occurs (after all the modules have been copied).

However, there are times when it is more convenient to use a separate command for each module, e.g., in a script generated by a program, or when the commands are being sent to MODLIB through calls to $executeModlibCommands. The "UPDATE" and "NOUPDATE" commands may be used to handle such situations.

The default is "UPDATE". Once the "NOUPDATE" command is given, all libraries referenced are kept open, and directories are not written to each altered library after each command. The "UPDATE" command turns off this effect of "NOUPDATE", and also immediately closes all libraries and updates the directories of all altered libraries. An altered library is updated when it is closed, even if "NOUPDATE" is in effect (all libraries are closed by MODLIB's final procedure, so all libraries are updated when MAINSAIL exits).

The commands:

```
NOUPDATE
COPY srcLib dstLib mod1
COPY srcLib dstLib mod2
...
COPY srcLib dstLib modn
UPDATE
```

execute faster than the same commands without the surrounding "NOUPDATE" and "UPDATE" commands. The speed-up may be significant when many modules are involved.

If the system should crash, or if the MAINSAIL execution is aborted, between "NOUPDATE" and "UPDATE", the libraries opened for output access may be in an unrecoverable state.

53.4. MODLIB as a Device Prefix

An objmod in a library can be treated as a file by specifying a MODLIB device prefix. Opening a module in an objmod library opens the library for read/write access.

The syntax of a MODLIB device prefix for a library named libName and a module named modName is given by:

```
"MODLIB(" & libName & ")" & $devModBrkStr & modName
```

The case of the letters in "MODLIB" and modName is not important.

For example, on systems where $devModBrk is the character ">", a module FOO in a library "proj-xxx.olb" would be specified by:

```
modlib(proj-xxx.olb)>foo
```

If no library is specified, e.g., if the syntax:

```
"MODLIB" & $devModBrkStr & modName
```

is used, all open objmod libraries are searched for modName. When a MODLIB file f is opened, f.name is modified to include the library name in the file name if it was not included in the call to "open".

MODLIB files cannot be opened for output access; i.e., they must be opened read-only.

53.5. MODLIB Program Interface

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

MODLIB can be controlled from a user program by calling the procedure $executeModlibCommands, declared as shown in Figure 53.5-1.

```
BOOLEAN PROCEDURE $executeModlibCommands
                        (OPTIONAL STRING cmds;
                         OPTIONAL POINTER(textFile) f;
                         OPTIONAL BITS ctrlBits)
```

Figure 53.5-1. Declaration of $executeModlibCommands

cmds is a string which contains exactly what would be typed by a user giving commands to MODLIB. In addition, f can indicate a file from which commands are read once cmds becomes empty. If f is omitted, cmdFile is used. To force MODLIB to return rather than read from f when cmds becomes empty, specify the "QUIT" command as the last command in cmds.

A return value of false indicates that an error occurred during processing of a command. errMsg is called to report the error; the invoking program can intercept the error using the MAINSAIL exception mechanism. In this case, $exceptionStringArg1 starts with the substring "MODLIB:".

54. OBJCOM

OBJCOM compares two object modules to see if they are the same except for their compilation dates. The module can be invoked interactively or it can be called from a program using the procedure $compareObjmods.

OBJCOM has a strict notion of whether or not two modules match; they must occupy the same number of MAINSAIL pages and, except for the compilation date and time fields embedded in the modules, each pair of corresponding storage units must be the same. This means, e.g., that if a module is compiled and then one of its procedures is recompiled without having been changed, OBJCOM would report that the original version and the subsequent version of the module do not match, because the code for the recompiled procedure is not in the same place.

54.1. Interactive Use of OBJCOM

If OBJCOM is run interactively, it first prompts for the target abbreviation of the operating system for which the two object modules to be compared were compiled. Responding with <eol> means that the modules were compiled to run on the host system.

Next, OBJCOM prompts for the names of the files containing the two object modules. The first prompt asks for a file name. The user can type the name of the file containing the object module, or if MAINSAIL can derive the object file name from the module name alone, the user can type just the module name. If the object module is in a library, the user can respond to the first prompt with <eol>, in which case OBJCOM prompts for the name of the library file containing the object module, and then the name of the module. OBJCOM goes through the same sequence to get the names of both object modules.

If the two object modules match, OBJCOM reports that they matched after comparing them. Otherwise, it issues a short message indicating why they did not match. In any event, OBJCOM then prompts for the names of two more object modules to compare. To exit OBJCOM, the user should type <eol> to the prompt for another file name and to the prompt for a library name.

If one of the object modules examined in the previous comparison was in a library, then when OBJCOM prompts for another pair of object module names, it assumes that the corresponding one of the object modules to be compared this time is in the same library, and so it prompts for a module name without first prompting for a file name and then a library name. For example, if during the previous comparison, the first object module was in a library "foo" and the second

was not in a library, OBJCOM assumes that the first module it prompts for during the next iteration is also in the library "foo", and prompts for the name of another module in "foo". It makes no such assumptions about the second module for which it prompts. If the user types <eol> in response to the module name prompt, OBJCOM prompts for the name of a file, as it did when it was first run.

54.2. Calling OBJCOM from a Program

The interface to $compareObjmods is shown in Table 54.2-1.

```
BOOLEAN
PROCEDURE   $compareObjmods
                        (STRING file1,lib1,mod1,file2;
                         OPTIONAL STRING lib2,mod2,target;
                         PRODUCES OPTIONAL STRING msg);
```

Table 54.2-1. $compareObjmods

$compareObjmods returns true if and only if the two object modules specified by its input parameters are the same except for their compilation dates.

file1, lib1, and mod1 specify the first object module to be compared, and file2, lib2, and mod2 the second object module.

file1 can specify the name of the object file, or just the name of the module if MAINSAIL can determine the name of the object file from the module name. If file1 is non-Zero, lib1 and mod1 are ignored. If the object module is in a library, lib1 can specify the library's file name and mod1 can specify the module name within the library, in which case file1 should be the null string. file2, lib2, and mod2 specify the second module in the same way that file1, lib1, and mod1 specify the first.

target specifies the target abbreviation of the operating system for which the two object modules were compiled. If target is omitted or is the null string, the host operating system is assumed.

msg is set only if $compareObjmods returns false. It contains a brief description of why the procedure did not return true, such as that it could not find one of the modules, or that one of them was not really an object module, or that the modules did not match.

55. PDFMOD, Portable Data Format Routines

PDFMOD implements the routines that access PDF data in strings and in memory. These routines are extensively used by the PDF I/O and Structure Blaster procedures. They are useful only for "low-level" manipulation of PDF data; most programs that perform PDF I/O may do so through the standard I/O procedures (read, write, cRead, cWrite, etc.). In particular, if the only use of PDF in a given application is for PDF or Structure Blaster I/O, then there is no reason to use any of the facilities provided by PDFMOD; a normal MAINSAIL file pointer opened for PDF I/O is sufficient, as described in the "MAINSAIL Language Manual".

PDFMOD is a collection of macros, forward procedures, and inline procedures, rather than a module with interface procedures, for efficiency reasons. For a given machine and data type, the PDF representation may be the same as the host data representation. Since this is known at compiletime, the PDF routines use conditional compilation to avoid translating between PDF and host data when it is unnecessary. The use of macros and inline procedures allow many simple routines to be called with no procedure call overhead.

PDFMOD provides procedures to:

- read PDF data from a string or charadr and store them as host data (the data types boolean, (long) integer, (long) real, and (long) bits are supported).
- read host data from a string or charadr and store them as PDF data (the data types boolean, (long) integer, (long) real, and (long) bits are supported).
- read PDF characters from a string or charadr and store them as host characters.
- read host characters from a string or charadr and store them as PDF characters.

To use the PDFMOD procedures described in this section, the following line must be included in the source code:

```
RESTOREFROM "pdfmod";
```

Before calling any of the procedures that read or write PDF data or text, the application program must first call the procedure pdfInit (see Section 55.8) to perform initialization required by those procedures. When no more PDF procedures are to be called, the application program should call pdfDeInit (see Section 55.6).

The tables showing the translation of PDF character codes to host codes and vice versa appear in the "MAINSAIL Language Manual".

55.1. pdfCharRead

```
PROCEDURE   pdfCharRead (MODIFIES CHARADR src,dst;
                         LONG INTEGER numChars);

PROCEDURE   pdfCharRead (MODIFIES STRING src;
                         MODIFIES CHARADR dst;
                         INTEGER numChars);
```

Table 55.1-1. pdfCharRead (Generic)

pdfCharRead reads numChars PDF characters from a string or a charadr and writes them to a charadr as host characters. No translation is done if the host character set and the PDF character set are the same.

In the string form, characters written to dst are removed from src. In the charadr form, src is displaced by the number of characters read. In both forms, dst is displaced by the number of characters written.

In the string form, numChars is automatically adjusted so that it does not exceed the length of the string.

In the charadr form, the result is undefined if the source and destination overlap.

The effect of writing to or reading from nullCharadr is undefined.

55.2. pdfChars

```
INTEGER
<macro>     pdfChars     (INTEGER typeCode);
```

Table 55.2-1. pdfChars

pdfChars returns the number of characters required to represent the host data specified by typeCode as PDF data. pdfChars is undefined if typeCode is not one of booleanCode, integerCode, longIntegerCode, realCode, longRealCode, bitsCode, longBitsCode, addressCode, charadrCode, or pointerCode. The types address, charadr, and pointer are treated as long bits.

55.3. pdfCharWrite

```
PROCEDURE   pdfCharWrite
                        (MODIFIES CHARADR src,dst;
                         LONG INTEGER numChars);

PROCEDURE   pdfCharWrite
                        (MODIFIES STRING dst;
                         MODIFIES CHARADR src;
                         INTEGER numChars);
```

Table 55.3-1. pdfCharWrite

pdfCharWrite reads numChars host characters from a charadr and writes them to a string or a charadr as PDF characters. No translation is done if the host character set and the PDF character set are the same.

In the string form, characters are appended to dst. In the charadr form, characters are written to dst and dst is displaced by the number of characters written. In both forms, src is displaced by the number of characters read.

In the charadr form, the result is undefined if the source and destination overlap.

The effect of writing to or reading from nullCharadr is undefined.

55.4. pdfcRead

```
INTEGER
PROCEDURE   pdfcRead    (MODIFIES CHARADR c);

INTEGER
PROCEDURE   pdfcRead    (MODIFIES STRING s);
```

Table 55.4-1. pdfcRead

pdfcRead returns the host character code for the PDF character read from a string or charadr.

In the string form, the character is removed from the string. -1 is returned by the string form if the string is the null string.

In the charadr form, the charadr is displaced by one character. pdfcRead from nullCharadr is undefined.

55.5. pdfcWrite

```
PROCEDURE   pdfcWrite   (MODIFIES CHARADR c;
                         REPEATABLE INTEGER ch);

PROCEDURE   pdfcWrite   (MODIFIES STRING s;
                         REPEATABLE INTEGER ch);

PROCEDURE   pdfcWrite   (MODIFIES STRING s;
                         POINTER($area) area;
                         REPEATABLE INTEGER ch);
```

Table 55.5-1. pdfcWrite

pdfcWrite writes the PDF character for a host character ch to a string or charadr.

The string forms append the character to s. In the area form, area specifies the destination area for the resulting string.

The charadr form writes the character to its charadr argument. The charadr is then positioned to the next character location. The effect is undefined if the charadr is nullCharadr.

55.6. pdfDeInit

```
PROCEDURE   pdfDeInit;
```

Table 55.6-1. pdfDeInit

pdfDeInit releases resources allocated by pdfInit. pdfDeInit may be called more than once; it has no effect if all resources have already been released.

55.7. pdfFldRead

```
STRING
PROCEDURE   pdfFldRead  (MODIFIES CHARADR c;
                         INTEGER width;
                         OPTIONAL POINTER($area) area);

STRING
PROCEDURE   pdfFldRead  (MODIFIES STRING s;
                         INTEGER width;
                         OPTIONAL POINTER($area) area);
```

Table 55.7-1. pdfFldRead

pdfFldRead reads a PDF field from a string or a charadr and returns a host string. A field is a string with the specified width.

If width is less than one, the null string is returned.

area specifies the destination area for the result string.

In the string form, width is automatically adjusted so that it does not exceed the length of the string. Characters written to the result string are removed from the source string.

In the charadr form, the source charadr is displaced by the number of characters read. The effect of reading from nullCharadr is undefined.

55.8. pdfInit

```
PROCEDURE    pdfInit;
```

Table 55.8-1. pdfInit

pdfInit performs initialization required by other PDFMOD procedures and must be called before any of the other PDFMOD procedures is called. pdfInit may be called more than once; it has no effect if all resources have already been allocated.

55.9. pdfRead

```
PROCEDURE    pdfRead      (MODIFIES CHARADR c;
                           PRODUCES REPEATABLE BOOLEAN v);

PROCEDURE    pdfRead      (MODIFIES CHARADR c;
                           PRODUCES REPEATABLE INTEGER v);

PROCEDURE    pdfRead      (MODIFIES CHARADR c;
                           PRODUCES
                           REPEATABLE LONG INTEGER v);

PROCEDURE    pdfRead      (MODIFIES CHARADR c;
                           PRODUCES REPEATABLE REAL v);
```

Table 55.9-1. pdfRead (continued)

```
PROCEDURE   pdfRead     (MODIFIES CHARADR c;
                         PRODUCES
                         REPEATABLE LONG REAL v);

PROCEDURE   pdfRead     (MODIFIES CHARADR c;
                         PRODUCES REPEATABLE BITS v);

PROCEDURE   pdfRead     (MODIFIES CHARADR c;
                         PRODUCES
                         REPEATABLE LONG BITS v);

PROCEDURE   pdfRead     (MODIFIES CHARADR c;
                         PRODUCES REPEATABLE STRING s);

PROCEDURE   pdfRead     (MODIFIES CHARADR c;
                         POINTER($area) area;
                         PRODUCES REPEATABLE STRING s);

PROCEDURE   pdfRead     (MODIFIES STRING s;
                         PRODUCES REPEATABLE BOOLEAN v);

PROCEDURE   pdfRead     (MODIFIES STRING s;
                         PRODUCES REPEATABLE INTEGER v);

PROCEDURE   pdfRead     (MODIFIES STRING s;
                         PRODUCES
                         REPEATABLE LONG INTEGER v);

PROCEDURE   pdfRead     (MODIFIES STRING s;
                         PRODUCES REPEATABLE REAL v);

PROCEDURE   pdfRead     (MODIFIES STRING s;
                         PRODUCES
                         REPEATABLE LONG REAL v);

PROCEDURE   pdfRead     (MODIFIES STRING s;
                         PRODUCES REPEATABLE BITS v);
```

Table 55.9-1. pdfRead (continued)

```
PROCEDURE   pdfRead     (MODIFIES STRING s;
                         PRODUCES
                         REPEATABLE LONG BITS v);

PROCEDURE   pdfRead     (MODIFIES STRING r;
                         PRODUCES REPEATABLE STRING s);

PROCEDURE   pdfRead
                        (MODIFIES STRING r;
                         POINTER($area) area;
                         PRODUCES REPEATABLE STRING s);
```

Table 55.9-1. pdfRead (end)

pdfRead reads PDF data from a string or charadr and produces host data.

The forms that read boolean, (long) integer, (long) real, or (long) bits read PDF data and the string forms read PDF characters from the source string or charadr.

In the forms that read from a string, characters read are removed from the string. In the forms that read from a charadr, the charadr is displaced by the number of characters read.

In the forms that read a string, the string is obtained by scanning the source for the PDF character eol. If the source is a string, all scanned characters, including the eol, are removed from the string. If the source is a charadr, the charadr is displaced by the number of scanned characters, including the eol. In the area forms, area specifies the destination area for the produced string.

In the forms that read boolean, (long) integer, (long) real, or (long) bits values from a string, the effect is undefined if the length of the string is less than the number of characters required to represent the host data as PDF data. In all forms that read a boolean, (long) integer, (long) real, or (long) bits value, the effect is undefined if the PDF datum is outside the MAINSAIL guaranteed range for the data type.

The effect of reading from nullCharadr is undefined.

55.10. pdfWrite

```
PROCEDURE   pdfWrite    (MODIFIES CHARADR c;
                         REPEATABLE BOOLEAN v);

PROCEDURE   pdfWrite    (MODIFIES CHARADR c;
                         REPEATABLE INTEGER v);

PROCEDURE   pdfWrite    (MODIFIES CHARADR c;
                         REPEATABLE LONG INTEGER v);

PROCEDURE   pdfWrite    (MODIFIES CHARADR c;
                         REPEATABLE REAL v);

PROCEDURE   pdfWrite    (MODIFIES CHARADR c;
                         REPEATABLE LONG REAL v);

PROCEDURE   pdfWrite    (MODIFIES CHARADR c;
                         REPEATABLE BITS v);

PROCEDURE   pdfWrite    (MODIFIES CHARADR c;
                         REPEATABLE LONG BITS v);

PROCEDURE   pdfWrite    (MODIFIES CHARADR c;
                         REPEATABLE STRING s);

PROCEDURE   pdfWrite    (MODIFIES STRING s;
                         REPEATABLE BOOLEAN v);

PROCEDURE   pdfWrite    (MODIFIES STRING s;
                         REPEATABLE INTEGER v);

PROCEDURE   pdfWrite    (MODIFIES STRING s;
                         REPEATABLE LONG INTEGER v);

PROCEDURE   pdfWrite    (MODIFIES STRING s;
                         REPEATABLE REAL v);
```

Table 55.10-1. pdfWrite (continued)

```
PROCEDURE   pdfWrite    (MODIFIES STRING s;
                         REPEATABLE LONG REAL v);

PROCEDURE   pdfWrite    (MODIFIES STRING s;
                         REPEATABLE BITS v);

PROCEDURE   pdfWrite    (MODIFIES STRING s;
                         REPEATABLE LONG BITS bb);

PROCEDURE   pdfWrite    (MODIFIES STRING r;
                         REPEATABLE STRING s);

PROCEDURE   pdfWrite    (MODIFIES STRING s;
                         POINTER($area) area;
                         REPEATABLE BOOLEAN v);

PROCEDURE   pdfWrite    (MODIFIES STRING s;
                         POINTER($area) area;
                         REPEATABLE INTEGER v);

PROCEDURE   pdfWrite    (MODIFIES STRING s;
                         POINTER($area) area;
                         REPEATABLE LONG INTEGER v);

PROCEDURE   pdfWrite    (MODIFIES STRING s;
                         POINTER($area) area;
                         REPEATABLE REAL v);

PROCEDURE   pdfWrite    (MODIFIES STRING s;
                         POINTER($area) area;
                         REPEATABLE LONG REAL v);

PROCEDURE   pdfWrite    (MODIFIES STRING s;
                         POINTER($area) area;
                         REPEATABLE BITS v);

PROCEDURE   pdfWrite    (MODIFIES STRING s;
                         POINTER($area) area;
                         REPEATABLE LONG BITS v);
```

Table 55.10-1. pdfWrite (continued)

```
PROCEDURE   pdfWrite    (MODIFIES STRING r;
                         POINTER($area) area;
                         REPEATABLE STRING s);
```

Table 55.10-1. pdfWrite (end)

pdfWrite writes host data to a string or charadr as PDF data.

The forms that write boolean, (long) integer, (long) real, or (long) bits write PDF data and the string forms write PDF characters to the destination string or charadr.

In the forms that write to a string, the characters written are appended to the string. In the forms that write to a charadr, the charadr is displaced by the number of characters written.

In the forms that write to a string and in which area is specified, area is the destination area for the resulting string.

The effect of the forms that write boolean, (long) integer, (long) real, or (long) bits is undefined if the value is outside the MAINSAIL guaranteed range.

The effect of writing to nullCharadr is undefined.

55.11. The PDF Charadr Read Procedures

```
PROCEDURE   pdfBoRead   (MODIFIES CHARADR src,dst);

PROCEDURE   pdfiRead    (MODIFIES CHARADR src,dst);

PROCEDURE   pdfLiRead   (MODIFIES CHARADR src,dst);

PROCEDURE   pdfrRead    (MODIFIES CHARADR src,dst);

PROCEDURE   pdfLrRead   (MODIFIES CHARADR src,dst);
```

Table 55.11-1. The PDF Charadr Read Procedures (continued)

```
PROCEDURE   pdfbRead    (MODIFIES CHARADR src,dst);

PROCEDURE   pdfLbRead   (MODIFIES CHARADR src,dst);
```

Table 55.11-1. The PDF Charadr Read Procedures (end)

The PDF charadr read procedures read PDF data from a source and write them to a destination as host data.

The source charadr is displaced by the number of characters read. The destination charadr is displaced by the number of characters written.

The effect of reading from or writing to nullCharadr is undefined.

55.12. The PDF Charadr Write Procedures

```
PROCEDURE   pdfBoWrite  (MODIFIES CHARADR src,dst);

PROCEDURE   pdfiWrite   (MODIFIES CHARADR src,dst);

PROCEDURE   pdfLiWrite  (MODIFIES CHARADR src,dst);

PROCEDURE   pdfrWrite   (MODIFIES CHARADR src,dst);

PROCEDURE   pdfLrWrite  (MODIFIES CHARADR src,dst);

PROCEDURE   pdfbWrite   (MODIFIES CHARADR src,dst);

PROCEDURE   pdfLbWrite  (MODIFIES CHARADR src,dst);
```

Table 55.12-1. The PDF Charadr Write Procedures

The PDF charadr write procedures read host data from a source and write them to a destination as PDF data.

The source charadr is displaced by the number of characters read. The destination charadr is displaced by the number of characters written.

The effect of reading from or writing to nullCharadr is undefined.

56. PMERGE, Page Merger

The page merger module, PMERGE, forms an output text file from selected pages of existing input text files. PMERGE prompts for the name of the output file to be created. A "?" typed in response to this prompt causes a brief introduction to be displayed.

Once the output file has been specified, PMERGE prompts for input files and the pages to be included from each. To exit PMERGE, type <eol> in response to the "Next input file" prompt.

After an input file has been specified, PMERGE prompts for the pages to be included from that file. The "Pages" prompt expects either a line of page specifications, an <eol> or "?". If <eol> is entered, PMERGE prompts for the next input file. Table 56-1 lists the available page specification forms and is displayed when "?" is typed in response to the "Pages" prompt.

Page Spec	Description
m	copy page m
m-n	copy page m through page n
*	copy the whole file
m-*	copy page m to the end of the file
-<pageSpec>	copy pages in pageSpec only if not already copied

Table 56-1. PMERGE Page Specifications

Any number of page specifications may be entered on a line, separated by commas (no spaces). The first page in a file is page 1. Example 56-2 gives an example of a page specification line.

Example 56-3 shows a sample session with PMERGE. Pages from "file1" and "file2" are written to "outFile". PMERGE displays the pages that it includes after each page specification line.

```
If a file is 13 pages long and the following page
specification is given:

    9-10,5,7,9,11,-5-11,-*

pages are copied from the input file as follows:

    9,10,5,7,9,11,6,8,1,2,3,4,12,13
```

Example 56-2. PMERGE Page Specification Example

```
MAINSAIL (R) Version 12.10 (? for help)
Copyright (c) 1984, 1985, 1986, 1987, 1988, and 1989 by
 XIDAK, Inc., Menlo Park, California, USA.
*pmerge<eol>
Output file (? for help): outFile<eol>

Next input file (eol to stop): file1<eol>

Pages (eol to stop,? for help): 3-5,8,2,-4-10<eol>
 3 4 5 8 2 6 7 9 10
Pages (eol to stop,? for help): <eol>

Next input file (eol to stop): file2<eol>

Pages (eol to stop,? for help): 1,3<eol>
 1 3
Pages (eol to stop,? for help): <eol>

Next input file (eol to stop): <eol>
```

Example 56-3. PMERGE Example

57. $ranCls and $ranMod, Pseudo-Random Number Generator

The standard MAINSAIL pseudo-random number generator is a module $ranMod of class $ranCls. The declarations of $ranCls and $ranMod are shown in Table 57-1.

```
CLASS $ranCls (

LONG INTEGER
PROCEDURE          $rand;

PROCEDURE          $initRand          (LONG INTEGER newX;
                                       OPTIONAL LONG INTEGER
                                           newX2);

LONG INTEGER
PROCEDURE          $sRand;

PROCEDURE          $initsRand         (LONG INTEGER ix);

);

MODULE($ranCls)  $ranMod;
```

Table 57-1. $ranCls and $ranMod

Each of the procedures $rand and $sRand returns the next value produced by a pseudo-random number generator. The two generators are different. $initRand specifies a seed for the $rand generator, and $initsRand specifies a seed for the $sRand generator. Both pseudo-random number generators are derived from Volume 2 of Donald Knuth's "The Art of Computer Programming" (second edition. Reading, Massachusetts: Addison-Wesley Publishing Company, 1973).

The $rand generator uses a linear congruence algorithm:

```
x[n + 1] = (x[n] * a + c) MOD m
c = 6364136223846793005
m = 2 ^ 64
```

where the xi are successive values computed by $rand; $rand returns the high-order (most random) 31 bits of xi on each call. The values returned by $rand thus range from 0L to 2147483647L. The generator has a period of 2 ^ 64.

The $srand generator uses the "subtractive method" generator algorithm presented in Section 3.6 of "The Art of Computer Programming". It returns values in the range 0L to 999999999L (10 ^ 9 - 1). The period is probably quite long.

The properties of the $rand generator are somewhat better understood mathematically than those of the $sRand generator. The $rand generator is therefore, in some sense, "more certainly" random than the $sRand generator; however, it executes somewhat more slowly. For most practical purposes, either generator is satisfactorily random.

The parameters to $initRand may be any nonnegative integers (although only the low-order 28 bits of each are examined; i.e., the numbers are taken modulo 2 ^ 28). The parameter to $initsRand may be any number in the range 0L to 999999999L.

Applications that require that the same sequence of pseudo-random numbers always be produced from a given seed (perhaps for debugging purposes) should create a private copy of the pseudo-random number generator by declaring a pointer to $ranCls and initializing the pointer by calling "new":

```
POINTER($ranCls) p;
        .
        .
        .
p := new($ranMod);
```

The pointer p may then be used to call procedures in the private copy of $ranMod, e.g.:

```
li := p.$rand
```

or:

```
li := p.$sRand
```

The common (bound) copy of $ranMod may be in use simultaneously by a number of modules. If it is not important to an application that the same sequence of pseudo-random numbers be

produced on each execution, then it may share the bound copy of $ranMod, and need not use a direct pointer to call the procedures in $ranMod, e.g.:

```
li := $rand
```

or:

```
li := $sRand
```

XIDAK reserves the right to add new fields to $ranCls or to add new optional parameters to the procedures in $ranCls.

58. RNMFIL, File Renamer

Command line syntax: rnmfil {fileName {newFileName}}*

The module RNMFIL allows the user to rename one or more files from within MAINSAIL. The same device module must be used for both the old and new files; i.e., renaming files across device modules is not supported.

RNMFIL accepts pairs of file names on the command line; each file named fileName is renamed to newFileName, if possible. If no arguments are given, RNMFIL prompts for the name of each file to be renamed. Example 58-1 shows how to use RNMFIL to rename files from within MAINSAIL. The file "foo" is renamed to be "bar", and the file "baz" is renamed to be "newbaz".

```
MAINSAIL (R) Version 12.10 (? for help)
Copyright (c) 1984, 1985, 1986, 1987, 1988, and 1989 by
 XIDAK, Inc., Menlo Park, California, USA.
*rnmfil<eol>
Next file to be renamed (just eol to stop): foo<eol>
New file name: bar<eol>
Next file to be renamed (just eol to stop): baz<eol>
New file name: newbaz<eol>
Next file to be renamed (just eol to stop): <eol>
*

                Alternatively:

*rnmfil foo bar baz newbaz<eol>
*
```

Example 58-1. RNMFIL Example

If an error occurs when trying to rename the file specified, RNMFIL issues an error message and prompts for an error response. If <eol> is typed in response to the "Error Response:" prompt, the user is given a chance to respecify the file names. Example 58-2 shows this interaction. The example assumes that the file "foo" does not exist and that the file "baz" exists and can be opened. An error message is given saying that the file "foo" could not be found.

When respecifying the file names, typing <eol> causes the previously specified name to be used (<eol> in response to the "New file name (just eol for bar):" prompt causes the file "baz" to be renamed to be "bar").

```
MAINSAIL (R) Version 12.10 (? for help)
Copyright (c) 1984, 1985, 1986, 1987, 1988, and 1989 by
 XIDAK, Inc., Menlo Park, California, USA.
*rnmfil<eol>
Next file to be renamed (just eol to stop): foo<eol>
New file name: bar<eol>

File not found

ERROR: $rename: could not find old file foo => bar
Error Response: <eol>
Want to specify new names (Yes or No): y<eol>
File to rename (just eol for foo): baz<eol>
New name (just eol for bar): <eol>
Next file to be renamed (just eol to stop): <eol>
*
```

Example 58-2. RNMFIL Example with a Rename Error

59. SRTMOD, Sorting Package

The sorting package provides the ability to sort one-dimensional arrays based on the values of their elements, and facilities from which a programmer may generate sorting procedures for other data structures. The sort may be an in-place sort (the elements of the array are rearranged) or an index sort (a parallel array of subscripts (indices) is rearranged to represent the sorted arrangement of the array to be sorted). The sorting algorithm used is a mixture of quicksort and bubble sort. Procedures are also provided to reverse the order of elements of a one-dimensional array or to rearrange them based on an index array.

The facilities in the sorting package are located in an intmod called SRTMOD. Therefore, to use the identifiers described herein, either use the directive:

```
RESTOREFROM "srtmod";
```

or prefix each identifier described herein with "srtmod$", e.g., "srtmod$sort".

In the description of the procedures in this chapter, where instances of a generic procedure have analogous parameters of a number of different types, the notation "<data type>" may be used in place of the actual data type names in the procedure headers. The description of the procedure enumerates the possible values for "<data type>".

59.1. Sorting Arrays of Ordered Data Types

Ordered data types, for the purpose of this section, are boolean, string, (long) integer, and (long) real. In the case of boolean, true is considered greater than false; strings are compared caselessly, as by the system procedure "compare" with the upperCase bit set; the other data types are ordered as by the MAINSAIL comparison operators.

<data type> in the declaration of sort may be boolean, integer, long integer, real, long real, or string. All forms sort the elements of a from lb to ub, or from a.$lb1 to a.$ub1 if lb and ub are not specified. An error message is issued if lb and ub are specified and either is outside the bounds of the array.

The in-place forms of sort take only one (long) array parameter a. The elements of a are reordered so that every element has a value greater than or equal to the values at all lesser subscripts; i.e., greater values are at greater subscripts. An error occurs if a is Zero.

```
PROCEDURE     sort        (<data type> LONG ARRAY(*) a;
                           OPTIONAL LONG INTEGER lb,ub);

PROCEDURE     sort        (<data type> ARRAY(*) a;
                           OPTIONAL INTEGER lb,ub);

PROCEDURE     sort        (<data type> LONG ARRAY(*) a;
                           LONG INTEGER lb,ub;
                           MODIFIES LONG INTEGER
                                LONG ARRAY(*) ind);

PROCEDURE     sort        (<data type> ARRAY(*) a;
                           INTEGER lb,ub;
                           MODIFIES INTEGER ARRAY(*) ind);
```

Table 59.1-1. sort (Generic)

The index forms of sort take a (long) array parameter a and a (long) array index array parameter ind. a is not modified. If ind is Zero or if its bounds do not equal the bounds of the subrange of a to be examined, it is reallocated to be the right size. The elements of ind are set to the ordered subscripts of a; i.e., successive elements of ind, when used as subscripts of a, result in a sorted series of values, greater values at greater subscripts of ind.

Index sorts are useful if several parallel arrays with related values are to be sorted, e.g., arrays of names and dates, where the date attached to a name has the same index in one array as the name in the other. Sorting each array in place would destroy the relationship between the names and the dates. An index sort does not modify the name and date arrays, but can produce the ordering of the two arrays based on either the name or the date. If the program of Example 59.1-2 is given the input file shown in Example 59.1-3, the output is as shown in Example 59.1-4.

```
BEGIN "parSrt"

# Display a sorted listing of modules along with dates of
# creation.  An unsorted list is read from a file in which
# the first line has the number of module names in the
# file and each subsequent line has the name of a module
# followed by a date.

RESTOREFROM "srtmod";

INITIAL PROCEDURE;
BEGIN
INTEGER i,j;
STRING s;
POINTER(textFile) f;
INTEGER ARRAY(1 TO *) ind;
LONG INTEGER ARRAY(1 TO *) date;
STRING ARRAY(1 TO *) name;
open(f,"Unsorted file: ",prompt!input);
read(f,s); j := cvi(s); new(date,1,j); new(name,1,j);
FOR i := 1 UPTO j DOB
    read(f,s); name[i] := scan(s," " & tab);
    date[i] := $strToDate(s) END;
close(f);
IF confirm("Sort by name (else by date)") THEN
    sort(name,1,j,ind)
EL  sort(date,1,j,ind);
FOR i := 1 UPTO j DOB
    fldWrite(logFile,name[ind[i]],8,- ' ');
    write(logFile,$dateToStr(date[ind[i]],$hyphenateDate),
        eol) END;
END;

END "parSrt"
```

Example 59.1-2. A Parallel Sorting Program

```
8
AMOD 20-oct-86
ZMOD 20-jan-87
X 13-feb-87
PHOTO 12-jun-85
TIMER 9-oct-86
POST 11-jan-85
PRE 6-jul-86
INTER 9-aug-86
```

Example 59.1-3. Input File "parsrt.dat" for PARSRT

```
*parsrt<eol>
Unsorted file: parsrt.dat<eol>
Sort by name (else by date) (Yes or No): y<eol>
AMOD     20-Oct-86
INTER    9-Aug-86
PHOTO    12-Jun-85
POST     11-Jan-85
PRE      6-Jul-86
TIMER    9-Oct-86
X        13-Feb-87
ZMOD     20-Jan-87
*parsrt<eol>
Unsorted file: parsrt.dat<eol>
Sort by name (else by date) (Yes or No): n<eol>
POST     11-Jan-85
PHOTO    12-Jun-85
PRE      6-Jul-86
INTER    9-Aug-86
TIMER    9-Oct-86
AMOD     20-Oct-86
ZMOD     20-Jan-87
X        13-Feb-87
```

Example 59.1-4. PARSRT Execution

59.2. Sorting Arrays with User-Defined Ordering

The macro generateQuickSort is used to generate a generic procedure to perform a sort with a user-defined ordering. This is useful, e.g., if the elements of the array to be sorted are records of which one or more fields must be examined to determine their relationship. generateQuickSort may be called wherever a procedure declaration is legal.

generateQuickSort takes five arguments as follows:

```
generateQuickSort (
    <procName>,
    <elementType>,
    <isLessThan>,
    <isLEQ>,
    <isEqual>);
```

<procName> is the name of the generic procedure to be generated. <elementType> is the data type of the elements of the array, e.g., "POINTER(foo)". <isLessThan>, <isLEQ>, and <isEqual> are the names of boolean macros or procedures that take two arguments of type <elementType>; they return true if and only if the first element is, respectively, less than, less than or equal to, or equal to the second.

The generated generic procedure declaration and the headers of the generated procedures look like:

```
GENERIC PROCEDURE <procName>
    "<procName>InPlace,<procName>InPlaceShort," &
    "<procName>Indexed,<procName>IndexedShort";

PROCEDURE <procName>InPlace
        (<elementType> LONG ARRAY(*) a;
         OPTIONAL LONG INTEGER alb,aub);

PROCEDURE <procName>InPlaceShort
        (<elementType> ARRAY(*) a;
         OPTIONAL INTEGER alb,aub);

PROCEDURE <procName>Indexed
        (<elementType> LONG ARRAY(*) a;
         LONG INTEGER alb,aub;
         MODIFIES LONG INTEGER LONG ARRAY(*) ind);

PROCEDURE <procName>IndexedShort
        (<elementType> ARRAY(*) a;
         INTEGER alb,aub;
         MODIFIES INTEGER ARRAY(*) ind);
```

Additional procedures, the names of which begin with <procName>, may be generated but are not intended to be called directly.

The parameters a, alb, aub, and ind correspond to the parameters a, lb, ub, and ind to sort; provided that the arguments to generateQuickSort are well-formed and correct, the generated procedure functions exactly like sort except that the ordering criteria are provided by the programmer. For example, the standard real array forms of sort could be generated by:

```
DEFINE
    rLessThan(a,b) = [(a < b)],
    rLEQ(a,b) = [(a LEQ b)],
    rEqual(a,b) = [(a = b)];
generateQuickSort(sort,REAL,rLessThan,rLEQ,rEqual);
```

59.3. Multiple-Array or Non-Array Sorting

The macro generateMultipleQuickSort may be used to generate a sort based on multiple arrays, two- or three-dimensional arrays, or non-array data structures. generateMultipleQuickSort takes four or five parameters as follows:

```
generateMultipleQuickSort (
    <procName>,
    <isLessThan>,
    <isLEQ>,
    <isEqual>);
```

or:

```
generateMultipleQuickSort (
    <procName>,
    <isLessThan>,
    <isLEQ>,
    <isEqual>,
    LONG);
```

<procName> is the name of the generic procedure to be generated. <isLessThan>, <isLEQ>, and <isEqual> are the names of boolean macros or procedures that take two arguments of type integer (if "LONG" is absent) or long integer (if "LONG" is present); they return true if and only if the (program-defined) quantity corresponding to the first argument is, respectively, less than, less than or equal to, or equal to the second.

If "LONG" is present, the generated generic procedure declaration and the header of the generated procedure look like:

```
GENERIC PROCEDURE <procName> "<procName>Indexed";

PROCEDURE <procName>Indexed
    (LONG INTEGER alb,aub;
     MODIFIES LONG INTEGER LONG ARRAY(*) ind);
```

If "LONG" is absent, the generated generic procedure declaration and the header of the generated procedure look like:

```
GENERIC PROCEDURE <procName> "<procName>IndexedShort";

PROCEDURE <procName>IndexedShort
    (INTEGER alb,aub;
     MODIFIES INTEGER ARRAY(*) ind);
```

By calling generateMultipleQuickSort twice with the same <procName>, once with "LONG" present and once with it absent, a generic procedure with two instances is generated. The generated procedure sets ind in the range alb to aub to reflect the ordering specified by <isLessThan>, <isLEQ>, and <isEqual>.

Example 59.3-1 shows how to sort two parallel arrays a and b, with a as the primary and b the seconday sort. The arrays a and b must be outer variables, since they cannot be passed as arguments to the generated sort procedures.

```
INTEGER ARRAY(*) a,b;

DEFINE
    myLT(i,j) =
        [a[i] < a[j] OR (a[i] = a[j] AND b[i] < b[j])],
    myLEQ(i,j) =
        [a[i] < a[j] OR (a[i] = a[j] AND b[i] LEQ b[j])],
    myEQ(i,j) =
        [a[i] = a[j] AND b[i] = b[j]];

generateMultipleQuickSort(myGenSort,myLT,myLEQ,myEQ);

        .
        .
        .

PROCEDURE foo;
BEGIN
INTEGER ARRAY(*) ind;
        .
        .
        .
myGenSort(1,1000,ind);
# ind now contains the sorted subscripts.  To reorder both
# a and b according to ind:
reorder(a,ind); reorder(b,ind);
        .
        .
        .
```

Example 59.3-1. Use of generateMultipleQuickSort

59.4. Reversing an Array

```
PROCEDURE    reverse      (<data type> ARRAY(*) a;
                           OPTIONAL INTEGER lb,ub);

PROCEDURE    reverse      (<data type> LONG ARRAY(*) a;
                           OPTIONAL LONG INTEGER lb,ub);
```

Table 59.4-1. reverse (Generic)

<data type> in the declaration of reverse may be any MAINSAIL data type: boolean, integer, long integer, real, long real, bits, long bits, string, address, charadr, or pointer. reverse reverses an array a from lb to ub, or from a.$lbl to a.$ubl if lb and ub are not specified. The elements of a are rearranged to put the value at the highest subscript at the lowest subscript and vice versa, and the second highest at the second lowest and vice versa, and so on. An error occurs if a is Zero or lb or ub falls outside the range a.$lbl to a.$ubl.

59.5. Reordering an Array According to an Index Array

```
PROCEDURE    reorder      (<data type> ARRAY(*) a;
                           INTEGER ARRAY(*) ind);

PROCEDURE    reorder      (<data type> LONG ARRAY(*) a;
                           LONG INTEGER LONG ARRAY(*) ind);
```

Table 59.5-1. reorder (Generic)

<data type> in the declaration of reorder may be any MAINSAIL data type: boolean, integer, long integer, real, long real, bits, long bits, string, address, charadr, or pointer. reorder rearranges the elements of a so that the value at i is moved to ind[i] for i in the range ind.$lbl to ind.$ubl. ind is not modified. An error occurs if a or ind is Zero or if the bounds of ind fall outside the bounds of a. The effect is undefined if ind is not a valid index array, i.e., does not contain exactly one occurrence of each value in the range ind.$lbl to ind.$ubl.

60. STAMP and Object Module Security

Several features are provided to give a software distributor some control over the remote use of individual object modules. The utility module STAMP is used to set the security fields in an object module.

An expiration date can limit the use of an object module for up to five years from the date the expiration date is put into the object module.

A password can be put into an object module. If a password is present, then whenever the module is initialized for execution, MAINSAIL prompts for the password and allows use only if the user provides the password.

A set of CPU ID's (as would be returned by the $cpuId procedure) can be put into an object module. MAINSAIL does not allow use of the module on a CPU that is not included in the specified set.

The CPU ID is unavailable on many operating systems; therefore, it is not possible to CPU-stamp modules for many target systems (unless an $alternateCpuID procedure is provided by the user, as is possible on some operating systems; see the operating-system-specific MAINSAIL user's guide for details).

The security fields are stored into the object module in an encrypted form so that the unscrupulous user cannot easily alter their values. STAMP is designed so that the distributor can send information to a customer about how to change the security fields in one way, and yet the customer cannot use that information to alter the fields in other ways.

STAMP stores an encrypted key into each object module. This key must be kept secret. The purpose of the key is to allow secure change access by the customer. A different key should be used for every stamped module/customer pair; the distributor must remember the keys for each module and customer. Typically, only one critical module per "package" need be stamped to control the package's use. For example, a critical module of an application can be stamped with an expiration date two months in the future. At that time, the expiration date can be extended if the customer buys the package.

There is a many-to-one mapping between passwords and CPU ID's and the encrypted values stored in the object module, so that it is possible (although unlikely) that a bogus value would be accepted by the MAINSAIL runtime system.

The security features are designed more for low runtime overhead than absolute security, and hence should not be relied upon to protect truly sensitive data or code. At the very least, a customer could use STAMP with random values until he or she eventually came up with a value that allowed use on an unauthorized CPU (though this might take quite some time). The purpose of the security fields is to require the customer to undertake a non-trivial effort to alter the security fields to his or her advantage, thereby providing clear evidence that the customer has purposely broken a contract. This avoids the situation of a well-intentioned but undisciplined customer claiming that a package was used beyond an agreed-upon expiration date or on an unauthorized CPU due to forgetfulness or uninformed users.

60.1. Expiration Date

An encrypted expiration date can be put into an object module, along with an encrypted key that provides secure change access for the expiration date. The encrypted key and expiration date depend on the contents of the object module, so that neither is valid in another object module. The encrypted expiration date also depends on the key. When an object module is first created, the compiler sets the encrypted date and key to encrypted "match-all" values.

Whenever a module is initialized for execution, MAINSAIL aborts the module allocation if the key is not match-all, and the current date is greater than the expiration date or the expiration date is more than five years in the future (this makes it more difficult for a user to put a randomly selected bit pattern for an encrypted expiration date into the object module and successfully extend the expiration date). Thus, an expiration date can be at most five years from the current date, which is consistent with the purpose of an expiration date as a means of limiting use of a program to a reasonably short trial period.

The module STAMP is used to set the expiration date in an object module. The expiration date can be set only if the user knows the key, or the key is match-all. Example 60.1-1 shows how to stamp a module FOO with an expiration date, and at the same time set the key if it is currently match-all.

```
*stamp<eol>
MAINSAIL (R) Module Security (? for help)
STAMP: setExpirationDate<eol>
Module file: foo-<sys>.obj<eol>
Expiration date (just <eol> for none): dec 31 1983<eol>
Key: 'H12A8CD0<eol>
```

Example 60.1-1. "setExpirationDate"

"?" may be typed to the "STAMP:" prompt to get a summary of the valid responses. Only as much of a command as is needed to make it unique need be typed; for example, "sete" is sufficient for "setExpirationDate" (case is irrelevant). The expiration date can be any valid MAINSAIL date string format. If just <eol> is typed for the date, then the expiration date is set to match-all; i.e., there is no expiration date.

STAMP gives an error message if the expiration date is more than five years away, or if the key encrypted in the object module is neither match-all nor the supplied key. The key may be any one- to eight-digit hexadecimal value (case is irrelevant). Once a key is set for a module, that key must be used in all subsequent commands for the module (unless a "setToCompiledState" command is given, which sets the key to match-all and therefore allows a new key to be given for the next command).

Since the above interaction makes explicit use of the key, it is not safe to be carried out by a customer, since the customer, knowing the key, could use STAMP to change the expiration date again. However, STAMP can also be used to change the expiration date in a way that does not reveal the key. This is a two-step process. The first step is carried out privately by the software distributor, who knows the key, and the second by the customer, who does not. First, the distributor secretly gets a "mixed-up" expiration date, as shown in Example 60.1-2.

```
*stamp<eol>
MAINSAIL (R) Module Security (? for help)
STAMP: mixupExpirationDate<eol>
Expiration date (just <eol> for none): dec 31 1983<eol>
Key: 'H12A8CD0<eol>
Mixed up date is 'H7A39F10E
```

Example 60.1-2. "mixupExpirationDate"

The distributor then tells the customer the mixed-up date value, and the customer uses STAMP to set the expiration date, as in Example 60.1-3.

The effect is the same as if the "setExpirationDate" command for the originally specified date had been used.

```
*stamp<eol>
MAINSAIL (R) Module Security (? for help)
STAMP: setMixedupExpirationDate<eol>
Module file: foo-<sys>.obj<eol>
Mixed up expiration date: 'H7A39F10E<eol>
```

Example 60.1-3. "setMixedupExpirationDate"

60.2. Password

An encrypted password can be put into an object module, along with an encrypted key (the same key used with the expiration date). Whenever a module is initialized for execution, MAINSAIL checks whether the module contains other than the "match-all" password for that module, and if so, prompts the user for the password. Use of the module is allowed only if the user password matches. The compiler initializes the password to the encrypted match-all value.

The encrypted password depends on the contents of the object module and the key, so that it is not valid in another object module.

STAMP is used to set the password in an object module. The password can be set only if the user knows the key, or the key is match-all. Example 60.2-1 shows how to stamp a module FOO with a keyword, and at the same time set the key if it is currently match-all.

```
*stamp<eol>
MAINSAIL (R) Module Security (? for help)
STAMP: setPassword<eol>
Module file: foo-<sys>.obj<eol>
New password (just <eol> for none): monkey<eol>
Key: 'H12A8CD0<eol>
```

Example 60.2-1. "setPassword"

STAMP gives an error message if the key encrypted in the object module is neither match-all nor the supplied key. A password is any string. If just <eol> is typed, then the password is set to match-all; i.e., there is no password.

Just as for setting the expiration date, STAMP can be used to have the customer set the password. This is not as useful as remote setting of the expiration date, since while it is in the customer's best interest to extend the expiration date, it is not in his or her interest to stamp a password into a module. Nevertheless, the capability of remotely setting the password is provided should it be useful. The distributor does as shown in Example 60.2-2.

```
*stamp<eol>
MAINSAIL (R) Module Security (? for help)
STAMP: mixupPassword<eol>
Password (just <eol> for none): monkey<eol>
Key: 'H12A8CD0<eol>
Mixed up password is 'H7A39F10E
```

Example 60.2-2. "mixupPassword"

The distributor then tells the customer the mixed-up password, and the customer uses STAMP to set the password, as in Example 60.2-3.

```
*stamp<eol>
MAINSAIL (R) Module Security (? for help)
STAMP: setMixedupPassword<eol>
Module file: foo-<sys>.obj<eol>
Mixed up password: 'H7A39F10E<eol>
```

Example 60.2-3. "setMixedupPassword"

The effect is the same as if the "setPassword" command had been used.

60.3. Target CPU ID's

An encrypted set of CPU ID's can be put into an object module, along with an encrypted key (the same key used with the expiration date). Whenever a module is initialized for execution, MAINSAIL aborts the module allocation if the encrypted CPU ID's are not "match-all" and the value returned by $cpuId is not contained in the encrypted set. The compiler sets the encrypted CPU ID's to match-all.

The encrypted CPU ID's depend on the contents of the object module and the key, so that they are not valid in another object module.

The module STAMP is used to set the CPU ID's in an object module. The CPU ID's can be set only if the user knows the key, or the key is match-all. To stamp a module FOO with three CPU ID's, and at the same time set the key if it is currently match-all, do as in Example 60.3-1.

```
*stamp<eol>
MAINSAIL (R) Module Security (? for help)
STAMP: setCpuIds<eol>
Module file: foo-<sys>.obj<eol>
Next CPU Id: AE-3451<eol>
Next CPU Id: DD-1298<eol>
Next CPU Id: AD-8632<eol>
Next CPU Id: <eol>
Key: 'H12A8CD0<eol>
```

Example 60.3-1. "setCpuIds"

A CPU ID is a string. Terminate the CPU ID list with <eol>. Use just <eol> for the first CPU ID to indicate the match-all value. STAMP gives an error message if the key encrypted in the object module is neither match-all nor the supplied key.

Just as for setting the expiration date, STAMP can be used to have the customer set the CPU ID's. The software distributor does as shown in Example 60.3-2.

The distributor then tells the customer the mixed-up CPU ID values (there are always four), and the customer uses STAMP to set the CPU ID's, as shown in Example 60.3-3.

The CPU ID's must be given in the same order as printed out by "mixupCpuIds". The effect is the same as if the "setCpuIds" command had been used.

60.4. Target System

The command "target <target system abbreviation>" specifies the system for which the module to be stamped was compiled. If <target system abbreviation> is omitted, the host system is assumed. The command "target ?" displays a list of possible system abbreviations. The

```
*stamp<eol>
MAINSAIL (R) Module Security (? for help)
STAMP: mixupCpuIds<eol>
Next CPU Id: AE-3451<eol>
Next CPU Id: DD-1298<eol>
Next CPU Id: AD-8632<eol>
Next CPU Id: <eol>
Key: 'H12A8CD0<eol>
Mixed up CPU Ids are
            'H983C879F 'HB3187993 'H8F474CCA 'HCB278901
```

Example 60.3-2. "mixupCpuIds"

```
*stamp<eol>
MAINSAIL (R) Module Security (? for help)
STAMP: setMixedupCpuIds<eol>
Module file: foo-<sys>.obj<eol>
Mixed up CPU Id 1: 'H983C879F<eol>
Mixed up CPU Id 2: 'HB3187993<eol>
Mixed up CPU Id 3: 'H8F474CCA<eol>
Mixed up CPU Id 4: 'HCB278901<eol>
```

Example 60.3-3. "setMixedupCpuIds"

"target" command remains in effect until either a new "target" command is given or the STAMP session is terminated.

60.5. Stamping Modules in Libraries

The STAMP command "library" allows modules to be stamped directly in libraries. The forms of the command are shown in Table 60.5-1 (<library name> is the file name of a module library; <module list> is a series of module names separated by spaces or tabs).

Syntax	Description
library <library name>	stamp all modules in <library name>.
library <library name> <module list>	stamp the modules in <library name> specified in <module list>.
library	resume stamping modules not contained in libraries.

Table 60.5-1. Forms of the STAMP "library" Command

The "library" command stays in effect until either a new "library" command is given or the STAMP session is terminated. All subsequent STAMP commands affect the modules specified in the "library" command.

The first time a module is stamped, the user is asked to specify a key, which is encrypted and stored somewhere in the module. The STAMP commands "setExpirationDate", "setPassword", "setCpuIds", and "setToCompiledState" require the user to specify a key the encrypted value of which must match the value stored in a module in order for the commands to be performed on that module. All modules specified in a single "library" command are assumed to have the same key, so that if modules in a library are being stamped, the user is requested to enter a key only during the first STAMP command that matches a key against the key stored in a module. If all modules being stamped match that key or have not been stamped before, then subsequent such STAMP commands use that key instead of requiring that the key be respecified. Thus, the dialogue to stamp a module FOO in a library "bar.lib" with both an expiration date and a password would resemble that shown in Example 60.5-2.

If the key specified by the user does not match the key stored in a module, a message to that effect is written to logFile, and that module is not stamped. STAMP still tries to stamp the remaining modules; i.e., it does not stop in the middle of stamping a list of modules because it cannot stamp one. However, at the next STAMP command that uses a key, the user is prompted for another key.

The commands "mixUpExpirationDate", "mixUpPassword", and "mixUpCpuIds" always request a key, since they do not match the key against the key stored in a module.

```
*stamp<eol>
MAINSAIL (tm) Module Security (? for help)
STAMP: library bar.lib foo<eol>
STAMP: setexpirationdate<eol>
Expiration date (just <eol> for none): jan 1 1990<eol>
Key: 'H12345678<eol>
STAMP: setpassword<eol>
New password (just <eol> for none): meringue<eol>
STAMP:
```

Example 60.5-2. Stamping a Module in a Library

60.6. Information about an Object Module

The command "show" displays an object module's expiration date, whether or not the module requires a password to be run, and whether or not it can only be run on a restricted set of CPU's (the exact list of CPU's cannot be displayed, because it cannot be derived from the information in the module).

If no "lib" command is in effect, STAMP prompts for the name of the object file to examine. Otherwise, it examines the modules specified by the "lib" command.

The syntax of the command is "show {all}" (the curly braces mean that the "all" is optional). If the "all" is omitted, only modules that have actually been stamped in some way are displayed. If no modules have been stamped, STAMP displays a message to that effect. If "all" is specified, then information is displayed about all modules being examined, even the modules that have not been stamped.

When STAMP decodes a module's expiration date, it displays it only if the expiration date is within five years, either way, of the current date. This is to foil people who might try to beat the system by patching random values into an expired object module's expiration date field and then using STAMP's "show" command to see what the effect was.

What follows is a session illustrating the use of the new command to find out whether or not the module "mainex" in the library "/11/system-16.lib" is stamped:

```
*stamp<eol>
MAINSAIL (R) Module Security (? for help)
STAMP: lib /11/system-16.lib mainex<eol>
STAMP: show<eol>
MODULE   EXPIRATION DATE   PASSWORD   RESTRICTED CPU IDS

MAINEX       1-May-86         no              yes
STAMP: q<eol>
*
```

60.7. Miscellaneous

60.7.1. List of Commands

Typing question mark to the command prompt gives a summary of the available commands, as shown in Example 60.7.1-1.

```
*stamp<eol>
MAINSAIL (R) Module Security (? for help)
STAMP: ?<eol>

target {targetSystem}
library libName {modList}
setExpirationDate
mixupExpirationDate
setMixedupExpirationDate
setPassword
mixupPassword
setMixedupPassword
setCpuIds
mixupCpuIds
setMixedupCpuIds
setToCompiledState
show {all}
quit
```

Example 60.7.1-1. STAMP Help

60.7.2. Long Command Lines

A very long STAMP command may be broken across several lines by terminating each line with a backslash ("\"). Thus, the command lines:

```
library foo.lib\
 abc\
 def\
 ghi
```

are equivalent to the single command line:

```
library foo.lib abc def ghi
```

60.7.3. "setToCompiledState" Example

In Example 60.7.3-1, the "setToCompiledState" command is used to set the encrypted key, expiration date, password, and CPU ID to the match-all values that are set by the compiler.

```
*stamp<eol>
MAINSAIL (R) Module Security (? for help)
STAMP: setToCompiledState<eol>
Module file: foo-<sys>.obj<eol>
Key: 'H12A8CD0<eol>
```

Example 60.7.3-1. "setToCompiledState"

60.7.4. Match-All Keys

Whenever a module is initialized for execution, MAINSAIL first checks whether the key is match-all, and if so, it does not bother checking any of the other security values, thereby saving the overhead of doing so. Thus it is a good idea to use "setToCompiledState" if a module has a key other than match-all, and all of the security fields are match-all (and will never be set again, though there is little reason to set them again since the unscrupulous customer could just continue to use the copy with no values set).

Note that once the key is set to match-all, there is no way to use the "remote" commands to set the key to a secret value. Thus if there is a chance that a module that is sent to a customer may at some later date be stamped with some values, even though this is not done when the module is sent to the customer, then the module should be stamped with a key before it is shipped to the customer (e.g., use the "setPassword" command and use just <eol> for the password). The rule of thumb is to stamp a module before distribution if restricted access is desired, and then use the remote commands to change the values (not particularly for changing the password, since if it is discovered, then the customer will not want to change it, and if it is not discovered, then there is no particular reason to change it anyway).

The "quit" command is used to exit STAMP.

60.8. STAMP Program Interface

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

STAMP can be called from a program with the procedure $executeStampCommands, declared as shown in Table 60.8-1.

```
BOOLEAN
PROCEDURE       $executeStampCommands
                                (OPTIONAL STRING cmds;
                                 OPTIONAL
                                     POINTER(textFile) f);
```

Table 60.8-1. $executeStampCommands

$executeStampCommands executes a series of STAMP commands until either one of the commands fails, in which case it returns false and ignores the remaining commands, or it encounters a "quit" command or runs out of commands, in which case it returns true. Commands are first read from the string cmds until it is empty, at which point commands are read from the file f until end-of-file is reached.

A command can fail for a number of reasons; e.g., a file cannot be opened, or it is not a library or object file, or a specified key does not match the key stored in any of the modules being stamped. If a key does not match a single module's key, the other modules are still stamped if

the specified key matches their keys, but the command is considered to have failed, and the remaining commands are ignored.

The commands are exactly the same as if they were being entered from cmdFile. For example, the call of Example 60.8-2 stamps the module FOO in the library "bar.lib" with both an expiration date and a password, as done in Example 60.5-2.

```
$executeStampCommands(
    "library bar.lib foo" & eol &
    "setexpirationdate" & eol &
    "jan 1 1990" & eol &
    "12345678" & eol &
    "setpassword" & eol &
    "meringue" & eol &
    "quit");
```

Example 60.8-2. Use of $executeStampCommands

61. SUBCMD, Subcommand Processor

Command line syntax: subcmd {one subcommand}

The module SUBCMD allows the user to give any MAINEX subcommand. When invoked with a commmand line argument, the argument is interpreted as a single MAINEX subcommand.

SUBCMD is most useful when run from MAINEDIT or MAINDEBUG or from the "Error response:" prompt.

61.1. SUBCMD Example

When module SUBCMD is invoked with no arguments, it reads subcommands from logFile. When logFile is "TTY", the subcommand prompt (">") is issued and subcommand mode is entered. Any number of subcommands may be entered at this point. Subcommand mode is exited when <eol> is typed in response to the ">" prompt.

Example 61.1-1 shows the use of SUBCMD when subcommands are entered through "TTY" from MAINED, the MAINSAIL text editor front end. Since <eol> is given as the name of the file that contains the subcommands, subcommand mode is entered and subcommands are read from primary input. When all of the desired subcommands have been given, <eol> is typed to exit subcommand mode. At this point, the module SUBCMD is exited and the user is back in command mode.

If it is desired to read subcommands from a file, the subcommand "SUBCOMMANDS" may be used.

```
The MAINED command "QESUBCMD<eol>" is given.

----P.1-----L.1-----I----CMDLOG--------------------
:
:Enter subcommands (? for help).
:>enter foo realFileName<eol>
:><eol>
:
E
E
```

Example 61.1-1. SUBCMD Example with Subcommands Entered from "TTY" Using MAINED

62. TVIEW, Text File Viewer

The module TVIEW displays the contents of a text file. TVIEW prompts for the name of the file to be examined. Once the input file is successfully opened, the contents of position 0 are automatically displayed and the user is prompted for a command. TVIEW displays the current position in the file followed by "/" followed by 16 characters beginning at the current position. Printing characters are displayed exactly as they appear in the file; character codes delimited by angle brackets are displayed for non-printing characters. For example, a horizontal tab character is displayed as "<9>" (assuming an ASCII character set). Available commands are displayed when "?" is typed in response to the TVIEW command prompt, and are listed in Table 62-1.

```
n         Examine position n
eol       Step forward through file
^         Step backward through file
+n        Position forward n integers
-n        Position backward n integers
S xxx     Search for value xxx
W xxx     Write value xxx
F xxx     Examine file xxx
Q         Quit TVIEW
```

Table 62-1. TVIEW Commands

62.1. n, <eol>, "^", "+n", and "-n"

These commands tell TVIEW what position in the input file to display. If n is typed, 16 characters beginning at position n are displayed. Typing <eol> causes TVIEW to display the contents of the next 16 positions in the file. For example, if the contents of position 0 have just been displayed and the <eol> command is issued, TVIEW displays 16 characters beginning with position 16. "^" displays the previous 16 characters, "+n" displays the 16 characters beginning with the nth next position, and "-n" displays the 16 characters beginning with the nth previous position in the file.

62.2. "S xxx"

The "S" command searches forward for the string xxx. If found, 16 characters are displayed beginning with the search string. If the string is not found in the file, a message is printed and 16 characters beginning with the current position are redisplayed.

62.3. "W xxx"

The "W" command writes the string xxx to the current position of the input file. The message "Wrote xxx at n" is displayed and the new text beginning with the current position is redisplayed.

The file being examined is initially opened for input only. When the "W" command is issued, the file is closed and reopened for input and output access. After the value has been written, the file is again closed and then reopened for input only. Since TVIEW opens the file for random access, and random output is not supported for record-oriented files, the "W" command is not allowed for such files.

62.4. "F xxx"

The "F" command changes the input file to xxx. The new input file is opened and first 16 characters of the file are automatically displayed.

62.5. "Q"

The "Q" command exits TVIEW.

62.6. TVIEW Example

Example 62.6-1 shows how to use TVIEW to examine a text file. The file "text" is opened and the first 16 characters are automatically displayed. The 16th character is a carriage return, indicated by "<13>". <eol> displays the next 16 characters beginning with position 16, and "100<eol>" displays 16 characters beginning with location 100. The search command searches forward from the current position (100) for the string "proc" and finds it at position 138.

```
MAINSAIL (R) Version 12.10 (? for help)
Copyright (c) 1984, 1985, 1986, 1987, 1988, and 1989 by
 XIDAK, Inc., Menlo Park, California, USA.
*tview<eol>
Text file viewer (? for help)
<xxx> represents non-graphic decimal character code xxx

File to view: filcch

     0/   BEGIN "$filCch"<13>
TVIEW><eol>
    16/   <10><13><10>MODULE $filCc
TVIEW>100<eol>
   100/   <9>      <9>  LONG I
TVIEW>s proc<eol>
   138/   PROCEDURE $clear
TVIEW>q<eol>
*
```

Example 62.6-1. TVIEW Example

63. WRDCOM, Data File Comparer

Command line syntax: wrdcom {file1 {file2}}*

WRDCOM compares pairs of data files. If the file names are provided on the command line, WRDCOM compares the specified pairs of files. If the last file2 is omitted, WRDCOM prompts for it. If no files are specified on the command line, WRDCOM prompts for two file names.

Successive bits values from each file are read and if different, the values and file position (0-origin) are written to logFile. If the two files are identical, no output is generated.

Example 63-1 shows how to use WRDCOM. The bits values at offsets 12, 14, 18, and 20 of the files "file1" and "file2" are different.

```
MAINSAIL (R) Version 12.10 (? for help)
Copyright (c) 1984, 1985, 1986, 1987, 1988, and 1989 by
 XIDAK, Inc., Menlo Park, California, USA.
*wrdcom<eol>
File number 1 (just <eol> to stop): file1<eol>
File number 2: file2<eol>
12 ('HC):              'H312E    'H322E
14 ('HE):              'HA48     'HA57
18 ('H12):             'H2077    'H206F
20 ('H14):             'H6F6E    'H6E65
Found 4 differences.
File number 1 (just <eol> to stop): <eol>
*
```

Example 63-1. WRDCOM Example

64. XREF, Cross-Reference

XREF writes a cross-reference for a text file. The cross-reference lists each non-keyword MAINSAIL identifier along with each page and line on which it occurs. Identifiers within string quotes and in comments, and characters after single quotes, are ignored. The case of identifiers is ignored; e.g., "abc" is the same as "ABC" and "aBc".

XREF prompts for the keyword option to be used when scanning the input files (keywords are omitted from the cross-reference listing). Table 64-1 lists the available responses. Only enough of the option to make it uniquely identifiable need be typed.

```
no keywords
MAINSAIL keywords
just keyword file
keyword file plus MAINSAIL keywords
help
```

Table 64-1. XREF Keyword Options

Options include specifying MAINSAIL keywords, those in a user file, both MAINSAIL keywords and those in a user file, and no keywords. The keyword file is assumed to consist of keywords separated by any mixture of blanks, tabs, eol's or eop's. MAINSAIL keywords (reserved identifiers) are listed in an appendix to the "MAINSAIL Language Manual".

After the keyword option has been specified, XREF prompts for the start and stop columns for the "page.line" information for each identifier. The cross-referenced identifiers are output in alphabetic order, beginning in column 0. The "page.line" information is listed in ascending order beginning with column startCol and not exceeding column stopCol (startCol and stopCol are 0-origin). The default startCol is column 20; the default stopCol is 71.

When startCol and stopCol have been read, XREF prompts for input-output file pairs. To exit, type <eol> in response to the "Next input file" prompt.

Example 64-2 shows a sample use of XREF. User responses are underlined. A cross-reference listing for the file "sample" is written to the file "sample-xrf". MAINSAIL keywords are not

included in the list. XREF prints the number of identifiers in the file and the total number of occurrences.

```
MAINSAIL (R) Version 12.10 (? for help)
Copyright (c) 1984, 1985, 1986, 1987, 1988, and 1989 by
 XIDAK, Inc., Menlo Park, California, USA.
*xref<eol>
Keyword option (? for responses): m<eol>
startCol stopCol (just eol for 20 71, ? for help): <eol>
Next input file (just eol for none): sample<eol>
Output file: sample-xrf<eol>
12 different ids. 33 occurrences.

Next input file (just eol for none): <eol>
*
```

Example 64-2. XREF Example

Figure 64-3 gives the XREF output for the module SAMPLE.

For information on cross-reference information available directly from the MAINSAIL compiler, see the descriptions of the utility "XRFMRG" and of the "ITFXRF" and "FLDXRF" subcommands of the compiler in the "MAINSAIL Compiler User's Guide".

```
Keyword Options:    MAINSAIL keywords
Input File:         sample
Statistics:         12 different ids. 33 occurrences.

BUCKETSIZE          1.3 1.11 1.17
CREAD               1.10
CVI                 1.17
EOL                 1.20 1.20
H                   1.8 1.9 1.9 1.10 1.11
HASH                1.6 1.20
I                   1.8 1.9 1.10
J                   1.8 1.9 1.10 1.10
LENGTH              1.9
S                   1.6 1.9 1.10 1.16 1.19 1.20
TTYREAD             1.17 1.19
TTYWRITE            1.17 1.18 1.20
```

Figure 64-3. XREF Output for SAMPLE

Index

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