Inter-Office Memorandum

To	Mesa Users	Date	October 27, 1980
From	John Wick	Location	Palo Alto
Subject	**Mesa 6.0 Change Summary**	Organization	SDD/SS/Mesa

XEROX

Filed on: [Iris]<Mesa>Doc>Summary60.bravo (and .press)

This memo outlines changes made in Mesa since the last release (Mesa 5.0, April 9, 1979). The first section lists references which should be consulted for more detailed information. The second section contains an overview of the changes. The final section describes operational differences which you must know in order to use Mesa 6.0.

For most programs, you should be able to begin converting to Mesa 6.0 after reading only this memo; then consult the various updates when details of new features are required. If your application relies on the extended memory support previously provided by XMesa, you will probably need to read the System and XMesa updates before converting.

References

The following documents can be found on `<Mesa>Doc>`. (In Palo Alto, the official Mesa release directory is maintained on Iris (there is also a copy on Ivy); in El Segundo, it is on Isis. For other locations, consult your support group.) In addition, the file `Mesa60.press` is a compilation of this and other material (about 75 pages); hardcopy is available through your support group.

Mesa 6.0 Change Summary. `Summary60.press`
Mesa 6.0 Compiler Update. `Compiler60.press`
Mesa 6.0 Binder Update. `Binder60.press`
Mesa 6.0 System Update. `System60.press`
Mesa 6.0 XMesa Update. `XMesa60.press`
Mesa 6.0 Debugger Update. `Debugger60.press`
Mesa 6.0 Utilities Update. `Utilities60.press`
Mesa 6.0 Pup and Ftp Update. `PupFtp60.press`

The above memos describe changes since the last release. The `<Mesa>Doc>` directory also includes new versions of the following documents:

Mesa System Documentation. `System.press`
Mesa Debugger Documentation. `Debugger.press`
Mesa User's Handbook. `Muhb.press`
Mesa Pup Package Functional Specification. `Pup.press`

Mesa Ftp Functional Specification. `Ftp.press`
Integrated Mesa Environment. `CommandCentral.press` (in `Utilities60.press`)
Performance Mesaurement Tool. `PerformanceTool.press`
Control Transfer Counting Tool. `XferCounter.press`
Debugger: Extended Features. `XDF.press` (in `Debugger60.press`)

Because the language changes are upward compatible with Mesa 5.0, a new version of the *Mesa Language Manual* will not be issued with this release; consult the *Compiler Update* and the *Binder Update* for information on additions to Mesa and C/Mesa.

Highlights

The primary emphasis in this release has been on the following areas:

A number of significant new language features are included: extended defaults, floating point support, machine dependent records and enumerated types, dynamic storage allocation, sequences, and exported types. Less major revisions and extensions have been made in the following areas: syntactic and semantic glitches, keyword array constructors, packed array representation, successor and predecessor operations, the DIRECTORY statement, implicitly imported interfaces, type TEXT, loop control variables, extended NIL, the REJECT statement, and the ABORTED signal. The language changes are compatible with Mesa 5.0, and will not affect existing source code (there are some minor exceptions; see below). In addition, the Compiler's user interface and command line syntax have changed substantially. See the *Compiler Update* and Appendix A of the *Mesa User's Handbook* for details.

A Mesa source formatter is available with this release. It establishes a standard representation for source text, and will produce both plain and formatted `.bravo` and `.press` files. Complete documentation can be found in Appendix B of the *Mesa User's Handbook.*

The Binder has been updated to agree with the Compiler's treatment of the DIRECTORY statement and its user interface and command line syntax. In addition, it now supports multiple control modules in a single configuration. See the *Binder Update* and Appendix C of the *Mesa User's Handbook* for details.

Support for extended memory beyond 64K, formerly provided by XMesa, is now part of the standard system. The BasicMesa configuration has been abandoned in favor of facilities which dynamically delete non-essential components of the system. Support for subdirectories, fast directory scanning, floating point, Pup checksums, and 3K RAMs have been added. (Appendix G of the *Mesa User's Handbook* details extensions made to RunMesa.) Other than memory allocation facilities, changes to the System and Pup and Ftp interfaces are minor; see the updates for details.

The Debugger has a new user interface, a new interpreter, and a simple cut and paste editor. The Debugger's command language is essentially unchanged, but several extensions in window operations and source window facilities have been added. A description of the new **FileTool** is attached to the *Debugger Update*.

Several new commands have been added to the Lister, and the IncludeChecker has been extensively modified to support large configurations more efficiently. A new package, **CommandCentral**, is available for use with the Debugger; it serves as a small executive for controlling the editing, compilation, binding, and debugging of applications software. This package greatly speeds up the edit compile bind debug cycle when small changes are involved. Documentation is attached to the *Utilities Update*.

A new mode has been added to the Control Transfer Counting Tool which allows modules to be assigned to groups and information to be collected on cross group as well as cross module transfers. The performance measurement tools are documented separately as described in the first section of this memo.

Operational Changes

This section summarizes important operational differences which you must know in order to use Mesa 6.0*; do not begin to convert to Mesa 6.0 until you have read it*. More complete information is contained in the update memos listed in the first section of this document.

General

On large programs, performance of the Mesa 6.0 Compiler and Debugger on 64K Altos is considerably worse than in Mesa 5.0; *an extended memory machine with a minimum of 128K is recommended*.

Alto Operating System version 18 and Executive version 11 or later are required to run Mesa 6.0.

The new file creation date standard is now supported. The compiler inserts the creation date of the source file into the `.bcd` (as does the Binder), and the Debugger checks that the source and object file versions match. The IncludeChecker also makes use of these dates. *Therefore, you must use an editor and a file server that support the file date standard*. If you use dump files, be sure you have the latest version of Ftp.

Language

With two exceptions, Mesa 5.0 source files are compatible with Mesa 6.0. In file names in the DIRECTORY statement, names inside angle brackets are no longer ignored; they are treated as subdirectories of the Alto file system (use of this feature is not recommended). The following new reserved words have been added: FREE, PRED, PROC, REJECT, SEQUENCE, SUCC, UNCOUNTED, and ZONE.

Compiler

The order of evaluating the items in constructors (including argument lists) and the operands of infix operators (except AND and OR) has been changed, particularly in cases involving embedded procedure calls; it is no longer always left to right. In particular, expressions of the form word _ get[s]*256+get[s] are suspect, and probably incorrect.

Except during initialization, constructors for records containing MONITORLOCKs, CONDITIONs, and PORTs are not allowed (to prevent unintentionally overwriting these fields with their default values). Such records must be assigned to field by field.

For element sizes of four bits or less, the internal representation of packed arrays has changed. This is a potential problem in reading files containing packed arrays created by earlier versions of Mesa.

The Compiler no longer supports interactive command input; it reads commands only from the command line, and does not use the keyboard or display (limited feedback is via the cursor). The command language has been extended and switch processing has changed. The `/c` switch has been deleted; global switches must now be specified with a null file name, e.g. `Compile /p Defs Impl`.

Switches are restricted to a single letter. *Do not use complete switch names such as* `/pause` (each letter will be interpreted as a separate switch). Until you understand the full syntax of the command language, it is best to group all global switches at the beginning of the command line following a single slash.

The log is now written on `Compiler.log`, not `Mesa.typescript`; separate `.errlog` files are still produced. The error log (if any) is deleted if the compilation is successful; conversely, if the compilation fails, the `.bcd` (if any) is deleted.

The implementation of floating point has changed considerably; the IEEE standard format is now used, and the compiler generates calls directly to user-supplied microcode (this will produce undefined results if the proper microcode is not loaded). Calls to software floating point as in Mesa 5.0 can be generated with the `/-f` switch. *Do not use type* REAL *without first consulting with the supplier of your floating point package.*

Because of bug fixes, previously acceptable programs may no longer compile.

Binder

The Binder is now available only as a `.bcd` file; you must have `Mesa.image` to run it. Like the Compiler, the Binder takes commands only from the command line and does not use the keyboard or display; it writes its log file on `Binder.log`.

The meaning of switches used to copy code and symbols has changed substantially; read the *Binder Update* if you use these options. Compressed symbols have been compressed still further to include only procedure and signal names (without parameters or results); this substantially reduces the size of these `.symbol` files.

Because of bug fixes, previously acceptable configurations may no longer bind.

System

Mesa 6.0 is compatible with XMesa 5.0 microcode version 39 (but some new features are not available with the old microcode; e.g., extended memory BITBLT).

Features previously implemented by BasicMesa and XMesa are now a standard part of `Mesa.image`. **MakeImage** is no longer a part of the standard system; **ImageMaker** must be loaded or bound with the client configuration if it is needed.

The standard system now supports only command line input; the **MesaExec** can be loaded to provide interactive input. The command line switch `/b` can be used to convert the standard system to a basic one. The `/k` switch will disable the allocation of space for the Debugger's bitmap on extended memory machines (see below).

The default maximum numbers of processes (75) and modules (384) have been increased. A version of the System called **SmallMesa** allows 33 processes and 256 modules.

Interface changes are described in the *System Update*.

Debugger

The Debugger has also been enhanced to take advantage of extended memory. If you have more than two banks (128K) of memory, see the installation section of the *Debugger Update* for an explanation of the options available; otherwise, the standard defaults will "do the right thing".

The Debugger now requires a strike font named `SysFont.strike` or `MesaFont.strike`; a version of Gacha10 is available on `<Mesa>MesaFont.strike`. Additional fonts are available on `[Maxc]<AltoFonts>`. (Strike fonts that include kerning are not supported.)

The selection scheme and the assignment of function keys and mouse buttons has changed. Clicking RED once selects a character, clicking twice selects a word, three times a line, etc.; the selection can be extended to the left or right with BLUE. The menu button is now YELLOW (formerly BLUE). FL4 is no longer the stuff key; use FR4 (Spare2) or Keyset2.

Scrollbars no longer occupy a dedicated part of the window, but instead come up on top of the left edge. To obtain a scroll bar, move left just past the edge of the window, then move right slightly, back into the window.

New source window menu commands have been added, and they have been factored into several menus. The Debugger's wisk window has been replaced by a more general `Split` menu command. There are now also `Normalize Insertion` and `Normalize Selection` commands.

The interpreter can now be used when in `Display Stack` and `Display Process` subcommand mode. Several commands now invoke the interpreter automatically (e.g., `Octal Read: @p, n: SIZE[R]`). The interpreter does procedure calls (the `Interpret Call` command has been deleted).

The constructs ABS, ERROR, LONG, LOOPHOLE, MAX, MIN, NIL, POINTER TO, PROC, PROCEDURE, SIGNAL, WORD, and open and half open intervals have been added to the interpreter's grammar. Type expressions following **%** must be enclosed in parentheses. The interpreter syntax **Expression?** replaces the `Interpret Expression` command.

Each breakpoint is now assigned a unique number used for displaying and clearing it. There are new commands for attaching conditions to break and tracepoints. Break/Tracepoints can no longer be set by typing a source line, and the `Break Module` and `Break Procedure` commands and corresponding `Trace` and `Clear` commands have been deleted; the menu commands must be used. `Clear All Entries/Xits` clears both break and tracepoints.

Tracepoints now automatically invoke the normal `Display Stack` command processor (with subcommand `p(arameters)`, `v(ariables)`, or `r(esults)` as appropriate). The `q(uit)` subcommand (not `b(reak)`) exits to the Debugger's command level; it no longer continues execution of the client.

If the source window is loaded with the `s(ource)` subcommand of `Display Stack`, the window will remember the appropriate context for setting breakpoints. For an exit break, the `s(ource)` subcommand now displays the declaration line of the procedure.

The Debugger no longer ignores case, and the case commands have been deleted; *identifiers must be typed with their correct capitalization.*

Distribution:
- Mesa Users
- Mesa Group
- SDSupport

Inter-Office Memorandum

To	Mesa Users	Date	October 27, 1980
From	Ed Satterthwaite	Location	Palo Alto
Subject	**Mesa 6.0 Compiler Update**	Organization	PARC/CSL

XEROX

Filed on: [Iris]<Mesa>Doc>Compiler60c.bravo, Compiler60d.bravo, and Compiler60.press

This memo describes changes to the Mesa language and compiler that have been made since the release of Mesa 5.0 (April 9, 1979).

Definitions of syntactic phrase classes used but not defined in this document come from the *Mesa Language Manual, Version 5.0*, Appendix F.

Compatibility

Because of changes in symbol table and BCD formats, you must recompile all existing Mesa programs after obtaining recompiled versions of the interfaces and packages that they depend upon.

Language Changes

Our goal for language compatibility has been to accept any valid Mesa 5 source program as a valid Mesa 6 source program. We are aware of the following incompatibilities:

There are some new reserved words, as follows:

FREE PRED PROC REJECT SEQUENCE SUCC UNCOUNTED ZONE

Some of the quoted file names that appear in DIRECTORY clauses have different interpretations. Text inside angle brackets is no longer ignored; it is treated as the name of a local subdirectory (but we do not recommend using local subdirectories for Mesa programs at the present time).

The orders of evaluating the items in constructors (including argument lists) and the operands of infix operators (except AND and OR) have changed somewhat. In particular, programs that assumed a left-to-right order of procedure calls in these contexts (e.g., *Divide*[*Pop*[], *Pop*[]]) are unlikely to work correctly.

The assignment operation is no longer available for updating objects containing MONITORLOCK or CONDITION values; updating of such objects must be done component-by-component.

The granularity of packed arrays has changed. If the components of a packed array can be stored in four or less bits, the storage structures defined by the declaration of that array will differ between Mesa 5 and Mesa 6. This is a potential problem in reading files created by earlier versions of Mesa. Also, the DESCRIPTOR operator cannot be applied to packed arrays occupying less than a word.

If you have been using type REAL, check with the supplier of your floating point package to determine the effect of Mesa 6 changes in that area.

Bug Fixes

A large number of Mesa 5 bugs have been fixed. The most notable of these involve

expansion of inline procedures that are defined in DEFINITIONS modules,

expansion of inline procedures when an argument is itself an expanded inline,

proper retention of the tag of a variant record within an arm of a discriminating selection,

proper identification of object files that have been renamed.

Because of certain bug fixes, the compiler may reject previously acceptable programs or may issue new warning messages.

As usual, the list of compiler-related change requests closed by Mesa 6.0 will appear separately as part of the Software Release Description.

Language Rationalization

Mesa 6 attempts to remove certain minor anomalies and to add some obvious generalizations to the existing language.

Syntactic Glitches Removed

List Punctuation: For most lists that are explicitly bracketed by symbols other than [and], the allowable forms are described by the following meta-BNF:

list ::= empty | item | item separator list

In other words, the list may be empty or may be a sequence of items separated by, and optionally terminated by, a separator. This rule now applies to the following constructs:

form	*separator*	*notes*
VariantList	,	
CatchSeries	;	ANY must come last
ChoiceSeries	;	
ExitSeries	;	
StatementSeries	;	
StmtChoiceSeries	;	
ChoiceList	,	
ExprChoiceList	,	

When the bracketing symbols are [and] or when the length of a list is significant, a trailing separator is not allowed unless it is semantically meaningful (as in constructors and extractors), but empty lists are allowed. This change affects the following syntactic entities:

UsingClause: The form USING [ ] is now permitted (e.g., to emphasize that an interface is only being exported).

VariantFieldList: In the declaration of a variant record, the form [ ] (empty brackets) may be used instead of NULL (and is recommended).

FieldList: The form [ ] is permitted in the declaration of a ParameterList or ReturnsClause (it is equivalent to omitting the list or clause).

Directory: The form DIRECTORY ; may be used in place of an empty directory.

ImportsList (ExportsList): The form IMPORTS (EXPORTS) may be used when the corresponding list is empty.

The last two items reflect the view that DIRECTORY, IMPORTS and EXPORTS introduce formal parameter lists, even though their punctuation omits [and].

Declarations in Loop Bodies: A declaration series can now appear between DO and ENDLOOP (after the OPEN and ENABLE clauses, if any); no additional bracketing is required. Its scope is limited to the DeclarationSeries and StatementSeries in the loop body.

A Bit Less Verbiage

PROC: PROC is accepted as a short form of PROCEDURE.

Statement Brackets: The bracket pair { } can be used any place the bracket pair BEGIN END can be used (but not conversely).

Semantic Glitches Removed

LONG *Arithmetic*: Compile-time evaluation of expressions with constant operands now works for 32-bit quantities just as it does for 16-bit quantities.

Dereferencing with OPEN: A pointer expression following OPEN or WITH will be dereferenced an arbitrary number of times (not just once) to obtain an expression designating a record.

Renaming with OPEN: An OPEN clause may give an alternative local name to an interface that it opens (formerly, only to a record).

Nested Extractors: An ExtractItem may itself be an Extractor. This allows extraction from records embedded within records. In particular, this form is useful in situations where a single-component record is not automatically converted to its single component, e.g., extraction from a record that is the only component of a return record.

Extractor Expressions: An assignment with an Extractor as its left side may be used as an expression. This allows, among other things, multiple extraction. The value of such an assignment is the value of its right side.

Interface Aliases: An interface module may have multiple identifiers (preceding ": DEFINITIONS"). In the DIRECTORY clause of another module, you can reference that interface using any one of the identifiers. A type obtained from such an interface is equivalent to the same type obtained through a naming path that uses any other identifier of the interface.

New Language Features

Extended Defaults

You can associate a default initial value with a type (not just with a field of a record). If a type is constructed from other types using one of Mesa's type operators (e.g., RECORD), the default value for that type is determined by the default values of the component types and by rules associated with each operator. When you declare a named type, you have the option of explicitly specifying a default for that type.

With this extension, you will find that uses of defaults in Mesa generally fall into two classes. Default values for fields of records make the corresponding constructors more concise and more convenient to use. On the other hand, the usual reason for associating a default initial value with a type is to ensure that storage allocated for that type is well-formed, i.e., that any variable of such a type always has a meaningful value. There is some interaction between these uses; the default value of a record type is partly determined by any default values specified for its fields, and a record field may inherit its default value from the type of that field. The details appear below.

The rules for inheritance of defaults are designed to provide the following property (currently not quite preserved by sequence or variant record types): if a type *T* has been given a non-NULL default value, any type derived from *T* will have a defined and non-NULL default value for any embedded component of type *T*. Because of the potential cascading effect implied by this, you should carefully consider the relative costs and benefits of specifying a default, especially one that does not include NULL as an alternative.

Defaults are ignored in determining equivalence and conformance of types. Thus it is possible to have two compatible types with different default initializations.

Specification of Default Initialization

None of the built-in types (INTEGER, CARDINAL, BOOLEAN, CHARACTER, STRING and REAL) has a default initial value.

The following rules determine the default initial value of a type designated by an expression involving a type operator:

> The default initial value for a type constructed using RECORD (or ARRAY) is defined field-by-field (or element-by-element). For each field (element), it is the default value for that field if there is one; otherwise, it is the default initial value for the type of that field (element) or is undefined if there is no such default.
>
> The default initial value for any port type, constructed using PORT, is NIL (see below).
>
> Types constructed using other operators have no implied default initialization.

The default initial value of a type designated by a declared type identifier *T* depends upon the form of the declaration of *T*, as follows:

> *T:* TYPE = *TypeExpr*;
>
> *T* receives all the attributes of *TypeExpr* including any default.
>
> *T:* TYPE = *TypeExpr _ e*;
>
> *T* receives all the attributes of *TypeExpr* except that its default initial value is *e*.

Examples

```
Flag: TYPE = BOOLEAN _ FALSE;
Rec1: TYPE = RECORD [f: Flag];                    -- default value is [f: FALSE]
Rec2: TYPE = RECORD [f: Flag] _ [ ];              -- ditto (the field defaults)
Rec3: TYPE = RECORD [f: Flag] _ [TRUE];           -- explicit default
Rec4: TYPE = Rec3;                                -- default value is [f: TRUE]
Rec5: TYPE = Rec3 _ [f: FALSE];                   -- default value is [f: FALSE]
```

Any DefaultSpecification is acceptable in a type declaration (see *Mesa Language Manual, Version 5.0*, page 37). A declaration giving a type *T* a NULL default cannot, however, equate *T* to a type with a default that does not include NULL. A default appearing in a type definition within a DEFINITIONS module must be either NULL or an expression with a compile-time constant value.

Default values associated with types are used

- to initialize local variables of procedures and programs, in the absence of explicit initialization,
- to initialize variables that are dynamically allocated using NEW, in the absence of explicit initialization (see below),
- to construct records (except argument and result records), in the absence of an explicit value for a field in the constructor and of a default value for that field in the record declaration,
- to construct arrays, in the absence of an explicit value for an element (see below).

Defaults in Argument and Result Records

You may specify default values for the fields of argument and result records. Such default values must be constructed from constants or variables that are declared outside of the procedure type definition. In particular, you cannot use a value of another field of the same record or, in the case of a result record, a value from the associated argument record to define such a default.

You may omit a field in the constructor of an argument or result record only if the definition of that record specifies an explicit default value for the field; default initial values associated with the *types* of such fields are *not* inherited (for example, this protects you from assigning a value to a return variable and then forgetting to mention it in a RETURN statement, causing the default for its type to be returned). On the other hand, protection against ill-formed storage *is* inherited; you may not void or elide a field unless the type of that field allows a NULL initialization.

Any defaults that you specify in the declaration of a result record serve two purposes. Since the fields of such a record can be used as local variables within the procedure body, a default specification affects the initialization of those variables; in addition, it allows abbreviation in the constructors of the corresponding return records. The precise rules are the following:

- Upon entry to a procedure, each field of the result record is initialized with the default value specified for that field, if any; otherwise, with the default initial value for the type of that field, if there is one; otherwise, its initial value is undefined.
- If a RETURN is followed by an explicit constructor, the default specifications appearing in the declaration of the result record control the values of any omitted or elided fields, *even if other assignments have been made to the result variables within the procedure body*. If the RETURN stands by itself, without such a constructor, or if the RETURN is implicit, the return record is constructed using the current values of the result variables.

Examples

```
T: TYPE = INTEGER _ 1;

Proc1: PROC [i: INTEGER _ 0, j: T];

Proc2: PROC RETURNS [m: T, n: INTEGER _ 2] = {
  -- m initialized to 1 (from T), n to 2
  Proc1[j: 3];                  -- Proc1[i:0, j:3];
  Proc1[i: 3];                  -- illegal (j does not default to 1)
  ...
  m _ 4;  n _ 5;
  ... RETURN;                   -- returns [4, 5]
  ... RETURN [m, n];            -- also returns [4, 5]
  ... RETURN [m];               -- returns [4, 2]
  ... RETURN [NULL, n];         -- illegal (declaration of T disallows voiding of m)
  ... RETURN [, n];                        -- ditto (m does not default to 1 or 4)
  ... RETURN [6, 7];            -- returns [6, 7]
  ... };                        -- implicitly returns [4, 5]
```

Defaults and Variant Records

You may specify a default for the entire variant part in the declaration of a variant record type. In the absence of such a specification, the default value of that part, including the tag, is undefined with respect to the undiscriminated record type.

The default initial value of a discriminated variant record type has a tag value corresponding to the discriminating adjective, and defaults for the other fields of the variant part are those implied by the fields selected by that tag. In particular, the declaration or allocation of a variable with discriminated record type sets the tag correctly.

Example

```
VRec: TYPE = RECORD [
  common: INTEGER _ 0,
  variant: SELECT tag: * FROM
    red => [r1: BOOLEAN _ FALSE],
    green => [g1: INTEGER _ 0]
    ENDCASE _ red[TRUE] | NULL];

v: VRec;                        -- initial value is [common: 0, variant: red[r1: TRUE]]
v1: VRec _ [common: 10];        -- initial value is [common: 10, variant: red[r1: TRUE]]
v2: VRec _ [variant: NULL];     -- tag and variant part are undefined, v2.common = 0
v3: VRec _ NULL;                -- illegal (declaration of common does not allow NULL)
rv: red VRec;                   -- initial value is [common: 0, variant: red[r1: FALSE]]
gv: green VRec;                 -- initial value is [common: 0, variant: green[g1: 0]]
```

Defaulted Array Elements

Elements in an array constructor may be voided or elided. Omission of elements is permitted in a keyword constructor (see below) but not in a positional constructor. The empty constructor ([ ]) is a keyword constructor with all items omitted. An elided or omitted element receives the default value for the type of the components of the array (if any); the value of a voided element is undefined.

ALL abbreviates a positional constructor of the appropriate length; thus ALL[ ] elides all elements (defaulting if possible) and ALL[NULL] voids all positions.

Keyword Array Constructors

You can use keyword array constructors when the index type of the array is an enumeration or subrange thereof. The acceptable keywords are the constants appearing in the enumeration. In the case of a subrange, the endpoints must be defined by expressions involving only those constants, the operators FIRST, LAST, SUCC and PRED, and identifiers equated by declaration to such expressions. If the component type of the array has a defined default value (including NULL), keyword items can be omitted; the corresponding elements receive the default value.

Packed Arrays

If you specify the PACKED attribute for an array type, the granularity of packing is 1, 2, 4, 8 or 16*n* bits and is determined by the component type of the array (formerly just 8 or 16*n* bits).

The value of the construct SIZE[*T, n*] is the size, in words, of the storage required by a packed array of *n* items of type *T*. (SIZE[*T*] continues to yield the number of words occupied by a single item of type *T*.)

Example

```
Bit: TYPE = BOOLEAN _ FALSE;
BitSet: TYPE = PACKED ARRAY Color OF Bit;
AllBits: BitSet = ALL[TRUE];
threeBits: BitSet _ [yellow: TRUE, red: TRUE, blue: TRUE];
```

Successor and Predecessor Operations

The operators SUCC and PRED operate upon values of any ordered type except REAL. For numeric and character types, SUCC[*x*] and PRED[*x*] are equivalent to *x*+1 and *x* 1 respectively. For enumerated types, the values are the successor and predecessor of *x* in the enumeration; a bounds fault occurs if there is no such element *and* you requested bounds checking.

Directories

You can now override the association established by the DIRECTORY clause between the names of included modules and the names of the files containing those modules. Any file names implied by convention can be omitted entirely; they will be computed from the interface identifiers.

Syntax

```
IncludeList     ::=  IncludeItem
                |    IncludeList , IncludeItem

IncludeItem     ::=  identifier UsingClause
                |    identifier : FROM FileName UsingClause
                |    identifier : TYPE UsingClause
                |    identifier : TYPE identifier UsingClause

UsingClause     ::=  empty | USING [ IdList ] | USING [ ]
```

The (initial) identifier in an IncludeItem names a module. If you specify the name of the file containing that module when you invoke the compiler (as a keyword parameter with keyword identifier, see below), that name is used, even if a FileName appears in the IncludeItem. Otherwise, if such a FileName appears, it is used. If you supply neither a compile-time argument nor a FileName, the name `identifier.bcd` is used.

One approach to describing systems built from collections of Mesa modules views the DIRECTORY clause as the declaration of (compile-time) formal parameters of type TYPE. Mesa 6 provides the final two forms of IncludeItem primarily for compatibility with this view. The identifier preceding the colon names the formal parameter; it is also used to derive a file name as described above. The identifier following TYPE constrains the set of acceptable actual parameters; it must match the ModuleName used in the definition of the module that you intend to include (see the *Mesa Language Manual, Version 5.0*, Section 7.2).

You can use the final form to change the name by which one module is known within another, notably to avoid duplicate names in a DIRECTORY clause. For example, you might need to reference two different versions or parameterizations of an interface *Defs* within a single program. The IncludeItems

LongDefs: TYPE *Defs,*
ShortDefs: TYPE *Defs*

declare *LongDefs* and *ShortDefs* as identifiers within that program of possibly different modules, each with the ModuleId *Defs*. As IncludeItems, the forms

identifier
identifier: TYPE

each abbreviate

identifer: TYPE *identifier*

and the name in the DIRECTORY must be identical to the ModuleId if you use one of these forms.

Implicitly Imported Interfaces

An implicitly imported interface is one from which imported values are required for binding the free variables of another, explicitly imported interface. For example, interface *D1* might import interface *D2* to gain access to a procedure or exported variable supplied by the latter. *D2* is then implicitly imported by any program module *M* that imports *D1* (see the *Mesa Language Manual, Version 5.0*, Section 7.4.4).

In Mesa 6, the free variables of *D2* that are used by *D1* are bound to the *principal instance* of *D2* in *M*. An import is a principal instance if it is the only instance of that interface imported by a module *or* if it is unnamed. Furthermore, if *M* imports no instances of *D2*, a principal instance will be created automatically. If module *M* has no other reason to mention *D2*, *D2* then need not appear in either the DIRECTORY or the IMPORTS list of *M*. Explicitly importing a principal instance of *D2* in such a situation is not an error, and you must do so if

- you plan to use positional notation to specify the imports of *M* in a C/Mesa configuration description, since the positions of automatically created interface instances are not defined, or
- you already import more than one instance of *D2*, each of which is named.

In a C/Mesa configuration, principal instances of interfaces are *not* supplied automatically; you must import them explicitly if they cross (sub)configuration boundaries.

Real Numbers

Mesa 6 has adopted the proposed IEEE standard for floating-point arithmetic (see, e.g., Coonen, An implementation guide to a proposed standard for floating-point arithmetic, *Computer*, January 1980, pp. 68-79). In support of this, the language provides floating-point literals and the compiler performs a limited number of operations upon floating-point constants.

Syntax

primary	::=	... | realLiteral
realLiteral	::=	unscaledReal
	|	unscaledReal scaleFactor
	|	wholeNumber scaleFactor
unscaledReal	::=	wholeNumber fraction
	|	fraction
fraction	::=	. wholeNumber
scaleFactor	::=	E optSign wholeNumber | e optSign wholeNumber
optSign	::=	empty | + |
wholeNumber	::=	digit | wholeNumber digit

An unscaledReal has its usual interpretation as a decimal number. The scaleFactor, if present, indicates the power of 10 by which the unscaledReal or wholeNumber is to be multiplied to obtain the value of the literal.

Mesa 6 represents REAL numbers by 32 bit approximations as defined in the IEEE standard. The rounding mode used to convert literals is "round-to-nearest." A literal that overflows the internal representation is an error; one that underflows is replaced by its so-called "denormalized" approximation. In Mesa 6, the value of the unscaledReal in a literal must be a valid LONG INTEGER when the decimal point is deleted.

No spaces are allowed within a realLiteral. Note that such a literal can begin, but not end, with a decimal point. Thus the interpretation of [0...1) is unambiguous (but perhaps surprising; use [0 .. .1) or [0.0..0.1) instead).

Operations

The compiler performs the following operations involving floating-point constants:

Unary negation (with 0 = 0)
ABS
Fixed-to-Float (in "round-to-nearest" mode).

Other operations are deferred until runtime, even if all their operands are constant, so that the programmer can control the treatment of rounding and exceptions (see the proposed standard).

> Unless you specify the compilation option -f (see below), the compiler generates instructions for floating-point operations that require hardware or microcode support. If you are in doubt about the state of your machine or its microcode, see a local floating-point expert.

Machine Dependent Enumerations

Sometimes a programmer can enumerate the values of some type but requires control of the encoding of each value or of the number of bits used to represent the type (usually for future expansion). Mesa 6 provides machine-dependent enumerations for such applications.

Syntax

EnumerationTC	::=	MachineDependent { ElementList }
MachineDependent	::=	empty | MACHINE DEPENDENT
ElementList	::=	Element | ElementList , Element
Element	::=	identifier
	|	identifier (Expression)
	|	(Expression)

Examples

Status: TYPE = MACHINE DEPENDENT {*off*(0), *ready*(1), *busy*(2), *finished*(4), *broken*(7)}

Color: TYPE = MACHINE DEPENDENT {*red*, *blue*, *green*, (255)} -- reserve 8 bits

Each Expression in an EnumerationTC must denote a compile-time constant, the value of which is an unsigned integer.

In an enumerated type with the MACHINE DEPENDENT attribute, the values used to represent the enumeration constants are assigned according to the following rules. If a parenthesized expression follows the element identifier, the value of that expression is used; otherwise, the representation of an element is one greater than the representation of the preceding element. If you specify only a representation, the corresponding element (normally a place holder) is anonymous. If the representation of the initial element is not given, the value zero is used.

You cannot explicitly specify the representation of any element unless the attribute MACHINE DEPENDENT appears in the type constructor. Two element identifiers cannot be represented by the same value (either given explicitly or determined implicitly as described above). The ordering of elements determined by position in the ElementList must agree with the ordering determined by the (unsigned) arithmetic ordering of the representations.

Sparse Enumerations

A machine-dependent enumerated type is *sparse* if there are gaps within the set of values used to represent the constants of that type or if the smallest such value is not zero. Mesa 6 takes the following position on gaps: they are filled by valid but anonymous elements of the enumerated type. These elements can be generated only by the operators FIRST, SUCC and PRED (or by the iteration forms that implicitly use these operators).

> If you use a sparse enumerated type as the index type of an array, the array itself will have components for all elements of the enumeration, including the anonymous ones. The latter are awkward to access (except through ALL) and may cause problems in constructors, comparison operations, etc., as well as wasted space. (For example, ARRAY *Color* OF INTEGER would occupy 256 words.)

Machine Dependent Records

Machine-dependent records are provided for situations in which the exact position of each field is important. In Mesa 6, you can explicitly specify word- and bit-positions in the declaration of the record type. This form provides better documentation and usually is easier to use than the previous, purely positional form. You should use it in preference to the old form, which remains for compatiblity.

Syntax

VariantFieldList	::=	CommonPart FieldId : Access VariantPart
	|	VariantPart
	|	NamedFieldList
	|	UnnamedFieldList
	|	empty
CommonPart	::=	NamedFieldList , | empty
NamedFieldList	::=	NamedField | NamedFieldList , NamedField
NamedField	::=	FieldIdList : Access TypeSpecification DefaultOption
FieldIdList	::=	FieldId | FieldIdList , FieldId
FieldId	::=	identifier | identifier (FieldPosition)
Tag	::=	FieldId | ...
FieldPosition	::=	Expression : Expression .. Expression
	|	Expression

Examples

```
InterruptWord: TYPE = MACHINE DEPENDENT RECORD [
   channel (0: 8..10): [0..nChannels),          -- nChannels <= 8
   device (0: 0..7): DeviceNumber,
   stopCode (0: 11..15): MACHINE DEPENDENT {finishedOK(0), errorStop(1), powerOff(3)},
   command (1: 0..31): ChannelCommand];

Node: TYPE = MACHINE DEPENDENT RECORD [
   type (0: 0..15): TypeIndex,
   rator (1: 0..13): OpName,
   rands (1: 14..47): SELECT valence (1: 14..15): * FROM
      nonary => [],
      unary => [left (1: 16..31): POINTER TO Node],
      binary => [left (1: 16..31), right (1: 32..47): POINTER TO Node]
      ENDCASE]
```

An identifier with an explicitly specified FieldPosition can occur only in the declaration of a field of a record defined to have the MACHINE DEPENDENT attribute. If the position of any field of a record is specified, the positions of all must be. Each Expression in a FieldPosition must denote a compile-time constant, the value of which is an unsigned integer.

The first expression appearing in a FieldPosition specifies the (zero-origin) record-relative index of the word containing the start of the field; the second and third specify the indices (zero-origin) of the first and last bits of the field with respect to that word. The second and third expressions may specify a bit offset greater than the word size if the word offset is adjusted accordingly. Similarly, the difference between the second and third expressions may exceed the word size. If the bit

positions are not specified, a specification of 0..*n***WordSize*-1 is assumed, where *n* is the minimum number of words required by the type of the field.

Each field must be at least wide enough to store any value of the corresponding type. Values are stored right-justified within the fields. The current implementation of Mesa imposes the following additional restrictions on the sizes and alignment of fields:

> A field smaller than a word (16 bits) cannot cross a word boundary.
>
> Any field occupying a word or more must begin at bit zero of a word and have a size that is a multiple of the word size.
>
> A variant part may begin at any bit position (as determined by its tag field).
>
> If the sizes of all variants of a record type are less then a word, those sizes must be equal; otherwise, the size of each variant of the type must be a multiple of the word length.

In the definition of a machine-dependent record type, explicitly specified field positions must not overlap. For a variant record type, this requirement applies to the variant part (including the tag) considered in conjunction with the fields of the common part; the tag and fields particular to each variant must lie entirely within the variant part.

The order of fields in a record type declaration need not agree with the order of those fields in the representation of the record; however, no gaps are permitted. For variant records, the fields of at least one variant (including the tag field) must fill the position specified for the variant part.

Dynamic Storage Allocation

In Mesa 6, you can use special constructs to describe the dynamic allocation and deallocation of variables. You are still responsible for managing the storage and guarding against dangling pointers; the new features handle certain routine aspects of allocation and deallocation (such as computing sizes), provide proper default initialization of newly allocated variables, and reduce the total number of LOOPHOLEs required to deal with an allocator.

Zones

Allocation and deallocation are done with respect to *zones*. A zone need not be associated with any specific storage area; it is just an object characterized by procedures for allocation and deallocation as described below. The storage managed by a zone in Mesa 6 is said to be *uncounted*. In such zones, object management is the responsibility of the programmer, who must explicitly program the deallocation.

To use an uncounted zone, you must provide the procedures that manage the zone and implement the required set of operations. Many users will be able to import a suitable implementation from a standard package; the details of writing such packages are discussed below.

A zone object has a value and a type. You will normally obtain a zone value by calling a procedure exported by some package implementing zones. Typically, such a procedure constructs a zone (and perhaps an initial storage pool) according to user-supplied parameters.

The type of a zone value must belong to a new class of types, called zone types. Mesa 6 provides two such types, UNCOUNTED ZONE and *MDSZone*. Transactions with objects having these types are generally in terms of LONG POINTER and POINTER values respectively (see below).

Syntactically, UNCOUNTED ZONE is a type constructor. *MDSZone* is a predeclared identifier; you may think of it as a synonym for *MDS* RELATIVE UNCOUNTED ZONE (which you currently cannot write directly).

You may declare variables having zone types (for which fixed initialization is recommended). Zone types may also be used to construct other types. In particular, you may choose to deal with pointers to zones; the NEW and FREE constructs described below provide automatic dereferencing.

Allocating Storage

The operator NEW allocates new storage of a specified type, initializes it appropriately, and returns a pointer to that storage. The NEW operation is considered an attribute of a zone, which must be specified explicitly.

Syntax

Primary	::=	...
	|	Variable . NEW [TypeSpecification Initialization OptCatch]
	|	(Expression) . NEW [TypeSpecification Initialization OptCatch]
Initialization	::=	empty | _ InitExpr | = InitExpr
OptCatch	::=	empty | ! CatchSeries

The value of the Variable or Expression identifies the zone to be used, either directly or after an arbitrary number of dereferencing operations. The TypeSpecification determines the type of the allocated object. If an InitExpr is provided, it must conform to the specified type and its value is used to initialize the new object; otherwise, the default value associated with that type (if any) is used. Only signals raised or propagated by the allocation procedure activate a CatchSeries attached to NEW.

The value of the Primary is a pointer to the newly allocated object. The type of that pointer depends upon the type of the zone and the form of the Initialization. If the argument of NEW is some type *T*, the type of the result is

LONG POINTER TO *T*, if the type of the zone is equivalent to UNCOUNTED ZONE

POINTER TO *T*, if the type of the zone is equivalent to *MDSZone*.

If you specify fixed (=) initialization, the result is a read-only pointer with type LONG POINTER TO READONLY *T* or POINTER TO READONLY *T* respectively.

The InitExpr cannot be the special form for string body initialization ([Expression]). You can, however, allocate string bodies with dynamically computed sizes by using a new form of TypeSpecification (see below). If you do so, the Initialization must be empty.

Releasing Storage

Uncounted zones have FREE operations. When applied to an object, this operation releases the storage allocated for that object.

Syntax

Statement	::=	...
	|	Variable . FREE [Expression OptCatch]
	|	(Expression) . FREE [Expression OptCatch]

The zone used in a FREE operation is determined as described for NEW; it should be the zone from which the variable was originally allocated. The argument of FREE is the *address* of a pointer to the variable to be deallocated; FREE sets the pointer to NIL and deallocates the storage for the variable.

Only signals raised or propagated by the deallocation procedure activate a CatchSeries on a FREE.

Implementing Uncounted Zones

This section describes the assumptions currently made by the compiler about the user-supplied implementations of uncounted zones. These assumptions are compatible with the style of "object-oriented" programming that has proven successful in a number of applications. You need to read this section only if you are designing the interface between a storage management package and the zone features of the language.

An uncounted zone dealing with LONG POINTER values is represented by a two word value, which the compiler assumes to be a long pointer compatible with the following skeletal structure:

```
UncountedZoneRep: TYPE = LONG POINTER TO MACHINE DEPENDENT RECORD [
  procs (0:0..31): LONG POINTER TO MACHINE DEPENDENT RECORD [
    alloc (0): PROC [zone: UncountedZoneRep, size: CARDINAL] RETURNS [LONG POINTER],
    dealloc (1): PROC [zone: UncountedZoneRep, object: LONG   POINTER]
    -- possibly followed by other fields-- ],
  data (2:0..31): LONG POINTER                    -- optional, see below
  -- possibly followed by other fields-- ];
```

If *z* is an uncounted zone, the code generated for *p* _ *z*.NEW[*T*] is equivalent to

```
p _ z^.procs^.alloc[z, SIZE[T]]
```

and the code generated by z.FREE[@*p*] is equivalent to

```
{temp: LONG POINTER _ p; p _ NIL; z^.procs^.dealloc[z, temp]}.
```

Within this framework, you may design a representation of zone objects appropriate for your storage manager. In general, you should create an instance of a *finger* (the record with fields *procs* and *data*) for each instance of a zone. The record designated by the *procs* pointer can be shared by all zones with the same implementation. The *data* pointer normally designates a particular zone and/or the state information characterizing that zone. Note that the compiler makes no assumptions about the designated object and does not generate any code referencing the *data* field. The extra level of indirection provided by that field is not obligatory; you may replace it with state information contained directly in the finger (but following the *procs* field).

The compiler assumes a similar (but single word) representation for an *MDSZone* value; the skeletal structure is as follows:

```
MDSZoneRep: TYPE = POINTER TO MACHINE DEPENDENT RECORD [
  procs (0:0..15): POINTER TO MACHINE DEPENDENT RECORD [
    alloc (0): PROC [zone: MDSZoneRep, size: CARDINAL] RETURNS [POINTER],
    dealloc (1): PROC [zone: MDSZoneRep, object: POINTER]
    -- possibly followed by other fields-- ],
  data (1:0..15): POINTER                    -- optional
  -- possibly followed by other fields-- ];
```

Sequences

A *sequence* in Mesa is an indexable collection of objects, all of which have the same type. In this respect, a sequence resembles an array; however, you need not specify the length of the sequence when its type is declared, only when an instance of that type is created. Mesa 6 provides sequence-containing types for applications in which the size of a dynamically created array cannot be computed statically. Note, however, that only a subset of a more general design for sequences has been implemented. The contexts in which sequence types may appear are somewhat restricted, as are the available operations on them. We believe that the subset provides enough functionality to accomodate most uses of sequences, but you will encounter a number of annoying and sometimes inconvenient restrictions that you must take note of in your Mesa 6 programming.

One can view a sequence type as a union of some number of array types, just as the variant part of a variant record type can be viewed as a union of some (enumerated) collection of record types. Mesa adopts this view, particularly with respect to the declaration of sequence-containing types, with the following consequences:

> A sequence type can be used only to declare a field of a record. At most one such field may appear within a record, and it must occur last.
>
> A sequence-containing object has a tag field that specifies the length of that particular object and thus the set of valid indices for its elements.

To access the elements of a sequence, you use ordinary indexing operations; no discrimination is required. In this sense, all sequences are overlaid, but simple bounds checking is sufficient to validate each access.

Uses of sequence-containing variables must follow a more restrictive discipline than is currently enforced for variant records. The (maximum) length of a sequence is fixed when the object containing that sequence is created, and it cannot subsequently be changed. In addition, Mesa 6 imposes the following restrictions on the uses of sequences:

> You cannot embed a sequence-containing record within another data structure. You must allocate such records dynamically and reference them through pointers. (The NEW operation has been extended to make allocation convenient.)
>
> You cannot derive a new type from a sequence-containing type by fixing the (maximum) length; i.e., there is no analog of a discriminated variant record type.
>
> There are no constructors for sequence-valued components of records, nor are such components initialized automatically.

The following sections describe sequences in more detail.

Defining Sequence Types

You may use sequence types only to declare fields of records. A record may have at most one such field, and that field must be declared as the final component of the record:

Syntax

```
VariantPart     ::=   . . .
                |     PackingOption SEQUENCE SeqTag OF TypeSpecification
```

```
SeqTag              ::=   identifier : Access  BoundsType
                    |     COMPUTED BoundsType
BoundsType          ::=   IndexType

TypeSpecification   ::=   . . .
                    |     TypeIdentifier [ Expression ]
```

The TypeSpecification in VariantPart establishes the type of the sequence elements. The BoundsType appearing in the SeqTag determines the type of the indices used to select from those elements. It is also the type of a tag value that is associated with each particular sequence object to encode the length of that object. For any such object, all valid indices are smaller than the value of the tag. If *T* is the BoundsType, the sequence type is effectively a union of array types with the index types

T[FIRST[*T*] .. FIRST[*T*]), *T*[FIRST[*T*] .. SUCC[FIRST[*T*]]), ... *T*[FIRST[*T*] .. LAST[*T*])

and a sequence with tag value *v* has index type *T*[FIRST[*T*]..*v*). Note that the smallest interval in this union is empty.

If you use the first form of SeqTag, the value of the tag is stored with the sequence and is available for subscript checking. In the form using COMPUTED, no such value is stored, and no bounds checking is possible.

Examples:

```
StackRep: TYPE = RECORD [
  top: INTEGER _ 1,
  item: SEQUENCE size: [0..LAST[INTEGER]] OF T]

Number: TYPE = RECORD [
  sign: {plus, minus},
  magnitude: SELECT kind: * FROM
    short => [val: [0..1000)],
    long => [val: LONG CARDINAL],
    extended => [val: SEQUENCE length: CARDINAL OF CARDINAL]
    ENDCASE]

WordSeq: TYPE = RECORD [SEQUENCE COMPUTED CARDINAL OF Word]
```

The final example illustrates the recommended method for imposing an indexable structure on raw storage.

If *S* is a type containing a sequence field, and *n* is an expression with a type conforming to CARDINAL, both *S* and *S*[*n*] are TypeSpecifications. They denote different types, however, and the valid uses of those types are different, as described below.

MACHINE DEPENDENT *Sequences*

You may declare a field with a sequence type within a MACHINE DEPENDENT record. Such a field must come last, both in the declaration and in the layout of the record, and the total length of a record with a zero-component sequence part must be a multiple of the word length. If you explicitly specify bit positions, the size of the sequence field also must describe a zero-length sequence; i.e., it must account for just the space occupied by the tag field (if any).

Examples:

```
Node: TYPE = MACHINE DEPENDENT RECORD [
  info (0: 0..7): CHARACTER,
  sons (0: 8..15): SEQUENCE nSons (0: 8..15): [0..256) OF POINTER TO Node]

CharSeq: TYPE = MACHINE DEPENDENT RECORD [
  length (0): CARDINAL,
  char (1): PACKED SEQUENCE COMPUTED CARDINAL OF CHARACTER]
```

Allocating Sequences

If *S* designates a record type with a final component that is a sequence, *S*[*n*] is a type specification describing a record with a sequence part containing exactly *n* elements. The expression *n* must have a type conforming to CARDINAL. Its value need *not* be a compile-time constant; however, you can use specifications of this form only to allocate sequence-containing objects (as arguments of NEW) or to inquire about the size of such objects (as arguments of SIZE). In particular, you cannot use *S*[*n*] to define or construct a new type or to declare a variable.

The value of the expression SIZE[*S*[*n*]] has type CARDINAL and is the number of words required to store an object of type *S* having *n* components in its sequence part.

The value of the expression *z*.NEW[*S*[*n*]] has type POINTER TO *S* (or LONG POINTER TO *S*, depending upon the type of the zone *z*). The effect of its evaluation is to allocate SIZE[*S*[*n*]] words of storage from the zone *z* and to initialize that storage as follows:

Any fields in the common part of the record receive their default values.

The sequence tag field receives the value $\text{SUCC}^{n}[\text{FIRST}[T]]$, where *T* is the type of that field.

The elements of the sequence part have undefined values.

To supply initial values for the fields in the common part, you may use a constructor for type *S* in the call of NEW. There are currently no constructors for sequence parts, however, and you must void the corresponding field. In any case, you must explicitly program any required initialization of the elements of the sequence part. In Mesa 6, this is true even if the element type has non-NULL default value.

Examples:

```
ps: POINTER TO StackRep _ z.NEW[StackRep[100]];        -- s.top = 1

pn: POINTER TO Node _ z.NEW[Node[degree[c]] _ [info: c, sons: NULL]]

pxn: POINTER TO extended Number _ z.NEW[extended Number[2*k]]
```

Note that *n* specifies the maximum number of elements in the sequence part and must conform to CARDINAL no matter what BoundsType T_i appears in the SeqTag. The value assigned to the tag field is $\text{SUCC}^{n}[\text{FIRST}[T_i]]$. A bounds fault occurs if this is not a valid value of type T_i, i.e., if $n >$ *cardinality*(T_i), *and* you have requested bounds checking.

If $\text{FIRST}[T_i] = 0$, $\text{SUCC}^{n}[\text{FIRST}[T_i]]$ is just *n*, i.e., the interpretation of the tag is most intuitive if T_i is a zero-origin subrange. Usually you will specify a BoundsType (e.g., CARDINAL) with a range that comfortably exceeds the maximum expected sequence length. If, however, some maximum length *N* is important to you, you should consider using [0..*N*] as the BoundsType; then the value of the tag field in a sequence of length *n* ($n < N$) is just *n* and the valid indices are in the interval [0..*n*).

Operations on Sequences

You can use a sequence-containing type *S* only as the argument of the type constructor POINTER TO. Note that the type of *z*.NEW[*S*[*n*]] is POINTER TO *S* (not POINTER TO *S*[*n*]). If the type of an object is *S*, the operations defined upon that object are

ordinary access to fields in the common part

readonly access to the tag field (if not COMPUTED)

indexing of the sequence field

constructing a descriptor for the components of the sequence field (if not COMPUTED).

There are no other operations upon either type *S* or the sequence type embedded within *S*. In particular, you cannot assign or compare sequences or sequence-containing records (except by explicitly programming operations on the components).

Indexing: You may use indexing to select elements of the sequence-containing field of a record by using ordinary subscript notation, e.g., *s.seq*[*i*]. The type of the indexing expression *i* must conform to the BoundsType appearing in the declaration of the sequence field and must be less than the value of the tag, as described above. The result designates a variable with the type of the sequence elements. A bounds fault occurs if the index is out of range, the sequence is not COMPUTED, *and* you have requested bounds checking.

By convention, the indexing operation upon sequences extends to records containing sequence-valued fields. Thus you need not supply the field name in the indexing operation. Note too that both indexing and field selection provide automatic dereferencing.

Examples:

ps^.*item*[*ps.top*] *ps.item*[*ps.top*] *ps*[*ps.top*] -- all equivalent

Descriptors: You may apply the DESCRIPTOR operator to the sequence field of a record; the result is a descriptor for the elements of that field. The resulting value has a descriptor type with index and component types and PACKED attribute equal to the corresponding attributes of the sequence type. By extension, DESCRIPTOR may be applied to a sequence-containing record to obtain a descriptor for the sequence part. The DESCRIPTOR operator does not automatically dereference its argument.

You cannot use the single-argument form of the DESCRIPTOR operator if the sequence is COMPUTED. The multiple-argument form remains available for constructing such descriptor values explicitly (and without type checking).

In any new programming, you should consider the following style recommendation: use sequence-containing types for allocation of arrays with dynamically computed size; use array descriptor types only for parameter passing.

Examples:

DESCRIPTOR[*pn*^] DESCRIPTOR[*pn.sons*] -- equivalent

String Bodies and TEXT

The type *StringBody* provided by previous versions of Mesa illustrates the intended properties and uses of sequences. For compatibility reasons, it has not been redefined as a sequence; the declarations of the types STRING and *StringBody* remain as follows:

```
STRING: TYPE = POINTER TO StringBody;

StringBody: TYPE = MACHINE DEPENDENT RECORD [
  length (0): CARDINAL _ 0,
  maxlength (1): --READONLY-- CARDINAL,
  text (2): PACKED ARRAY [0..0) OF CHARACTER]
```

The operations upon sequence-containing types have, however, been extended to *StringBody* so that its operational behavior is similar. In these extensions, the common part of the record consists of the field *length*, *maxlength* serves as the tag, and *text* is the collection of indexable components (packed characters). Thus *z*.NEW[*StringBody*[*n*]] creates a *StringBody* with *maxlength* = *n* and returns a STRING; if *s* is a STRING, *s*[*i*] is an indexing operation upon the text of *s*, DESCRIPTOR[*s*^] creates a DESCRIPTOR FOR PACKED ARRAY OF CHARACTER, etc.

> There are two anomalies arising from the actual declaration of *StringBody*: *s.text*[*i*] *never* uses bounds checking, and DESCRIPTOR[*s.text*] produces a descriptor for an array of length 0. Use *s*[*i*] and DESCRIPTOR[*s*^] instead.

Type TEXT

The type TEXT, which describes a structure similar to a *StringBody* as a true sequence, is predeclared in Mesa 6. Its components *length* and *maxLength* are declared to have a type compatible with either signed or unsigned numbers (but with only half the range of INTEGER or CARDINAL).

```
TEXT: TYPE = MACHINE DEPENDENT RECORD [
  length (0): [0..LAST[INTEGER]] _ 0,
  text (1): PACKED SEQUENCE maxLength (1): [0..LAST[INTEGER]] OF CHARACTER]
```

Exported Types

An *exported type* is a type designated by an identifier that is declared in an interface and subsequently bound to some *concrete type* supplied by a module exporting that interface. This is analogous to the current treatment of procedures in interfaces, where the implementations of procedures (i.e., the procedure bodies) do not appear in the interface but are defined separately. The advantages are twofold:

> The internal structure of the type is guaranteed to be invisible to clients of the interface.
>
> There are no compilation dependencies between the definition of the concrete type and the interface module. The definition of that type can be changed and/or recompiled at any time (perhaps subject to a size constraint; see below) without requiring recompilation of either the interface or any client of the interface.

The uses of an exported type are the same as those of any other type, e.g, to construct other types. The value provided by the interface is constant but has no accessible internal structure. In Mesa 6, there are two other important differences between exported procedures and exported types.

The first is a restriction necessary to ensure type safety across module boundaries. Different exporters of an interface can supply different implementations of any particular procedure in that interface. In Mesa 6, *this is not true for exported types*; all exporters of a particular type within a configuration must supply the same concrete type, which is called the *standard implementation* of that exported type. Because of this restriction, clients can safely interassign values with exported type *T*, no matter how obtained. In addition, any exporter of *T* may convert a value of type *T* to a value of the concrete type it uses to represent *T* and conversely.

The second difference is that it is not necessary to import an interface to access an exported type defined within it or to distinguish among values of such a type coming from different imported

instances. This is another consequence of the fact that, in Mesa 6, all interfaces must reference the standard implementation of the exported type.

Interface Modules

An exported type is declared in an interface (DEFINITIONS) module using one of the following two forms:

T: TYPE;
T: TYPE [Expression];

The first of these introduces a type *T*, *no* properties of which are known in the interface or to any client of the interface. In particular, the size of *T* is not known; this is adequate (and desirable) if the interface and clients deal only with values of type POINTER TO *T*.

The second form specifies the size of the values used in the representation of the type. The value of Expression, which must denote a compile-time constant with an unsigned integer value, gives this size in units of machine words. Supplying the size of an exported type is a shorthand for exporting a set of fundamental operations (creation, _ , =, and #) upon that type. In Mesa 6, the eventual concrete type must supply the *standard implementations* of these operations, which are defined as follows:

creation	allocate the specified number of words, with no initialization
_	copy an uninterpreted bit string
=, #	compare uninterpreted bit strings

Note that a type with non-NULL default value does not have the standard creation operation. Such types cannot be exported with known size. You should therefore consider writing your interfaces in terms of POINTER TO *T*, where *T* is a completely opaque exported type and not subject to these restrictions.

Client Modules

A client has no knowledge of the type *T* beyond those properties specified in the interface. If the size is not specified there, no operations on *T* are permitted. If the size is available from the interface, SIZE[*T*] is legal; also declaration of variables (including record fields and array components) and the operations _, =, # are defined for type *T*.

Implementation Modules

An implementor exports a type *T* to some interface *Defs* by declaring the type with the required identifier, the PUBLIC attribute, and a value that is the concrete type; e.g., in

T: PUBLIC TYPE = *S*;

S specifies the concrete type. If the size of *T* appears in the interface, the definition of *T* in the exporter must specify a type with that size and with the standard fundamental operations (the compiler checks this).

Within an exporter, *Defs.T* and *T* conform freely and are assignment compatible. Otherwise, *Defs.T* is treated opaquely there and is *not* equivalent to *T* (except for the purpose of checking exports). You should therefore attempt to write an exporting module entirely in terms of concrete types. Consider the following example:

Interface Module (*Defs*):

```
T: TYPE;
H: TYPE = POINTER TO T;
R: TYPE = RECORD [f: H, ...];
Proc1: PROC [h: H];
Proc2: PROC [r: POINTER TO R];
...
```

Exporting Module:

```
T: PUBLIC TYPE = RECORD [v: ...];
P: TYPE = POINTER TO T;
Proc1: PUBLIC PROC [h: P] = {... h.v ...};
Proc2: PUBLIC PROC [r: POINTER TO Defs.R] = {
  q: P = r.f;
  ... q.v ...};
...
```

If the type of *h* were *Defs.H* in the implementation of *Proc1*, the reference to *h.v* would be illegal. By defining a type such as *P* and using it within the exporter instead of *H*, you can avoid most such problems. (Note that *Proc1* is still exported with the proper type.) This strategy of creating concrete types in one-to-one correspondence to interface types involving *T* fails for record types such as *R* (because of the uniqueness rule for record constructors). In this example, you must use *Defs.R* to define the type of *r* in the implementation of *Proc2*, but a reference to *r.f.v* is illegal. In such cases, a LOOPHOLE-free implementation may require redundant assignments, such as the one to *q*. Alternatively, you should consider making the record type another exported type, and defining its concrete type within the exporter also.

Binding

For each interface containing some exported type *T*, all exporters of that interface must provide equivalent concrete types for *T* (the binder and loader check this). In Mesa 6, the concrete types must in fact be identical; if two modules export *T*, they must obtain the same concrete definition of *T*, e.g., from another shared interface module (typically, a private one).

Control Variables

You can now declare the control variable of a loop as part of the FOR clause attached to that loop. Such an identifier cannot be accessed outside the loop and cannot be updated except by the FOR clause in which it is declared.

Syntax

```
Iteration     ::=   FOR identifier Direction IN LoopRange
                |   FOR identifier : TypeExpression Direction IN LoopRange
Assignation   ::=   FOR identifier _ Expression , Expression
                |   FOR identifier : TypeExpression _ Expression , Expression
```

The forms of Iteration and Assignation with ": TypeExpression" declare a new control variable. That variable cannot be explicitly updated (except by the FOR clause itself). Its scope is the entire LoopStmt introduced by the Iteration or Assignation including any LoopExitsClause. Note, however, that the value of a control variable used in an Iteration is undefined in the FinishedExit.

Extended NIL

In Mesa 6, null values are available for all address-containing types. An address-containing type is one constructed using POINTER, DESCRIPTOR, PROCEDURE, PROGRAM, SIGNAL, ERROR, PROCESS, PORT, ZONE or a LONG or subrange form of one of the preceding. The built-in type STRING is address-containing. A relative pointer or relative descriptor type is not considered to be address-containing in Mesa 6.

Null values are denoted as follows:

> If *T* designates any address-containing type, NIL[*T*] denotes the corresponding null value.
>
> Whenever *T* is implied by context, NIL abbreviates NIL[*T*].
>
> If *T* is not implied by context, NIL means NIL[POINTER TO UNSPECIFIED] and thus continues to match any POINTER or LONG POINTER type.

A fault will occur if you attempt to dereference a null value *and* have requested NIL checking; a fault will occur unconditionally if you attempt to transfer control through a null value.

Reject Statement

Within a catch phrase, you can use the statement REJECT to explicitly reject a signal, i.e., to terminate execution of that catch phrase and propagate the signal to the enclosing one. (Note that each catch phrase is currently terminated by an implicit REJECT.)

Process Extensions

Aborting a process now raises the predeclared signal ABORTED. The predeclared types MONITORLOCK and CONDITION are now defined with default initialization. The only client-visible field is *timeout* in CONDITION; its default initial value is 100 ticks.

Restrictions on Assignment

The assignment operations defined upon certain types have been restricted so that variables of those types can be initialized (either explicitly or by default) when they are created but cannot subsequently be updated. A variable is considered to be created at its point of declaration or, for dynamically allocated objects, by the corresponding NEW operation.

In Mesa 6, the following types have restricted assignment operations:

> MONITORLOCK
>
> CONDITION
>
> any type constructed using PORT
>
> any type constructed using SEQUENCE
>
> any type constructed using ARRAY in which the component type has a restricted assignment operation.
>
> any type constructed using RECORD in which one of the field types has a restricted assignment operation.

Note that the restrictions upon assignment for a type do not impose restrictions upon assignment to component types. Thus selective updating of fields of a variable may be possible even when the

entire variable cannot be updated; e.g., the *timeout* field of a CONDITION variable can be updated by ordinary assignment. Also, you may apply the operator @ to obtain the address of the entire variable in such a situation.

Operational Changes

User Interface

The standard Mesa 6 Compiler reads commands only from the executive's command line; it no longer supports interactive input. During compilation, the display and keyboard are disabled. The cursor provides a limited amount of feedback; it moves down the screen to indicate progress through a sequence of commands and to the right as errors are detected. At the end of compilation, the message "Type Key" is displayed in a flashing cursor if there are errors and you have requested the compiler to pause. Typing `Shift-Swat` aborts the Executive's current command sequence; `Ctrl-Swat` invokes the Mesa Debugger; any other character causes normal exit from the compiler.

A summary of compilation commands is written on the file `Compiler.log` (formerly, `Mesa.typescript`).

Command Line Arguments

The Mesa 6 Compiler allows you to control the association between modules and file names at the time you invoke the compiler. The compiler accepts a series of commands, each of which has the form

$$\texttt{outputFile \_ inputFile[}id_1\texttt{: }file_1\texttt{, ..., }id_n\texttt{: }file_n\texttt{]/switches}$$

Only `inputFile` is mandatory; it names the file containing the source text of the module to be compiled, and its default extension is `.mesa`. Any warning or error messages are written on the file `outputRoot.errlog`, where `outputRoot` is the string obtained by deleting any extension from `outputFile`, if given, otherwise from `inputFile`. If there are no errors or warnings, any existing error log with the same name is deleted at the end of the compilation.

If a list of keyword arguments appears between brackets, each item establishes a correspondence between the name id_i of an included module, as it appears in the DIRECTORY of the source program, and a file with name $file_i$; the default extension for such file names is `.bcd`. (If the name of an included module is not mentioned on the command line, its file name is computed from information in the DIRECTORY statement; see above).

The optional `switches` are a sequence of zero or more letters. Each letter is interpreted as a separate switch designator, and each may optionally be preceded by `-` or `~` to invert the sense of the switch.

If `outputFile` (and _) are omitted, the object code and symbol tables are written on the file `inputRoot.bcd`, where `inputRoot` is `inputFile` with any extension deleted. Otherwise code and symbols are written on `outputFile`, for which a default extension of `.bcd` is supplied. If the compiler detects any errors, the output file is not written and any existing file with the same name is deleted.

The compiler accepts a sequence of one or more commands from the executive's command line (through the file `Com.cm`). Commands are separated by semicolons, but you may omit a semicolon

between any two successive identifiers (file names or switches), or between a] and an identifier (but not between an identifier and a `/`). Note that any required semicolon in an Alto Executive command must be quoted.

You can set global switches by a command with an empty file name. In the form `/switches`, each letter designates a different switch. Unless a command to change the global switch settings comes first in the sequence of commands, you must separate it from the preceding command by an explicit semicolon. Note that the form switch/c is no longer available for setting global switches.

Switches

The following compilation options have been added:

Switch	*Option Controlled*
`f`	implementation of floating-point operations
`l`	treatment of long pointers
`y`	warning on runtime calls

If the `f` switch is set, the compiler generates byte code instructions for floating-point operations (these require microcode support); otherwise, it generates calls through the system dispatch vector (SD) to software routines implementing such operations. If the `a` and `l` switches are both set, the compiler generates code using an variant of the Alto Mesa instruction set that implements long pointer accesses to a virtual memory larger than 64K (code generated using the `l` switch cannot be executed on an Alto, even if long pointers are not used). If you specify `-a`, the `l` switch is ignored. The `y` switch indicates that a warning message should be issued whenever the compiler generates code to invoke a runtime procedure (including some "instructions" which are actually implemented in software).

The default settings for these switches are `f`, `-l` and `-y`.

Distribution:
Mesa Users
Mesa Group
SD Support

Inter-Office Memorandum

To	Mesa Users	Date	October 27, 1980
From	Brian Lewis, Ed Satterthwaite	Location	Palo Alto
Subject	**Mesa 6.0 Binder Update**	Organization	SDD/SS/Mesa

XEROX

Filed on: [Iris]<Mesa>Doc>Binder60.bravo (and .press)

This memo describes changes to the C/Mesa language and the Binder that have been made since the release of Mesa 5.0 (April 9, 1979).

Definitions of syntactic phrase classes used but not defined in this document come from the *Mesa Language Manual, Version 5.0*, Section 7.7.

Compatibility

Because of changes in BCD formats, you must rebind all your existing Mesa configurations after obtaining recompiled versions of their components. There is one potential incompatibility:

> Some of the quoted file names that appear in DIRECTORY clauses have different interpretations. Text inside angle brackets is no longer ignored; it is treated as the name of a local subdirectory (but we do not recommend using local subdirectories for Mesa programs at the present time).

A number of bugs have been fixed. As usual, the list of Binder-related change requests closed by Mesa 6.0 will appear separately as part of the Software Release Description. The most notable involve

> correctly checking the types of interfaces when positional notation is used,
>
> proper treatment of named imports or exports of a configuration,
>
> proper identification of object files that have been renamed.

Because of bug fixes, the Binder may reject previously acceptable configuration descriptions or may issue new warning messages. A number of error or warning messages now use symbolic names whenever the corresponding symbol tables are available.

New Language Features

Bracket Symbols

The bracket pair { } can be used in place of BEGIN END.

Syntax

CBody	::=	BEGIN CStatementSeries END
	|	{ CStatementSeries }

Multiple Control Modules

You can now specify an ordered list of control modules for any configuration.

Syntax

ControlClause	::=	CONTROL IdList
	|	empty
IdList	::=	identifier | IdList , identifier

When a (sub)configuration is started, either explicitly or as the result of a start trap (see the *Mesa Language Manual, Version 5.0*, Section 7.8.4), each of the modules named in the ControlClause is started in order.

Note that, if there are calls from a control module to one of its successors in the list, the order of starting will not necessarily follow the order of the ControlClause. In starting a configuration, any control modules that have already been started are skipped.

Operational Changes

The Binder is now available only as a `.bcd` file; you must have `Mesa.image` to run it. When compressing symbols, SymbolCompressor.bcd must be loaded first; see the description of the /x switch below.

User Interface

The Binder now reads commands only from the Alto Executive's command line; it no longer supports interactive input. At the start of the first binding, the message "Bind" is displayed in the cursor. If there are any warnings, "Warning" is displayed, and if there are errors "Errors" is shown. At the end of binding, the message "Type Key" is displayed in a flashing cursor if there are errors and you have requested the Binder to pause. Typing `Shift-Swat` aborts the executive's current command sequence; `Ctrl-Swat` invokes the Mesa Debugger; any other character causes normal exit from the Binder.

A summary of binder commands is written on the file `Binder.log` (formerly `Mesa.typescript`).

Commands

The Mesa 6.0 Binder allows you to control the names and contents of various output files when you invoke it; it accepts a series of commands, each of which usually has one of the following forms:

```
inputFile/switches
outputFile _ inputFile/switches
[key1: file1, ... keym: filem] _ inputFile/switches
```

(It is also possible to control the association between included modules and configurations and their file names; this is described below.) In the last form, `key` is one of `bcd`, `code`, or `symbols`.

The string `inputFile` names the file containing the source text of the configuration description, and its default extension is `.config`.

There is a *principal* output file, the name of which is determined as follows:

> If you use the first command form, it is `inputRoot.bcd`, where `inputRoot` is the string obtained by deleting any extension from `inputFile`.
>
> If you use the second form, it is `outputFile`, with default extension `.bcd`.
>
> If you use the third form and key_i is `bcd`, it is $file_i$, with default extension `.bcd`; otherwise, it is obtained as described for the first form.

If the Binder detects any errors, the principal output file is not written, and any existing file with the same name is deleted.

You may also request that the code and/or symbols of the constituent modules be copied to an output file, as follows:

> You request copying of code by specifying the `/c` switch *or* by using the third command form with keyword `code`. Code is copied to the principal output file unless you use the third form and key_i is `code`, in which case the code is copied to a file named $file_i$, with default extension `.code`.
>
> You request copying of symbols by specifying the `/s` or `/x` switch *or* by using the third command form with keyword `symbols`. Symbols are copied to the file `inputRoot.symbols` unless you use the third form and key_i is `symbols`, in which case they are copied to a file named $file_i$, with default extension `.symbols`. Compressed symbols are copied if the `/x` switch is specified.

Any warning or error messages are written on the file `outputRoot.errlog`, where `outputRoot` is the string obtained by deleting any extension from the name of the principal output file. If there are no errors or warnings, any existing error log with the same name is deleted at the end of the bind.

When more than one Binder command is given on the command line, the commands must be separated by semicolons. However, you may omit a semicolon between any two successive identifiers (file names or switches), or between a] and an identifier (but not between an identifier and a `/`). Note that any required semicolon in an Alto Executive command must be quoted.

Switches

The optional `switches` are a sequence of zero or more letters. Each letter is interpreted as a separate switch designator, and each may optionally be preceded by `-` or `~` to invert the sense of the switch.

The Binder recognizes the switches:

- `/c` - copy code
- `/s` - copy symbols
- `/x` - copy compressed symbols
- `/p` - pause after binding if there are errors or warnings
- `/r` - run the specified `.image` or `.run` file
- `/g` - (has no effect; retained for compatibility with Mesa 5)

The Mesa 5 Binder switch `/o` (output file) is no longer available.

In earlier versions of the Binder, copying symbols was an error if not all of the symbol files were available at bind time. The Binder now copies all symbols that it can find, leaves the symbol table references for the other modules in the original (unavailable) files, and issues a warning.

The switches `/c` and `/s` are interpreted differently in Mesa 6 than they were in Mesa 5. The following table outlines these changes.

Mesa 6	*Mesa 5*	*meaning*
`file/c`	`file/c`	code to file.bcd
`file/cs`	`file/c file/s`	code to file.bcd; symbols to file.symbols
`[symbols: file.bcd] _ file/c`	`file/cs`	code and symbols to file.bcd

Note that `file/cs` has quite a different meaning in Mesa 6 than before. Also, the common Mesa 5 command sequence `file/c file/s` will bind `file` *twice* in Mesa 6, the first time copying only code and the second time only symbols.

Compressed symbols are copied with the `/x` switch. In this mode, only the following symbols are included: all procedures, and the signals declared at the top level of modules. Symbols for their parameters and results are not copied. This option allows limited but often adequate debugging, and substantially reduces the size of the symbols file (typically by more than 50%). To copy compressed symbols, `SymbolCompressor.bcd` must be loaded ahead of the Binder. Thus

```
>Mesa.image SymbolCompressor.bcd Binder.bcd MySystem/x
```

will create `MySystem.bcd` and `MySystem.symbols` (compressed).

Global switches are set by a command with an empty file name. In the form `/switches`, each letter designates a different switch. The switches to copy code (`/c`), to copy symbols (`/s`), and to compress symbols (`/x`) may now be given as global switches, and hence apply to all source files thereafter. Unless a command to change the global switch settings comes first in the sequence of commands, it must be separated from the preceding command by an explicit semicolon.

Associating File Names with Modules and Configurations

The Binder now lets you control the association between file names and modules or subconfigurations when you call it. This is done by specifying a list of component identifier-file name pairs inside brackets after the input file name. For example, the command

```
MySystem[Test: UnpackedTest]
```

will bind `MySystem.config` using the previously bound configuration **Test** that is stored on the file `UnpackedTest.bcd`. A command that includes one of these optional component-file name lists will have one of the forms:

```
inputFile[id1: file1, ... idn: filen]/switches
outputFile _ inputFile[id1: file1, ... idn: filen]/switches
[key1: file1, ... keym: filem] _ inputFile[id1: file1, ...
                                           idn: filen]/switches
```

The module or configuration named by id_i in the configuration description will be read from the file $file_i$; `.bcd` is the default extension.

Distribution:
Mesa Users
Mesa Group
SD Support

Inter-Office Memorandum

To	Mesa Users	Date	October 27, 1980
From	John Wick	Location	Palo Alto
Subject	**Mesa 6.0 System Update**	Organization	SDD/SS/Mesa

XEROX

Filed on: [Iris]<Mesa>Doc>System60.bravo (and .press)

This memo outlines changes made in the Mesa system interfaces since the last release (Mesa 5.0, April 9, 1979). A number of internal changes made in the system and the microcode are also discussed.

This memo is intended as a quick guide to conversion, not a detailed specification of the changes. Names in square brackets refer to sections of the *Mesa System Documentation* or to other publicly available reference documents (e.g., the *Alto Operating System Reference Manual*).

External Interfaces

Major changes include integrated support for Alto extended memory and elimination of BasicMesa. The System also now exports versions of the following Development Software interfaces: **Ascii**, **Format**, **Inline**, **Process**, **Runtime**, **Storage**, **String**, **System**, and **Time**. (Note that implementation of these interfaces may not be complete.) Other changes are relatively minor.

AllocDefs

Private types and operations have been removed. **AllocInfo**, **MakeDataSegment**, and **MakeSwappedIn** are now defined in **SegmentDefs**, and temporarily duplicated here for compatibility. [**Segment Package**]

AltoDefs

MaxVMPage has been increased to support up to a million words of memory; **MaxMDSPage** and **PagesPerMDS** have been added. [**Segment Package**]

AltoDisplay

MaxBitsPerLine has been changed to 608 (it was 606). **Cursor**, **CursorBits**, and **CursorHandle** define the location and format of the cursor. **Coordinate**, **CursorXY**, and **MouseXY** define the location and format of the cursor and mouse coordinates. [**Display Package**]

AltoFileDefs

Support for the **DiskShape** and **PartitionName** properties of the directory's leader page has been added. The definition of a file serial number (**SN**) has been changed to isolate the flag bits (**directory**, **random**, and **nolog**) into a separate structure (**SNBits**). [*Alto Operating System Reference Manual*]

AltoHardware

This new interface defines most structures of the Alto hardware, including the processor, display, keyboard, mouse, keyset, printer interface, disk, and Ethernet. [*Alto Hardware Manual*]

Ascii

This new interface defines the ASCII control character codes; for compatibility, these continue to be defined in **IODefs**. [**StreamIO Package**]

BasicMesa

The facilities of **BasicMesa** have been replaced by procedures in the standard system and a command line switch (`/b`) which can be used to destroy the display and keyboard packages (see **DisplayDefs** and **StreamDefs**). **MakeImage** is no longer a standard part of `Mesa.image`, and must be loaded separately. [**Section 3**]

BitBltDefs

The extended memory option now supports use of the normal and alternate bank registers, whose values are supplied in the unused word of the **BBTable** (this option is *not* supported under XMesa 5.0 microcode (version 39)). **AlignedBBTable** (and **BBTableSpace**) can be used to properly align **BITBLT** argument records. [*Alto Hardware Manual*]

CharIO

This new interface provides many of the functions of **IODefs**, but each operation takes a **StreamDefs.StreamHandle** as its first parameter, allowing formatted input and output to any standard stream. [**StreamIO Package**]

DirectoryDefs

This interface has been changed slightly to speed up directory searches (by about a factor of 3). In addition, support for subdirectories was added (see *Alto Operating System Reference Manual*). The following items have changed (note that a NIL **DiskHandle** does *not* imply the system directory):

```
EnumerateEntries: PROCEDURE [
  dir: DiskHandle,
  proc: PROCEDURE [CARDINAL, StreamScan.Handle, DEptr] RETURNS [BOOLEAN],
  inspectFree: POINTER TO READONLY BOOLEAN,
  lengthFilter: CARDINAL _ 0] RETURNS [index: CARDINAL];
```

The procedure **proc** is called for each directory entry; free entries are passed only if **inspectFree^** is TRUE. If the **lengthFilter** is non-zero, only entries with a filename length equal to **lengthFilter** characters will be passed to **proc**.

The following procedure inserts an entry into the directory; unlike **Lookup**, it does not create a file. If the file already exists, TRUE is returned (and **fp^** is undisturbed).

```
Insert: PROCEDURE [
  dir: DiskHandle, fp: POINTER TO AltoFileDefs.FP, name: STRING]
  RETURNS [old: BOOLEAN];
```

ParseFileName replaces **ExpandFileName**; it strips the leading directory information from **name**, puts the result in **filename** (appending a period if necessary), and returns a stream (with access **dirAccess**) open on the directory in which the file should be looked up.

```
ParseFileName: PROCEDURE [
  name, filename: STRING, dirAccess: SegmentDefs.AccessOptions]
  RETURNS [StreamDefs.DiskHandle];
```

The following procedures set and return the directory used for looking up files which do not specify a directory name (initially set to `"<SysDir."`).

SetWorkingDir: PROCEDURE [dir: SegmentDefs.FileHandle];

GetWorkingDir: PROCEDURE RETURNS [dir: SegmentDefs.FileHandle];

Finally, the signal **BadDirectory** no longer takes a string parameter. [**Directory Package**, *Alto Operating System Reference Manual*]

DisplayDefs

DestroyDisplay can be used to delete the display package; it turns off the display, deallocates the bitmap, destroys the font, and UNNEWs all the display modules. [**Display Package**]

DoubleDefs

This interface is no longer implemented or supported, since **LONG** data types are now a standard part of the language and runtime support.

FrameDefs

Validate(Global)Frame and **Invalid(Global)Frame** now take (return) **UNSPECIFIED**. The procedure

LoadConfig: PROCEDURE [name: STRING] RETURNS [PROGRAM];

loads a configuration without starting it and returns its control module or control module list (or NIL if there is no control module). Note that this will not handle configurations whose control modules take parameters. [**Modules**]

FSPDefs

The error **ZoneTooLarge** is now raised by **Make(New)Zone** and **AddTo(New)Zone** when an attempt is made to make a zone of more than 32K words. [**Storage Management**]

ImageDefs

MakeImage takes an optional second parameter (**merge: BOOLEAN _ TRUE**); **MakeUnMergedImage** has been temporarily retained for compatibility. The **ImageMaker** package is no longer a part of the standard system; it must be loaded or bound with the client configuration. [**Image Files**]

InlineDefs

The **LongCOPY** operation for use with long pointers is now implemented by the extended memory microcode. The types **BytePair** and **BcplLongNumber** have been added; the procedures **MesaToBcplLongNumber** and **BcplToMesaLongNumber** implement conversion between Mesa and BCPL long numbers. [**Miscellaneous**]

IODefs

The procedure **WriteSubString** has been added. [**StreamIO Package**]

MiscDefs

The **ByteBlt** procedure has been added. [**Miscellaneous**]

MiscOps

ReleaseDebuggerBitmap can be used to free the storage normally allocated (on extended memory machines) for the Debugger's bitmap. [**Section 3**]

OsStaticDefs

The type of **ClockSecond** has been changed to use **InlineDefs.BcplLongNumber**. [*Alto Operating System Reference Manual*]

ProcessDefs

Aborted has been redefined to be equal to the predeclared error ABORTED. A **Pause** procedure has been added which delays execution of its caller by the specified number of ticks. **Detach** and **GetCurrent** now take (return) a **PROCESS** instead of an **UNSPECIFIED**. [**Processes and Monitors**]

SegmentDefs

The majority of changes in **SegmentDefs** are due to the incorporation of XMesa and extended memory support into the standard system. Clients of XMesa should see the XMesa update document.

The definition of **SegmentObjects** has changed to allow for a twelve bit page number. The **read** bit in **FileSegmentObjects** has been deleted (read access is always assumed) and **MaxSegLocks** has been reduced to fifteen (**MaxSegLocks** and **MaxFileLocks** replace **MaxLocks**).

A type field has been added to **DataSegmentObject**s with predefined values **UnknownDS**, **FrameDS**, **TableDS**, **HeapDS**, **SystemDS**, **BitmapDS**, **StreamBufferDS**, and **PupBufferDS**. These types are interpreted by the Debugger's `Coremap` command.

For clients of low level memory allocation, the definition of **AllocInfo** has changed and the constants **HardUp**, **HardDown**, **EasyUp**, and **EasyDown** have been defined. The procedures **MakeDataSegment** and **MakeSwappedIn** have been moved here from **AllocDefs**.

The access options **ReadWrite**, **WriteAppend** and **ReadWriteAppend** have been added. **NewFile** and **InsertFile** now default the **access** and **version** parameters.

The following two procedures have been added to provide access to file times:

```
GetFileTimes: PROCEDURE [file: FileHandle]
  RETURNS [read, write, create: TimeDefs.PackedTime];

SetFileTimes: PROCEDURE [
  file: FileHandle,
  read, write, create: TimeDefs.PackedTime _ TimeDefs.DefaultTime];
```

GetFileTimes does not modify any of the file's times. In **SetFileTimes**, if any of the times are defaulted, the current time is used. [**Segment Package**]

StreamDefs

DestroyKeyHandler can be used to delete the standard keyboard handler; it destroys the keyboard process and UNNEWs all the keyboard modules. [**Keyboard**]

The type **StreamPosition**, defined as a **LONG CARDINAL**, can be used in place of a **StreamIndex**. The operations **GetPosition**, **SetPosition**, and **ModifyPosition** are similar to the corresponding index operations; **IndexToPosition** and **PositionToIndex** perform conversions between positions and indicies. The access options **ReadWrite**, **WriteAppend**, and **ReadWriteAppend** have been added, as has the signal **FileNameError**. [**Disk Streams**]

StreamScan

This new interface allows overlapped disk input when reading from a stream. It is a transliteration of the same code from the Alto Operating System (version 17). The following are defined in StreamScan.mesa:

```
Descriptor: TYPE = RECORD [
  da: AltoFileDefs.vDA,
  pageNumber: CARDINAL,
  numChars: CARDINAL,
  -- private fields];

Handle: TYPE = POINTER TO READONLY Descriptor;

Init: PROCEDURE [
  stream: StreamDefs.StreamHandle, bufTable: POINTER, nBufs: CARDINAL]
  RETURNS [Handle];

GetBuffer: PUBLIC PROCEDURE [ssd: Handle] RETURNS [POINTER];

Finish: PROCEDURE [ssd: Handle];
```

Init sets up a scan stream from a disk stream. In addition to the stream, the client supplies a vector of pointers to 256 word areas useable as disk buffers (**bufTable**). The number of buffers supplied is **nBufs**. At least one buffer must be supplied (the normal stream buffer is also used). Each call to **GetBuffer** will return a pointer to the next sequential page of the file and returns the previous buffer page to the buffer pool (first call returns data page 0; file page 1). The public fields of the **Handle** are correct for the page returned by the most recent call to **GetBuffer**. **GetBuffer** returns **NIL** when there are no more pages to be read. A call to **Finish** terminates the scan. No other stream operations should be performed between **Init** and **Finish**. [**Disk Streams Package**]

StringDefs

CompareStrings lexically compares two strings and returns -1, 0, or 1 if the first is less than, equal to, or greater than the second; an optional parameter may be supplied to ignore case differences. All procedures in this interface now handle NIL string parameters. [**String Package**]

SystemDefs

CopyString allocates storage from the system heap and copies its argument into it, optionally making the new string longer. **ExpandString** performes a similar function, allocating a new string (and freeing the old one) if necessary. **Even** and **Quad** can be used to align pointers on double and quad word boundaries. [**Storage Management**]

TimeDefs

The type **HardwareTime** has been replaced by **InlineDefs.BcplLongNumber**. Default values have been added to **UnpackDT**, **PackDT**, and **AppendDayTime**. **ReadClock** returns the current value of the Alto's realtime clock (part of which can be found at location **RealTimeClock**). [**Time Package**]

TrapDefs

StackError no longer takes a parameter; **UnboundProcedure** now takes an **UNSPECIFIED**. The following signals have been added (not all of which can be generated by Alto/Mesa): **ZeroDivisor**, **DivideCheck**, **UnimplementedInst**, **WakeupError**, **PageFault**, **Write-ProtectFault**, and **HardwareError**. [**Traps**]

XMesa Extended Memory Support

Functions formerly provided by XMesa are now integrated with the standard system. A 3K RAM or Mesa microcode in ROM1 is required to support the extended memory option. [**Segment Package**]

Internal Interfaces

The following changes are internal to the implementation and do not affect public interfaces; they may affect performance and/or space requirements, however. Note that Mesa 6.0 continues to support version 39 of the XMesa microcode available with Mesa 5.0; obviously, certain new features listed below are not available if your ROM contains the old microcode (e.g., Long BitBlt).

3K RAM Support

Mesa now supports the Alto 3K RAM option (available only on extended memory machines).

Debugger Bitmap

If the extended memory option is present, the system allocates part of the client's memory for use by the debugger for its display bitmap; this improves the debugger's response times considerably. The debugger bitmap may be deallocated by the procedure **MiscOps.ReleaseDebuggerBitmap** or the command line switch `/k` (in the former case, the call must be made before the debugger is first entered).

Long Copy, Long BitBlt

These opcodes now include support for extended memory.

Misc Opcodes

Misc opcodes (except for **RCLK**) now provide a general escape to user microcode in the RAM if Mesa is running on a 2K ROM or 3K RAM machine; they produce undefined results otherwise. Alpha bytes for the currently implemented MISC functions are defined in **MiscAlpha**.

Overflow Microcode

RunMesa has been upgraded to include microcode support for Pup checksums, IEEE floating point, and HBlt (used by Griffin). This microcode is loaded with the XMesa overflow microcode on Altos with the 2K ROM (with version 41 microcode) or 3K RAM option. Users who have been loading microcode for these functions need no longer do so. This change affects Alto IIs only.

Range Checking

The bounds check instruction (**BNDCK**) is now implemented correctly.

Distribution:
Mesa Users
Mesa Group
SDSupport

Inter-Office Memorandum

To	Mesa Users	Date	October 27, 1980
From	Jim Sandman, John Wick	Location	Palo Alto
Subject	**Mesa 6.0 XMesa Update**	Organization	SDD/SS/Mesa

XEROX

Filed on: [Iris]<Mesa>Doc>XMesa60.bravo (and .press)

This memo describes the changes in Mesa 6.0 runtime support which incorporate the facilities of XMesa 5.0 into the standard system.

Overview of Extended Memory Support

Mesa now uses the extended memory of Alto II XMs as additional swapping space for code. This means that code and data need not co-exist in the MDS, the primary 64K of memory. Mesa takes advantage of any available extra space automatically; standard Alto programs do not need to be modified to run. Support is provided for up to one million words of memory in blocks of 64K words.

Because Mesa uses extended memory for code segments, it includes a page-level storage allocator for the additional banks. Client programs may request storage in the additional banks by using extensions of the standard procedures in **SegmentDefs**. Mesa provides primitive mechanisms to read and write words in extended memory and to copy blocks of data between banks of memory, but gives no other assistance in accessing information in the extended memory. In particular, arbitrary use of **LONG POINTER**s is *not* supported on the Alto.

Public Interfaces

Unless otherwise stated, all of the facilities in this section are defined in **SegmentDefs**.

Configuration Information

The Mesa runtime system has an internal data structure that contains information about the hardware configuration of the machine on which it is running. Clients may obtain a copy of this data structure by calling **GetMemoryConfig** and should normally test for the existence of extended memory by examining the **useXM** field. The extant banks of memory are indicated by **MemoryConfig.banks**, which is a bit mask (*e.g.*, **MemoryConfig.banks=140000B** implies that banks zero and one exist). Note that this bit mask has been expanded to allow for up to sixteen banks; *constants used to test against it must be changed*.

```
BankIndex: TYPE = [0..17B];

ControlStoreType: TYPE = {Ram0, RamandRom, Ram3k, unknown};

MachineType: TYPE = {unknown0, AltoI, AltoII, AltoIIXM, . . . };

MemoryConfig: TYPE = MACHINE DEPENDENT RECORD [
  reserved: [0..37B],
  AltoType: MachineType,
  xmMicroCode: BOOLEAN,
  useXM: BOOLEAN,
  mdsBank: BankIndex,
  controlStore: ControlStoreType,
  banks: [0..177777B],
  mesaMicrocodeVersion: [0..177777B]];

memConfig: PUBILC READONLY MemoryConfig;

GetMemoryConfig: PROCEDURE RETURNS [MemoryConfig] = INLINE
  BEGIN RETURN[memConfig] END;
```

The field **memConfig.useXM** is true if and only if the following conditions hold:

1) the machine is an Alto II with XM modifications (**AltoType = AltoIIXM**),
2) the Alto has more than one memory bank installed (**banks ~= 100000B**),
3) the Alto has a 3K RAM, or it has a second ROM containing an appropriate version of the XMesa microcode.

The microcode version field tells only the *microcode* version, *not* the Mesa release number. (For example, for Mesa 6.0, **mesaMicrocodeVersion** is 41; Mesa 5.0 version 39 microcode is also supported, although not all features are available.)

Extended Memory Management

The facilities described in this section can be used regardless of the state of **useXM**.

Segments in extended memory are created with the usual primitives in **SegmentDefs**. However, additional "default" parameter values for those procedures that expect a VM base page number have been provided. **DefaultMDSBase** requests allocation anywhere in the MDS. **DefaultXMBase** requests allocation anywhere in the extended memory banks but not in the MDS. **DefaultBase0**, **DefaultBase1**, **DefaultBase2** and **DefaultBase3** request allocation in particular banks. **DefaultANYBase** requests allocation anywhere in the extended memory banks or the MDS. **DefaultBase** is equivalent to **DefaultANYBase** *if the segment is a code segment*, otherwise, it is equivalent to **DefaultMDSBase**.

The following procedures convert between segment handles and long pointers, and work for segments anywhere in the 20-bit address space.

```
LongVMtoSegment: PROCEDURE [a: LONG POINTER] RETURNS [SegmentHandle];
```

```
LongSegmentAddress: PROCEDURE [seg: SegmentHandle] RETURNS [LONG POINTER];

LongVMtoDataSegment: PROCEDURE [a: LONG POINTER] RETURNS
    [DataSegmentHandle];

LongDataSegmentAddress: PROCEDURE [seg: DataSegmentHandle]
    RETURNS [LONG POINTER];

LongVMtoFileSegment: PROCEDURE [a: LONG POINTER] RETURNS
    [FileSegmentHandle];

LongFileSegmentAddress: PROCEDURE [seg: FileSegmentHandle]
    RETURNS [LONG POINTER];
```

The following definitions have been added to **AltoDefs**; they define parameters of the extended memory system.

```
MaxVMPage: CARDINAL = 7777B;
MaxMDSPage: CARDINAL = 377B;
PagesPerMDS: CARDINAL = MaxMDSPage+1;
```

The following procedures convert between page numbers and long pointers, and are analogous to **AddressFromPage** and **PageFromAddress**.

```
LongAddressFromPage: PROCEDURE [page: AltoDefs.PageNumber]
    RETURNS [lp: LONG POINTER];

PageFromLongAddress: PROCEDURE [lp: LONG POINTER]
    RETURNS [page: AltoDefs.PageNumber];
```

The following procedures check the validity of long pointers and page numbers and raise the indicated errors.

```
ValidateVMPage: PROCEDURE [page: UNSPECIFIED];

InvalidVMPage: ERROR [page: UNSPECIFIED];

ValidateLongPointer: PROCEDURE [a: LONG UNSPECIFIED];

InvalidLongPointer: ERROR [lp: LONG UNSPECIFIED];
```

The signal **ImmovableSegmentInXM** is raised when **MakeImage** (or **CheckPoint**) discovers a segment in the extended memory banks that cannot be swapped out. (See the section on restrictions, below, for more information about image files).

Long Pointer Support

The facilities described in this section should be used only when **useXM** (see above) is TRUE.

XCOPY is no longer implemented; clients should use **InlineDefs.LongCOPY**. It may only be called when **memConfig.xmMicrocode** is TRUE.

LongCOPY: PROCEDURE **[from:** LONG POINTER, **nwords:** CARDINAL, **to:** LONG POINTER**];**

LongCOPY makes no attempt to validate the long pointers; if they exceed 20 bits or reference non-existent memory, **LongCOPY** will produce unpredictable results.

XBitBlt is no longer implemented; the following extension is not supported by XMesa 5.0 ROMs. The normal AltoIIXM **sourcealt** and **destalt** fields of the BitBlt record (**BitBltDefs.BBTable**) should be used (*do not use the long pointer options*). In addition, if the **unused** word in the **BBTable** is nonzero, the microcode sets the emulator bank register to that value for the duration of the BitBlt. In effect, BitBlt can only be used to move data within a single bank or between the MDS (bank zero) and some other bank.

Restrictions, Limitations, and "Features"

Images and Checkpoints. **MakeImage** cannot preserve the contents of extended memory in the image file it constructs. If **MakeImage** is invoked when **useXM** is TRUE, it will swap out all unlocked file segments in extended memory. (It will also move any locked code segments to the MDS.) If any segments then remain in extended memory, **MakeImage** will refuse to build the image file. Analogous comments apply to **CheckPoint**.

Bank Registers. Mesa assumes it has exclusive control of the emulator bank register on AltoIIXMs. Client programs must not attempt to alter the bank register, but rather must use the public interfaces for moving data to and from extended memory (see **LongCOPY** and **BitBlt**, above).

Segment Alignment. Segments may not cross bank boundaries. The first page of each non-MDS bank is reserved for internal allocation tables.

Swapper Algorithms. The swapper loads a segment into extended memory by first swapping it into primary memory, then copying it to extended memory and releasing the MDS memory space. Thus, if the MDS is so full that the requested segment cannot be swapped in, **InsufficientVM** will be raised, even though sufficient space for the segment may exist in other banks. (Analogous comments apply when swapping out segments that must be written to disk.)

Distribution:
Mesa Users
Mesa Group
SDSupport

Inter-Office Memorandum

To	Mesa Users	Date	October 27, 1980
From	Bruce Malasky, John Wick	Location	Palo Alto
Subject	**Mesa 6.0 Debugger Update**	Organization	SDD/SS/Mesa

XEROX

Filed on: [Iris]<Mesa>Doc>Debugger60.bravo (and .press)

This memo outlines changes made in the Mesa Debugger since the last release (Mesa 5.0, April 9, 1979); it is intended as a concise guide to conversion, not a detailed specification of the changes. Complete documentation on the Mesa 6.0 Debugger can be found in the *Mesa Debugger Documentation*.

User Interface

The Debugger's user interface incorporates changes made in Tajo (the Tools Environment); the window package **Wisk** has been converted to use **Vista**, the new window package. For more complete documentation on the Tajo design, see the *Tajo User's Guide* and the *Tajo Functional Specification.*

Typein

The assignment of some function keys and mouse buttons has changed. The menu button is now YELLOW (formerly BLUE). FL4 is no longer the stuff key; use FR4 (Spare2), Keyset2, or ^S. The following function keys are implemented (see the section on editing for an explanation of the functions):

Function	ADL Keyboard	Microswitch Keyboard	Keyset	Control Key
Cut	DEL	DEL	Keyset5	^C
Paste	LF	LF	Keyset1	^F
Next	FL3	(none)	Keyset3	^N
Replace	FL4	(none)	Keyset4	^R
Swat	FR1	Spare3	(none)	(none)
Stuff	FR4	Spare2	Keyset2	^S
Back Word	BW	Spare1	(none)	^W
Replace/Next	FR5	(none)	(none)	^K

Typein is directed to the Debugger if the cursor is not in any window. Source windows will accept input until a file is loaded; they then direct typein to the Debugger (unless they are editable; see below).

Selections

The selection scheme has changed. Clicking RED once selects a character, clicking twice selects a word, three times a line, etc. The selection can be extended to the left or right with BLUE; a character selection is extended by characters, a word selection by words, and so on. The current selection is now video reversed.

Scrollbars

Scrollbars no longer occupy a dedicated part of the window, but instead come up on top of the left edge. They are twice as wide as before, and you can "see through" them. To obtain a scroll bar, move left just past the edge of the window, then move right slightly, back into the window.

Name Stripe

The name stripe and tiny windows now video reverse when the cursor is in the sections that function as accelerators for the window manager menu commands (`Move`, `Grow`, `Size`, `Top`, `Bottom`, and `Zoom`).

Menus

Except for the change from BLUE to YELLOW, the way menus are invoked has not changed. However, some new menus and commands have been added.

Standard Menus

In addition to `Move`, `Grow`, `Size`, `Top`, `Bottom`, and `Zoom`, the standard window manager menu now also includes the following command:

> `Deactivate`
>
> This command deactivates the selected window; it will no longer appear on the screen and the resources used by it will be freed. The window's name is added to a menu of deactivated windows, which is available outside all windows. The window may be made active again by selecting its menu item.

A new *Text Ops* menu is now supplied with the `Debug.log` and source windows in addition to the Window Manager menu. It contains `Find`, `Position`, `Split`, `Normalize Insertion`, `Normalize Selection`, and `Wrap`; the following commands are new:

> `Split`
>
> The Debugger's wisk window has been replaced by the more general `Split` window command. Feedback is similar to that in Laurel: the split line can be picked up using RED and moved vertically. The subwindow is destroyed by moving the split line off the top or bottom of the (sub)window.
>
> `Normalize Insertion`
>
> For windows containing an insert point (`Debug.log` and editable source windows), this command will position the text in the (sub)window so that the line containing the insert point is at the top.

`Normalize Selection`

This command positions the text in the (sub)window so that the line containing the left most position of the current selection is at the top.

Debugger Menu

A separate *Debugger* menu no longer exists; the `Alter Bitmap` function has been deleted, `Move Boundary` has been superseded by `Split` (see above), and `Stuff It` is now available only on the keyboard.

Source Menus

In addition to the standard menus, the source window has two additional menus, *Source Ops* and *File Ops*. The *Source Ops* menu contains the following commands, which are unchanged: `Create`, `Destroy`, `SetBreak`, `SetTrace`, and `ClearBreak`; the last three commands are available only if a file has been loaded into the window. The *Source Ops* menu contains the following new command:

`Attach`

Causes the Debugger to ignore the creation date of the current source file when setting breakpoints or positioning to a source line. This command is essentially a LOOPHOLE; *because the source-object correspondence may not be correct, it should be used with caution*. If, after using `Attach`, the Debugger sets breakpoints in strange places, chances are that the source file does not match the version of the object in the system you are debugging.

The *File Ops* menu includes the following new commands (plus `Load`, which functions as before):

`Edit`

Enables editing of the currently loaded read only file (see below). Empty windows are always editable, but because they have no backing store (until they are `Save`d or `Store`d on a file), the amount of information in the window should be kept small.

`Save`

Outputs the contents of the window to its current file; overwriting the file requires confirmation. A backup "$" file is created that is a copy of the unedited version. After the `Save` command completes, access reverts to read only.

`Store`

Outputs the contents of the window to the file named by the current selection; if the file already exists, overwriting it requires confirmation. After the `Store` command completes, access reverts to read only.

`Reset`

Discards all edits that have been made to the window (during this session) and resets access to read only. If the file is not editable, the window is made empty.

The `Edit` command is available only if a file has been associated with the window (by a previous `Load`, `Store`, or `Save`); `Store` and `Save` apply only if the window has been edited.

Editing

The standard source window facilities now provide a simple cut and paste editor. Editing is modeless and is accomplished by moving the insert point and typing the desired text. (Note that unlike Bravo, the insert point is independent of the location of the current selection.) Backspace and backword functions (BS and BW) are always available. The following functions are provided:

^RED	Moves the insert point (represented by a blinking caret) to the cursor position.
DEL (Keyset5) ^C	Cut deletes the current selection and puts the deleted text in the TrashBin (see LF).
LF (Keyset1) ^F	Paste inserts the TrashBin at the insert point (see DEL).
FL4 (Keyset4) ^R	Replace does a cut and moves the insert point to the place where the text was deleted.
FR4 (Spare2) ^S	Stuff inserts the current selection at the insert point.

The message `Please terminate editing of` <filename> appears in the `Debug.log` if you try to `Kill` or `Quit` from the Debugger while editing a file.

Caution: The editing facilities are designed not to alter the original file until it is `Saved` or `Stored`, much like Bravo; the original contents are copied to a file with "$" appended to its name. This is however, a new facility and should be used with caution. It is designed to support a moderate number of localized changes to programs, not to replace your favorite document creation system.

Debugger Commands

Changes in Debugger commands are relatively minor. The Debugger's interpreter is more generally available and more consistent with the language. Tracepoints have been re-implemented as a minor extension of the standard breakpoint facilities.

Old Commands

`Ascii/Octal Read`

The `Ascii` and `Octal Read` commands no longer automatically increment the default value produced by ESC.

`Break/Trace Points`

Break and trace points can no longer be set by typing a source line, and the `Break Module` and `Break Procedure` commands and corresponding `Trace` and `Clear` commands have been deleted; the menu commands must be used.

The distinction between trace and breakpoints has been removed. An optional command string can now be attached to each breakpoint which will be executed when the breakpoint is taken. A tracepoint then becomes a breakpoint with a standard default command string. `LIst Breaks` lists both break and tracepoints (`List Traces` has been deleted). `Clear All Entries/Xits` clears both break and tracepoints.

Tracepoints automatically invoke the normal `Display Stack` command processor (with subcommand `p(arameters), v(ariables),` or `r(esults)` as appropriate). The `q(uit)` subcommand (not `b(reak)`) exits to the Debugger's command level, where the normal `Proceed` command continues execution of the client.

The method of specifying conditional break and tracepoints has changed; see the `ATtach Condition` command in the next section.

When an exit break is set, the Debugger breaks on any return of the procedure by setting the actual breakpoint on a common return instruction. The Debugger has no way of telling which return was taken if there is more than one. When asked to display the source line when at an exit break, the Debugger now shows the declaration line of the procedure instead of the last return statement.

`Case On/Off`

The Debugger no longer ignores case, and the case commands have been deleted; *identifiers must be typed with their correct capitalization.*

`Control DEL`

Typing **^DEL** will now abort the display of long arrays and strings, as well as most searches. This key combination no longer has to be held down to be recognized.

`COremap`

This command now prints more information about some data segments; the (system-assigned) types currently recognized are heap, system, frame, table, bitmap, stream buffer, and Pup buffer. Unrecognized types (assigned by the user) are displayed as `data(t)`; an unknown type is displayed as `data(?)`.

`Display Process [`**process**`]`

The subcommand space (**SP**) can now be used to invoke the interpreter.

`Display Stack`

The new subcommand "`g`" displays the global variables of the module containing the current procedure. A space (**SP**) invokes the interpreter. If the source window is loaded with the `s(ource)` subcommand, the window will remember the appropriate context for setting breakpoints.

`Interpret Call`

The `Interpret Call` command has been deleted; the Debugger's interpreter should be used. There are no longer any restrictions on when the interpreter may be called.

`ReSet Context [confirm]`

This command now requires two keystrokes, to avoid conflict with the `ReMote debuggee` command (not yet implemented on the Alto).

`STart [`**address**`] [Confirm]`

This command now requires confirmation.

New Commands

`AScii Display [`**address, count**`]`

Interprets **address** as POINTER TO PACKED ARRAY OF CHARACTER and displays **count** characters (each character separately, not as a string).

`ATtach Condition [`**number, condition**`]`

This command replaces old style conditional breaks; it changes a normal breakpoint into a conditional one. Arguments are a breakpoint number and a condition, which is evaluated in the context of the breakpoint. The breakpoint number is displayed when the break/tracepoint is set, and may also be obtained using the `LIst Breaks` command.

`ATtach Keystrokes [`**number, command**`]`

Arbitrary command strings can now be attached to break and tracepoints; they are executed by the Debugger when the breakpoint is taken. Arguments are a breakpoint number and a command string terminated with a **CR**. A **CR** can be embedded in the command string by quoting it with **^V**.

`ATtach Loadstate [`**filename**`]`

Like `ATtach Image`, except that the initial rather than the current loadstate of the image file is used; this command is for wizards only.

`Break All Entries/Xits [`**module**`]`

This new command is the same as `Trace All Entries/Xits`, except that breakpoints are set.

`CLear Break [`**number**`]`

This command clears breakpoints by number. Typing **CR** in place of a number will clear the current breakpoint, i.e., the one that transferred control into the Debugger.

`CLear Condition [`**number**`]`

This command changes a conditional breakpoint into a normal one. Typing **CR** in place of a number behaves as in `CLear Break`.

`CLear Keystrokes [`**number**`]`

This command clears any command string associated with the breakpoint. Typing **CR** in place of a number behaves as in `CLear Break`.

`LOgin [`**user, password**`]`

This command sets the default user name and password for the debugging session. The new user name and password are not written into the client's core image or onto the disk.

`ReMote Debugee [`**host**`]`

This command is not implemented on the Alto.

`Trace Stack`

This command is used when the Debugger breaks and enters the debugger nub ("`//`" mode); it dumps the Debugger's call stack in octal to the log. Change requests reporting Debugger problems that result in an uncaught signal or other problem should be accompanied by a `Debug.log` which includes the output of this command.

Interpreter

The interpreter provides support for all of the new language features introduced in Mesa 6. All commands requiring numeric input now invoke the interpreter automatically (e.g., `Octal Read: @p, n: SIZE[r]`).

Grammar

A summary of the revised grammar is attached. The constructs ABS, ERROR, LONG, LOOPHOLE, MAX, MIN, NIL, POINTER TO, PROC, PROCEDURE, SIGNAL, WORD, and open and half open intervals have been added to the interpreter's grammar; type REAL has been added for output only. Type expressions following **%** must be enclosed in parentheses. The interpreter syntax **Expression?** has replaced the `Interpret Expression` command; it prints the value of the expression in several formats including octal and decimal.

Target Typing

The interpreter now does a much better job of target typing. As a result, arguments to procedure calls and right hand sides of assignments are type checked. In addition, assignments to enumerated types now work correctly.

The interpreter also does a better job of determining signed/unsigned representation. For example, any octal number is assumed to be unsigned.

Symbol Lookup

Even if a module has compressed symbols, the debugger will first look for the file `modulename.bcd` to see if it is the original compiler output for that module (by checking the version stamp). If so, it will use those symbols. Thus, there is no need to `Attach Symbols` if the proper file is on the disk. It makes sense to use compressed symbols for large systems and to also have present the complete symbol files for the specific modules undergoing detailed debugging.

Output Conventions

In display stack mode, variables declared in nested blocks are now shown indented according to their nesting level.

A "?" in a variable display now uniformly means that the value is out of range; "`. . .`" indicates that there are additional fields present which cannot be displayed due to lack of symbol table information.

When the debugger refers to a program module, it usually gives the address of its global frame, e.g., "`G: nnnnnB`". If the module has not been started, the debugger now prints a "~" after the `B`. If a module has not been started, the user *should not modify* the global variables of that module, nor should they be displayed, as they are uninitialized.

New Error Messages

The warning `Eval stack not empty!` will be printed if the debugger is entered via either an interrupt or a breakpoint with variables still on the evaluation stack; this indicates that the current value of some variables may not be in main memory, where the interpreter normally looks. Exceptions to this are at entry and exit breaks; the debugger has enough information to decode the

argument records that are on the stack in this case (if the appropriate symbol tables are available).

Before the debugger permits any breakpoints to be set using the source window, the creation date in the source file is checked against the corresponding date recorded by the compiler in the BCD. The message `Can't use` <module> `of` <time> `instead of version created` <time> will result if the versions do not match (but see the `Attach` source menu command above).

The message `Resetting symbol table!` is displayed when the interpreter's scratch symbol table overflows; the command is retried automatically. The Debugger's performance decreases somewhat until the symbol table is reinitialized.

If a program is compiled with cross-jumping, the debugger will print the warning `Cross jumped!` before displaying the source.

Installation

Fonts

The Debugger now requires a strike font named `MesaFont.strike` or `SysFont.strike`; a version of Gacha10 is available on `<Mesa>MesaFont.strike`. Additional strike fonts are stored on `[Maxc]<AltoFonts>`. (Strike fonts which include kerning are not supported.)

Switches

Installing the debugger with the `/b` switch will video reverse the display (i.e., white characters on a black background).

Memory Bank Management

When running on machines with more than 64K of memory, the client system supplies space to the Debugger for its bitmap (unless all but one bank has been disabled; see below); the client can disable this option by using the `/k` switch or by calling a system procedure before the Debugger is first invoked (see the *Mesa 6.0 System Update*).

It is also possible for the Debugger to be installed with more than one bank of memory available for code swapping; this is done by reducing the amount of memory available to the client using the RunMesa bank switches or the Alto Executive `MesaBanks.~` command (in Executive version 11 or later).

> `MesaBanks.~`
>
> This command establishes the default memory allocation available to client programs. Arguments can be in two forms: a sixteen bit octal mask (followed by an optional `/b` switch) indicating the *available* banks; a one in bit position *n* of the mask (counting from the left) indicates that bank *n* is available. Form two is a series of decimal bank numbers each followed by the `/x` switch; each bank mentioned is *excluded* from use by the client. Note that a request to exclude bank zero will be ignored. If no argument is present, the command will display the current value of the bank mask.
>
> The `MesaBanks.~` command establishes the available memory for each `.image` or `.bcd` program invoked directly by the Alto Executive. The default may be overridden by explicitly using RunMesa to invoke the program and optionally specifying bank switches on its command line, before the `.image` file name. The bank switches have the same format as the arguments to `MesaBanks.~` (except that the `/b` switch is required in the case of a

bitmask). In the absense of any bank switches, RunMesa always assumes that all banks are available to the client.

Using these facilities, it is possible to set up the defaults so that the Debugger has extra banks of memory at the expense of the client program. For example, on a three bank Alto, the following commands might be used to set the default and then install the Debugger:

```
MesaBanks.~ 2/x
RunMesa.run 1/x XDebug.image
```

Under this arrangement, the client would use banks zero and one, and the Debugger would use banks zero and two (because bank zero is swapped onto `Swatee`, it can be used by both). Actually, because the client (by default) is also allocating space for the Debugger's display bitmap, the client actually has only one-and-one-half banks, and the Debugger has two-and-one-half; this can be changed by running the client with the `/k` switch, resulting in two banks available to each.

Note that the `MesaBanks.~` command affects *all* Mesa programs invoked by the Alto Executive, including the Compiler and Binder. So the above example would run the Compiler in only two banks, not three; this can be changed by saying `RunMesa Compiler` on the command line, which, because there are no bank switches specified, defaults to all banks available (not really necessary in this case, since the Compiler runs almost as well in two banks as in three). On the next new session, the Debugger is smart enough to notice that the Compiler (or whoever) has smashed what it thought was in bank two. (It is also smart enough not to use any memory that the client owns, so that the `1/x` switch on the command line above is actually unnecessary.)

Since there are a lot of options here, some "standard" examples of client and Debugger configurations might be helpful:

Two Banks: Normally, do nothing; client and Debugger will each have one-and-one-half banks. For small clients and better Debugger performance, use `RunMesa 1/x Mesa.image Client.bcd`, which will give the client one bank and the Debugger two. (If you were to use `MesaBanks.~ 1/x` in this case, the Compiler would also be restricted to one bank).

Three Banks: As in the three bank example above.

Four Banks: Use `MesaBanks.~ 3/x` to give the client and the Debugger two-and-one-half banks each and the Compiler three; use `MesaBanks.~ 2/x 3/x` and `Client/k` to increase the Debugger's allocation to three banks and restrict the client to two. Obviously, this can be adjusted based on the size of the client and the desired performance of the Debugger.

Extended Features

Nearly all of **Alto/Tajo** is now included in the Debugger (Librarian support and communications are not). Accordingly, there is little (if any) distinction between UserProcs and Tools, and **Fetch** (the **FileTool** plus communications) which runs in the Debugger is the same as the **FileTool** provided by **Alto/Tajo**. A copy of section 10 of the *Tajo User's Guide* describing the **FileTool** is attached to this memo.

Distribution:
Mesa Users
Mesa Group
SDSupport

Debugger Summary

Version 6.0

AScii
- **R**ead [**address**, **count**]
- **D**isplay [**address**, **count**]

ATtach
- **I**mage [**filename**]
- **C**ondition [**number**, **condition**]
- **K**eystrokes [**number**, **command**]
- **L**oadstate [**filename**]
- **S**ymbols [**globalframe**, **filename**]

Break
- **A**ll
  - **E**ntries [**module/frame**]
  - **X**its [**module/frame**]
- **E**ntry [**procedure**]
- **X**it [**procedure**]

CLear
- **A**ll
  - **B**reaks [confirm]
  - **E**ntries [**module/frame**]
  - **T**races [confirm]
  - **X**its [**module/frame**]
- **B**reak [**number**]
- **C**ondition [**number**]
- **E**ntry
  - **B**reak [**procedure**]
  - **T**race [**procedure**]
- **K**eystrokes [**number**]
- **X**it
- **B**reak [**procedure**]
- **T**race [**procedure**]

COremap [confirm]

CUrrent context

Display
- **B**reak [**number**]
- **C**onfiguration
- **E**val-stack
- **F**rame [**address**] (**g,j,l,n,p,q,r,s,v**)
- **G**lobalFrameTable
- **M**odule [**module**]
- **P**rocess [**process**] (**l,n,p,q,r,s**)
- **Q**ueue [**identifier**] (**l,n,p,q,r,s**)
- **R**eadyList (**l,n,p,q,r,s**)
- **S**tack (**g,j,l,n,p,q,r,s,v**)

Find variable [**identifier**]

Kill session [confirm]

LIst
- **B**reaks [confirm]
- **C**onfigurations [confirm]
- **P**rocesses [confirm]

LOgon [**user**, **password**]

Octal
- **C**lear break [**globalframe**, **bytepc**]
- **R**ead [**address**, **number**]
- **S**et break [**globalframe**, **bytepc**]
- **W**rite [**address**, **value**]

Proceed [confirm]

Quit [confirm]

Re**S**et context [confirm]

Re**M**ote debuggee [**host**] [confirm]

SEt
- **C**onfiguration [**config**]
- **M**odule context [**module/frame**]
- **O**ctal context [**address**]
- **P**rocess context [**process**]
- **R**oot configuration [**config**]

STart [**address**] [confirm]

Trace
- **A**ll
  - **E**ntries [**module/frame**]
  - **X**its [**module/frame**]
- **E**ntry [**procedure**]
- **S**tack
- **X**it [**procedure**]

Userscreen [confirm]

Worry
- off [confirm]
- on [confirm]

^Debug [confirm]

Debugger Interpreter Grammar

Version 6.0

StatementList	**::=**	**Statement | StatementList; | StatementList; Statement**
Statement	**::=**	**LeftSide Interval | LeftSide _ Expression |** MEMORY **Interval | Expression | Expression ?**
LeftSide	**::=**	**identifier | (Expression) | LeftSide Qualifier |** **identifier $ identifier | number $ identifier |** MEMORY **[Expression] |** LOOPHOLE **[Expression] |** LOOPHOLE **[Expression , TypeExpression]**
Qualifier	**::=**	**^ | . identifier | [ExpressionList]**
Interval	**::=**	**[Bounds] | [Bounds) | (Bounds] | (Bounds) |** **[Expression ! Expression]**
Bounds	**::=**	**Expression .. Expression**
Expression	**::=**	**Sum**
Sum	**::=**	**Product | Sum AddOp Product**
AddOp	**::=**	**+ |**
Product	**::=**	**Factor | Product MultOp Factor**
MultOp	**::=**	*** | / |** MOD
Factor	**::=**	**Primary | Primary**
Primary	**::=**	**Literal | LeftSide | @ LeftSide | BuiltinCall |** **Primary % | Primary % (TypeExpression)**
Literal	**::=**	**number | character | string**
BuiltinCall	**::=**	NIL **|** NIL **[TypeExpression] | PrefixOp [ExpressionList] |** **TypeOp [TypeExpression]**
PrefixOp	**::=**	ABS **|** BASE **|** LENGTH **|** LONG **|** MAX **|** MIN
ExpressionList	**::=**	**empty | Expression | ExpressionList, Expression**
TypeOp	**::=**	SIZE
TypeExpression	**::=**	**identifier | TypeIdentifier | TypeConstructor**
TypeIdentifier	**::=**	BOOLEAN **|** INTEGER **|** CARDINAL **|** WORD **|** REAL **|** CHARACTER **|** STRING **|** UNSPECIFIED **|** PROC **|** PROCEDURE **|** SIGNAL **|** ERROR **|** **identifier identifier | identifier TypeIdentifier |** **identifier . identifier | identifier $ identifier**
TypeConstructor	**::=**	LONG **TypeExpression | @ TypeExpression |** POINTER TO **TypeExpression**

Wisk Summary

Version 6.0

WHAT WISK MOUSE BUTTONS DO:

	Scroll Bar	Text Area
RED	Scroll Up	Select
YELLOW	Thumb	Menu
BLUE	Scroll Down	Extend

NAME STRIPE/SMALL WINDOW COMMANDS:

	Left	Middle	Right
RED	Top/Bottom	Zoom	Top/Bottom
YELLOW	Grow (corner)	Grow (edge)	Grow (corner)
BLUE	Move	Size	Move

STANDARD WINDOW MENU COMMANDS:

Move Size Bottom Grow Top Zoom Deactivate

STANDARD TEXT OPS MENU COMMANDS:

Find [selection]	Normalize Insertion	Split
Position [selection]	Normalize Selection	Wrap

SOURCE WINDOW SOURCE OPS MENU COMMANDS:

Create	Set Break [selection]	Clear Break [selection]
Destroy	Set Trace [selection]	Attach

SOURCE WINDOW FILE OPS MENU COMMANDS:

Load [selection]	Store [selection]	Reset
Edit	Save	

R	Y	B	R	Y	B	R	Y	B
T/B	⌜G	⌜M	Zoom	T	Size	T/B	⌜G	⌜M

10.0 File Tool

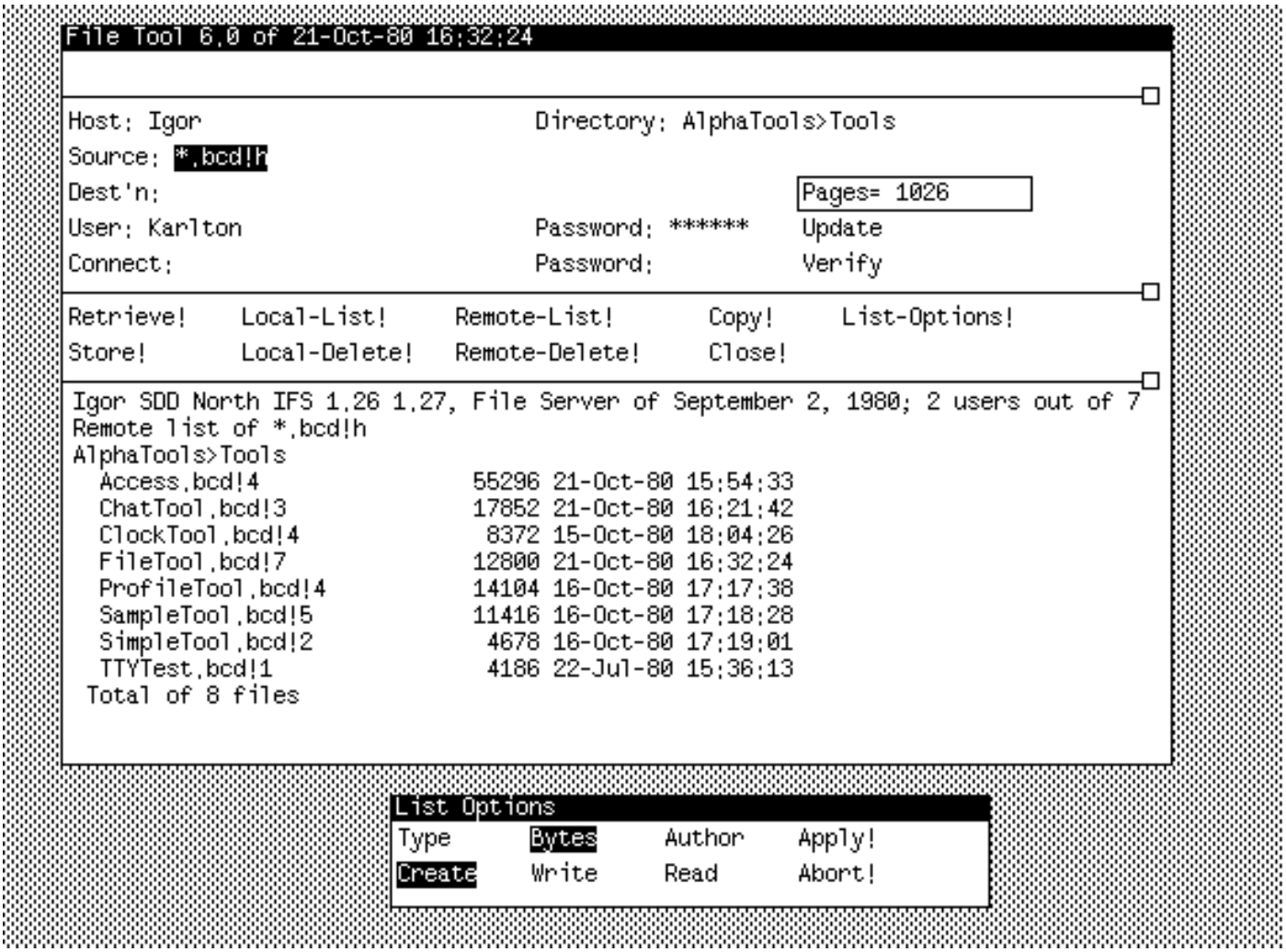
The File Tool provides a means of dealing with local as well as remote file systems from within the Development Environment.

10.1 The User Illusion

The File Tool employs the standard features of the Development Environment. See section 3 for further details.

10.2 Tool Appearance

Below is an illustration of a File Tool with the List Options window (explained below) visible.



10.3 Parameter Subwindow

The upper form subwindow contains parameters that can be set by the user; they will be used by the next File Tool command.

Host: the name of the host to be used for remote file operations. If a connection is already open, any editing of this field causes it to be closed; if a transfer is in progress, the connection will not be closed until it is complete.

Directory: the default remote directory. If empty, the value in the User: field is used.

Source: a list of files (separated by spaces or returns) to be operated on. If the first character of a file name is "@", then the file is taken to be an indirect file and its contents are used as a list of files. Indirect files may nest.

Dest'n: file name for the destination of a transfer. If this field is left blank, the file name is the same as the source.

Pages= number of free pages left on the disk. This item is read only.

User:, Password: the primary directory and the associated password. This field is initialized from the value of the user's last Alto Operating System login. Editing of this field is local to the File Tool and does *not* affect the user's login in the Alto Operating System.

Update only store or retrieve the file if the source is newer than the destination (comparing creation dates). The default is false.

Connect:, Password: the secondary directory and the associated password.

Verify request confirmation for each file operation. The default is false.

10.4 Command Subwindow

File Tool commands are available in the second form subwindow. Some of the commands are accomplished by a background process. Those commands clear the Command subwindow so that a second operation cannot be invoked while one is under way. The Copy! command operates only on the local disk. It does not take a list of files to operate upon. Close! closes a remote connection (if there is one).

It is important to remember that the commands are postfix; e.g., fill in the Host: and Source: fields before invoking the Retrieve! command. The following commands are available:

Retrieve! transfers the file specified in Source: from the remote file system to the local disk. The file name must conform to the file-naming conventions on the remote host. You may designate multiple files by the use of * expansion only to the extent that the remote server supports it. If the local file is already in use, the transfer will not be made and the message "<filename>: can't be modified" will be displayed in both the message window and the log window. See warning in Section 10.6

Local-List! lists all files on the local disk corresponding to the name in Source:. This command will expand *s and #s.

Remote-List! lists all files on the remote file system corresponding to the name in Source:. This must conform to the file naming conventions of the remote host. You may designate multiple files by the use of * expansion only to the extent that the remote server supports it (currently Maxc and IFS do, but differently).

Copy! makes a copy of a local file on the local disk. Only a single file may be copied and *s and #s are not allowed.

List-Options! creates a List Options window if one does not already exist.

Store! transfers the file specified in Source: from the local disk to the remote Host. Alto file name conventions apply to the local file.

Local-Delete! deletes the files specified in Source: from the local disk. If the local file is already in use, the delete will be skipped and the message "<filename>: can't be modified" will be displayed in both the message window and the log window. See warning in Section 10.6

Remote-Delete! deletes the file specified in Source: from the remote file system. You may designate multiple files by the use of * expansion only to the extent that the remote server supports it.

Close! closes the currently open FTP connection.

If Verify is TRUE, then for each file that might be acted upon, the following commands are displayed

Confirm! do the operation.

Deny! abort the operation.

Stop! abort the operation and terminate the command. This will close the connection with the server if a retrieve is being aborted.

10.5 List Options window

The List Options window is created by the List-Options! command. The properties that will be displayed, in addition to the file name, by a Local-List! or Remote-List! are governed by the booleans in this window. After changing the options, use Apply! to effect those changes. The Abort! command will restore the options which existed before the List-Options! command was selected. Choosing either of the commands in the List Options window will cause that window to be removed.

If the Type attribute is requested for a Local-List! and the type is unknown, it will be listed as such to prevent the time it would take to read the file and determine the type.

10.6 Exceptions

The actual transfer takes place in a background process, so the user is free to issue other commands or even change the values in the parameter subwindow without affecting the command currently executing. Changing a parameter while the File Tool is waiting for Confirm! will **not** affect the name of the destination file; you should skip the transfer (by using Deny!) and reissue the command with the desired parameter correctly set.

Warning: Do not attempt to use a file while it is being retrieved. This includes issuing commands to the Debugger that cause it to try to reference the file. For example, Display Stack may cause the Debugger to reference symbols contained in the file being retrieved.

Warning: If you are using the File Tool in the Debugger, be careful not to change any files out from under the program you are debugging; the file tool makes no provisions for checking if the file is in use *in the client world* when you modify a local file.

Inter-Office Memorandum

To	Mesa Users	Date	October 27, 1980
From	Bruce Malasky	Location	Palo Alto
Subject	Debugger: Extended Features	Organization	SDD/SS/Mesa

XEROX

Filed on: [Iris]<Mesa>Doc>XDF.bravo (and .press)

DRAFT

This memo discusses *Debugger User Procedures* (UserProcs) and contains a sample *Printer*, a special type of UserProc.

The Debugger is now the functional equivalent of the Alto/Tajo environment (with the exception of Librarian support and communications). As a result, there are no longer any differences between the **FileTool** and **ChatTool** that run in Alto/Tajo and the versions that run in the Alto/Mesa Debugger.

Loading User Procedures

To install the Debugger from the command line with some UserProcs, type:

```
XDebug YourProc1[/l] YourProc2[/l] ...
```

to the Alto Executive. To load files in an installed debugger, simply enter the Debugger nub; then do a `>New` **filename**, followed by `>Start` **globalframe**. More information on the mechanism for loading programs into the Debugger can be found in the *Mesa User's Handbook* and the *Mesa Debugger Documentation*.

Hints for Writing User Procedures

The Debugger gives you added help in gaining access to the information it already knows about your program. The Debugger's configuration exports all of the Debugger's and Tajo's interfaces; see `XDebug.config` for details. A user program can access any of the Debugger's public procedures simply by importing the definitions modules of the procedures that you want to use. When writing your own debugging routines, look carefully at some of the utility routines that the Debugger already provides (e.g., **Name, Frame**, **ShortREAD**, etc.). In particular, **DebugUsefulDefs** contains most of the interesting procedures you might want. The interface **DOutput** contains utility procedures for displaying information in the `Debug.log` (a la **IODefs**). You should also look at the `<MesaLib>` and `<AlphaHacks>` directories for UserProcs that other Mesa users have already written and debugged.

Warning: *The Mesa Group makes no guarantees about the stability of these interfaces between releases. Use at your own risk!*

Printers

The Debugger is capable of calling a user supplied procedure to print variables of specific types. To do this, a program must first register any type it will display by calling

```
AddPrinter: PROC [type: STRING, proc: PROC [DebugOps.Foo]]
```

from the interface **Dump**. The Debugger's interpreter evaluates **type** at the beginning of each session and remembers the target type of the result. Unfortunately, **type** is not a simple type expression, but rather a statement evaluated by the interpreter; the type is extracted from the result. Any additional information such as the address of a variable used when evaluating the statement is ignored.

Later, whenever the Debugger encounters a variable of that type, it will call **proc** to display it. If, for a given printer, calling **proc** or evaluating **type** ever causes an UNWIND, the printer is never called again. The parameter to **proc** is defined as follows:

```
Foo: TYPE = POINTER TO Fob;

Fob: TYPE = RECORD [
  there: BOOLEAN,
  addr: BitAddress,
  words: CARDINAL,
  bits: [0..WordLength),
  . . .]];

BitAddress: TYPE = RECORD [
  base: LONG POINTER,
  offset: [0..WordLength],
  . . .];
```

If **there** is **TRUE**, the **BitAddress** is a location in the user core image. For large structures, **LongREAD** and **LongCopyREAD** from **DebugUsefulDefs** should be used to access the data; for small structures the procedure **GetValue** in the interface **DI** (it takes a **Foo** as its argument) copies the information into the Debugger's core image and updates the **addr**. The Debugger owns the storage for **Foo**s and the values copied into them from the user's core image; they are freed by the Debugger between commands.

A good technique for debugging the string used in the call to **AddPrinter** is to actually try it out using the interpreter. All REALs could be intercepted by supplying the following STRING to **AddPrinter**:

```
0%(REAL)
```

The following STRING is used by the sample printer attached at the end of this memo.

```
LOOPHOLE[1400B,  StackFormat$Stack]^
```

The constant 1400B is simply a location that is always mapped; **AddPrinter**'s evaluation of the STRING does not actually use that location.

Once **StackPrinter** is instantiated in the Debugger, **PrintStack** is called whenever the Debugger wants to display a **StackObject**. Since **PrintStack** understands the format of **StackObject**s, it can show the complete contents of a **stack**, something the Debugger is unable to do because of the zero length array.

```
-- StackFormat.mesa
-- Last Edited:  Keith, October 21, 1980  10:30 PM

StackFormat: DEFINITIONS =
  BEGIN

  Stack: TYPE = POINTER TO StackObject;

  StackObject: TYPE = RECORD [
    top: CARDINAL _ 0,
    max: CARDINAL _ 0,
    overflowed: BOOLEAN _ FALSE,
    stack: ARRAY [0..0) OF CARDINAL];

  END.


-- StackPrinter.mesa
-- Last Edited:  Keith, October 21, 1980  10:38 PM

DIRECTORY
  DebugOps USING [Foo, LongREAD],
  DI USING [GetValue],
  DOutput USING [Char, Line, Octal, Text],
  Dump USING [AddPrinter],
  StackFormat USING [StackEntry, StackObject];

StackPrinter: PROGRAM IMPORTS DebugOps, DI, DOutput, Dump =
  BEGIN

  PrintRecord: PROC [lp, lps: LONG POINTER TO StackFormat.StackObject] =
{
    lpStack: LONG POINTER TO CARDINAL _ LOOPHOLE[@lps.stack];
    IF lp.top = 0 THEN DOutput.Text["empty "L]
    ELSE
      FOR i: CARDINAL DECREASING IN [0..lp.top) DO
        DOutput.Octal[DebugOps.LongREAD[lpStack + i]]; DOutput.Char[' ];
        ENDLOOP;
    IF lp.overflowed THEN DOutput.Text["(overflow!) "L];
    IF lp.max = lp.top THEN DOutput.Text["(full!)"L];
    DOutput.Line[" "L]};

  PrintStack: PROC [f: DebugOps.Foo] = {
    g: LONG POINTER _ f.addr.base;
    DI.GetValue[f]; PrintRecord[f.addr.base, g]};

  Dump.AddPrinter[
    type: "LOOPHOLE[1400B, StackFormat$Stack]^", proc: PrintStack];

  END.
```

Inter-Office Memorandum

To	Mesa Users	Date	October 27, 1980
From	Brian Lewis, Jim Sandman, Dick Sweet	Location	Palo Alto
Subject	**Mesa 6.0 Utilities Update**	Organization	SDD/SS/Mesa

XEROX

Filed on: [Iris]<Mesa>Doc>Utilities60.bravo (and .press)

This memo outlines the changes made in the utility packages since the last release (Mesa 5.0, April 9, 1979). More complete information can be found in the *Mesa User's Handbook.*

Major changes include some new commands in the Lister and an extensively reworked IncludeChecker. In addition, there is a version of the SignalLister that reads `.bcd` files.

Lister

The lister is now available only as a `.bcd` file. The user interface has been changed slightly: commands that take string parameters no longer need string quotes. The command scanner takes all characters up to the next comma or right bracket as the parameter. Thus

`Code[ListerRoutines]` and `Code["ListerRoutines"]`

are equivalent. The new commands (and "improved" old ones) are listed below. Note that several of the new commands (and some of the old ones) are useful only for internal (Compiler) debugging.

`CodeInConfig[`**config, module**`]`
`OctalCodeInConfig[`**config, module**`]`

> **Config** names a bound configuration; **module** is a module within that configuration. A code listing is produced for the module (see the `Code` and `OctalCode` commands). This is of particular interest for packaged configurations where the code has been rearranged among segments and code packs.

`CompressUsing[`**file**`]`

> The named file should contain a list of BCD file names. The using lists of the directory statement are generated for each module in the list; they are then sorted to show for each interface, and for each item in the interface, which modules reference that item. The same caveat about implicitly included symbols applies as for the `Using` command (see below). The output is written to `file.ul`.

`Hexify[], Octify[]`

> The code lister normally prints addresses (and opcodes for "octal" listings) in base eight. For microcode debugging, base sixteen is sometimes preferable. `Hexify`[] puts the code lister into hexadecimal mode; `Octify[]` reverts to octal mode.

`Implementors[`**file**`]`

The named file should contain a list of compiler output BCDs (interfaces and program modules). This command creates a file, `File.iml`, showing for each interface exported by any program in the list, where the various interface items are implemented. If the list also includes the BCD for a particular interface, the interface items not implemented by any program are also shown. In order to run this command, one needs not only the BCDs in the list, but also the BCDs for the interfaces exported by the programs therein. Missing BCDs are reported and the command attempts to forge on.

`Stamps[`**file**`]`

File names a compiler or binder output BCD. This command generates a file, `File.bl`, that shows the version stamps of any modules bound in the file, and of all imports and exports of the top level configuration in the file.

`UnboundExports[`**file**`]`

File names a compiler or binder output BCD. This command examines all of the exported interfaces and enumerates interface items in those interfaces that are not exported by this module or configuration.

`Using[`**file**`]`

Given a compiler output BCD, this command generates a directory statement with its included identifier lists (on `file.ul`). Since there is not enough information in the BCD to tell which symbols were implicitly included, the USING clauses will contain a superset of those items actually needed. The Mesa 6 Lister does a much better job of weeding out extraneous names in the USING clauses.

`Version[`**file**`]`

File names a compiler or binder output BCD, or an IMAGE file. This command shows, on `Mesa.typescript`, the object, source, and creator version stamps of the file.

`XrefByCallee[`**file**`], XrefByCaller[`**file**`]`

File names a file that contains a list of BCD file names. For each module in the list, a scan is made of the code to find all procedure calls. The <caller, callee> pairs are then sorted by either caller or callee. These commands produce output on `file.xle` and `file.xlr`, respectively.

There are three kinds of procedure calls: local, external, and stack. The program can figure out which procedure is being called for local and external calls. Stack function calls are used for procedure variables (e.g., **stream.get[...]**) and for nested procedure calls. The program ignores nested calls and indicates a callee of `*` for procedure variables.

Include Checker

The most significant differences between this version of the IncludeChecker and the one released with Mesa 5.0 are the following:

1. It handles more files, and requires less processing time for large numbers of files.

2. It executes either from the command line or interactively.

3. It obtains the creation dates for source files from their leader page, rather than from the first few lines of the source text (however, see the description of the new switch `/t` below).

There have been other minor changes to command syntax. The entire section of the *Mesa User's Handbook* on the IncludeChecker is included below.

The IncludeChecker is a program that examines a collection of Mesa source and BCDs for consistency. It produces an output listing that gives a compilation order for the files, and for each BCD, a list of all the BCDs that it includes, and a list of the BCDs which include it. Any inconsistencies (which are described below) are flagged in this listing by an asterisk. As an option, the IncludeChecker will also generate a compilation command on `Line.cm` that can be executed to make the files consistent.

The IncludeChecker determines that an inconsistency exists among the input files if either:

1. A BCD includes another BCD with a version different from the one currently on the disk. This might happen, for example, if the included BCD had been recompiled.

2. A source file is "newer" than the corresponding BCD. This could happen if the source had been edited, or if the source had been retrieved from a remote file server. The IncludeChecker compares the creation date of the source file against the creation date recorded in the BCD of the source file from which the BCD was derived.

The IncludeChecker operates in either command line or interactive mode. To use it in command line mode, type to the Alto executive:

```
>IncludeChecker [outputfile][/switches] [filename1 filename2 ...]
```

where

`outputfile` is the name of the file written. If no extension or switches are given, `.list` is assumed. If no file name is specified, the file `Includes.list` is assumed.

`filename1 filename2 . . .` is the list of file names specifying the source and `.bcd` files to be checked. It is not necessary to give an extension, since the IncludeChecker will look for any `.mesa` or `.bcd` file with the specified name. If no input files are specified, all `.mesa` and `.bcd` files on the disk are examined.

To use the IncludeChecker interactively just type:

```
>IncludeChecker
```

It will then prompt for the output file name and switches, and then a list of the files to check. These are typed one at a time, and the list of file names is terminated by a `CR`. Typing ? `CR` in interactive mode displays a short summary of the IncludeChecker's parameters and use.

Each switch can be preceded by a `-` or `~` to turn it off. The switches are:

`/o` Print a compilation order in the output file (this is the default); `-o` suppresses this listing.

`/i` Print both the includes and included by relationships in the output file (default).

`/t` Obtain the creation date of source files from their leader page (default); `-t` will attempt to get the creation date from the first few lines of the source text.

`/c` Write a consistent compilation command in `Line.cm` (`-c` is the default). In addition, list as comments any BCDs and source files not on the disk which are needed to do the compilation.

`/m` Use multiple output files (`-m` is default). The compilation order is written on `source.outputfile`. The includes and included by relations are written onto `outputfile.includes` and `outputfile.includedBy`, respectively. This switch is useful if the output would otherwise be too large to fit into Bravo.

`/n` Do not compile source files that do not currently have corresponding `.bcd`s on the disk (`-n` is default).

`/p` Place a `/p` after every change of inclusion depth (see below) in the compilation command (`-p` is default). This will cause the Compiler to pause if errors are found while compiling that or any previous module.

`/s` Same as `/c-i-o`. This is used when only a consistent compilation command is needed.

The default switches are `/oit-c-m-n-p-s`.

Note: *The IncludeChecker only checks for consistency of the files that you specify*. Thus, the list of files that you give should include, for example, any important system files upon which your files are dependent.

You should also inspect the compilation command before executing it, since the IncludeChecker's idea of what should be recompiled may not be the same as yours.

If a source file but no BCD is found on the disk, the IncludeChecker outputs a warning on the display; in addition, it adds that file to the compilation command if `/c` and `/-n` are in effect. A warning is also displayed if a BCD is found that was created by an obsolete version of the Compiler; its source file is also added to the compilation command.

The IncludeChecker lists the file names of the compilation order and the consistent compilation command by inclusion depth, with the files that are the most deeply included first. Within that constraint, definitions modules are printed before program modules. In general, then, the "lowest level" definitions modules appear first, while the "highest level" program modules appear last.

As an example of the IncludeChecker's use, the command line

```
>IncludeChecker IC/c IODefs IOPkg LexiconDefs Lexicon
LexiconClient
```

will produce a consistent compilation command in `Line.cm` and the output shown below on `IC.list`.

```
Compilation Order (by inclusion depth):
  LexiconDefs streamdefs stringdefs
  IODefs
  oldstringdefs systemdefs tty windowdefs
  IOPkg Lexicon LexiconClient


IODefs (4-May-80 16:20:37 60#203#) (compilation source: 14-Apr-80 17:37:16)
```

```
includes
  streamdefs
  stringdefs

IOPkg (28-May-80 9:30:01 60#203#) (compilation source: 28-May-80 9:08:43)
    (source on disk: [same]) includes
  IODefs (4-May-80 16:20:37 60#203#)
  oldstringdefs
  streamdefs
  tty
  windowdefs

Lexicon (28-May-80 9:30:29 60#203#) (compilation source: 28-Apr-80
17:02:20)
    (source on disk: [same]) includes
  IODefs (4-May-80 16:20:37 60#203#)
  LexiconDefs (14-May-80 10:48:49 60#205#)
  oldstringdefs
  systemdefs

LexiconClient (28-May-80 10:02:50 60#203#) (compilation source: 28-May-80
10:02:14)
    (source on disk: [same]) includes
  IODefs (4-May-80 16:20:37 60#203#)
  LexiconDefs (14-May-80 10:48:49 60#205#)
  oldstringdefs

LexiconDefs (14-May-80 10:48:49 60#205#) (compilation source: 18-Apr-79
19:19:11)
    (source on disk: [same]) includes nothing


IODefs is included by
  IOPkg                 Lexicon
  LexiconClient

IOPkg is included by nothing

Lexicon is included by nothing

LexiconClient is included by nothing

LexiconDefs is included by
  Lexicon               LexiconClient
```

BcdSignals

BcdSignals is an Alto/Mesa program which will produce a signal listing from a `.bcd` file; it works much like the Alto/Mesa SignalLister for listing the signals in a `.image` file (see the *Mesa User's Handbook*). To produce the signal listing `Foo.signals` from `Foo.bcd`, type to the Alto Executive:

```
>BcdSignals [octalNumber/switch] Foo[/switches]
```

where

/n	takes `octalNumber` to be the global frame index of the first frame in this BCD. This will normally be the first free global frame index in the system into which the BCD will be loaded.
/x	takes `octalNumber` to be the StartPilot loadmap form of a global frame index. This is the number in brackets beside the module name in the loadmap. It should be 200B times the octal number used with the /n switch.
/p	lists the name, byte PC, and length of each procedure in `Foo` on `Foo.procs`.
/s	list the signals of `Foo` on `Foo.signals` (default).

As usual, a `-` or ~ can be used to invert the sense of the `/s` switch.

Distribution:
Mesa Users
Mesa Group
SDSupport

Inter-Office Memorandum

To	Mesa Users	Date	October 27, 1980
From	Jim Sandman, John Wick	Location	Palo Alto
Subject	**Integrated Mesa Environment**	Organization	SDD/SS/Mesa

XEROX

Filed on: [Iris]<Mesa>Doc>CommandCentral.bravo (and .press)

This memo documents a small executive called Command Central; this Tool is intended to be installed with the Debugger and can be used to invoke the Compiler, the Binder, and client programs, all of which upon completion are directed to return to Command Central rather than to the Alto Executive. The idea is that, while programming in Mesa, you enter Command Central's control only once, and you rarely have to leave it; this is made possible by the editor that is now included in the Debugger, as well as by the context switching facilities provided by Command Central.

Installation

To include Command Central in the Debugger, type the following Alto Executive command when installing, after retrieving `<Mesa>Fetch.bcd` (which contains **TinyPup**, **Stps**, and the **FileTool**) and `<Mesa>Utilities>CommandCentral.bcd`. (If you have more than 64K of memory, be sure to consult the Installation section of the Debugger documentation before proceeding.)

```
>XDebug Fetch/l CommandCentral/l
```

While it is possible to use Command Central without also installing the **FileTool**, including it will help minimize the number of times you have to leave the Mesa environment. If you have enough memory on your machine, you might consider installing other Tools with your Debugger as well (e.g., **ChatTool**, **SendMessageTool**).

Note: Tools loaded via the command line are initially inactive (i.e., no window is showing); move the cursor into the gray area outside all windows and use the menu found there to activate them.

Entering Command Central

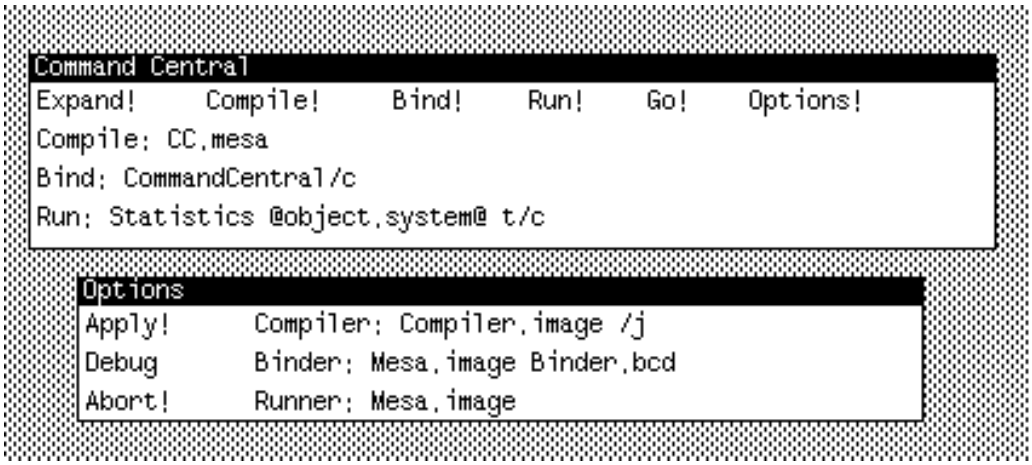
When using Command Central, the Debugger becomes the executive from which all programs are invoked. To first enter this environment, type

```
>Mesa/d
```

to the Alto Executive. You can now use the **FileTool** to retrieve the modules you wish to work on and the Tools editor to modify them. When you have finished your changes, turn your attention to the Command Central window.

Command Central Window

This window provides command lines for compiling, binding, and running your program, the context switching commands, and an option sheet; it also supports the standard window operations (scrolling, growing, etc.).



The three fields contain command lines for the Compiler, the Binder, and the System; their contents are written to `Com.cm` when the commands are invoked. (A special global switch /q is added so that control is returned to the Debugger rather than to the Alto Executive.)

The `Compile!`, `Bind!`, and `Run!` commands invoke the appropriate programs using the command lines constucted from the `Compile:`, `Bind:`, and `Run:` fields, respectively. The `Run!` command can be used to invoke `.bcd`, `.image`, and `.run` files (see below). The `Go!` command constructs a combined command line using all non-null fields and executes the appropriate programs in order. Note that the Debugger will complain if you issue any of these commands while a file is being edited (since the edits might be lost as a result of executing them).

The parameter fields also recognize command files preceeded by the traditional at-sign (e.g., `@file.cm`); the `Expand!` command will expand all such references into their contents and write the result back into the window. Indirect references are also automatically expanded when any of the other commands are invoked.

The `Options!` command produces the options window which allows you to specify the names of the compiler, binder and system you are using. The names are parsed to allow default switches to be included. The `Apply!` command will save the new names and the `Abort!` command will restore the names to their previous state (the default names are shown above). Both `Apply!` and `Abort!` will remove the options window.

If the Compiler or the Binder detect errors (and the pause switch is in effect), they will invoke the Debugger with an appropriate message instead of pausing. You can then load the appropriate error log into a window and step through it and your source file together. Because the file index of the error is included in each message, the `position` menu command can be used to find the source of the error quickly.

Invoking Other Programs

Any Mesa `.bcd` which expects to be loaded into `Mesa.image` and obtains its commands from the command line (`Com.cm`) can be invoked by Command Central using the `Run:` field and the `Run!` command. (As above, the global /q switch is added to the command line so that control will return to the Debugger.) Some obvious programs which you might include on your disk are **Access** and **Print**.

You can also run arbitrary `.image` and `.run` files using Command Central, but unless they have made provision to return control to the Debugger, they will exit to the Alto Executive upon completion. Use the `Mesa/d` command to reenter Command Central.

Limitations

If you use the `Compile!`, `Bind!`, `Run!`, or `Go!` commands when you are in the middle of a debugging session (at a breakpoint or an uncaught signal, for example), the state of the client will be lost. In particular, normal termination processing of the client will not take place (e.g., open files will be left dangling).

Distribution:
- Mesa Users
- Mesa Group
- SDSupport

Inter-Office Memorandum

To	Mesa Users	Date	October 27, 1980
From	Hal Murray, Mark Sapsford	Location	Palo Alto
Subject	**Mesa 6.0 Pup and Ftp Update**	Organization	SDD/SS/Mesa

XEROX

Filed on: [Iris]<Mesa>Doc>PupFtp60.bravo (and .press)

This memo outlines changes made in the Mesa Pup and Ftp Packages since the University release of December 20, 1979. This release is essentially a recompilation of the Pup and Ftp packages using Mesa 6.0. As usual, a number of bugs have also been fixed, and minor changes have been made to the interfaces. More complete information is available in the functional specifications stored on `<Mesa>Doc>` as `PupPackage.press` and `FtpPackage.press`.

Pup Summary

Changes

The following changes were made to **Stream**. The procedure types **GetProcedure**, **PutProcedure**, **SetSSTProcedure**, **SendAttentionProcedure**, and **WaitAttention-Procedure** now take a new initial argument **sh: Handle**. In addition, **SendAttention-Procedure** now takes a second argument **byte: Byte** (which is ignored), and **WaitAttentionProcedure** now returns a **Byte** (which should be ignored). As a result of these type redefinitions, the inline procedures **SendAttention** and **WaitAttention** have different calling sequences, and a new return value has been added to **WaitAttention**.

The following changes were made to **PupDefs**. **PupRouterSendThis** no longer returns a **SendReturnCode**. **SendReturnCode** (a TYPE) has been deleted. If the buffer could not be sent, it is discarded.

The arguments to **EnumeratePupAddresses** are now named. The order of the two arguments to **PupAddressLookup** has been reversed. The second argument to **AppendPupAddress** is now a **PupAddress**, instead of a POINTER TO **PupAddress**. Similarly, the argument to **Print-PupAddress** is now a **PupAddress**, instead of a POINTER TO **PupAddress**. The **local: PupSocketID** argument to **PupPktStreamCreate** and **PupByteStreamCreate** has been removed.

Additions

The procedures **AppendHostName** and **AppendMyName** have been added to **PupDefs**.

UseAltoChecksumMicrocode has been added to speed up processing if you are running on an Alto with XMesa in the ROM or an Alto with a 3K RAM. The overflow microcode loaded into the RAM by `RunMesa.run` includes the necessary additions. (Beware if you load your own microcode.)

Subtle Implementation Changes

GetPupAddress will no longer return an address for a dying net. **EnumeratePupAddresses** will now pass the client supplied procedure addresses on dead or dying nets, but only after processing all the addresses on nets that are reachable. (It used to skip addresses on unreachable nets.)

The byte stream internals have been reworked to eliminate several unpleasant delays while opening and closing connections. It is now possible to open a connection, send a thousand words, and close the connection in less than a second. (Since closing deletes three module instances, it will take longer if there are many active global frames.) A byproduct of this cleanup is that **SendNow** will send an empty **aData** packet to request an acknowledgment even if the previous **SendBlock** happened to end on a convenient packet boundary.

When sending a packet whose destination is the local machine, the Ethernet driver puts a copy on the input queue and acts as though it had sent the packet. It now also copies broadcast packets to the input queue, so clients should check to be sure that their programs will not take undesired actions if they hear their own broadcasts. Packets that are sent to the local machine are now also sent out over the wire; this allows PeekPup.run to be used when analyzing timing problems.

Bug Fixes

The following change requests are closed by this release:

2848 **GetMyName** (actually net address) procedure
2906 Slow on Dorado
2998 Error strings in **NameConversion**
3311 Pup ByteStream close
3341 Trivial bug in **PupTypes**
3708 **PupNameLookup**+**PupAddressLookup PupGlitch**
4456 Recompile packages to fix long return record bug
4552 ByteStream timeout
5005 Delays when creating byte stream
5093 NameLookup vs dying nets
5098 Change priority of interrupt routine in EthernetDriver(s)

Ftp Summary

The arguments to the (client supplied) procedure passed to **FTPInventoryDumpFile** have been extended to allow proper processing of create dates. It is now compatable with the procedure passed to **FTPEnumerateFiles**.

The following change requests are closed by this release:

2906 Slow on Dorado
3584 Ftp vs IFS 1.23
3626 Sending mail
3664 MTP user: require sender property on **SendMessage**
3900 UNWIND from **FTPEnumerate**/retrieve
3987 **StringBoundsFault** from **TimeExtras.PacketTimeFromString**
4335 **FTPUtilities.TransferBytes**
4352 **FTPAltoFile.PreProcessFile** does a blind **ReleaseFile**
4444 **FTPInventorryDumpFile** needs create date
4456 Recompile packages to fix long return record bug

4499 **FTPTransferFile** doesn’t pass through the creation date
4544 **FTPRetrieve** hangs on a timeout on a "no" mark
4763 Troubles if forget to call **IdentifyNextRejectedRecipient**
5152 **TimeExtras.PacketTimeFromString** zone screwup
5198 Config with server and user things

Distribution:
Mesa Users
Mesa Group
SDSupport