Inter-Office Memorandum

To	Mesa Users	Date	May 31, 1978
From	Dave Redell, John Wick	Location	Palo Alto
Subject	Mesa 4.0 Change Summary	Organization	SDD/SD

XEROX

Filed on: [IRIS]<MESA>DOC>SUMMARY40.BRAVO

This memo outlines changes made in Mesa since the last release (October 17, 1977).

References

The following documents can be found on [IRIS]<MESA>DOC>; all files are in Bravo format. Hardcopy is available through your support group; in addition, the PRESS files MESA40A, MESA40B, and MESA40C are a compilation of this material (about 75 pages).

Mesa 4.0 Change Summary. SUMMARY40.BRAVO

Mesa 4.0 Compiler Update. COMPILER40.BRAVO, ARITHMETIC40.BRAVO

Mesa 4.0 Process Update. PROCESS40.BRAVO

Mesa 4.0 Binder Update. BINDER40.BRAVO

Mesa 4.0 System Update. SYSTEM40.BRAVO

Mesa 4.0 Microcode Update. MICROCODE40.BRAVO

Mesa 4.0 Debugger Update. DEBUGGER40.BRAVO

The section on processes is a preliminary draft of a new chapter of the *Mesa Language Manual* (which will be sent to the printer shortly); thanks are due to Dave Redell and the *Pilot Functional Specification* for contributing much of this material.

The MESA>DOC directory also includes new versions of the *Mesa System Documentation* and the *Mesa Debugger Documentation* (the relevant PRESS files are SYSTEM1, SYSTEM2, and DEBUGGER).

Highlights

The primary emphasis in this release has been on three areas: implementation of features required by Pilot and Dstar applications for effective use of the new machine architecture (processes, monitors, long pointers, etc.), reduction of overhead in the basic system structures and improved performance of the Mesa runtime environment (faster microcode, smaller global frames, more efficient memory management), and extension of the debugger's capabilities (primarily an interpreter for a subset of the Mesa language).

The primary impact of Mesa 4.0 on existing systems is in the area of concurrent programming. A brief intoduction to the new process mechanism appears below. It is intended to present enough information to enable programmers to experiment with the new features of the language and the runtime system. However, before attempting to revise or redesign existing systems to use these facilities, programmers are urged to carefully examine the material in the *Mesa 4.0 Process Update* and the *Mesa System Documentation.*

Warning: Because Pilot will be available soon, the Alto/Mesa operating system software has not been revised and redesigned to fully exploit the capabilities of the new process mechanism. In particular, arbitrary preemptive processes are not supported, and the restrictions of Mesa 3.0 on processes running at interrupt level still apply.

A Brief Introduction to Processes in Mesa

Mesa 4.0 introduces three new facilities for concurrent programming:

Processes, which provide the basic framework for concurrent programming.

Monitors, which provide the fundamental interprocess synchronization facility.

Condition variables, which build upon monitors to provide more flexible forms of interprocess synchronization.

As compared with the mechanisms provided in earlier releases of Mesa, the new concurrency facilities are more extensive, and are much more thoroughly integrated into the language. The purpose of the new facilities is to allow easy use of concurrency as a basic control structure in Mesa programs. Concurrency can be an important consideration in progam design, especially when input/output or user interactions may cause unpredictable delays.

Processes

For example, consider an application with a front-end routine providing interactive composition and editing of input lines:

```
ReadLine: PROCEDURE [s: STRING] RETURNS [CARDINAL] =
  BEGIN
  c: CHARACTER;
  s.length ← 0;
  DO
    c ← ReadChar[];
    IF ControlCharacter[c] THEN DoAction[c]
    ELSE AppendChar[s, c];
    IF c = CR THEN RETURN[s.length];
    ENDLOOP;
  END;
```

Thus, the call:

```
n ← ReadLine[s];
```

would collect a line of user typing up to a CR and return it to the caller. Of course, the caller cannot get anything else accomplished during the type-in of the line. If there was anything else that needed doing, it could be done concurrently with the type-in by *forking* to ReadLine instead of calling it:

```
p ← FORK ReadLine[s];
. . .
<concurrent computation>
. . .
n ← JOIN p;
```

This would allow the statements labeled <concurrent computation> to proceed in parallel with user typing. The FORK statement spawns a new process whose result type matches that of ReadLine. (ReadLine is referred to as the "root procedure" of the new process.)

```
p: PROCESS RETURNS [CARDINAL];
```

Later, the results are retrieved by the JOIN statement, which also deletes the spawned process. Obviously, this must not occur until both processes are ready (i.e. have reached the JOIN and the RETURN, respectively); this rendevous is synchronized automatically by the process facility.

Note that the types of the arguments and results of ReadLine are *always* checked at compile time, whether it is called or forked.

Monitors

Further investigation of ReadLine reveals another example of interprocess interaction; the ReadChar routine it uses inspects an input character buffer, which is filled by an independent dedicated keyboard process. (Such dedicated processes replace the "hard processes" of earlier releases of Mesa.) To avoid conflict over the buffer, appropriate synchronization is needed. A *monitor* can be used to insure that neither process will ever access the buffer while the other has it in a "bad state" (e.g. inconsistent pointers, etc.). The keyboard monitor might look like:

```
Keyboard: MONITOR =
BEGIN
buffer: STRING;
ReadChar: PUBLIC ENTRY PROCEDURE RETURNS [c: CHARACTER] =
  BEGIN
  c ← <get character from buffer>
  END;
PutChar: PUBLIC ENTRY PROCEDURE [c: CHARACTER] =
  BEGIN
  <put c in buffer>
  END;
END.
```

The keyword MONITOR confers upon the Keyboard module some special properties. The most fundamental is the presence of *entry* procedures, identified by the keyword ENTRY. These procedures have the property that calls on them are *mutually exclusive*; that is, a new call cannot commence while any previous call is in progress. In effect, the monitor module is made temporarily private to a single process, and any other processes wishing to use it are delayed until the first process is finished. In this example, the client's call to ReadChar and the keyboard process' call to PutChar are guaranteed mutually exclusive access to the buffer.

Condition variables

As long as it finds some characters in the buffer, ReadChar as shown above will work correctly without conflict over the buffer. If it finds the buffer empty, however, it cannot simply loop in the monitor waiting for a character to arrive; not only would this be inefficient, but it would lock out the keyboard process from ever delivering the desired next character! What is needed is some way for ReadChar to pause and release the mutual exclusion temporarily until PutChar has delivered the next character. This facility is provided by *condition variables*. Condition variables serve as the basic building blocks out of which the programmer can fashion whatever generalized synchronization machinery proves necessary in a given situation. For example, the Keyboard monitor can be modified to use the WAIT and NOTIFY operations on condition variables as follows:

```
Keyboard: MONITOR =
BEGIN
buffer: STRING;
bufferNonEmpty: CONDITION;
ReadChar: PUBLIC ENTRY PROCEDURE RETURNS [c: CHARACTER] =
  BEGIN
  WHILE <buffer empty> DO
    WAIT bufferNonEmpty
    ENDLOOP;
  c ← <get character from buffer>
  END;
PutChar: PUBLIC ENTRY PROCEDURE [c: CHARACTER] =
  BEGIN
  <put c in buffer>
  NOTIFY bufferNonEmpty;
  END;
END.
```

Note that the WAIT statement is embedded in a WHILE-loop which repeatedly tests for the desired condition. *This is the only recommended usage pattern for the* WAIT *statement.* In particular, it would have been incorrect to replace the loop above by:

```
IF <buffer empty> THEN WAIT bufferNonEmpty;
c ← <get character from buffer>
```

This rule exemplifies a fundamental property of condition variables in Mesa: a condition variable always corresponds to some Boolean expression describing a desired state of the monitor data, and suggests that any interested process(es) might do well to reevaluate it. *It does not guarantee that the Boolean expression has become true*, hence programmers should *never* write programs (such as the fragment above) that implicitly assume the truth of the desired condition upon awakening from a WAIT.

Priorities

The set of existing processes grows and shrinks dynamically as FORKs and JOINs occur. At any given time, some of the processes are *ready* and compete for use of the processor. The choice of which one to run is done on the basis of priority. A process starts life with the priority of its parent (who executed the FORK), and may change its own priority by calling SetPriority.

CAUTION: *Use of multiple priorities in the Alto/Mesa implementation is severely restricted.* Any process running at other than the default priority (currently, 1) is forbidden to use many of the standard runtime support features of the Mesa environment. In practice, this means that non-standard priorities should be used only for interrupt handling, while all "normal" processing takes place concurrently at the default priority level.

More general features

More complex situations will sometimes require more flexible use of the concurrency facilities. Such use involves more complicated rules and syntactic constructs, which are described in the *Mesa 4.0 Process Update.*

Distribution:
 Mesa Users
 Mesa Group

Inter-Office Memorandum

To	Mesa Users	Date	May 31, 1978
From	Ed Satterthwaite	Location	Palo Alto
Subject	Mesa 4.0 Compiler Update	Organization	SDD/SD

XEROX

Filed on: [IRIS]<MESA>DOC>COMPILER40.BRAVO

This memo describes changes to the Mesa language and compiler that have been made since the last release (October 17, 1977). As usual, the list of compiler-related change requests closed by Mesa 4.0 will appear separately as part of the Software Release Description.

The language accepted by the Mesa 4.0 compiler has several significant extensions and a few minor changes. It features a process mechanism, enhanced arithmetic capabilities, long and base-relative pointers, and more general block structure.

Because of changes in symbol table and BCD formats, all existing Mesa programs must be recompiled. There are minor incompatibilities with Mesa 3.0 at the source level in the areas of signed/unsigned arithmetic and the scope of OPEN in an iterative statement. These incompatibilities should have negligible impact on existing programs. The syntax and semantics of declaring (but not calling) machine-coded procedures have changed substantially.

Page and section numbers in this update not otherwise qualified refer to the *Mesa Language Manual, Version 3.0.* The BNF descriptions of new or revised syntax follow the conventions introduced in that manual. For phrase classes used but not redefined here, see its Appendix D. Revisions of phrase class definitions are cumulative; except as noted, the appearance of "..." as an alternative indicates that an existing definition is being augmented. A definition without "..." supersedes any definition of the same phrase class in the manual.

Arithmetic

Mesa 4.0 supports double-precision integer arithmetic (type LONG INTEGER) and provides some help with floating-point computations (type REAL). In conjunction with these changes, the rules governing combination of signed and unsigned values have been more carefully defined (see the Appendix to this memo).

Syntax

```
PredefinedType ::=   INTEGER | CARDINAL | LONG INTEGER | REAL |
                     BOOLEAN | CHARACTER | STRING | UNSPECIFIED | WORD

Primary   ::=  ... | identifier [ Expression ] | LONG [ Expression ]
```

Signed and Unsigned Arithmetic

The rules governing the use of signed and unsigned representations in single-precision arithmetic have been reformulated. In previous versions of Mesa, conditions under which an operation was considered to overflow were not well defined. As a consequence, options such as overflow detection and reliable range checking were precluded. Mesa 4.0 does *not* offer these options, but it does remedy the defects in the language definition.

The precise rules governing signed/unsigned arithmetic are somewhat lengthy. They appear in an appendix to this memo with some background information explaining the motivation and philosophy. In their effect on the acceptance or rejection of source text, the new rules differ little from those in previous versions of Mesa; the main change is that CARDINAL - CARDINAL is now assumed to produce a result with unsigned (instead of unknown) representation (see Section 2.5.1, pages 10-12). Thus the immediate practical effect of the new rules is minor; however, programmers should read the appendix carefully so that their code will work correctly even when it becomes possible to request overflow and range checks.

The effects of the new rules with respect to subtraction are worth emphasizing. If both operands have valid signed representations, the result is an INTEGER. If both have only unsigned representations, the result is a CARDINAL and is considered to overflow if the first operand is less than the second.

```
i: INTEGER;   m, n: CARDINAL;   s, t: [0..10);
i ← m-n;                    -- should be used only if it is known that m >= n
i ← IF m >=n THEN m-n ELSE -(n-m);        -- should be used otherwise
IF m-n > 0 ...              -- comparison (and subtraction) are unsigned
IF m > n ...                -- a better and safer test
IF s-t < 0 ...              -- comparison (and subtraction) are signed
```

Range Assertions

The new rules mentioned above assume that there are implicit conversion functions mapping CARDINAL to INTEGER and vice-versa. In both directions, the "conversion" amounts to an assertion that the argument is an element of INTEGER ∩ CARDINAL. The programmer can make such a *range assertion* explicit. If *S* is an identifier of a subrange type and *e* is an expression with compatible type *T*, the form *S*[*e*] has the same value as *e* and is additionally an assertion that *e* IN [FIRST[*S*∩*T*] .. LAST[*S*∩*T*]] is TRUE.

Note that this is not equivalent to LOOPHOLE[*e*, *S*] but is an assertion about the range of a value that already has an appropriate type.

In Mesa 4.0, *such assertions must be verified by the programmer.* There is *not* an option to generate code that checks these assertions, whether implicit or explicit. An assertion can be used to control the assumed representation of a subexpression; otherwise, it is currently treated as a comment by the compiler.

Examples

INTEGER[*n*], *IndexType*[*i-j*]

Long Integers

Mesa 4.0 supports double-precision integers. There is a new predeclared type LONG INTEGER, values of which occupy two words (32 bits) of storage and range over $[-2^{31} .. 2^{31})$. There is no special denotation for LONG INTEGER constants. The type of any decimal or octal constant in $[2^{16} .. 2^{31})$ is LONG INTEGER; smaller constants are converted as required by context. The arithmetic operators +, -, *, /, MOD, MIN, MAX, (unary) - and ABS have double-precision extensions that perform the mapping

(LONG INTEGER)^n → LONG INTEGER;

furthermore, LONG INTEGERs are ordered, and the relational operators =, #, <, <=, >, >=and IN have extensions that perform the mapping

(LONG INTEGER)^n → BOOLEAN.

Some fine points:

> All LONG INTEGERs have a signed representation; the Mesa 4.0 language does not provide LONG CARDINAL.
>
> Addition, subtraction, and comparison of LONG INTEGERs is fast; multiplication and division are done by software and are relatively slow.
>
> In Mesa 4.0, it is not possible to declare a type that is a subrange of LONG INTEGER.

Mesa provides an automatic coercion from any single-precision numeric type (INTEGER, CARDINAL, etc.) to LONG INTEGER. This coercion is called *widening* and is discussed in more detail below. It is applied when necessary to match inherent and target types (e.g., in assignments). Also, if any operand of an arithmetic or relational operator is a LONG INTEGER, the double-precision operation is used. In most cases, widening of any shorter operands is automatic. Thus single- and double-precision quantities can be mixed freely within expressions to yield double-precision results.

The form LONG[*e*] explicitly forces the widening of any expression *e* with a single-precision numeric type. There are no automatic conversions from LONG INTEGER to any single-precision type (but see the *Mesa 4.0 System Documentation* for some standard procedures).

> Widening of a single-precision constant is done at compile-time. Currently, no other arithmetic or relational operations on LONG INTEGERs are performed at compile-time, even if all operands are constant.
>
> Widening of a single-precision expression is substantially more efficient if that expression has an unsigned representation.

Examples

```
i: INTEGER;
ii: LONG INTEGER;
c2: LONG INTEGER =2;                -- a compile-time constant
c4: LONG INTEGER =c2*c2;            -- not a compile-time constant

ii ← 0; ii ← ii+1; ii ← i; ii ← (ii+i)/c2;     -- all valid

ii ← LONG[0]; ii ← (ii+LONG[i])/c2;            -- also valid (and explicit)

i ← ii; ii ← LONG[c4];                         -- invalid
```

Reals

A standard representation for floating-point values has not yet been chosen. Mesa 4.0 nevertheless provides some help with floating-point computation. It allows declaration and assignment of REAL values; furthermore, REAL expressions constructed using the standard infix operators (except MOD) are converted to sequences of procedure calls by the compiler.

A REAL value is assumed to occupy two words (32 bits) of storage. Beyond this, no assumptions are made about the representation of REALS. Users of real arithmetic must provide and install an appropriate set of procedures for performing the arithmetic operations (see the *Mesa 4.0 System Documentation* also). The procedures must be assignable to variables declared as follows:

> *FADD, FSUB, FMUL, FDIV:* PROCEDURE [REAL, REAL] RETURNS [REAL];
>
> *FCOMP:* PROCEDURE [REAL, REAL] RETURNS [INTEGER];
> -- returns a value that is: 0 if equal, negative if the first is less, positive otherwise
>
> *FLOAT:* PROCEDURE [LONG INTEGER] RETURNS [REAL];

This scheme has the following consequences:

> All other arithmetic operations (ABS, MIN, etc.) are fabricated from these primitives.
>
> The source language provides no denotation for real constants, since the compiler does not know the internal format expected by the user-supplied procedures. As discussed below, values of type INTEGER or LONG INTEGER are automatically converted to type REAL at run-time; thus integer constants can appear in real expressions but will be reconverted each time the expression is evaluated.

Of course, implementers of floating-point packages are free to provide their own procedures for constructing REAL values from, e.g., octal constants, but a REAL "constant" currently cannot be a compile-time constant and cannot appear in a DEFINITIONS module (unless it is defined using a LOOPHOLE).

Examples

Two: REAL =2;	-- means *Two:* REAL = *FLOAT*[2];
Half: REAL =1/*Two*;	-- means *Half:* REAL = *FLOAT*[1]/*Two*;
Bug: REAL =1/2;	-- means *Bug:* REAL = *FLOAT*[0]; (integer division)

Implicit Conversions

Conversions from INTEGER or CARDINAL to LONG INTEGER and from LONG INTEGER to REAL are called *widening*. Widening is automatic in the following situations:

> An expression will be widened from its inherent type to match its target type (see Section 3.5, pages 37-39). This occurs in assignments and assignment-like contexts (such as record construction or extraction).
>
> The types of the operands of an arithmetic operator will be balanced by widening until all match the type of the widest operand (but not further, even if the target type is wider).

In Mesa 4.0, automatic widening is not completely implemented in the following situations:

> Operands of MIN and MAX will be widened to match the target type if one is well defined and otherwise to match the type of the first operand, but there is no general balancing.

The endpoints in the right operand of IN will be widened to match the type of the left operand, but there is no general balancing.

Expressions appearing in the arms of conditionals will be widened as required by the target type, but there is no general balancing when the target type is ill-defined.

The expressions selecting the arms of a **SelectExpr** or **SelectStmt** will be widened to match the type of the selector, but the selector itself is never widened.

The following examples illustrate widening.

```
i, j: INTEGER;   ii: LONG INTEGER;   x: REAL;
ii ← i;   x ← i;   x ← ii;   x ← IF i < j THEN i ELSE ii
i +ii, ii +1             -- added as LONG INTEGERS (for any target type)
i +x, x +1, ii +x        -- added as REALS
x > i*j +ii  -- multiplied as INTEGERS, added as LONG INTEGERS, compared as REALS
```

The following are currently considered errors.

```
ii IN [i .. x)
(IF i < j THEN i ELSE ii) < x        -- ill-defined target for IfExpr
SELECT i FROM x => ...; > ii => ...; ENDCASE
```

In cases in which automatic widening is not implemented or does not give the desired result, the operator LONG or user-supplied procedure *FLOAT* can be used.

```
m, n: CARDINAL;   ii: LONG INTEGER;
ii ← m +n                        -- added as CARDINALS (overflow lost)
ii ← LONG[m +n]                  -- ditto
ii ← LONG[m] +LONG[n]            -- added as LONG INTEGERS (overflow captured)
```

A fine point: There are system-provided procedures for performing certain multiplication and division operations in which the operands and results do not all have the same precision. These procedures provide less expensive equivalents of, e.g., LONG[*m*]*LONG[*n*]. See the *Mesa 4.0 System Documentation.*

Long Pointers and Array Descriptors

Mesa 4.0 implements both long pointers and array descriptors with long pointers as base components. These pointers provide access to the entire virtual memory of the Dstar. For compatibility, long pointers are also supported on the Alto, but they do not provide any additional addressing capability.

Syntax

```
TypeConstructor  ::=  ... | LongTC
LongTC  ::=  LONG TypeSpecification
ArrayDescriptorTC ::=  DESCRIPTOR FOR TypeSpecification |
                       DESCRIPTOR FOR PackingOption ARRAY OF TypeSpecification
```

The type constructor LONG can be applied to INTEGER (discussed in the preceding section), any pointer type, or any array descriptor type. An attempt to lengthen any other type is an error.

The type constructor DESCRIPTOR FOR can be applied to any array type, including one designated by a type identifier. (This corrects an oversight in previous versions of Mesa). In addition, specification of an **IndexType** for the described array type can be omitted if its constructor follows immediately. In this case, a subrange of CARDINAL with zero origin and indefinite upper bound is assumed for the index type.

Long Pointers

A long pointer value occupies two words (32 bits) of storage. Long pointers are typically created by lengthening (short) pointers as described below. In particular, NIL is automatically lengthened to provide a null long pointer when required by context. The standard operations on pointers (dereferencing, assignment, testing equality, comparison if ORDERED, etc.) all extend to long pointers

> On the Dstar, NIL is lengthened by prefixing a word of zeros and thus has an MDS-independent representation. All other pointers are lengthened by adding the MDS base. Every pointer generated in this way is represented by an 8 bit field of zeros followed by a 24 bit virtual address. Long pointers with certain other formats can be created using LOOPHOLE and will be correctly dereferenced by the hardware. There is no normalization prior to operations on pointers, however, and such pointers will give anomolous results in, e.g., comparisons.
>
> On the Alto, pointers are lengthened by prefixing a word of zeros. In all dereferencing operations, that prefix is discarded (without a check for zero) and the remaining word is interpreted as the actual address.

Both automatic widening and explicit widening (using the operator LONG) are provided for pointer types as well as for numeric types. Widening an expression of type POINTER TO *T* produces a value of type LONG POINTER TO *T*, i.e., only the length attribute is changed by the widening. The rules and restrictions governing widening in Mesa 4.0 that are discussed in the preceding section apply equally to pointers.

The operator @ applied to a variable of type *T* produces a pointer of type LONG POINTER TO *T* if the access path to that variable itself involves a long pointer (other than the implicitly accessed MDS pointer) and of type POINTER TO *T* otherwise.

Limited pointer arithmetic continues to be supported in Mesa 4.0, but programmers are encouraged to use BASE and RELATIVE pointers (described in the next section) if the purpose of the arithmetic is simple relocation. If either operand in a pointer addition or subtraction is long, all operands are widened and the result is long.

> *Examples*
>
> *R:* TYPE = RECORD [*f:* *T*, ...];
> *p*, *q:* POINTER TO *R*;
> *pp*, *qq:* LONG POINTER TO *R*;
> *pT:* POINTER TO *T*;
> *ppT:* LONG POINTER TO *T*;
>
> The following are valid.

pp ← *qq*; *pp* ← NIL; *pp* ← *p*

pp =*qq*, *pp* =NIL, *pp* =*q* -- long comparisons

pT ← @*p.f*; *ppT* ← @*pp.f*

ppT ← @*p.f* -- pointer lengthened

pp+*ii*, *pp*+*i*, *p*+*ii*, *pp*-*qq*, *pp*-*q* -- long results

The following are not valid.

pp =*ppT* -- type clash

p ← *pp*; *pT* ← @*pp.f* -- no automatic shortening

Long Array Descriptors

In a long array descriptor, the BASE component is a long pointer and the descriptor occupies three words (48 bits) of storage. All the standard operations on array descriptors (indexing, assignment, testing equality, LENGTH, etc.) extend to long array descriptors. The type of BASE[*desc*] is long if the type of *desc* is long.

Array descriptors are widened, either automatically or explicitly, according to the usual rules and restrictions. Long array descriptors are created by applying DESCRIPTOR[] to an array that is only accessible through a long pointer (other than the MDS pointer), by applying DESCRIPTOR[,,] to operands the first of which is long, or by widening a (short) array descriptor.

Examples

d: DESCRIPTOR FOR ARRAY OF *T*;
dd: LONG DESCRIPTOR FOR ARRAY OF *T*;
i, *n:* CARDINAL;
pp: LONG POINTER TO ARRAY [0..0) OF *T*;
x: *T*;

dd ← DESCRIPTOR[*pp*, 10, *T*]; *dd* ← *d*

x ← *dd*[*i*]

pp ← BASE[*dd*]; *n* ← LENGTH[*dd*]

Base and Relative Pointers

Mesa 4.0 deals more satisfactorily with base-relative pointers, i.e., pointers that must be *relocated* by adding some base value before they are dereferenced. Such pointers are useful for reducing the number of bits stored when objects can be identified by small offsets, and for dealing with collections of interlinked data items that are subject to relocation as entire aggregates.

Syntax

PointerTC ::= **Ordered BaseOption** POINTER **OptionalInterval PointerTail**

BaseOption ::= **empty** | BASE.

TypeConstructor ::= ... | **RelativeTC**

RelativeTC ::= **TypeIdentifier** RELATIVE **TypeSpecification**

In a **PointerTC**, a nonempty **OptionalInterval** declares a subrange of a pointer type, the values of which are restricted to the indicated interval (and can potentially be stored in smaller fields). Normally, such a subrange type should be used only in constructing a relative pointer type as described below, since its values cannot span an MDS.

The **BaseOption** BASE indicates that pointer values of that type can be used to relocate relative pointers. Such values behave as ordinary pointers in all other respects with one exception: subscript brackets never force implicit dereferencing (see below). The attribute BASE is ignored in determining the assignability of pointer types.

A **RelativeTC** constructs a *relative pointer* or *relative array descriptor* type. The **TypeIdentifier** must evaluate to some (possibly long) pointer type which is the type of the base, and the **TypeSpecification** must evaluate to a (possibly long) pointer or array descriptor type.

Note that the form

LONG **TypeIdentifier** RELATIVE **TypeSpecification**

is always in error, since LONG cannot be applied to a relative type. The type designated by the **TypeSpecification** can be lengthened (to give a relative long pointer) using the form

TypeIdentifier RELATIVE LONG **TypeSpecification** .

Relative Pointers

In the following discussion, assume the declarations

BaseType: TYPE = BASE POINTER TO ...;
FullType: TYPE = POINTER TO ...;
RelativeType: TYPE = *BaseType* RELATIVE *FullType*;
base: BaseType;
offset: RelativeType;
p: FullType.

If *FullType* is some pointer, long pointer, or pointer subrange type, *RelativeType* is declared to be a relative pointer type. Values with type *RelativeType* are pointers that must be relocated, by adding some value of type *BaseType*, before they can be dereferenced. Also, relative pointers are never widened automatically. With respect to other operations (assignment, testing equality, comparison if *FullType* is ORDERED, etc.), relative pointers behave like pointers of type *FullType*. In particular, the amount of storage required to store such a pointer is determined by *FullType*. Note, however, that *RelativeType* and *FullType* are distinct types, incompatible with respect to, e.g., assignment and comparison.

Relocation of a relative pointer is specified by using subscript-like notation in which the type of the "array" is *BaseType* and that of the "index" is *RelativeType*, i.e., the absolute pointer is denoted by an expression with the form

base[*offset*]

This expression has the type *FullType* and the value LOOPHOLE[*base*] +*offset*. Note that *base*[*offset*] is not a variable; typical variable designators are *base*[*offset*]↑ or *base*[*offset*].*field*. (In addition, the usual rules for implicit dereferencing apply in, e.g., an **OpenItem**). Relocation prior to dereferencing is mandatory; *offset*↑, *offset.field*, etc. are errors.

Some fine points:

The type of *base*[*offset*] is more precisely defined as follows: if *FullType* is a subrange pointer type, the subrange is discarded to obtain some type *T*; otherwise, *T* is *FullType*. If *FullType* is not a long pointer type but *BaseType* is, then the final type is LONG *T*; otherwise, it is *T*. In other words, the resulting type is long if either the base type or the relative type is.

The declaration of a relative pointer does not associate a particular base value with that pointer, only a basing type. Thus some care is necessary if multiple base values are in use. Note that the final type of the relocated pointer is largely independent of the type of the base pointer; the relative pointer determines the type. Sometimes this observation can be used to help distinguish different classes of base values without producing relocated pointers with incompatible types.

The base type must have the attribute BASE. Conversely, the attribute BASE always takes precedence in the interpretation of brackets following a pointer expression. Consider the following declarations:

> *p:* POINTER TO ARRAY *IndexType* OF ...;
> *q:* BASE POINTER TO ARRAY *IndexType* OF ... .

The expression *p*[*e*] will cause implicit dereferencing of *p* and is equivalent to *p*↑[*e*]. On the other hand, *q*[*e*] is taken to specify relocation of a pointer, even if the type of *e* is *IndexType* and not an appropriate relative pointer type. In such cases, the array must (and always can) be accessed by adding sufficient qualification, e.g., *q*↑[*e*]; nevertheless, users should exercise caution in using pointers to arrays as base pointers.

Mesa 4.0 supplies no mechanisms for constructing relative pointers. It is expected that such values will be created by user-supplied allocators that pass their results through a LOOPHOLE or from pointer arithmetic involving LOOPHOLES.

Examples

p↑ ← *base*[*offset*]↑

p ← *base*[*offset*] -- valid pointer assignment (but often unwise)

The following are invalid.

p ← *offset*; *p*↑ ← *offset*↑

p[*offset*] -- *p* has incorrect type

Relative Array Descriptors

Relative array descriptor types are entirely analogous to relative pointer types; indeed, values of such types can be viewed as array descriptors in which the base components are relative pointers. Note the following:

In the constructor of a relative array descriptor type, the **TypeSpecification** must evaluate to a (possibly long) array descriptor type.

In the notation introduced above, a reference to an element of the described array has the form

> *base*[*offset*][*i*]

where *i* is the index of the element.

Relative array descriptors are constructed using the DESCRIPTOR operator. If *p* is *B* RELATIVE pointer, the form DESCRIPTOR[*p*, *n*, *T*] produces a value with type *B* RELATIVE DESCRIPTOR FOR ARRAY OF *T*. Also, the operators BASE and LENGTH can be applied to a *B* RELATIVE array descriptor; the former produces a *B* RELATIVE pointer.

Block Structure

The previous concepts of procedure body and compound statement have been merged. A *block* can appear anywhere a statement is acceptable and can introduce new identifiers with scope smaller than an entire procedure (or module) body. In addition, catch phrases and exit labels can now appear at the outermost level of a procedure body.

The syntax for declaring procedures with bodies expressed in machine code has also been revised (in anticipation of more general inline procedures). The corresponding semantics are machine dependent and are not specified here.

Syntax

```
ModuleBody  ::=  Block

ProcedureBody  ::=  Block

Statement  ::=  ... | Block | ...        -- replaces CompoundStmt

Block  ::=   BEGIN
             OpenClause
             EnableClause
             DeclarationSeries
             StatementSeries
             ExitsClause
             END

EnableClause  ::=    empty |
                     ENABLE Catchltem ; |
                     ENABLE BEGIN CatchSeries END ; |
                     ENABLE BEGIN CatchSeries ; END ;

MachineCode   ::=  MACHINE CODE BEGIN InstructionSeries END

InstructionSeries ::=   empty | ByteList |
                        ByteList ; InstructionSeries

ByteList   ::=  Expression | ByteList , Expression
```

In addition, the phrase classes **Body**, **CompoundStmt** and **MachineCodeTC** are deleted.

During the execution of a Mesa program, frames are allocated at the procedure and module level only. Any storage required by variables declared in an internal **Block** (one used as a **Statement**) is allocated in the frame of the smallest enclosing procedure or module. When such internal blocks are disjoint, the areas of the frame used for their variables overlay one another.

The scopes of identifiers introduced in the various components of a block are summarized by the following diagram, where indentation is used to show the scope of each phrase:

```
BEGIN
OpenClause
    EnableClause
        DeclarationSeries
            StatementSeries
   ExitsClause
END
```

Note that any newly declared identifiers are visible only in the **DeclarationSeries** and **StatementSeries** of the block. Any exit labels are visible within the **EnableClause** (as well as the more deeply indented constructs); on the other hand, any catch phrase in the **EnableClause** is not enabled within the **ExitsClause.** If the **Block** is used as a module or procedure body, the parameters and results are visible throughout the **Block.** Thus it is possible to open records designated by parameters or to assign return values within an **ExitsClause** (but the assigned values cannot involve internally declared variables).

A CONTINUE statement appearing in the **EnableClause** of a **Block** causes exit from that block. A similarly placed RETRY statement causes reexecution of the block. In the latter case, any initializing values in the **DeclarationSeries** are recomputed.

Note that an optional semicolon can now terminate a **CatchSeries** in an **EnableClause.**

Nested Block Structure

With the introduction of blocks, procedure bodies can appear where they were syntactically prohibited in previous versions of Mesa. Special rules apply to the inheritance of scope when a procedure body is declared within the **DeclarationSeries** or (with nesting) within the **StatementSeries** of a **Block.** Within the inner procedure body:

Identifiers made visible by the **OpenClause** remain visible (unless redeclared).

Catch phrases in the **EnableClause** are not inherited and not enabled.

Identifiers declared in the **DeclarationSeries** remain visible (unless redeclared).

Jumps to labels in the **ExitsClause** are prohibited.

Assume the following skeletal declaration:

```
Outer: PROCEDURE [...] =
   BEGIN
   ENABLE s => Handler[];
   ...
   Inner: PROCEDURE [...] = BEGIN ... END;
   ...
   EXITS
      Label => ...
   END
```

If the signal *s* is raised in an instance of *Inner*, *Handler* is not invoked there. *Handler* will, of course, be invoked eventually if *s* propagates to the enclosing instance of *Outer*. (This noninheritance rule prevents double execution of handlers in such situations.) In Mesa 4.0, the statement GO TO *Label* is considered an error within the body of *Inner*.

Iterative Statements

For consistency with blocks, the scope rules for iterative statements have been revised slightly. In addition, a new statement form that terminates one iteration of the loop body and initiates the next has been added.

Syntax

```
Statement    ::=    ... | LoopCloseStmt

LoopStmt ::=LoopControl
                 DO
                 OpenClause
                 EnableClause
                 StatementSeries
                 LoopExitsClause
                 ENDLOOP

LoopCloseStmt   ::=  LOOP
```

The scopes of identifiers introduced in the various components of a loop are summarized by the following diagram (cf. **Blocks**):

```
LoopControl
  DO
  OpenClause
      EnableClause
         StatementSeries
    LoopExitsClause
  ENDLOOP
```

In previous versions of Mesa, the scope of the **OpenClause** excluded the **LoopExitsClause.** As in the case of blocks, any exit labels are visible within the **EnableClause**, and any catch phrase in the **EnableClause** is not enabled within the **ExitsClause.**

The statement LOOP can appear only within the body of an iterative statement. Executing it terminates the current iteration of the smallest enclosing **LoopStmt**, after which the **LoopControl** is updated/reevaluated and, if appropriate, the next iteration is started. Thus the construct

```
DO ... LOOP ... ENDLOOP
```

is an abbreviation for

```
DO
  BEGIN
  ... GO TO Skip ...
  EXITS   Skip => NULL;
  END
ENDLOOP .
```

Included Identifier Lists

In Mesa 4.0, an item in the DIRECTORY clause can explicitly list the identifiers eligible for inclusion from a designated module. Such *included identifier lists* serve as compiler-checked (but programmer-maintained) lists of intermodular connections and dependencies.

Syntax

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

IncludeItem  ::=        identifier : FROM FileName |
                        identifier : FROM FileName USING [ IdList ]
```

If the USING clause is absent, the item's **identifier** has all the properties and uses described in Sections 7.2.1 and 7.2.2. The only effect of a USING clause is to enumerate (and potentially restrict) the set of identifiers made accessible to the including module. Use of the **identifier**, either within an OPEN clause or for explicit qualification, makes visible only those identifiers in the **IdList**.

Some fine points.

Only identifiers declared in the **DeclarationSeries** that is part of the **ModuleBody** of the included module are mentioned in the **IdList**; in particular, neither the included module's own identifier nor identifiers of record fields, enumeration constants, etc. appear in this list.

Each identifier appearing in the **IdList** must be defined in the module designated by the **IncludeItem**.

A warning is generated for each identifier appearing in the **IdList** but not used explicitly in the including module. Identifiers used only implicitly (to describe attributes of explicitly included identifiers) should not be listed.

The **IdList** restricts the set of identifiers available for inclusion from a module. It does not restrict export into an included interface. The identifier of an exported item should not appear in the list unless the intention is to reference a different item with the same name through an imported instance of the interface.

The following example assumes the declaration of *SimpleDefs* appearing on page 92.

```
DIRECTORY
  SimpleDefs: FROM "simpledefs" USING [Range, PairPtr];
Example: PROGRAM =
  BEGIN
  First: PROCEDURE [p: SimpleDefs.PairPtr] RETURNS [SimpleDefs.Range] =
    BEGIN
    RETURN [IF p = NIL THEN 0 ELSE p.first]
    END;
  END.
```

Note that *Pair* does not appear in the included identifier list (because it is only referenced implicitly, through the definition of *PairPtr*), nor does *first* (because it is declared in a record, not in the body of *SimpleDefs* itself). Any reference to *SimpleDefs.limit* would be an error in this example.

Processes

Mesa 4.0 supports a process mechanism in which processes are created by forking to procedures and are synchronized by entry to monitors. Most of the information about the semantics and intended usage of Mesa processes appears in the *Mesa 4.0 Process Update* (henceforth cited as *Process*). The *Mesa 4.0 Change Summary* contains a complete example, and additional examples appear in the *Process* document. This section summarizes the syntax and deals with a few linguistic details.

Syntax

```
PredefinedType   ::=  ... | MONITORLOCK | CONDITION

ProgramTC ::=         ... |
                      MONITOR ParameterList ReturnsClause LocksClause

LocksClause ::=       empty |
                      LOCKS Expression |
                      LOCKS Expression USING identifier : TypeSpecification

TypeConstructor  ::=  ... | ProcessTC

ProcessTC  ::=  PROCESS ReturnsClause

Declaration ::=       IdList : Access EntryOption TypeSpecification Initialization ; |
                      IdList : Access TYPE = Access TypeSpecification ;

EntryOption  ::=  empty | ENTRY | INTERNAL

RecordTC  ::=  MonitoredOption MachineDependent RECORD [ VariantFieldList ]

MonitoredOption  ::=  empty | MONITORED

Statement  ::=  ... | WaitStmt | NotifyStmt | JoinCall

Expression  ::=  ... | ForkCall | JoinCall

WaitStmt  ::=  WAIT Variable OptCatchPhrase

NotifyStmt  ::=  NOTIFY Variable | BROADCAST Variable

ForkCall  ::=  FORK Call

JoinCall  ::=  JOIN Call
```

Forking and Joining

Processes are created and destroyed by FORK and JOIN operations. If procedure *P* has type PROCEDURE *T* RETURNS *T'*, then the expression FORK *P*[...] produces a *process handle h* with type PROCESS RETURNS *T'*. JOIN requires a process handle as its operand. The form JOIN *h* produces an argument record of type *T'* (or stands as a statement if the **ReturnsClause** is empty). As type mappings,

FORK:	PROCEDURE *T* RETURNS *T'* × *T* → PROCESS RETURNS *T'*
JOIN:	PROCESS RETURNS *T'* → *T'*.

Some fine points:

A catch phrase can be attached to a FORK or JOIN (by specifying it in the **Call**).

Unlike an ordinary procedure call, a FORK returns a value with some process type (not a record type), and that value cannot be discarded by writing an empty extractor.

Monitored Modules

A **ProgramTC** containing MONITOR can be used only in a **ModuleHead** to specify the type of a program module. The **LocksClause** provides additional information about the program body and is not part of the module's type. If a monitor is to be exported, the correct type for the interface item in the DEFINITIONS module is obtained by replacing MONITOR by PROGRAM and deleting the **LocksClause**.

Synchronization of processes is based upon variables with the system-defined types MONITORLOCK and CONDITION. A distinguished MONITORLOCK with the identifier *LOCK* is implicitly declared in the global frame of any MONITOR with an empty **LocksClause**. If the **MonitoredOption** MONITORED appears in the definition of a record type, each record of that type similarly contains an implicitly declared and distinguished MONITORLOCK with identifier *LOCK*. Lock and condition variables can also be declared explicitly, but any MONITORLOCK so declared is not distinguished, even if its identifier is *LOCK* (see below).

When a variable with type MONITORLOCK or CONDITION is a component of a (local or global) frame, it is initialized automatically when the frame is created. In all other cases, a system procedure must be called to establish appropriate initial values (see *Process*, Section A.6).

Entry Procedures

The **EntryOption** ENTRY can appear only in a declaration within a monitor; when it does, the **TypeSpecification** must evaluate to a procedure type and the initialization must specify a procedure body (**Block**). Note that ENTRY does not imply PUBLIC, but PUBLIC ENTRY is a permissible (and common) combination.

Entry into a monitor through an ENTRY procedure is protected by a monitor lock. The identity of that lock is determined by the declaration of the monitor. If the **LocksClause** is empty, entry is controlled by the distinguished variable *LOCK*. Otherwise, the **LocksClause** must designate a variable with type MONITORLOCK, a record containing a distinguished lock field, or a pointer that can be dereferenced (perhaps several times) to yield one of the preceding. There are two cases (see *Process*, Section A.4.2):

> If the USING clause is absent, the monitor is a multi-module one. The lock is located by evaluating the LOCKS expression in the context of the monitor's main body; i.e., the monitor's parameters, imports, and global variables are visible, as are any identifiers made accessible by a global OPEN. Evaluation occurs upon entry to, and again upon exit from, the ENTRY procedure (and for any internal WAITs). The location of the designated lock can thus be affected by assignments within the procedure to variables in the LOCKS expression. To avoid disaster, it is essential that each reevaluation yield a designator of the same MONITORLOCK.
>
> If the USING clause is present, the monitor is an object monitor. The lock is located as above with one exception: any occurrence of the identifier declared in the USING clause is bound to that argument of the ENTRY procedure having the same identifier and a compatible type. If there is no such parameter, the ENTRY is in error. The same care is necessary with respect to reevaluation; to emphasize this, the distinguished argument is treated as a read-only value within the body of the ENTRY procedure.

The following examples illustrate the selection of locks.

```
R: TYPE = MONITORED RECORD [...];
RR: TYPE = RECORD [..., specialLock: MONITORLOCK, ...];

M1: MONITOR =
  BEGIN
  -- LOCK: MONITORLOCK implicitly declared here
  P1: PUBLIC ENTRY PROCEDURE [...] =
    BEGIN -- locks LOCK -- ... END;
  END.
```

```
M2a: MONITOR [p: POINTER TO POINTER TO R] LOCKS p =
  BEGIN
  P2: PUBLIC ENTRY PROCEDURE [...] =
    BEGIN -- locks p↑↑.LOCK -- ... END;
  END.

M2b: MONITOR [p: POINTER TO POINTER TO RR] LOCKS p↑↑.specialLock =
  -- specification of the lock is mandatory here
  BEGIN
  P2: PUBLIC ENTRY PROCEDURE [...] =
    BEGIN -- locks p↑↑.specialLock -- ... END;
  END.

M3: MONITOR LOCKS p USING p: POINTER TO R =
  BEGIN
  P3: PUBLIC ENTRY PROCEDURE [p: POINTER TO R, ...] =
    BEGIN -- locks p.LOCK -- ... END;
  END.
```

Signals require special attention within the body of an ENTRY procedure. A signal raised with the monitor lock held will propagate without releasing the lock and possibly invoke arbitrary computations. For errors, this can be avoided by using the RETURN WITH ERROR construct described in the next section.

When an instance of an ENTRY procedure is to be destroyed because of a remote exit from a catch phrase (unwinding), the lock should also be released. In Mesa 4.0, it is the programmer's responsibility to determine if unwinding is possible and, if so, to provide a catch phrase for UNWIND that restores the monitor invariant. Code to actually release the monitor lock is automatically appended to the outermost enabled catch phrase for UNWIND in an ENTRY procedure. That catch phrase can have a NULL body if no other cleanup actions are required.

Internal Procedures

The **EntryOption** INTERNAL can appear only in a declaration within a monitor; when it does, the **TypeSpecification** must evaluate to a procedure type and the initialization must specify a procedure body (**Block**). Note that INTERNAL does not imply PRIVATE (if the default is PUBLIC), but PUBLIC INTERNAL is considered an improper combination of attributes (warning only).

A call of an INTERNAL procedure is permitted only within an ENTRY procedure or another INTERNAL procedure. Forking to an INTERNAL procedure is never allowed. An INTERNAL procedure can safely access monitored data and can perform WAIT, NOTIFY and BROADCAST operations. A WAIT operation implicitly references the monitor lock; thus an INTERNAL procedure of an object monitor that contains a WAIT must have a parameter designating the locked object as described above.

Some fine points:

> In Mesa 4.0, the attribute INTERNAL is associated with a procedure's body, not its type. Thus INTERNAL cannot be specified in a DEFINITIONS module, and checks on intermodular calls of internal procedures are not performed (except for the PUBLIC INTERNAL warning). Also, the attribute INTERNAL is lost when a procedure value is assigned to a variable or passed as an argument of a procedure. Such assignments should be done with caution.

Signals raised by INTERNAL procedures require special consideration. When the construct RETURN WITH ERROR is executed within an INTERNAL procedure, the monitor lock is *not* released prior to signal propagation.

Wait and Notify

Only ENTRY and INTERNAL procedures within a monitor can contain WAIT, NOTIFY and BROADCAST statements.

Error Returns

It is possible to delete a procedure instance before raising an error detected by that procedure. Within an ENTRY procedure of a monitor, the monitor lock is released before the error is raised. (Such procedures are expected to be the primary users of this facility.)

Syntax

```
ReturnStmt   ::=  ... | RETURN WITH ERROR Call
```

Consider the following skeletal code:

```
Failure: ERROR [...] = CODE;

Proc: ENTRY PROCEDURE [...] RETURNS [...] =
   BEGIN
   ENABLE UNWIND => ...;
   ..
   IF cond1 THEN ERROR Failure[...];
   IF cond2 THEN RETURN WITH ERROR Failure[...];
   ...
   END;
```

Execution of the construct ERROR *Failure*[...] raises a signal that propagates until some catch phrase specifies an exit. At that time, unwinding begins; the catch phrase for UNWIND in *Proc* is executed and then *Proc*'s frame is destroyed. Within an entry procedure such as *Proc*, the lock is held until the unwind (and thus through unpredictable computation performed by catch phrases).

Execution of the construct RETURN WITH ERROR *Failure*[...] releases the monitor lock and destroys the frame of *Proc* before propagation of the signal begins. Note that the argument list in this construct is determined by the declaration of *Failure* (not by *Proc*'s RETURNS clause). The catch phrase for UNWIND is not executed in this case. The signal *Failure* is actually raised by the system, after which *Failure* propagates as an ordinary error (beginning with *Proc*'s caller).

Multiword Constants

Record and array constructors in which all components are themselves constant define so-called *multiword constants*. Such constants are now constructed during compilation and can be encoded within Mesa symbol tables. This has the following consequences:

A declaration equating an identifier to a multiword constant (but not to a string literal) can appear in a DEFINITIONS module, and the constant value thereby becomes available to users of that module.

Constant selection from such values (by field selection or by indexing with a constant subscript) is also done during compilation.

Furthermore, if an identifier is equated to a multiword constant in a program module, exactly one copy of that constant appears in the code, and its components can be read (using, e.g., a computed index) directly from the code segment. This allows table driven programming in which the tables are automatically swapped.

A fine point: A packed array or an array of multiword elements is currently copied into a data area each time one of its elements is accessed.

The following declarations define multiword constants and can appear in a DEFINITIONS module.

Ident: RECORD [*version:* CARDINAL, *id:* CHARACTER, *released:* BOOLEAN] =
[1, '#, FALSE];

Powers: ARRAY [1..4] OF CARDINAL = [2, 4, 8, 16];

Nonsense: CARDINAL = IF *Ident.released* THEN *Ident.version* ELSE *Powers*[2];

The following are not compile-time constants in Mesa 4.0.

"abc", ("abc")[1].

Miscellaneous Language Changes

Local Strings

The body of a string literal is ordinarily placed in the global frame of the module in which the literal appears. Pointers to that body (the actual STRING values) can then be used freely with little danger that the body will move or be destroyed. Unfortunately, this scheme can consume substantial amounts of space in the (permanent and unmovable) global frame area.

If a string literal is followed by 'L (e.g., "abc"L), a copy of the string body is moved from the code to the local frame of the smallest enclosing procedure whenever an instance of that procedure is created. As a corollary, the space is freed and the string body disappears when the procedure returns. Thus it is important to insure that pointers to local string literals are not assigned to STRING variables with lifetimes longer than that of the procedure. Programmers should avoid using local string literals until performance tuning is necessary (except perhaps in calls of straightforward output procedures).

Character Arithmetic

The following arithmetic operations are now defined for values of type CHARACTER:

CHARACTER + INTEGER → CHARACTER
INTEGER + CHARACTER → CHARACTER
CHARACTER - INTEGER → CHARACTER
CHARACTER - CHARACTER → INTEGER.

Other arithmetic operations do not allow characters as operands, and values of type INTEGER and CHARACTER cannot be cross-assigned.

Examples

c: CHARACTER;
d: INTEGER ← *c* - '0; -- consider a translation table instead

```
IF c IN ['a..'z] THEN c ← 'A + (c-'a)
```

Selections

More general expressions are allowed to label arms of selections when there is no initial relational operator.

```
Test ::= Expression | RelationTail                -- formerly Sum | RelationTail
```

Example

```
SELECT TRUE FROM
  i > 0, j > 0 ⇒ s1;              -- previously required (i > 0), (j > 0)
  p AND q      ⇒ s2;              -- previously required (p AND q)
  k > 0 OR q   ⇒ s3;
  ...
  ENDCASE      ⇒ sN
```

This is equivalent to (and perhaps more readable than)

```
IF i > 0 OR j > 0 THEN s1
ELSE IF p AND q THEN s2
ELSE IF k > 0 OR q THEN s3
...
ELSE sN .
```

Discriminations

Previous versions of Mesa have required that all adjectives labeling an arm of a discrimination name identically structured variants; in Mesa 4.0, this restriction is lifted. If, however, the labels identify more than one variant structure, the record is not considered to be discriminated within that arm and only the common fields are visible (cf. ENDCASE).

Example

```
R: TYPE = RECORD [
  v: T,
  variant: SELECT tag:* FROM
    red, pink ⇒ [vRP: T],
    green ⇒ [vG: T],
    yellow ⇒ [vY: T],
    ENDCASE];

r: R;

WITH x: r SELECT FROM
  red, pink ⇒ ...;          -- x.v and x.vRP accessible
  green, yellow ⇒ ...;      -- only x.v accessible
  ENDCASE ⇒ ...;            -- only x.v accessible
```

Mesa 4.0 also allows computed or overlaid variant records to be compared without discrimination if all variants have the same length. As usual, caution is advised; two records interpreted as different variants can be represented by the same bit pattern when computed tags are used.

Compilation Options

The following compiler options have been added; they are controlled by switches in the usual way:

Switch	*Option Controlled*
alto	Generating code for an Alto or Dstar
run	Terminating compilation by running another program
sort	Sorting global variables and entry indices

The Alto/Dstar switch primarily affects the treatment of long pointers in the object code.

The run switch specifies running another program without returning to the executive. This switch is primarily intended for use in command files. The file name preceding the switch specifies the program to be run. The file is assumed to contain a program requiring standard (Bcpl) microcode if the file name's extension is ".RUN" and requiring Mesa microcode otherwise. The default extension is ".IMAGE". Prior to execution of the specified program, a new command file (COM.CM) is constructed containing the full file name plus any switches following the 'r. In the case of command-line input, the remainder of the command line is also appended.

The sorting switch has been added in anticipation of tools that will expedite updating a module in a configuration or subsystem when the new and old versions of the object code are sufficiently similar. When sorting is suppressed, the assignment of global frame offsets and entry indices depends only upon order of declaration in the source text; on the other hand, the generated code is likely to be somewhat less compact.

> Sorting of local variables is not suppressed. Unless a module uses global variables extensively, the object code expansion is unlikely to exceed ?%.

The defaults are to generate code for an Alto, to terminate by returning to the executive, and to sort global variables and entry points.

Internal Changes

The following internal changes are mentioned for completeness; see the *Mesa 4.0 System Update* for more information.

Main Body Procedure

The main body of a module is now executed in a separate local frame. Note however, that any storage required by blocks or local strings in the main body is still allocated in the global frame.

External Links

External links (for imported procedures, signals or frames) are now stored and indexed backwards from the global frame base or code base (as selected by a binding/loading option).

Alto/ Mesa Microcode

Both the instruction set and the opcode numbers have changed substantially.

Frame Allocation

Instructions for allocating and freeing frames are now implemented in microcode; this greatly inceases the speed of any transfer involving a large argument record.

Distribution:
Mesa Users
Mesa Group

Appendix: Signed and Unsigned Arithmetic

Background and Overview

In any implementation of Mesa, the number of bits available for representing a value of a given type is fixed. Each numeric type of the language thus is restricted to some subrange of $\underline{Z}$, the set of integers as understood in mathematics. The following types, corresponding to the indicated subranges, are built into the language:

INTEGER	$[-2^{N-1} .. 2^{N-1})$	-- "signed integers"
CARDINAL	$[0 .. 2^{N})$	-- "unsigned integers"
LONG INTEGER	$[-2^{2N-1} .. 2^{2N-1})$	-- "double-precision integers"

Here N is the word length of the machine (N=16 for the Alto and Dstar). The programmer can also declare types that are themselves subranges of CARDINAL or INTEGER (but not LONG INTEGER), e.g., *T*: TYPE = [0..10).

Let *v*, *x*, and *y* be variables with numeric subrange types. In principle, execution of the assignment $v \leftarrow x \ominus y$ proceeds as follows:

The values of *x* and *y* are taken as elements of $\underline{Z}$.

Those values are combined using some function *f* that defines the operator $\ominus$ over $\underline{Z}$ and produces a result $f(x,y)$, also in $\underline{Z}$.

If the result is in the subrange of $\underline{Z}$ spanned by the type of *v*, $f(x,y)$ is assigned to *v*; otherwise a *range failure* occurs.

Unfortunately, the underlying hardware does not provide the function *f* but only a partial function *f'* over some subrange of $\underline{Z}$ with the property that *f'* agrees with *f* wherever both are defined; *f'* is said to *overflow* (or *underflow*) elsewhere. In fact, the hardware generally provides a family of partial functions related to *f*, one each for INTEGER, CARDINAL, and LONG INTEGER. The operator $\ominus$ thus is generic at the hardware level, and the compiler must choose the appropriate partial function for preserving the abstraction being used by the programmer (or for detecting its breakdown). The choice is made by considering an attribute of each operand called its *representation.*

If the type of any operand is LONG INTEGER, the rule is simple: all other operands are converted to LONG INTEGER and the result is computed in that domain. For INTEGERS (with *signed* representation), CARDINALS (with *unsigned* representation) and subrange types such as *T* (with both representations), the issues are more subtle. Some operators, such as the relationals, are clearly generic and were recognized as such in previous versions of Mesa. Many other operators produce the correct result modulo 2^{N} (i.e., the "right" bit pattern) no matter what representation is assumed; the representation affects only the definition of overflow.

Examples (N=16)

The bit patterns representing -1 and 177777B are identical, but (177777B > 1) is TRUE while (-1 > 1) is FALSE . Also, (-1 +1) =0 and there is no overflow, but (177777B + 1) cannot be represented as an unsigned number.

In a critique of Mesa [Wirth], Niklaus Wirth has argued strongly that the language should be defined so that the overflow condition can always be specified. Note that this is a necessary condition for implementing reliable range checking (also advocated by Wirth) but not a sufficient one. Mesa 4.0 *does not* provide options for overflow detection or range checking but *does* revise the language definition so that future versions can offer such options.

While we have found no rules for mixing signed and unsigned values that are entirely satisfactory, we believe that those presented in the following section are reasonably unobtrusive, compatible with existing code and relatively free of surprises.

Signed and Unsigned Numbers

This section discusses the rules now used by Mesa for choosing between signed and unsigned versions of operations on single-precision numbers. The new rules assume that there are conversion functions performing the following mappings:

CARDINAL → INTEGER

INTEGER → CARDINAL .

In both cases, the "conversion" amounts to an assertion that the argument is an element of INTEGER ∩ CARDINAL. The programmer can also make such a range assertion explicit as described in the main body of this memo. In Mesa 4.0, *such assertions must be verified by the programmer.* There is *not* an option to generate code that checks these assertions, whether implicit or explicit, or code that detects overflow in arithmetic operations.

For each of the operators +, -, *, /, MOD, MIN, and MAX, there are two single-precision operations, mapping as follows:

INTEGER^n → INTEGER	(signed arithmetic)
CARDINAL^n → CARDINAL	(unsigned arithmetic).

Similarly, there are two operations for each of the operators =, #, <, <=, >, >= and IN:

INTEGER^n → BOOLEAN	(signed comparisons)
CARDINAL^n → BOOLEAN	(unsigned comparisons).

There are no operations upon mixed representations in any case; thus all operands must be forced to have some common representation. The arithmetic operators also propagate that same representation to the result.

> A possible surprise is that CARDINAL is taken to be closed under subtraction; i.e., $m-n$ is considered to overflow if m and n are CARDINALS and $m < n$.

For any arithmetic expression, the *inherent representations* of the operands and the *target representation* of the result are used to choose between the signed and unsigned operations (cf. the discussion of inherent and target types, Section 3.1, pages 37-39).

> The target type determines the target representation. The target type is derived from the type of the variable to which an expression is to be assigned, from a range assertion applied to a subexpression, etc. If all valid values of the target type are nonnegative, the target representation is *unsigned*; otherwise, it is *signed*. The arithmetic operators listed above propagate target representations unchanged to their operands, but the target representation of an operand of a relational operator is undefined. Thus each (sub)expression has at most one target representation.

> The inherent representation of a primary is determined by its type (if a variable, function call, etc.), by its value (if a compile-time constant), or explicitly (if a range assertion). Possible inherent representations are signed and unsigned; in addition,

compile-time constants in $[0 .. 2^{N-1})$ and primaries with types that are subranges of INTEGER $\cap$ CARDINAL are considered to have *both* inherent representations. Inherent representations of operands are propagated to results as described below.

The basic idea is that generic operations are disambiguated first by the inherent representations of their operands, next by the target representation, and finally by a default convention. If the operation cannot be disambiguated in any of these ways, the expression is considered to be in error. The exact rules follow:

If the operands have exactly one common inherent representation, the operation defined for that representation is selected (and the target representation is ignored).

If the operands have no common inherent representation but the target representation is well-defined, the operation yielding that representation is chosen, and each operand is "converted" to that representation (in the weak sense discussed above).

If the operands have both inherent representations in common, then
if the target representation is well-defined it selects the operation;
otherwise the signed operation is chosen.

If the operands have no representation in common and the target representation is ill-defined, the expression is in error.

In all cases, the inherent representation of the result is determined by the mapping performed by the selected operation.

The unary operators require special treatment. Unary minus converts its argument to a signed representation if necessary and produces a signed result. ABS is a null operation (with warning message) on an operand with an unsigned representation, and it yields an unsigned representation in any case. The target representation for the operand of LONG (or of an implied widening operation) is unsigned.

Examples

Assume the following declarations:

i, j: INTEGER; *m, n:* CARDINAL; *s, t:* [0..77777B]; *b:* BOOLEAN

The statements on each of the following lines are equivalent.

```
i ← m+n;  i ← INTEGER[m+n]                        -- unsigned addition
i ← j+n;  i ← n+j;  i ← j+INTEGER[n]              -- signed addition
i ← s+t;  i ← INTEGER[s] +INTEGER[t]              -- signed (overflow possible)
n ← s+t;  n ← CARDINAL[s] +CARDINAL[t]            -- unsigned (overflow impossible)
s ← s-t;  s ← CARDINAL[s] - CARDINAL[t]           -- unsigned (overflow possible)
b ← s-t > 0;  b ← INTEGER[s] - INTEGER[t]  > 0    -- signed (overflow impossible)
i ← -m;   i ← -INTEGER[m]
i ← m+n*(j+n);  i ← INTEGER[m] + (INTEGER[n]*(j+INTEGER[n]))
n ← m+n*(j+n);  n ← m + (n*(CARDINAL[j] +n))
```

```
i ← m+n*(s+n);   i ← INTEGER[m+(n*(CARDINAL[s]+n))]
b ← s IN [t-1 .. t+1];   b ← INTEGER[s] IN [INTEGER[t-1] .. INTEGER[t+1]]
FOR s IN [t-1 .. t+1] ...;   FOR s IN [CARDINAL[t-1] .. CARDINAL[t+1]] ...
```

The following statements are incorrect because of representational ambiguities.

```
b ← i > n;   b ← i+n IN [s .. j]
SELECT i FROM m => ...;  t => ...; ENDCASE
```

Both the following are legal and assign the same bit pattern to *i*, but the first overflows if $m < n$.

```
i ← m-n;   i ← IF m >= n THEN m-n ELSE -(n-m) .
```

Reference

Wirth, N. *On the peaceful coexistence of integers and cardinals*, Xerox PARC, 29 June 1977.

Inter-Office Memorandum

To	Mesa Users	Date	May 31, 1978
From	Dave Redell, Dick Sweet	Location	Palo Alto
Subject	Mesa 4.0 Process Update	Organization	SDD/SD

XEROX

Filed on: [IRIS]<MESA>DOC>PROCESS40.BRAVO

Mesa provides language support for concurrent execution of multiple processes. This allows programs that are inherently parallel in nature to be clearly expressed. The language also provides facilities for synchronizing such processes by means of entry to monitors and waiting on condition variables.

The next section discusses the forking and joining of concurrent process. Later sections deal with monitors, how their locks are specified, and how they are entered and exited. Condition variables are discussed, along with their associated operations.

10.1. Concurent execution, FORK and JOIN.

The FORK and JOIN statements allow parallel execution of two procedures. Their use also requires the new data type PROCESS. Since the Mesa process facilities provide considerable flexibility, it is easiest to understand them by first looking at a simple example.

10.1.1. A Process Example

Consider an application with a front-end routine providing interactive composition and editing of input lines:

```
ReadLine: PROCEDURE [s: STRING] RETURNS [CARDINAL] =
    BEGIN
    c: CHARACTER;
    s.length ← 0;
    DO
        c ← ReadChar[];
        IF ControlCharacter[c] THEN DoAction[c]
        ELSE AppendChar[s,c];
        IF c =CR THEN RETURN [s.length];
        ENDLOOP;
    END;
```

The call

```
n ← ReadLine[buffer];
```

will collect a line of user type-in up to a *CR* and put it in some string named *buffer*. Of course, the caller cannot get anything else accomplished during the type-in of the line. If there is anything else that needs doing, it can be done concurrently with the type-in by *forking* to *ReadLine* instead of calling it:

```
p ← FORK ReadLine[buffer];
. . .
<concurrent computation>
. . .
n ← JOIN p;
```

This allows the statements labeled <concurrent computation> to proceed in parallel with user typing (clearly, the concurrent computation should not reference the string *buffer*). The FORK construct spawns a new process whose result type matches that of *ReadLine*. (*ReadLine* is referred to as the "root procedure" of the new process.)

```
p: PROCESS RETURNS [CARDINAL];
```

Later, the results are retrieved by the JOIN statement, which also deletes the spawned process. Obviously, this must not occur until both processes are ready (i.e. have reached the JOIN and the RETURN, respectively); this rendevous is synchronized automatically by the process facility.

Note that the types of the arguments and results of *ReadLine* are always checked at compile time, whether it is called or forked.

The one major difference between calling a procedure and forking to it is in the handling of signals; see section 10.5.1 for details.

10.1.2. Process Language Constructs

The declaration of a PROCESS is similar to the declaration of a PROCEDURE, except that only the return record is specified. The syntax is formally specified as follows:

```
TypeConstructor   ::= ...  | ProcessTC
ProcessTC         ::= PROCESS ReturnsClause
ReturnsClause     ::= empty | RETURNS ResultList   -- from sec. 5.1.
ResultList        ::= FieldList                    -- from sec. 5.1.
```

Suppose that *f* is a procedure and *p* a process. In order to fork *f* and assign the resulting process to *p*, the **ReturnClause** of *f* and that of *p* must be compatible, as described in sec 5.2.

The syntax for the FORK and JOIN statements is straightforward:

```
Statement         ::= ...  | JoinCall
Expression        ::= ...  | ForkCall | JoinCall
ForkCall          ::= FORK Call
JoinCall          ::= JOIN Call
Call              ::= (see sections 5.4 and 8.2.1)
```

The **ForkCall** always returns a value (of type PROCESS) and thus a FORK cannot stand alone as a statement. Unlike a procedure call, which returns a RECORD, the value of the FORK cannot be discarded by writing an empty extractor. The action specified by the FORK is to spawn a process parallel to the current one, and to begin it executing the named procedure.

The **JoinCall** appears as either a statement or an expression, depending upon whether or not the process being joined has an empty **ReturnsClause**. It has the following meaning: When the forked procedure has executed a RETURN *and* the JOIN is executed (in either order),

the returning process is deleted, and

the joining process receives the results, and continues execution.

A catchphrase can be attached to either a FORK or JOIN by specifying it in the **Call**. Note, nowever, that such a catchphrase does not catch signals incurred during the execution of the procedure; see section 10.5.1 for further details.

There are several other important similarities with normal procedure calls which are worth noting:

The types of all arguments and results are checked at compile time.

There is no *intrinsic* rule against multiple activations (calls and/or forks) of the same procedure coexisting at once. Of course, it is always possible to write procedures which will work incorrectly if used in this way, but the mechanism itself does not prohibit such use.

One expected pattern of usage of the above mechanism is to place a matching FORK/JOIN pair at the beginning and end of a single textual unit (i.e. procedure, compound statement, etc.) so that the computation within the textual unit occurs in parallel with that of the spawned process. This style is encouraged, but is *not* mandatory; in fact, the matching FORK and JOIN need not even be done by the same process. Care must be taken, of course, to insure that each spawned process is joined only once, since the result of joining an already deleted process is undefined. Note that the spawned process always begins and ends its life in the same textual unit (i.e. the target procedure of the FORK).

While many processes will tend to follow the FORK/JOIN paradigm, there will be others whose role is better cast as continuing provision of services, rather than one-time calculation of results. Such a "detached" process is never joined. If its lifetime is bounded at all, its deletion is a private matter, since it involves neither synchronization nor delivery of results. No language features are required for this operation; see the runtime documentation for the description of the system procedure provided for detaching a process.

10.2. Monitors

Generally, when two or more processes are cooperating, they need to interact in more complicated ways than simply forking and joining. Some more general mechanism is needed to allow orderly, synchronized interaction among processes. The interprocess synchronization mechanism provided in Mesa is a variant of *monitors* adapted from the work of Hoare, Brinch Hansen, and Dijkstra. The underlying view is that interaction among processes always reduces to carefully synchronized access to shared data, and that a proper vehicle for this interaction is one which unifies:

- the synchronization
- the shared data
- the body of code which performs the accesses

The Mesa monitor facility allows considerable flexibility in its use. Before getting into the details, let us first look at a slightly over-simplified description of the mechanism and a simple example. The remainder of this section deals with the basics of monitors (more complex uses are described in section 10.4); WAIT and NOTIFY are described in section 10.3.

10.2.1. An overview of Monitors

A *monitor* is a module instance. It thus has its own data in its global frame, and its own procedures for accessing that data. Some of the procedures are public, allowing calls into the monitor from outside. Obviously, conflicts could arise if two processes were executing in the same monitor at the same time. To prevent this, a *monitor lock* is used for mutual exclusion (i.e. to insure that only one process may be in each monitor at any one time). A call into a monitor (to an *entry procedure*) implicitly acquires its lock (waiting if necessary), and returning from the monitor releases it. The monitor lock serves to guarantee the integrity of the global data, which is expressed as the *monitor invariant* -- i.e an assertion defining what constitutes a "good state" of the data for that particular monitor. It is the responsibility of *every* entry procedure to restore the monitor invariant before returning, for the benefit of the next process entering the monitor.

Things are complicated slightly by the possibility that one process may enter the monitor and find that the monitor data, while in a good state, nevertheless indicates that that process cannot continue until some other process enters the monitor and improves the situation. The WAIT operation allows the first process to release the monitor lock and await the desired condition. The WAIT is performed on a *condition variable*, which is associated by agreement with the actual condition needed. When another process makes that condition true, it will perform a NOTIFY on the condition variable, and the waiting process will continue from where it left off (after reacquiring the lock, of course.)

For example, consider a fixed block storage allocator providing two entry procedures: *Allocate* and *Free.* A caller of *Allocate* may find the free storage exhausted and be obliged to wait until some caller of *Free* returns a block of storage.

```
StorageAllocator: MONITOR =
    BEGIN
    StorageAvailable: CONDITION;
    FreeList: POINTER;

    Allocate: ENTRY PROCEDURE RETURNS [p: POINTER] =
        BEGIN
        WHILE FreeList = NIL DO
            WAIT StorageAvailable
            ENDLOOP;
        p ← FreeList; FreeList ← p.next;
        END;

    Free: ENTRY PROCEDURE [p: POINTER] =
        BEGIN
        p.next ← FreeList; FreeList ← p;
        NOTIFY StorageAvailable
        END;
    END.
```

Note that it is clearly undesirable for two asynchonous processes to be executing in the *StorageAllocator* at the same time. The use of entry procedures for *Allocate* and *Free* assures mutual exclusion. The monitor lock is released while WAITing in *Allocate* in order to allow *Free* to be called (this also allows other processes to call *Allocate* as well, leading to several processes waiting on the queue for *StorageAvailable*).

10.2.2. Monitor Locks

The most basic component of a monitor is its *monitor lock.* A monitor lock is a predefined type, which can be thought of as a small record:

```
MONITORLOCK: TYPE =PRIVATE RECORD [locked: BOOLEAN, queue: Queue];
```

The monitor lock is private; its fields are never accessed explicitly by the Mesa programmer. Instead, it is used implicitly to synchronize entry into the monitor code, thereby authorizing access to the monitor data (and in some cases, other resources, such as I/O devices, etc.) The next section describes several kinds of monitors which can be constructed from this basic mechanism. In all of these, the idea is the same: during entry to a monitor, it is necessary to acquire the monitor lock by:

1. waiting (in the queue) until: *locked* = FALSE,
2. setting: *locked* ← TRUE.

10.2.3. Declaring monitor modules, ENTRY *and* INTERNAL *procedures*

In addition to a collection of data and an associated lock, a monitor contains a set of procedure that do operations on the data. *Monitor modules* are declared much like program or definitions modules; for example:

```
M: MONITOR [arguments] =
  BEGIN
  . . .
  END.
```

The procedures in a monitor module are of three kinds:

Entry procedures

Internal procedures

External procedures

Every monitor has one or more *entry* procedures; these acquire the monitor lock when called, and are declared as:

```
P: ENTRY PROCEDURE [arguments] =. . .
```

The entry procedures will usually comprise the set of public procedures visible to clients of the monitor module. (There are some situations in which this is not the case; see external procedures, below). The usual Mesa default rules for PUBLIC and PRIVATE procedures apply.

Many monitors will also have *internal* procedures: common routines shared among the several entry procedures. These execute with the monitor lock held, and may thus freely access the monitor data (including condition variables) as necessary. Internal procedures should be private, since direct calls to them from outside the monitor would bypass the acquisition of the lock (for monitors implemented as multiple modules, this is not quite right; see section 10.4, below). internal procedures can be called only from an entry procedure or another internal procedure. They are declared as follows:

```
Q: INTERNAL PROCEDURE [arguments] =. . .
```

The attributes ENTRY or INTERNAL may be specified on a procedure only in a monitor module. Section 10.2.4 describes how one declares an interface for a monitor.

Some monitor modules may wish to have *external* procedures. These are declared as normal non-monitor procedures:

```
R: INTERNAL PROCEDURE [arguments] =. . .
```

Such procedures are logically outside the monitor, but are declared within the same module for reasons of logical packaging. For example, a public external procedure might do some preliminary processing and then make repeated calls into the monitor proper (via a private entry procedure) before returning to its client. Being outside the monitor, an external procedure must *not* reference any monitor data (including condition variables), nor call any internal procedures. The compiler checks for calls to internal procedures and usage of the condition variable operations (WAIT, NOTIFY, etc.) within external procedures, but does not check for accesses to monitor data.

A fine point:

Actually, unchanging read-only global variables *may* be accessed by external procedures; it is changeable monitor data that is strictly off-limits.

Generally speaking, a chain of procedure calls involving a monitor module has the general form:

Client procedure -- outside module
↓
External procedure(s) -- inside module but outside monitor
↓
Entry procedure -- inside monitor
↓
Internal procedure(s) -- inside monitor

Any deviation from this pattern is likely to be a mistake. A useful technique to avoid bugs and increase the readibility of a monitor module is to structure the source text in the corresponding order:

```
M: MONITOR =
   BEGIN
   <External procedures>
   <Entry procedures>
   <Internal procedures>
   <Initialization (main-body) code>
   END.
```

10.2.4. Interfaces to monitors

In Mesa, the attributes ENTRY and INTERNAL are associated with a procedure's body, not with its type. Thus they cannot be specified in a DEFINITIONS module. Typically, internal procedures are not exported anyway, although they may be for a multi-module monitor (see section 10.4.4). In fact, the compiler will issue a warning when the combination PUBLIC INTERNAL occurs.

From the client side of an interface, a monitor appears to be a normal program module, hence the keywords MONITOR and ENTRY do not appear. For example, a monitor *M* with entry procedures *P* and *Q* might appear as:

```
MDefs: DEFINITIONS =
    BEGIN
    M: PROGRAM [arguments];
    P, Q: PROCEDURE [arguments] RETURNS [results];
    .
    .
    END.
```

10.2.5. Interactions of processes and monitors

One interaction should be noted between the process spawning and monitor mechanisms as defined so far. If a process executing within a monitor forked to an internal procedure of the same monitor, the result would be two processes inside the monitor at the same time, which is the exact situation that monitors are supposed to avoid. The following rule is therefore enforced:

> A FORK may have as its target any procedure *except an internal procedure of a monitor.*

A fine point:

> In the case of a multi-module monitor (see section 10.4.4) calls to other monitor procedures through an interface cannot be checked for the INTERNAL attribute, since this information is not available in the interface (see section 10.2.4).

10.3. Condition Variables

Condition variables are declared as:

```
c: CONDITION;
```

The content of a condition variable is private to the process mechanism; condition variables may be accessed only via the operations defined below. It is important to note that it is the condition *variable* which is the basic construct; a condition (i.e. the contents of a condition variable) should *not* itself be thought of as a meaningful object; it may *not* be assigned to a condition variable, passed as a parameter, etc.

10.3.1. Wait, Notify, and Broadcast

A process executing in a monitor may find some condition of the monitor data which forces it to wait until another process enters the monitor and improves the situation. This can be accomplished using a condition variable, and the three basic operations: WAIT, NOTIFY, and

BROADCAST, defined by the following syntax:

```
Statement        ::= ... | WaitStmt | NotifyStmt
WaitStmt         ::= WAIT Variable OptCatchPhrase
NotifyStmt       ::= NOTIFY Variable | BROADCAST Variable
```

A condition variable *c* is always associated with some Boolean expression describing a desired state of the monitor data, yielding the general pattern:

Process waiting for condition:

```
WHILE ~BooleanExpression DO
  WAIT c
  ENDLOOP;
```

Process making condition true:

```
make BooleanExpression true; -- i.e. as side effect of modifying global data
NOTIFY c;
```

Consider the storage allocator example from section 10.2.1. In this case, the desired **BooleanExpression** is "*FreeList* # NIL". There are several important points regarding WAIT and NOTIFY, some of which are illustrated by that example:

WAIT always releases the lock while waiting, in order to allow entry by other processes, including the process which will do the NOTIFY (e.g. *Allocate* must not lock out the caller of *Free* while waiting, or a deadlock will result). Thus, the programmer is always obliged to restore the monitor invariant (return the monitor data to a "good state") before doing a WAIT.

NOTIFY, on the other hand, retains the lock, and may thus be invoked *without* restoring the invariant; the monitor data may be left in in an arbitrary state, so long as the invariant is restored before the next time the lock is released (by exiting an entry procedure, for example).

A NOTIFY directed to a condition variable on which no one is waiting is simply discarded. Moreover, the built-in test for this case is more efficient than any explicit test that the programmer could make to avoid doing the extra NOTIFY. (Thus, in the example above, *Free* always does a NOTIFY, without attempting to determine if it was actually needed.)

Each WAIT *must* be embedded in a loop checking the corresponding condition. (E.g. *Allocate*, upon being notified of the *StorageAvailable* condition, still loops back and tests again to insure that the freelist is actually non-empty.) This rechecking is necessary because the condition, even if true when the NOTIFY is done, may become false again by the time the awakened process gets to run. (Even though the freelist is always non-empty when *Free* does its NOTIFY, a third process could have called *Allocate* and emptied the freelist before the waiting process got a chance to inspect it.)

Given that a process awakening from a WAIT must be careful to recheck its desired condition, the process doing the NOTIFY can be somewhat more casual about insuring that the condition is actually true when it does the NOTIFY. This leads to the notion of a *covering condition variable*, which is notified whenever the condition desired by the waiting process is *likely* to be true; this approach is useful if the expected cost of false alarms (i.e. extra wakeups that test the condition and wait again) is lower than the cost of having the notifier always know precisely what the waiter is waiting for.

The last two points are somewhat subtle, but quite important; condition variables in Mesa act as *suggestions* that their associated Boolean expressions are likely to be true and should therefore be rechecked. They do *not* guarantee that a process, upon awakening from a WAIT, will necessarily find the condition it expects. The programmer should never write code which implicitly assumes the truth of some condition simply because a NOTIFY has occurred.

It is often the case that the user will wish to notify *all* processes waiting on a condition variable. This can be done using:

```
BROADCAST c;
```

This operation can be used when several of the waiting processes should run, or when *some* waiting process should run, but not necessarily the head of the queue.

Consider a variation of the *StorageAllocator* example:

```
StorageAllocator: MONITOR =
    BEGIN
    StorageAvailable: CONDITION;
    . . .

    Allocate: ENTRY PROCEDURE [size: CARDINAL] RETURNS [p: POINTER] =
        BEGIN
        UNTIL <storage chunk of size words is available> DO
            WAIT StorageAvailable
            ENDLOOP;
        p ← <remove chunk of size words>;
        END;

    Free: ENTRY PROCEDURE [p: POINTER, size: CARDINAL] =
        BEGIN
        . . .
        <put back storage chunk of size words>
        . . .
        BROADCAST StorageAvailable
        END;
    END.
```

In this example, there may be several processes waiting on the queue of *StorageAvailable*, each with a different *size* requirement. It is not sufficient to simply NOTIFY the head of the queue, since that process may not be satisfied with the newly available storage while another waiting process might be. This is a case in which BROADCAST is needed instead of NOTIFY.

An important rule of thumb: *it is always correct to use a* BROADCAST. NOTIFY should be used instead of BROADCAST if *both* of the following conditions hold:

It is expected that there will typically be several processes waiting in the condition variable queue (making it expensive to notify all of them with a BROADCAST), and

It is known that the process at the head of the condition variable queue will always be the right one to respond to the situation (making the multiple notification unnecessary);

If both of these conditions are met, a NOTIFY is sufficient, and may represent a significant efficiency improvement over a BROADCAST. The allocator example in section 10.2.1 is a situation in which NOTIFY is preferrable to BROADCAST.

As described above, the condition variable mechanism, and the programs using it, are intended to be robust in the face of "extra" NOTIFYS. The next section explores the opposite problem: "missing" NOTIFYS.

10.3.2. Timeouts

One potential problem with waiting on a condition variable is the possibility that one may wait "too long." There are several ways this could happen, including:

- Hardware error (e.g. "lost interrupt")
- Software error (e.g. failure to do a NOTIFY)
- Communication error (e.g. lost packet)

To handle such situations, waits on condition variables are allowed to *time out*. This is done by associating a *timeout interval* with each condition variable, which limits the delay that a process can experience on a given WAIT operation. If no NOTIFY has arrived within this time interval, one will be generated automatically. The Mesa language does not currently have a facility for setting the timeout field of a CONDITION variable. See the runtime documentation for the description of the system procedure provided for this operation.

The waiting process will perceive this event as a normal NOTIFY. (Some programs may wish to distinguish timeouts from normal NOTIFYS; this requires checking the time as well as the desired condition on each iteration of the loop.)

No facility is provided to time out waits for monitor locks. This is because there would be, in general, no way to recover from such a timeout.

10.4. More about Monitors

The next few sections deal with the full generality of monitor locks and monitors.

10.4.1. The LOCKS Clause

Normally, a monitor's data comprises its global variables, protected by the special global variable *LOCK*:

```
LOCK: MONITORLOCK;
```

This implicit variable is declared automatically in the global frame of any module whose heading is of the form:

```
M: MONITOR [arguments] IMPORTS . . . EXPORTS . . . =
```

In such a monitor it is generally not necessary to mention *LOCK* explicitly at all. For more general use of the monitor mechanism, it is necessary to declare at the beginning of the monitor module exactly which MONITORLOCK is to be acquired by entry procedures. This declaration appears as part of the program type constructor that is at the head of the module. The syntax is as follows:

```
ProgramTC      ::= ... | MONITOR ParameterList ReturnsClause LocksClause
LocksClause    ::= empty | LOCKS Expression |
                   LOCKS Expression USING identifier : TypeSpecification
```

If the **LocksClause** is empty, entry to the monitor is controlled by the distinguished variable *LOCK* (automatically supplied by the compiler). Otherwise, the **LocksClause** must designate a variable of type MONITORLOCK, a record containing a distinguished lock field (see section 10.4.2), or a pointer that can be dereferenced (perhaps several times) to yield one of the preceding. If a **LocksClause** is present, the compiler does not generate the variable *LOCK*.

If the USING clause is absent, the lock is located by evaluating the LOCKS expression in the context of the monitor's main body; i.e., the monitor's parameters, imports, and global variables are visible, as are any identifiers made accessible by a global OPEN. Evaluation occurs upon entry to, and again upon exit from, the entry procedures (and for any WAITs in entry or internal procedures). The location of the designated lock can thus be affected by assignments within the procedure to variables in the LOCKS expression. To avoid disaster, it is essential that each reevaluation yield a designator of the same MONITORLOCK. This case is described further in section 10.4.4.

If the USING clause is present, the lock is located in the following way: every entry or internal procedure must have a parameter with the same identifier and a compatible type as that specified in the USING clause. The occurrences of that identifier in the LOCKS clause are bound to that procedure parameter in every entry procedure (and internal procedure doing a WAIT). The same care is necessary with respect to reevaluation; to emphasize this, the distinguished argument is treated as a read-only value within the body of the procedure. See section 10.4.5 for further details.

10.4.2. Monitored Records

For situations in which the monitor data cannot simply be the global variables of the monitor module, a *monitored record* can be used:

```
r: MONITORED RECORD [x: INTEGER, . . . ];
```

A monitored record is a normal Mesa record, except that it contains an automatically declared field of type MONITORLOCK. As usual, the monitor lock is used implicitly to synchronize entry into the monitor code, which may then access the other fields in the monitored record. The fields of the monitored record must *not* be accessed except from within a monitor which first acquires its lock. In analogy with the global variable case, the monitor lock field in a monitored record is given the special name *LOCK*; generally, it need not be referred to explicitly (except during initialization; see section 10.6).

A fine point:

A more general form of monitor lock declaration is discussed in section 10.4.6

CAUTION: If a monitored record is to be passed around (e.g. as an argument to a procedure) this should always be done by reference using a POINTER TO MONITORED RECORD. Copying a monitored record (e.g. passing it by value) will generally lead to chaos.

10.4.3. Monitors and module instances

Even when all the procedures of a monitor are in one module, it is not quite correct to think of the module and the monitor as identical. For one thing, a monitor module, like an ordinary program module, may have several instances. In the most straightforward case, each instance constitutes a separate monitor. More generally, through the use of monitored records, the number of monitors may be larger or smaller than the number of instances of the corresponding module(s). The crucial observation is that in all cases:

> *There is a one-to-one correspondence between monitors and monitor locks.*

The generalization of monitors through the use of monitored records tends to follow one of two patterns:

> *Multi-module monitors*, in which several module instances implement a single monitor.
>
> *Object monitors*, in which a single module instance implements several monitors.

A fine point:

> These two patterns are *not* mutually exclusive; multi-module object monitors are possible, and may occasionally prove necessary.

10.4.4. Multi-module monitors

In implementing a monitor, the most obvious approach is to package all the data and procedures of the monitor within a single module instance (if there are multiple instances of such a module, they constitute separate monitors and share nothing except code.) While this will doubtless be the most common technique, the monitor may grow too large to be treated as a single module.

Typically, this leads to multiple modules. In this case the mechanics of constructing the monitor are changed somewhat. There must be a central location that contains the monitor lock for the monitor implemented by the multiple modules. This can be done either by using a MONITORED RECORD or by choosing one of the modules to be the "root" of the monitor. Consider the following example:

```
BigMonRoot: MONITOR IMPORTS . . . EXPORTS . . . =
  BEGIN
  monitorDatum1: . . .
  monitorDatum2: . . .
  . . .
  p1: PUBLIC ENTRY PROCEDURE . . .
  . . .
  END.

BigMonA: MONITOR
  LOCKS root     -- could equivalently say root.LOCK
  IMPORTS root: BigMonRoot . . . EXPORTS . . . =
  BEGIN
  . . .
```

```
    p2: PUBLIC ENTRY PROCEDURE . . .
        x ← root.monitorDatum1; -- access the protected data of the monitor
    . . .
    END.

BigMonB: MONITOR
    LOCKS root
    IMPORTS root: BigMonRoot . . . EXPORTS . . . =
    BEGIN OPEN root;
    . . .
    p3: PUBLIC ENTRY PROCEDURE . . .
        monitorDatum2 ← . . .; -- access the protected data via an OPEN
    . . .
    END.
```

The monitor *BigMon* is implemented by three modules. The modules *BigMonA* and *BigMonB* have a LOCKS clause to specify the location of the monitor lock: in this case, the distinguished variable *LOCK* in *BigMonRoot.* When any of the entry procedures *p*1, *p*2, or *p*3 is called, this lock is acquired (waiting if necessary), and is released upon returning. The reader can verify that no two independent processes can be in the monitor at the same time.

Another means of implementing multi-module monitors is by means of a MONITORED RECORD. Use of OPEN allows the fields of the record to be referenced without qualification. Such a monitor is written as:

```
MonitorData: TYPE =MONITORED RECORD [x: INTEGER, . . . ];

MonA: MONITOR [pm: POINTER TO MonitorData]
LOCKS pm
IMPORTS . . .
EXPORTS . . . =
BEGIN OPEN pm;
P: ENTRY PROCEDURE [. . .] =
    BEGIN
    . . .
    x ← x+1; -- access to a monitor variable
    . . .
    END;
. . .
END.
```

The LOCKS clause in the heading of this module (and each other module of this monitor) leads to a MONITORED RECORD. Of course, in all such multi-module monitors, the LOCKS clause will involve one or more levels of indirection (POINTER TO MONITORED RECORD, etc.) since passing a monitor lock by value is not meaningful. As usual, Mesa will provide one or more levels of automatic dereferencing as needed.

More generally, the target of the LOCKS clause can evaluate to a MONITORLOCK (i.e. the example above is equivalent to writing "LOCKS *pm.LOCK*").

CAUTION: The meaning of the target expression of the LOCKS clause *must not change* between the call to the entry procedure and the subsequent return (i.e. in the above example, changing *pm* would invariably be an error) since this would lead to a different monitor lock being released than was acquired, resulting in total chaos.

There are a few other issues regarding multi-module monitors which arise any time a tightly coupled piece of Mesa code must be split into multiple module instances and then spliced back together. For example:

> If the lock is in a MONITORED RECORD, the monitor data will probably need to be in the record also. While the global variables of such a multi-module monitor are covered by the monitor lock, they do *not* constitute monitor data in the normal sense of the term, since they are not uniformly visible to all the module instances.
>
> Making the internal procedures of a multi-instance monitor PRIVATE will not work if one instance wishes to call an internal procedure in another instance. (Such a call is perfectly acceptable so long as the caller already holds the monitor lock). Instead, a second interface (hidden from the clients) is needed as part of the "glue" holding the monitor together. Note however, that Mesa cannot currently check that the procedure being called through the interface is an internal one (see section 10.2.4).
>
> A fine point:
>
> The compiler will complain about the PUBLIC INTERNAL procedures, but this is just a warning.

10.4.5. Object monitors

Some applications deal with *objects,* implemented, say, as records named by pointers. Often it is necessary to insure that operations on these objects are *atomic*, i.e., once the operation has begun, the object will not be otherwise referenced until the operation is finished. If a module instance provides operations on some class of objects, the simplest way of guaranteeing such atomicity is to make the module instance a monitor. This is logically correct, but if a high degree of concurrency is expected, it may create a bottleneck; it will serialize the operations on *all* objects in the class, rather than on *each* of them individually. If this problem is deemed serious, it can be solved by implementing the objects as monitored records, thus effectively creating a separate monitor for each object. A single module instance can implement the operations on all the objects as entry procedures, each taking as a parameter the object to be locked. The locking of the parameter is specified in the module heading via a **LocksClause** with a USING clause. For example:

```
ObjectRecord: TYPE = MONITORED RECORD [ . . . ];

ObjectHandle: TYPE = POINTER TO ObjectRecord;

ObjectManager: MONITOR [arguments]
    LOCKS object USING object: ObjectHandle
    IMPORTS . . .
    EXPORTS . . . =
    BEGIN
    Operation: PUBLIC ENTRY PROCEDURE [object: ObjectHandle, . . . ] =
        BEGIN
        . . .
        END;
    . . .
    END.
```

Note that the argument of USING is evaluated in the scope of the arguments to the entry procedures, rather than the global scope of the module. In order for this to make sense, each entry procedure, and each internal procedure that does a WAIT, must have an argument which matches exactly the name and type specified in the USING subclause. All other components of the argument of LOCKS are evaluated in the global scope, as usual.

As with the simpler form of LOCKS clause, the target may be a more complicated expression and/or may evaluate to a monitor lock rather than a monitored record. For example:

```
LOCKS p.q.LOCK USING p: POINTER TO ComplexRecord . . .
```

CAUTION: Again, the meaning of the target expression of the LOCKS clause *must not change* between the call to the entry procedure and the subsequent return. (I.e. in the above example, changing *p* or *p.q* would almost surely be an error.)

CAUTION: It is important to note that global variables of object monitors are very dangerous; they are *not* covered by a monitor lock, and thus do *not* constitute monitor data. If used at all, they must be set only at module initialization time and must be read-only thereafter.

10.4.6. Explicit declaration of monitor locks

It is possible to declare monitor locks explicitly:

```
myLock: MONITORLOCK;
```

The normal cases of monitors and monitored records are essentially stylized uses of this facility via the automatic declaration of *LOCK*, and should cover all but the most obscure situations. For example, explicit delarations are useful in defining MACHINE DEPENDENT monitored records. (Note that the LOCKS clause becomes mandatory when an explicitly declared monitor lock is used.) More generally, explicit declarations allow the programmer to declare records with several monitor locks, declare locks in local frames, and so on; this flexibility can lead to a wide variety of subtle bugs, hence use of the standard constructs whenever possible is strongly advised.

10.5. Signals

10.5.1. Signals and Processes

Each process has its own call stack, down which signals propagate. If the signaller scans to the bottom of the stack and finds no catch phrase, the signal is propagated to the debugger. The important point to note is that forking to a procedure is different from calling it, in that the forking creates a gap across which signals cannot propagate. This implies that in practice, one cannot casually fork to any arbitrary procedure. The only suitable targets for forks are procedures which catch any signals they incur, and which never generate any signals of their own.

10.5.2. Signals and Monitors

Signals require special attention within the body of an entry procedure. A signal raised with the monitor lock held will propagate without releasing the lock and possibly invoke arbitrary computations. For errors, this can be avoided by using the RETURN WITH ERROR construct.

```
RETURN WITH ERROR NoSuchObject;
```

Recall from Chapter 8 that this statement has the effect of removing the currently executing frame from the call chain before issuing the ERROR. If the statement appears within an entry procedure, the monitor lock is released before the error is started as well. Naturally, the monitor invariant must be restored before this operation is performed.

For example, consider the following program segment:

```
Failure: ERROR [kind: CARDINAL] = CODE;

Proc: ENTRY PROCEDURE [. . .] RETURNS [c1, c2: CHARACTER] =
   BEGIN
   ENABLE UNWIND ⇒ . . .
   . . .
   IF cond1 THEN ERROR Failure[1];
   IF cond2 THEN RETURN WITH ERROR Failure[2];
   . . .
   END;
```

Execution of the construct ERROR *Failure*[1] raises a signal that propagates until some catch phrase specifies an exit. At that time, unwinding begins; the catch phrase for UNWIND in *Proc* is executed and then *Proc*'s frame is destroyed. Within an entry procedure such as *Proc*, the lock is held until the unwind (and thus through unpredictable computation performed by catch phrases).

Execution of the construct RETURN WITH ERROR *Failure*[2] releases the monitor lock and destroys the frame of *Proc* before propagation of the signal begins. Note that the argument list in this construct is determined by the declaration of *Failure* (not by *Proc*'s RETURNS clause). The catch phrase for UNWIND is not executed in this case. The signal *Failure* is actually raised by the system, after which *Failure* propagates as an ordinary error (beginning with *Proc*'s caller).

When the RETURN WITH ERROR construct is used from within an internal procedure, the monitor lock is *not* released; RETURN WITH ERROR will release the monitor lock in precisely those cases that RETURN will.

Another important issue regarding signals is the handling of UNWINDS; any entry procedure that may experience an UNWIND must catch it and clean up the monitor data (restore the monitor invariant):

```
P: ENTRY PROCEDURE [ ... ] =
   BEGIN ENABLE UNWIND ⇒ BEGIN <restore invariant> END;
   .
   .
   .
   END;
```

At the end of the UNWIND catchphrase, the compiler will append code to release the monitor lock before the frame is unwound. It is important to note that a monitor always has at least one cleanup task to perform when catching an UNWIND signal: the monitor lock must be released. To this end, the programmer should be sure to place an enable-clause on the body of every entry procedure that might evoke an UNWIND (directly or indirectly). If the monitor invariant is already satisfied, no further cleanup need be specified, but *the null catch-phrase must be written* so that the compiler will generate the code to unlock the monitor:

```
BEGIN ENABLE UNWIND ⇒ NULL;
```

This should be omitted *only* when it is certain that no UNWINDS can occur.

Another point is that signals caught by the **OptCatchPhrase** of a WAIT operation should be thought of as occurring after reacquisition of the monitor lock. Thus, like all other monitor code, catch phrases within a monitor are always executed with the monitor lock held.

10.6. Initialization

When a new monitor comes into existence, its monitor data will generally need to be set to some appropriate initial values; in particular, the monitor lock and any condition variables must be initialized. As usual, Mesa takes responsibility for initializing the simple common cases; for the cases not handled automatically, it is the responsibility of the programmer to provide appropriate initialization code, and to arrange that it be executed at the proper time. The two types of initialization apply in the following situations:

Monitor data in global variables can be initialized using the normal Mesa initial value constructs in declarations. Monitor locks and condition variables in the global frame will also be initialized automatically (although in this case, the programmer does not write any explicit initial value in the declaration).

Monitor data in records must be initialized by the programmer. System procedures must be used to initialize the monitor lock and condition variables. See the runtime documentation for the descriptions of appropriate procedures.

A fine point:

If a variable containing a record is declared in a frame, it is normally possible to initialize it in the declaration (i.e. using a constructor as the initial value); however, this does *not* apply if the record contains monitor locks or condition variables, which must be initialized via calls to system procedures.

Since initialization code modifies the monitor data, it must have exclusive access to it. The programmer should insure this by arranging that the monitor not be called by its client processes until it is ready for use.

Inter-Office Memorandum

To	Mesa Users	Date	May 31, 1978
From	John Wick	Location	Palo Alto
Subject	Mesa 4.0 Binder Update	Organization	SDD/SD

XEROX

Filed on: [IRIS]<MESA>DOC>BINDER40.BRAVO

This memo outlines changes made in the Mesa binder since the last release (October 17, 1977). (In addition, the list of change requests closed by Mesa 4.0 will appear as part of the Software Release Description.)

Except for the internal BCD file format, there are no known incompatabilities with the Mesa 3.0 binder. No changes to existing configuration descriptions are required; but because of the file format change, all configurations must be rebound.

If you are not concerned with the new features described in the major headings below, and you want to get on with Mesa 4.0, skip the rest of this memo for now, and come back to it later.

Code Packing

It is now possible to pack together the code for several modules into a single segment. This is useful for two reasons:

> Since the code is allocated an integral number of pages, there is some wasted space in the last page ("breakage"). If several modules are combined into a single segment, the breakage is amortized over all the modules, and there is less waste on the average.
>
> All the modules will be brought into and out of memory together, as a unit; a reference to any module in the pack will cause all the code to be brought in. Modules which are tightly coupled dynamically are good candidates for packing (for example, resident code should probably always be packed).

Of course, it is possible to "over pack" a configuration; the segments might become so large that there will never be room in memory for more than one of them at a time (this should remind you of an overlay system). *Packing is a tradeoff, and should be used with caution.*

Syntax

The segments are specified at the beginning of the configuration by giving a list of the modules which comprise each one. Any number of PACK statements may appear. The scope of the packing specification is the whole configuration, and not subconfigurations or individual module instances, because there is at most one copy of a module's code in any configuration (if all goes well).

```
ConfigDescription  ::=  Directory Packing Configuration .
Packing            ::=  empty | PackSeries ;
PackSeries         ::=  PackList | PackSeries ; PackList
PackList           ::=  PACK IdList
```

Each **PackList** defines a single segment; the code for all the modules in the **IdList** will be packed into it. The identifiers in the **IdList** must refer to modules in the configuration, and not to module instances; it is the code and not the global frames that are being packed (the frames are always packed when they are allocated by the loader).

It is illegal to specify the same module in more than one **PackList**. Even though there may be multiple instances of the module (i.e., multiple global frames) in the configuration, the code is shared by all of them, and therefore can only appear in one pack.

Finally, it is perfectly fine to reach inside a previously bound configuration that is being instantiated and single out some or all of its modules for packing. Of course, you must know something about the structure of that configuration in order to do this.

Restrictions

Obviously, the PACK statements apply only if the code is being moved to the output file; otherwise, the pack lists are ignored (and no warning message is given). This allows the programmer to debug the configuration without shuffling the code from file to file, thereby saving time. When making the final version, the packing can be effected with a binder switch, without having to modify the source of the configuration description.

Once some modules have been packed together, they cannot be taken apart and repacked with other modules later on, when they are bound into some other configuration.

Fine point:

> If a previously bound configuration contains a pack, referencing any module of the pack gets the whole thing. So it is possible to pack a module and a pack together, or even to pack two packs. It is never possible to unpack a pack.

In general, code packing should be specified only to the extent that no unpacking will ever be desired. Once the packing is done, it can't be undone, unless you start over with the individual modules.

External Links

In previous Mesa systems, links to the externals referenced by a program (imported procedures, signals, errors, frames, and programs) were always stored in the module's global frame. This allows each instance of a module to be bound differently, and it allows binding to be done at runtime without modification of the module's code segment. However, it has two drawbacks:

> The links are only referenced by the module's code, and are therefore not needed when the code is swapped out. Hence, the links logically belong in the code segment.
>
> If two instances of a module are bound identically (the usual case), the links must be stored twice.

Fine Point:

> To determine the amount of space required for external links, see the compiler's typescript file. Each link occupies one word.

The Mesa 4.0 binder therefore optionally places links in the code segment. This option is enabled by constructs in the configuration language, and is further controlled by binder and loader command modifiers (switches).

Syntax

For each component of a configuration, the link location is specified using the LINKS construct defined below. The default is frame links, as in Mesa 3.0.

```
Links          ::= empty | LINKS : CODE | LINKS : FRAME
```

A link specification can optionally be attached to each instantiation of a module, overriding the current default, so that the link location can be different for each instance.

```
CRightSide     ::= Item Links | Item [ ] Links | Item [ IdList ] Links
```

Alternately, the link option can be specified in the configuration header. This merely changes the default option for the configuration; it will apply to all components (including nested configurations) unless it is explicitly overridden.

```
CHead          ::= CONFIGURATION Links Imports CExports ControlClause
```

This construction works much like the PUBLIC / PRIVATE options in Mesa, and it nests in the same way. A link option attached to a configuration changes the default for all components within it, but that default can be overriden for a particular module (or nested configuration) by specifying a different link option.

Restrictions

This scheme has the consequence that, *if a module with code links has multiple instances, each instance must be bound the same.* For example, it is usually not meaningful to specify code links if the code is shared by frames residing in several different Main Data Spaces.

As with code packing, the code links option takes effect only when the code is being moved to the output file. At this point, the binder will make room for the links as it copies the code if any module sharing that code has requested code links. Again, this allows a programmer to debug without the expense of moving the code (using frame links), and then to effect the code links option with a binder switch, without changing the source of the configuration description.

Fine point:

> Once space for code links has been added to a configuration, it cannot be undone by a later binding. On the other hand, space for code links can always be added to a (previously bound) configuration, even if it did not specify code links in its description.

Using code links has one drawback: it slows down the binding and loading process, as the code must be swapped in and rewritten. The binder must make room in the code segment for the links, as described above. And because the loader resolves imports of previously loaded modules, as well as the imports of the module being loaded, it may have to swap in (and perhaps update and swapout) the code segment for every module in the system.

Fine point:

> In an experiment, it took about 2.5 seconds to load a medium size configuration (Mesa itself) with frame links (this includes a fixed overhead of about 1.5 seconds for a directory search). With code links, factoring out the fixed overhead, it was almost eight times longer (but it still took only nine seconds).

Finally, the loader will not automatically attempt to use code links, even if the space is available in the code segment. A loader switch ("l") must be used to effect this option.

Context Switching

The command line switch /R (for run) is used to specify that the Binder should run some other program rather than returning to the Alto Executive. Both ".image" and ".run" files may be specified. If there is no explicit extension, ".image" is assumed. Any switches after the R and any other text remaining in the command line after the file with the /R switch will be copied to Com.Cm for inspection by the new program.

Examples:

> "Binder SomeConfig/g Mesa/r SomeConfig" will bind SomeConfig and then run Mesa.image as if you had typed "Mesa SomeConfig".
>
> "Binder SomeConfig/g Mesa/rd OtherConfig/-s SomeConfig" will bind SomeConfig and then run Mesa.image as if you had typed "Mesa/d OtherConfig/-s SomeConfig"
>
> "Binder SomeConfig/g Ftp.run/r Store SomeConfig.bcd" will bind SomeConfig and then run Ftp.run as if you had typed "Ftp.run Store SomeConfig.bcd"

Fine points:

> The last specification before the file with the /R switch must have the /G (go) switch to indicate the end of the previous command.
>
> You can run Bravo using the /R switch, but the current version (7.1) will not correctly find switches or arguments on the command line.

Error Messages

The binder's error messages have been improved substantially. Each message includes the corresponding source line of the configuration description (if available), and more information from the data base is available for most common errors.

Fatal errors are now reported in a fashion similar to the compiler; the signal and message are given in octal, and should be included in any change request reporting a fatal binder error.

Distribution:
Mesa Users
Mesa Group

Inter-Office Memorandum

To	Mesa Users	Date	May 31, 1978
From	John Wick	Location	Palo Alto
Subject	Mesa 4.0 System Update	Organization	SDD/SD

XEROX

Filed on: [IRIS]<MESA>DOC>SYSTEM40.BRAVO

This memo outlines changes made in the Mesa system code since the last release (October 17, 1977). It also dicusses a number of internal changes made in the system and the microcode; see also the Mesa 4.0 Microcode Update. (In addition, the list of change requests closed by Mesa 4.0 will appear as part of the Software Release Description.)

External Interfaces

Names in square brackets refer to sections of the *Mesa System Documentation* which has also been updated. More details can be found there and in other documentation accompanying this release.

Alto Reserved Locations

Mesa software now conforms to the most recent allocation of Alto reserved locations (*Alto: A Personal Computer System, Hardware Manual*, February, 1978, Appendix H). The only page one location reserved by Mesa is the disaster flag (location 456B).

Basic Mesa

Basic Mesa has been reclassified as released software. To facilitate development of special purpose systems, Basic Mesa no longer includes a keyboard handler; the procedure for adding one is described in the documentation on the Keyboard Package. [Section 3]

CheckPoint/ Restart

Procedures have been added for writing checkpoint image files, and the bootstrap loader has been modified to load them. Checkpoint files contain only the data (so creating them and loading them is fast); unlike **MakeImage**, **MakeCheckPoint** does not copy code or any other files to the image file. Note this means that none of the files referenced by the checkpoint can be updated or modified in any way. [Image Files]

Code Links

The loader has been extended to optionally write external links in the code rather than the global frame (assuming code links were specified in the configuration description). To effect this option, the loader /l switch must be used. Note that if a module calls for code links, loading it will be slower, as the code segment must be swapped in and rewritten. To decrease resident storage requirements, all standard Mesa systems are configured with links in the code. [Modules]

Convert

Due to extremely limited microcode space, the convert instruction is no longer part of the Mesa instruction set. It can be simulated with **BitBlt** (see the system display package). [Display]

Debugger Call

A method of invoking the debugger explicitly, without generating a signal, is now available; it causes minimal disruption of the current state of the debuggee. The inline **CallDebugger** is defined in **MiscDefs**. [Miscellaneous]

Deleting Configurations

The procedure **UnNewConfig** is now available for unloading a configuration from a running system. The configuration's global frames are deallocated and its code segments are released. In addition, all existing modules are checked for bindings to the configuration being unloaded. [Modules]

Display Package

A smaller display package, similar to the Bcpl version, is now standard. It supports a simple display oriented teletype-like interface and an optional typescript file (see **FontDefs** and **DisplayDefs**). The window package is available as a separate configuration (which is no longer supported); see *Window Package.* [Display Package]

File Lengths

File lengths and file length hints are no longer kept as a permanent part of each file object. A separate (and optional) length object is allocated only when the length of a file is requested. Since length objects are not required for files containing code segments, this substantially reduces the amount of resident object space required to handle a large number of BCDS. [File Package]

Free Storage Package

The free storage package has been modified so that it protects each zone (including the system supplied free storage heap) with a monitor. This enables several processes to share the heap. [Storage Management]

Interrupts

For compatability with the new process mechanism, a Nova interrupt now causes a "naked notify" to one of sixteen condition variables. Pointers to these condition variables are contained in fixed locations in page zero (see **ProcessDefs**). [Processes and Monitors]

Images

All symbol table references and options have been removed from the system, and the interface to **MakeImage** has been changed to reflect this (the **symbolsToImage** parameter has been dropped). The image file format has also been revised to support checkpoint/restart files (see above). [Image Files]

KeyStreams

The keyboard routines have been revised to utilize the new process mechanism; a condition variable is notified when characters are available in the current keystream. The "idle procedure" has been replaced by a WAIT on this condition variable. [Keyboard Package]

Memory Management

A number of options have been added to the memory management and swapping facilities. Unlocked read-only segments, such as fonts, are now swapped automatically (previously, this applied only to code segments). The interface to swapping procedures has been expanded to include an (optional) **AllocInfo** parameter, which provides more information about how the memory should be allocated. The new facilities are defined in **AllocDefs**; swap stratagies and swapping procedures are now defined there. [Segment Package]

Pause to Debugger

A switch has been added to the NEW command which will invoke the debugger as soon as a configuration has been loaded (in command line mode, before it is started). Also, holding down the control-swat keys while an image file is loaded will now work correctly (the debugger will be invoked as soon as possible). [Section 4]

Process Structure

A new process mechanism which supports monitors and condition variables, WAITs and NOTIFYs, and FORK and JOIN has been implemented. The BLOCK operation has been eliminated (a **Yield** procedure is available). Note that several refinements (and some revisions) of the original proposal (in the *Pilot Functional Specification*) have been made. A new chapter in the *Mesa Language Manual* provides complete documentation; additional facilities which are not part of the language are described in the system documention (and **ProcessDefs**). [Processes and Monitors]

Warning: In general, facilities provided by the system *are not protected* by monitors. Since Pilot will be available soon, we have not redesigned and retrofitted the Alto/Mesa system to support preemptive processes (other than simple interrupt routines, as before). Except for the free storage package (see above), system facilities shared by more than one process must be protected by a user supplied set of monitor entry procedures.

Run Image

A module is now available which will invoke a Mesa image file (or a Bcpl run file) from the Mesa environment, without returning to the Alto Executive. Any Mesa subsystem which supports command line input (e.g., the compiler or binder, or even the Mesa system itself) can be invoked considerably faster using this facility. [Image Files]

StreamIO

To make instantiating multiple instances of this module easier, **StreamIO** no longer takes parameters specifying the input and output streams. Procedures are available which override the default settings. [StreamIO Package]

StringDefs

To facilitate conversion from binary to alphanumeric data, **AppendNumber** and other related procedures have been added to **StringDefs**. **WordsForString** now expects a cardinal. Bcpl strings now use packed arrays. [String Package]

SystemDefs

A simplified interface to the new memory management facilities has been added in the form of two additional procedures: **AllocateResidentSegment** and **AllocateResidentPages**. [Storage Management]

TrapDefs

Documentation on a number of system generated traps is now available in a new section of the system document. Most traps are converted into signals of the same name: **StartFault**, **ControlFault**, **UnboundProcedure**, **StackError**, etc. [Traps]

UnNew

This procedure no longer supports the option of adding the module's frame to the free frame heap (it will be returned there only if it was allocated from there). Note that this procedure does not check for other modules bound to the one being deleted. *Beware of dangling references!* [Modules]

Unsigned Compare

The unsigned compare operation (USC) has been removed from **InlineDefs**. Use of the appropriate signed or unsigned comparison operators should be controlled by the type of the variables involved: CARDINAL (unsigned) or INTEGER (signed). See the *Mesa Language Manual* and the *Mesa 4.0 Compiler Update* for more information.

User Interface

Modules to be loaded into the standard Mesa system (and Basic Mesa) can now be specified on the command line (and in command files). A number of switches are available to control loading options; these are described in the *Mesa User's Handbook.*

VMnotFree

The signal **VMnotFree** was inadvertently respelled (it used to be **VMNotFree**). [Segment Package]

Window Package

The Mesa window package is no longer part of the standard Mesa system; it is available as a separate configuration. **WindEx** replaces **WManager** for optional use in the debugger. The definitions of **BitBlt** and **Convert** have been removed from **RectangleDefs**. Support for a blinking cursor has been added. [Window 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. For several of these items, furthur information can be founded in the *Mesa 4.0 Microcode Update.*

Alto/Mesa Microcode

The microcode has been completely rewritten to improve its execution speed. The major changes are: 1) several instructions must now be aligned on word boundaries and, 2) certain instructions (notably jumps) require the evaluation stack to be empty except for their operands. (As a side effect of the reorganization, new opcode numbers have been assigned to most instructions.) We have observed improvements in raw execution speed of 20-50%, depending on the dynamic instruction mix.

Alto File System

The hint in the DiskDescriptor containing the number of free disk pages is now maintained properly. The declarations describing Sys.Log have been deleted, since it is no longer supported by the Bcpl OS (versions 14 or later).

Alto Time Standard

The time conversion package is now part of standard system. **UnpackedTime** and **PackDT** have been extended to support GMT, time zones, and daylight savings time (for compatability with Bcpl OS versions 14 or greater).

Bcd Format

This structure has been revised to achieve a space reduction of about 10%. A segment table and name table have been added, as well as support for packed code segments and code links. Provision for a source version stamp has also been included.

BitBlt

BitBlts are now performed entirely in microcode, using the ROM subroutine. They are not only faster, but interruptable as well. **BitBltDefs** now contains the interface to this operation; it has been updated to include the extended memory option.

Cleanup Procedures

Adding a cleanup procedure must now specify the conditions under which the procedure should be called. Several cleanup procedures are no longer called on swapping to and from the debugger.

Warning: Since the interrupt key may preempt a process holding a monitor lock, cleanup procedures must not attempt to enter any monitor. This severly restricts the operations that can be safely performed by cleanup procedures.

Code Packing

All resident code now employs the packed code option implemented by the binder.

Code Segments

The format of code segments has been revised to accomodate the new options for handling external links (storing them in the code and storing them backwards from the frame or code base). Also note that when several modules are packed into the same code segment, only the LRU bit of the first module is examined by the swapper.

ControlDefs

The declarations of control links and local and global frames now use overlaid variant records. The **AV**, **SD**, and **GFT** have been preassigned constant values as in the PrincOps; the REGISTER construct has been revised accordingly (see also *Trap Parameters*). The assignment of System Data indicies is now contained in **SDDefs**.

Descriptor Instructions

The descriptor instructions (DESCB and DESCBS) described in the PrincOps have been implemented.

External Links

External links are now stored and indexed backwards from the global frame base (or code base); this eliminates the "effective" minimum frame size overhead of eighteen. The total number of external procedures, programs, signals, and errors per module must be less than 256.

Field Descriptors

The format of field descriptors has been revised to agree with the D0 design. The read field stack (RFS) and read field code (RFC) instructions have been implemented.

Frame Allocation

The ALLOC and FREE instructions are now implemented in microcode; thus the overhead for large (greater than five word) parameter and result records has been drastically reduced.

Global Frame Format

The global frame overhead has been reduced from ten to three words; all fields relating to the old binding scheme have been eliminated (see also *External Links* and *Main Body Procedure*).

Kernel Function Calls

Several new kernel functions have been added as a result of other extensions (see **SDDefs**). Most entries of public interest are now defined as inlines in definitions modules (e.g. **FrameDefs**, **LoaderDefs**). Provision has been made for all of the traps defined in the PrincOps.

Load State Format

A change in format has reduced the size of the load state substantially. The maximum number of BCDS which can be loaded into a single image file is now about forty.

Long Integers

Addition, subtraction, and comparison of long (32-bit) integers are now implemented in microcode. Multiplication and division are done by software (and are therefore slow).

Main Body Procedure

The main body of a module is now executed in a separate local frame, instead of using the global frame. This eliminates three words from the global frame overhead (the access link, saved pc, and return link).

Nil Pointers

To enable conversion and comparison of both long and short values of null pointers, the value of NIL has been changed.

Novacode interface

To accomodate the implementation of the process opcodes in Nova code (and the removal of block, convert, and bitblt), the interface to the Nova has been revised.

OSStaticDefs

The format of the OS statics region has been revised to reflect the changes in OS version 14 (see the *Alto Operating System Reference Manual*).

Pair Instructions

A number of the pair instructions described in the PrincOps have been implemented on the Alto (notably the RXLP, RILP, and RIGP families).

Processes and Monitors

Due to severe space limitations, all of the process/monitor opcodes (enter, wait, reenter, notify, broadcast, exit, and requeue) are implemented in Nova code, and therefore are considerably slower (relative to other instructions) than they will be on the D0.

Real Data Type

The compiler now generates KFCBs to perform real arithmetic. The SD contains entries for the following floating point operations: FADD, FSUB, FMUL, FDIV, FCOMP, FIX, FLOAT. Note, however, that no implementation of these operations is provided or planned.

Realtime Clock

The low order ten realtime clock bits (maintained in a micro processor register) can now be read by the programmer using the REGISTER construct. This feature was added in conjunction with the program monitoring facilities (see *Xfer Traps*).

Segment and File Objects

Descriptors for files and segments are now allocated from a single pool, rather than from separate tables. This eliminates considerable breakage, at some cost in the speed of enumeration procedures (which are performed rarely). As a consequence, objects are now represented as true variant records (not computed or overlaid), and a number of procedures take either data or file segments as parameters.

Shared Code Segments

A bit has been added to the global frame which indicates if the code is shared by other module instances. In the common case (a single instance of each module), this will eliminate searches of the global frame table every time the code is swapped out.

Start Command

The Mesa Executive's Start command now FORKS to the module, running it as a separate process. This insures that the executive will survive and continue to accept commands even if the user's process is aborted.

Trap Parameters

To eliminate the possibility of clobbering the stack when (possibly nested) traps occur, all trap parameters are now passed in registers (of the micro processor). They are made available to the trap routine using the REGISTER construct.

Uncaught Signals

The method of handling uncaught signals has been revised to accomodate the new process mechanism. Each process no longer includes an instance of the debugger's "nub" as its root; instead, a nub is spliced into the call stack dynamically when the uncaught signal occurs.

Xfer Traps

A mechanism has been added which will optionally cause a trap routine to be invoked for each XFER operation. In addition to performance monitoring, this facility is also useful for finding a large class of bugs, especially clobbers (see **TrapDefs**).

Distribution:
Mesa Users
Mesa Group

Inter-Office Memorandum

To	Mesa Users	Date	May 31, 1978
From	Roy Levin	Location	Palo Alto
Subject	Mesa 4.0 Microcode Update	Organization	CSL

XEROX

Filed on: [IRIS]<MESA>DOC>MICROCODE40.BRAVO

This memo outlines the differences in the (Alto) Mesa microcode for release 4.0. The *OIS Processor Principles of Operation* [1] ("PrincOps") is scheduled to be revised soon, and some of the changes indicated below will be incorporated in that revision. Others are peculiar to the Alto implementation of Mesa and are indicated as such. This document is only a summary of the changes in Mesa 4.0; additional details may be found in the references cited at the end of this memo.

Definitions of New Notions

Two new instruction properties have been introduced in the Mesa 4.0 instruction set:

Alignment

An aligned 1-byte instruction must be the last significant byte in the word. Thus, if an aligned, 1-byte instruction appears in an even byte position, the microcode will ignore the contents of the odd byte in the same word. An aligned 2-byte instruction must have both bytes in the same memory word. An aligned 3-byte instruction consists of an aligned 1-byte opcode followed by a word containing the α and β bytes. In this case, the α byte must be in the *odd* byte of the word following the opcode. In the 2-byte case, padding is accomplished by use of the new instruction NOOP, which is discussed below.

Minimal stack

A minimal stack instruction expects its operands to be the only quantities on the stack, and leaves the stack empty, except for any results it explicitly supplies.

These properties are peculiar to the Alto implementation of Mesa and will *not* be included in the revision of the PrincOps.

Changes to the Instruction Set

Many of the changes in the Mesa 4.0 microcode bring the instruction set closer to the PrincOps. In some cases, however, constraints imposed by the Alto architecture have prevented an exact emulation of the PrincOps semantics. The following sections define the differences between the Mesa 3.0 and 4.0 instruction sets, and relate those differences to the PrincOps.

Mesa 3.0 bytecodes not present in Mesa 4.0

LGS, SGS, LLS, SLS
LGDS, SGDS, LLDS, SLDS
WSDS

Space constraints in the Alto implementation have forced the elimination of these bytecodes.

ADDL, ADDG

These instructions have been superseded by the PrincOps instructions LADRB and GADRB (see below).

RXL0-3, WXL0
RIG0-3, WIG0, WIL0

These instructions have been superseded by the PrincOps instructions RXLP, WXLP, RIGP, RILP, and WILP (see below). Note that RIL0 has been retained, because of its high static frequency.

RIL1-3

Space constraints in the Alto implementation have forced the elimination of these bytecodes. They will, however, remain in the PrincOps when it is revised. Note that RIL0 has been retained in the Alto implementation.

Jumps

Mesa 4.0 has adopted byte distances for specifying jump targets. As a result, the even and odd byte forms of the Mesa 3.0 jump instructions have been eliminated, and the entire set of jumps revised to conform almost completely to the PrincOps. JIB and JIW are the only Mesa 3.0 jumps that have been retained, though their semantics have been modified to agree with the notion of byte distances. Details appear below.

GFC0-15, GFCB

The GFCn instructions have been uniformly replaced with EFCn instructions, which have similar semantics but support destination links in either the global frame or the code segment. This substitution will occur in the revision of the PrincOps as well.

CVT

Space constraints in the Alto implementation have forced the elimination of CVT.

BLK

The Mesa 4.0 process machinery makes BLK obsolete, and it has been eliminated. It will be eliminated in the revision of the PrincOps as well.

Mesa 4.0 bytecodes not present in Mesa 3.0

NOOP

Introduced to accommodate alignment requirements of the Alto implementation. NOOP will not be incorporated in the PrincOps revision. See the section on "Interrupts" for additional requirements affecting the execution of NOOP. Note: this opcode is meaningful only when it appears in the odd byte of a word.

LADRB, GADRB

Behave as described in [1], except:
Both are aligned instructions.

This difference will not be included in the PrincOps revision.

ADD01

Identical to ADD as described in [1], except:
ADD01 is a minimal stack instruction, assuming precisely 2 elements on the stack.

It is possible that ADD01 will be included in the PrincOps revision.

DADD, DSUB, DCOMP

DADD and DSUB behave as described in [1], except:
Both require minimal stack.
No carry bit is left on stack above stack pointer.

DCOMP expects two double precision values on the stack. Call them A and B. DCOMP performs: Sign[DSUB[A,B]]. Sign(x) = {-1 if x<0, 0 if x=0, +1 if x>0}. The (single precision) result of the Sign is left on the stack. DCOMP has a *minimal stack requirement.* DCOMP will probably be included in the PrincOps revision.

RXLP, WXLP
RILP, RIGP, WILP

Behave exactly as described in [1].

RFS, WFS
RFC

RFS and WFS are aligned, 1-byte instructions that expect the top word of the stack to be an <α, β> pair, with α in the left byte. RFS and WFS pop this word from the stack, then behave exactly like RF and WF, respectively, using the α and β values obtained from the stack word.

RFC is an aligned, 3-byte instruction that is identical to RF in all respects except that the code base register, C, is added to the computed address before the field is accessed.

All of these instructions use a new field descriptor format, which is described in the following section under RF, WF, and WFS. All of these instructions will be included in the PrincOps revision.

J2-J9, JB, JW

Behave as described in [1], except:
JB and JW are aligned instructions.
All jump distances are signed values, measured from the last byte of the jump instruction instead of the first (as in the PrincOps).

These differences will not be included in the PrincOps revision.

JEQ2-9, JEQB
JNE2-9, JNEB

Behave as described in [1], except:
JEQB and JNEB are aligned instructions.
All jump targets are signed, PC-relative distances in bytes, measured from the last byte of the jump instruction instead of the first (as in the PrincOps).

These differences will not be included in the PrincOps revision.

JLB, JGEB, JGB, JLEB
JULB, JUGEB, JUGB, JULEB
JZEQB, JZNEB

Behave as described in [1], except:
All are aligned instructions.
All jump targets are signed, PC-relative distances in bytes, measured from the last byte of the jump instruction instead of the first (as in the PrincOps).

These differences will not be included in the PrincOps revision.

DESCB, DESCBS

Behave as described in [1], except:
Both are aligned instructions.
The result left on the stack is (gfiword $\wedge$ 177B)+2*α+1, *where* gfiword *is word 0 of the global frame used by the instruction (see the section on "Global Frame Format", below).*

The difference in the result produced by these instructions will be included in the PrincOps revision.

EFC0-15, EFCB
LLKB

EFC0-15 and EFCB replace the GFC0-15 and GFCB instructions of Mesa 3.0. EFCn behaves identically to GFCn except in the way the destination link is determined. To locate the destination link, the microcode examines the low-order bit of the gfi word (word 0) of the global frame. If this bit is 0, the destination link is taken from location G-n-1, where G is the address of the current global frame. If the bit is 1, the destination link is taken from location C-n-1, where C is the address of the current code segment. EFC0-15 and EFCB will preplace GFC0-15 and GFCB in the PrincOps revision.

LLKB is an aligned, 2-byte instruction that computes a destination link in the same way that EFCB does. Instead of using it as the destination of an Xfer, however, LLKB simply pushes the destination link on the stack, and performs no additional actions upon it. LLKB, with the alignment requirement dropped, will be included in the PrincOps revision.

ME, MRE, MXW, MXD, NOTIFY, BCAST, REQUEUE

These are process-related opcodes, and are described separately below. All will be included in the PrincOps revision.

Bytecodes whose semantics have changed from Mesa 3.0 to Mesa 4.0

LGDB, SGDB, LLDB, SLDB
LIW

Identical to Mesa 3.0, except:
All are aligned instructions.

This difference will not be included in the PrincOps revision.

JIB, JIW

Identical to Mesa 3.0, except:
Both are aligned instructions.
The jump target for JIB is an <u>unsigned</u> distance in bytes, measured from the last byte of the JIB instruction instead of the first (as in the PrincOps).

The jump target for JIW is a *signed* *distance in bytes, measured from the last byte of the JIW instruction instead of the first (as in the PrincOps).*

These differences will not be included in the PrincOps revision.

RDB, WDB
WSB, WSDB

Identical to Mesa 3.0, except:
All are aligned instructions.

This difference will not be included in the PrincOps revision.

RSTR, WSTR

Identical to Mesa 3.0, except:
All are aligned instructions.

This difference will not be included in the PrincOps revision.

RF, WF, WSF

Identical to Mesa 3.0, except:
All are aligned instructions.
Field descriptor in β byte is $\langle p,s\rangle$, where p =bits to the left of desired field and s = bits in field minus 1.

The difference in field descriptor format will be included in the PrincOps revision.

BITBLT

Identical to Mesa 3.0, except:
BITBLT is an aligned instruction.
BITBLT is a minimal stack, 2-argument instruction.
The first argument to BITBLT is unchanged from Mesa 3.0; the second is a word containing the value zero.
BITBLT may be interrupted and subsequently restarted.

These differences are not relevant to the PrincOps revision.

DST, LST, LSTF

Identical to Mesa 3.0, except:
All are aligned instructions.
Only the active portion of the stack is saved or loaded (including 2 words above the top of the stack).

Except for the alignment requirement, these differences will be included in the PrincOps revision.

RR, WR

All α byte interpretations have changed - see the section on Trap Handlers and Parameters, below.

These differences will not be included in the PrincOps revision.

Process Instructions and the Mesa/Nova Interface [4]

Entry points to the Mesa emulator

Three entry points are defined. Control is transferred to these entry points by means of the Nova JMPRAM instruction. The addresses of these entry points are unchanged from Mesa 3.0, but some of the necessary conditions at entry are different.

a) Entry point 'Mgo' (location 420B): Nova AC0 must contain the address of a process state vector. The emulator will load the process state from this address, then perform a control transfer using the destination link at <AC0>+11B and source link at <AC0>+12B.

b) Entry point 'Minterpret' (location 400B): Mesa emulation continues using the current state in the emulator's internal registers. It is the responsibility of the invoker to ensure that the proper process state has been loaded.

c) Entry point 'SWRET' (location 777B): Used only when the Mesa emulator resides in ROM1. See Alto Hardware Manual, section 9.2.4.

Exits to Nova code

The Mesa emulator may cease execution and transfer control to the Nova emulator for one of two reasons:

1) A pending interrupt must be serviced.

2) A bytecode whose actions are implemented by Nova code has been interpreted.

In both cases the Nova program counter is set to a fixed value (in page 0) before control is passed to the Nova emulator. (In the second case, the PC value is different for each distinct bytecode.) In addition, the following conditions hold:

a) Any parameters expected by the Nova code appear in consecutive ACs beginning with AC0. Any Nova accumulator whose content is not explicitly supplied by the microcode will have undefined contents. In particular, no parameters are passed for a case 1 exit (pending interrupts). The microcode does not interpret the values transferred to the Nova ACs.

b) The state of the current process has been dumped to a process state vector whose starting address is stored in main memory at location 'CurrentState' (location 23B).

c) Unless a pending interrupt is being serviced (case 1), Nova interrupts have been disabled.

d) Any results generated by the Nova code must be transferred to the saved process state vector *before* the Mesa emulator is restarted. The emulator supplies no explicit mechanism to the Nova code for returning results.

Nova dispatch vector

When the Nova emulator receives control, the PC reflects the intended action to be performed. The Mesa emulator uses a dispatch vector beginning at 'NovaDVloc' (location 25B), and indexes it by the particular action required. The entries in the dispatch vector may be any Nova instructions, but in most cases will be a JMP to a Nova program that implements the desired semantics. The particular indices and relevant parameters passed are given by the following table:

Index	Action	Parameters
0	interrupt	none
1	STOP	none
2-3	unused	
4	ME	AC0
5	MRE	AC0, AC1
6	MXW	AC0, AC1, AC2
7	MXD	AC0
10B	NOTIFY	AC0
11B	BCAST	AC0
12B	REQUEUE	AC0, AC1, AC2

Global Frame Format

The global frame overhead has been reduced to 3 words, which have the following contents:

<G>+0	bits 0-8 contain the GFI,
	bits 9-14 are used by software,
	bit 15 is the frame links/code links indicator.
<G>+1	is the code base (odd ⇒ swapped out, even ⇒ address of code segment).
<G>+2	not used by Mesa emulator microcode.

Xfer Traps

A mechanism to implement trapping of control transfers has been implemented in Mesa 4.0. It is described in detail in a separate document [8].

Interrupts

After any Xfer, regardless of the cause, the Mesa emulator guarantees that at least one "useful" instruction will be executed. "Useful" means an instruction which is not a NOOP. In Mesa 3.0, no padding instructions were necessary and thus every instruction was considered "useful". In Mesa 4.0 this is no longer true, and this guarantee is needed to preserve certain properties within trap handlers (see below).

Trap Handlers and Parameters [7]

Mesa 4.0 trap handlers obtain their arguments by means of the RR and WR instructions. Four internal registers have been assigned for trap parameters:

XTSreg:	holds Xfer trap state (see [8]).
XTPreg:	holds Xfer trap parameters (see [8]).
ATPreg:	holds allocation trap parameter.
OTPreg:	holds parameters for all other traps.

Thus, Xfer traps use XTSreg and XTPreg, allocation traps use ATPreg, and code-swapped-out and unbound procedure traps use OTPreg.

An additional constraint on trap handlers is that they cannot assume that interrupts have been disabled by the microcode (this was true for some trap handlers in Mesa 3.0). However, the microcode continues to guarantee that one instruction following a trap will be executed before any interrupts are taken. Therefore, if an IWDC instruction is the first instruction of a trap handler, no interrupts will occur before the trap handler has had the opportunity to obtain its parameter(s).

As a result of these changes, the α byte interpretations of RR and WR have been redefined. The new meanings are:

α	RR	WR	
1	WDC	WDC	
2	XTSreg	XTSreg	
3	XTPreg	---	
4	ATPreg	---	
5	OTPreg	---	
6	clock	---	(see below)
7	code-base	---	

The clock value returned by RR6 is the low-order 16 bits of the Alto clock, regardless of the model. Thus the format of this value will be different on the Alto I and Alto II.

Stack overflow and underflow are reported by the same trap mechanism used in Mesa 3.0, but may not occur immediately after the instruction that caused the stack error. However, the trap *will* occur before or during the next instruction that either manipulates the stack or terminates a straight-line execution sequence.

AV, SD, and GFT

The addresses of these tables were stored in internal registers by the Mesa 3.0 emulator and could be accessed and modified by RR and WR. In Mesa 4.0, fixed locations have been established for each of these tables, as follows:

AV	1000B
SD	1060B
GFT	1400B

References

[1] Thacker, Chuck. *OIS Processor Principles of Operation.* Version 2.0, April 9, 1977.

[2] Levin, Roy. **Comparison of Old and New Alto/Mesa Microcode.** November 4, 1977. Filed on [Maxc]<Levin>UCodeComparison.bravo.

[3] Levin, Roy. Extensions to Alto/Mesa Microcode. November 7, 1977. Filed on [Maxc]<Levin>UCodeExtensions.bravo.

[4] Levin, Roy. Mesa/Nova Interface. November 15, 1977. Filed on [Maxc]<Levin>MesaNovaInterface.bravo.

[5] Levin, Roy. **Incompatibilities in Alto/Mesa microcode, version 24.** November 23, 1977. Filed on [Maxc]<Levin>Incompatibilities24.bravo.

[6] Levin, Roy. Extensions to Alto/Mesa Microcode for version 26. December 5, 1977. Filed on [Maxc]<Levin>UCodeExtensions26.bravo.

[7] Wick, John. Mesa 4.0 microcode (loose ends). March 14, 1978. Filed on [Maxc]<Wick>MICRO40.bravo.

[8] Levin, Roy. A Mechanism for Monitoring Transfers of Control in (Alto) Mesa. March 29, 1978. Filed on [Ivy]<Levin>Mesa40>XferTrap.bravo.

Distribution
Mesa Users
Mesa Group

c: Belleville
Charnley
Lampson
Thacker

Inter-Office Memorandum

To	Mesa Users	Date	May 31, 1978
From	Barbara Koalkin	Location	Palo Alto
Subject	Mesa 4.0 Debugger Update	Organization	SDD/SD

XEROX

Filed on: [IRIS]<MESA>DOC>DEBUGGER40.BRAVO

This release of the Mesa debugger introduces many changes of interest and importance to all Mesa programmers. The purpose of this memo is to make you aware of the changes that have taken place. More complete explanations may be found in the *Mesa Debugger Documentation.*

Interpreter

The major addition to the Mesa 4.0 debugger is an interpreter that handles a subset of the Mesa language; it is useful for common operations such as assignments, dereferencing, indexing, field access, addressing, and simple type conversion. It is a powerful extension to the current debugger command language, as it allows you to more closely specify your variables while debugging, thus giving you more complete information with fewer keystrokes. A subset of the Mesa language has been specified as being acceptable to the interpreter (a copy of the grammar is attached to this memo).

Statement Syntax

Typing space (SP) to the command processor enables interpreting mode. At this point the debugger is ready to interpret any expression that is valid in the (debugger) grammar.

Multiple statements are separated by semicolons; the last statement on a line should be followed by a carriage return (CR). If the statement is a simple expression (ie., not an assignment), the result is displayed after evaluation.

For example, to perform an assignment and print the result in one command, you would type **foo ← exp; foo.**

Loopholes

A more concise LOOPHOLE notation has been introduced to make it easy to display arbitrary data in any format. The character "%" is used to denote LOOPHOLE[**exp, type**], with the expression on the left of the %, and the **type** on the right.

For example, the expression **foo % short red Foo** means LOOPHOLE the type of the variable **foo** to be a **short red Foo** and display its value.

Subscripting

There are two types of interval notation acceptable to the interpreter. The notation **[a . . b]** means start at index **a** and end at index **b**. The notation **[a ! b]** means start at index **a** and end at index **(a+b-1)**.

For example, the expressions MEMORY**[4 . . 7]** and MEMORY**[4 ! 4]** both display the octal contents of memory locations **4** through **7**. Note that the interval notation is only valid for display purposes, and therefore is not allowed as a **LeftSide** or embedded inside other expressions.

Module Qualification

To improve the performance of the interpreter, the **$** notation has been introduced to distinguish between module and record qualification. The character **$** indicates that the name on the left is a module, in which to look up the identifier or TYPE on the right. If a module cannot be found, it uses the name as a file (usually a definitions file). A valid octal frame address is also accepted as the left argument of **$**.

For example, **FSP$TheHeap** means look in the module **FSP** to find the value of the variable **TheHeap**. In dealing with variant records, be sure to specify the variant part of the record before the record name itself (ie., **foo % short red FooDefs$Foo**, *not* **foo % FooDefs$short red Foo**).

Type Expressions

The notation "**@ type**" is used to construct a POINTER TO **type**. This notation is used for constructing types in LOOPHOLEs (ie., **@foo** will give you the type POINTER TO **foo**).

Examples

Some of the old commands may now be simplified as follows:

Interpret Array [array,index,n] becomes **array[index ! n]**
Interpret Call [proc] becomes **proc[param1, ..., paramN]**
Interpret Dereference [ptr] becomes **ptr↑**
Interpret Expression [exp] becomes **exp**
Interpret Pointer [address,type] becomes **address%@type↑**
Interpret SIze [var] becomes SIZE**[type]**
Interpret STring [string,index,n] becomes **string[index ! n]**
Interpret @ [var] becomes **@var**
Display Variable [var] becomes **var**
Octal Read [address,n] becomes MEMORY**[address ! n]**
Octal Write [address,rhs] becomes MEMORY**[address] ← rhs.**

Here are some sample expressions which combine several of the rules into useful combinations:

If you were interested in seeing which procedure was associated with the third keyword of the menu belonging to a particular window called **myWindow**, you would type:

myWindow.menu.array[3].proc

which might give you the following output:

```
CreateWindow (PROCEDURE in WEWindows).
```

If you wanted to look at one of your procedure descriptors, you might type:

```
4601B%@procedure ControlDefs$ControlLink↑
```

which might produce the following output:

```
ControlLink[procedure[gfi: 23B, ep: 0, tag: procedure]].
```

The basic arithmetic operations are provided by the interpreter (with the same precedence rules as followed by the Mesa compiler).

```
3+4 MOD 2 ;  (3+4) MOD 2
```

would produce the following output:

```
3
1.
```

Radix conversion between octal and decimal can be forced using the loophole construct; for example, exp%CARDINAL will force octal output and exp%INTEGER will force decimal.

A typical sequence of expressions one might use to initialize a record containing an array of Foos and display some of them would be:

```
rec.array ← DESCRIPTOR[FSP$AllocateHeapNode[n*SIZE[FooDefs$Foo]], n];
InitArray[rec.array]; rec.array[first..last].
```

Process Commands

The debugger has added a set of commands for use with the new process capabilities of the language and the system. Display of the new data types is as follows: condition variables and monitor locks are displayed in octal; a process is displayed as PROCESS [octal number]. In all of the process commands, the message "! is an invalid ProcessHandle" or "! not a process" appears if the process is invalid.

Set Process Context process

sets the current process context to be process and sets the corresponding frame context for symbol lookup to be the frame associated with process. Upon entering the debugger for the first time, the process context is set to the currently running process. Note that either a variable of type PROCESS (returned as the result of a FORK) or an octal ProcessHandle is acceptable as input to this command. Note also that when you set the octal context or module context, the process context is set to NIL; however, it is restored when you reset the context.

Display Process process

is a specialized version of Display Variable that displays interesting things about a process. This command shows you the ProcessHandle and the frame associated with process, and whether the process is waiting on a monitor or a condition variable (waiting ML or waiting CV). Then you are prompted with a ">" and enter process subcommand mode. A response of N displays the next process in the array of psbs; R displays the root frame of the process; S displays the source text; P displays the priority of the process; and Q or DEL terminates the display and returns you to the command processor. A variable of type PROCESS (returned as the result of a FORK) or an octal **ProcessHandle** is acceptable as input to this command (note that process is an interpreted expression).

`Display Queue` **id**

displays all the processes waiting on the queue associated with **id**. For each process, you enter subcommand mode. The semantics of the subcommands remain the same as in `Display Process`, with the exception of N, which in this case follows the link in the process. This command is prepared to accept either a condition variable, a monitor lock, a monitored record, a monitored program, or an octal pointer (as in a pointer to the **ReadyList**). Note that **id** is an interpreted expression; if **id** is simply an octal number, you are asked whether it is a condition variable in order for the debugger to know where to find the head of the queue (i.e., `Display Queue: 175034B, condition variable? [Y or N]`).

`List Processes [confirm]`

lists all processes by telling you the **ProcessHandle** and its frame. If you wish to see more information about a particular process use the `Display Process` command.

Conditional Breakpoints, Multiple Proceeds, and new Breakpoint Syntax

The Mesa 4.0 debugger has extended the former set of breakpoint commands to include the capability to set conditional breakpoints.

The syntax for all of the Mesa 3.0 breakpoint commands remains basically the same with the following extension: if you type a SP after the procedure or module name you receive a prompt for the condition; if you type a CR it terminates the command input (in the case of entry/exit breaks) or just prompts for the source (in the case of text breaks), ie.,

```
Break Procedure: proc(SP), condition: x < 2 (CR), source: IF a =b (CR)
Break Procedure: proc(CR), source: IF a =b (CR).
```

The three valid formats for a conditional expression are:

variable relation variable - *eg., stop when x < y*

variable relation number - *eg., stop when x >= 10*

number - *eg., stop the 5th time you reach this breakpoint*

These commands accept relations belonging to the set: {<, >, =, #, <=, >=}, corresponding to: less than, greater than, equal, not equal, less than or equal, greater than or equal.

This gives us the ability to do multiple proceeds with the same syntax as simple conditional breakpoints.

Since the variables are interpreted expressions, they are looked up in the current context. However, if you are in a module context and wish to specify a local variable of the procedure you are setting the breakpoint in, you may do this by saying:

`proc.var` - *ie., use the local variable var defined in proc*

You may change the condition on a particular breakpoint or change a breakpoint from a conditional to a non-conditional one or vice-versa simply by setting the breakpoint again using the new condition.

There has also been a simplification to the breakpoint syntax as follows:

All commands to set breakpoints begin with the key letter `B`,
eg., `Break Procedure` instead of `SEt Procedure Break`

All commands to set tracepoints begin with the key letter `T`,
eg., `Trace Procedure` instead of `SEt Procedure Trace`

The keyword `Program` has been replaced by the word `Module`,
eg., `CLear Module Break` instead of `CLear Program Break`

All of the breakpoint commands now accept a valid **GlobalFrameHandle** as input when prompted for a module name.

New Commands / Changes to Existing Commands

`Kill session [confirm]`

ends your debugging session, cleans up the state as much as possible, and returns to the Alto Executive. Use this command instead of shift-Swat or the boot button to leave the debugger.

`ATtach Symbols` [globalframe, filename]

attaches the globalframe to filename. This is useful for allowing you to bring in additional symbols for debugging purposes not initially anticipated.

`ATtach Image` [filename]

specifies the filename to use as an image file when the debugger has been bootloaded. It is useful when the user core image has been clobbered. The default extension for filename is ".image".

`Display Stack` subcommands

The `Display Stack` command now makes a distinction between displaying module (global) and procedure (local) contexts. The valid subcommands for local contexts remain as in Mesa 3.0: n,p,v,r,s,q; with the addition of the subcommand `>j, n(10)` which means jump down the stack n levels. Note that if n is greater than the number of levels it can advance, the debugger tells you how far it was able to go. Most of these subcommands apply to `Display Stack` on a global context with the exception of j and n. If the debugger cannot find a symboltable for some frame on the call stack, you get the message "`No symbols for nnnnnnB`" and enter restricted `Display Stack` subcommand mode in which only the subcommands j, n, and q are allowed. Note that a local context is displayed as the procedure name with its local frame, followed by the module name and its global frame; a global context is displayed as the module name and its global frame. For example,

```
>Display Stack -- on a global frame
StreamsA, G: 172674B  >?--Options are: p,q,r,s,v.
>Display Stack -- on a local frame
TArrays, L: 165064B (in TArrays, G:166514B)
  >? --Options are: p, v, r, s, q, j, n.
```

Notice that the convention, `proc, L:nnnnnnB (in module, G: nnnnnnB)`, applies throughout debugger output, wherever procedures and modules are displayed.

`CUrrent context`

The notion of the current context has been extended to include the current ProcessHandle as well as the name and corresponding global frame address of the current module and the current configuration.

Old commands that are gone

`Join Ports` and `Interpret SIze` have been taken out of the debugger's command language since their functions have been taken over by the debugger interpreter; `Display Binding path` has been removed since the concept of a binding path has gone away.

Additional Capabilities

More breakpoints - local procedures

Due to a change in the lookup algorithm for procedures, it is now possible to set breakpoints/tracepoints on a local (nested) procedure without being in the context of its enclosing procedure. However, in order to display a local procedure you must still be in its enclosing context.

Validity checking

The debugger makes a considerable effort to check if the user core image has been smashed in any way. When it determines that something is wrong, rather than printing out incorrect information it sets the context to NIL and disables all of the commands that rely on getting information either from symbol tables or from the loadstate. The user gets a message that says "`Current context invalid.`" or "`Command not allowed.`" whenever this situation occurs. At this time you might want to attach an image file or some symbols (see the `ATtach` comands) to find out what is wrong.

Installing

To install the debugger with a command line to the Alto Executive, use the "I" switch; use the "L" switch to load programs with code links (to save space). For example, typing `XDebug WindEx/i` installs the debugger with the window manager (WINDEX.BCD); typing `XDebug WindEx/il` installs WINDEX with code links.

Missing definition files

Instead of simply refusing to give you any information about your variables when you are missing a definitions file or have the wrong version of a file, the debugger prints a "?" to give you an indication that something is missing.

Invalid values

The debugger will print "?[value]" when displaying enumerated types that appear to be wrong (ie., out of range).

Comments in the typescript file

A comment command has been added to the debugger command language. Use "--" to ignore type-in until a carriage return (CR). This is useful for saving your own notes along with your typescript file as well as type-in to be used for window selections.

Current Date and Time

The current date and time is inserted at the beginning of your typescript file along with the date and time that your version of the Mesa 4.0 debugger was created.

Default command

Typing the escape character (ESC) to the command processor of the debugger, uses the last command as the next valid command (i.e., you receive the prompts for the parameters (if any) for the previously executed command).

Confirming commands

When a command requires a [confirm] (CR), the debugger goes into wait-for-DEL mode if an invalid character is typed.

Extended Features

See the *Debugger - Extended Features* memo for further details on the following.

FTP in the debugger

The ↑FTP command (control-F) is used to provide file transfer capabilities from within the debugger using the standard FTP package. Any comments and/or problems regarding FTP itself should be addressed to the Communications Group. If FTP has not been loaded, trying to use any of the FTP commands will give you the message "-- FTP not installed".

UserProcs

The ↑UserProc command (control-U) allows you to load your own debugging package into the debugger. If you have only one user proc loaded when you type control-U, it will be invoked automatically. If you have several user procs loaded, typing "?" will give you a list of the command names for the user procs that you have loaded. If it can't find any procedures that have been loaded, you will just get the message, " !No user procs are currently loaded".

Window manager

The new window manager **WindEx** has several commands which allow you to set breakpoints and tracepoints by selecting text locations. Confirmation is given by moving the selection to the place at which the breakpoint is actually set.

Internal Changes

Debugger Nub

The Mesa 4.0 debugger has been able to realize a significant space reduction by removing its own internal debugging facilities and replacing them with a nub. Typing ↑D to the command processor brings you into the debugger nub with a "//" prompt. The following limited set of commands are available in the nub: Install, Bitmap, New, Start, Proceed, and Quit. Bitmap[n(10)] reallocates the bitmap to n pages (the default size is about 50 pages). The nub also provides a minimal signal catcher and interrupt handler as well as primitive debugging facilities. It is possible to install a different version of the debugger to use for debugging the debugger itself (see a member of the Mesa Group if you are interested in knowing more about how this works).

Savings in space

The global frame size of the Mesa 4.0 debugger has been reduced by over 50% from the previous release. This has created space for the interpreter and for a larger bitmap. Some of the savings is due to the split of the internal and external debugger. Another reason is due to the way in which the debugger handles strings. By putting the command strings, command prompts, signal and error messages, and debugger FTP commands into a separate file (and running this file through the string compactor), the string segment can be swapped in only when needed. Additional space was saved by making many of the remaining strings into local strings so that they do not take up space in the global frame.

Documentation

More complete documentation on the Mesa 4.0 Debugger may be found in the *Mesa Debugger Documentation.* Bug fixes may be found in the closed change requests maintained by <SDSupport>.

Debugger Summary

Version 4.0

```
ASciі read [address, n]
ATtach Image [filename]
       Symbols [globalframe, filename]
Break Entry [proc, condition]
      Module [module, condition, source]
      Procedure [proc, condition, source]
      Xit [proc, condition]
CAse off [confirm]
     on [confirm]
CLear All Breaks [confirm]
          Entry traces [module]
          Traces [confirm]
          Xit traces [module]
      Break [proc, source]
      Entry Break [proc]
            Trace [proc]
      Module Break [module, source]
             Trace [module, source]
      Trace [proc, source]
      Xit Break [proc]
          Trace [proc]
COremap [confirm]
CUrrent context
Display Configuration
        Eval-stack
        Frame [address]
        GlobalFrameTable
        Module [module]
        Process [process] - n,p,q,r,s
        Queue [id]
        Stack - j,n,p,v,r,s,q
        Variable [id]
Find variable [id]
Interpret Array [array, index, n]
          Call [proc]
          De-reference [ptr]
          Expression [exp]
          Pointer [address, type]
          String [string, index, n]
          @ [var]
Kill session [confirm]
List Breaks [confirm]
     Configurations [confirm]
     Processes [confirm]
     Traces [confirm]
Octal Clear break [globalframe, bytepc]
      Read [address, n]
      Set break [globalframe, bytepc]
      Write [address, rhs]
Proceed [confirm]
Quit [confirm]
Reset context [confirm]
SEt Configuration [config]
    Module context [module]
    Octal context [address]
    Process context [process]
    Root configuration [config]
STart [address]
Trace All Entries [module]
          Xits [module]
      Entry [proc,condition]
      Module [module, condition, source]
      Procedure [proc, condition, source]
      Xit [proc, condition]
Userscreen [confirm]
Worry off [confirm]
      on [confirm]
-- [comment]
```

Debugger Interpreter Grammar

Version 4.0

```
StmtList            ::=  Stmt | StmtList; Stmt

AddingOp            ::=  + | -

BuiltinCall         ::=  LENGTH [ LeftSide ] | BASE [ LeftSide ] |
                         DESCRIPTOR [ Expression ] |
                         DESCRIPTOR [ Expression , Expression ] |
                         SIZE [ TypeSpecification ]

Expression          ::=  Sum

ExpressionList      ::=  Expression | ExpressionList,  Expression |

Factor              ::=  - Primary | Primary

Interval            ::=  Expression .. Expression  | Expression ! Expression

LeftSide            ::=  identifier | Literal | MEMORY [ Expression ] |
                         LeftSide Qualifier |  ( Expression ) Qualifier |
                         identifier $ identifier | numericLiteral $ identifier

Literal             ::=  numericLiteral |
                         stringLiteral |  -- all defined outside the grammar
                         characterLiteral

MultiplyingOp       ::=  * | / | MOD

Primary             ::=  LeftSide | ( Expression ) | @ LeftSide | BuiltinCall

Product             ::=  Factor | Product MultiplyingOp Factor

Qualifier           ::=   . identifier | ↑ | % | % TypeSpecification | [ ExpressionList ]

Stmt                ::=  Expression | LeftSide ← Expression | MEMORY [ Interval ] |
                         LeftSide [ Interval ] | ( Expression ) [ Interval ]

Sum                 ::=  Product | Sum AddingOp Product

TypeConstructor     ::=  @ TypeSpecification

TypeIdentifier      ::=  INTEGER | BOOLEAN | CARDINAL |
                         CHARACTER | STRING | UNSPECIFIED |
                         identifier | identifier $ identifier |
                         identifier TypeIdentifier

TypeSpecification   ::=  TypeIdentifier | TypeConstructor
```

Windex Summary

Version 4.0

WHAT WINDEX MOUSE BUTTONS DO:

	Scroll Bar	Text Area
RED	ScrollUp	Select/Extend characters
YELLOW	Thumb	Select/Extend words
BLUE	ScrollDown	Menu Commands
YELLOW/BLUE	NormalizeSelection	

MENU COMMANDS:

Create [window]	Find [selection, window]
Destroy [window]	Set Brk [selection]
Move [window]	Clr Brk [selection]
Grow [window]	Set Trc [selection]
Load [selection, window]	Set Pos [index, window]
Stuff It [selection, window]	Keys On/Off

WHAT MENU MOUSE BUTTONS DO:

RED	"Do it" - in this window/ at this spot
BLUE	Reset to previous state

WHAT KEYSET BUTTONS DO:

BS	DEL	ESC	CR	STUFF IT

DURING TYPE IN:

BS	Backspace character
CONTROL-W	Backspace word
FL4	Stuff current selection into default window

Fetch Command Summary

Close connection [confirm]
DElete filename [filename]
DUmp from remote file [dumpfile]
Free pages
LIst remote file designator [filelist]
LOad from remote file [dumpfile]
Open connection [host, directory]
Quit [confirm]
Retrieve filename [filename]
Store filename [filename]

Inter-Office Memorandum

To	Distribution	Date	September 7, 1978
From	John Wick	Location	Palo Alto
Subject	Mesa 4.1 update	Organization	SDD/SD

XEROX

Filed on: [Iris]<Mesa>Doc>Mesa41.bravo

This memo summarizes the changes contained in Mesa 4.1. This is a maintenance release, and contains primarily bug fixes documented elsewhere (by SDSupport). There have been no changes to public interfaces since Mesa 4.0. However, there are a few highlights that are worth pointing out.

In the paragraphs below, numbers in square brackets refer to change requests maintained by SDSupport.

Microcode

The Mesa 4.1 compiler now generates the BLTC instruction. This means that 4.1 BCDs are not backward compatible with 4.0; that is, the output of the 4.1 compiler will not run with 4.0 microcode. (However, 4.0 BCDs will run with 4.1 microcode, so there is no need to recompile.) Note that all Mesa 4.1 image files require 4.1 microcode. [4.0.148]

Users are strongly encouraged to update to the 4.1 microcode, as there is a rather nasty bug in the 4.0 signed compare instructions. [4.0.167]

Compiler

There is one change in the semantics of relative pointers. To more closely parallel array subscripting, a relocated relative pointer is now automatically dereferenced. If *b* is a base pointer and *p* a relative pointer to *Foo*, the construct *b*[*p*] is now of type *Foo* instead of type POINTER TO *Foo*. (The compiler will point out all the constructs where an @ operator is needed or where an ↑ should be removed.) [4.0.273]

The constructs FIRST and LAST now apply to (LONG) INTEGERs, CARDINALs, and CHARACTERs; they yield the minimum and maximum values, respectively. For example, LAST[LONG INTEGER] has the value 2147483647 (2^{31}-1); these constructs should be used in place of MaxLongInteger and the like. [4.1.322]

Binder

The binder now enforces quad-word code alignment. This will affect only systems running on the D0, although Alto/Mesa users may notice a very small increase in the size of packed code segments.

The binder now pauses when warnings are detected (under control of the /p switch).

Debugger

The debugger has been updated to support Pilot on the D0. Alto users are unaffected by these extensions.

The string parameter passed to CallDebugger is now printed by the debugger; a 4.1 system (Mesa or BasicMesa) is required to use this feature. [4.0.26, 4.0.301]

Distribution:
Mesa Users
Mesa Group

Inter-Office Memorandum

To	Mesa Users	Date	December 5, 1978
From	J. Sandman	Location	Palo Alto
Subject	Performance Measurement Tool	Organization	SDD/SS/DE

XEROX

Filed on: [IRIS]<Mesa>Doc>PerformanceTool.bravo .press

A tool for the performance measurement of Mesa programs is described below. It allows users to identify places in their programs and then collect timing and frequency statistics of program execution between these places. The system is implemented as a set of commands that can be executed from the Mesa Debugger, plus a routine that intercepts all conditional breakpoints and collects statistics about them. Existing Debugger commands are used to specify what points are to be monitored, and additional commands are provided for controlling the measurements and outputting the results. Both Alto/Mesa programs as well as Pilot programs may use this measurement tool.

Concepts

A *node* is defined to be a place in a program where a breakpoint can be set by the Mesa Debugger. In fact, nodes are implemented via conditional breakpoints, so that while measurements are turned on, the *functioning of all conditional breakpoints is different.* In particular, conditional breakpoints behave as if they were never encountered or as if the stated condition is tested but is always found to be FALSE. (Also the count is not decremented for multiple proceed conditional breakpoints).

A *leg* is defined by a pair of nodes, one called the *from* node and the other the *to* node. A leg is the code executed between these nodes. Interesting items measured about a leg include the number of times this leg was executed and the time required to execute the leg.

Facilities are also provided for associating a *histogram* with any node or leg, thereby providing more detailed distribution information about the entry than is provided by counts, sums, and averages.

Since *processor time* or *task time* is not available on the *Alto* or the *D0*, the measure of computing is simply the *elapsed time* from the time the *from* node is executed to the time the *to* node is executed.

The concept of nodes and legs is borrowed from the Diamond ETM module. This tool was first written by Paul Jalics and transferred to the Mesa Group.

Terminology

Node Table

A table maintained by the measurement module containing information about each node. A node for each conditional breakpoint is entered into this table by the `Collect nodes` command or by the measurement module when it encounters a conditional breakpoint that is not already in the table. The node table has 20 entries.

NodeID

The index of a node in the node table. The NodeID for a particular conditional breakpoint does not change during a measurement session and is used in commands to identify a particular node.

Node pair

A pair of nodes defining a Leg. The syntax is N1-N2, where N1 and N2 are both NodeIDs. The character "*" may be used as a wildcard node designator in a Node pair. For example, the pair *-* designates all possible pairs and 1-* designates all pairs with node 1 as the *from* node.

Leg Table

A table maintained by the measurement module containing various information about each leg. Legs are entered into this table by the command `Add Legs` or by the measurement module when it encounters a new leg and automatic insertion is enabled. The leg table has 41 entries, one of which is reserved.

LegID

The index of a leg in the leg table. The LegID for a particular conditional breakpoint does not change during a measurement session and is used in commands to identify a particular leg.

Histogram

An optional table that may be associated with either a node or leg that records the distribution of the *value* of the node or leg by incrementing counters in a number of *buckets.* The distribution may be either *simple* or *logarithmic.* In a simple distribuiton, a *base* may be specified which will be used as the offset for the first bucket. In a logarithmic distribution, the buckets are indexed by the number of leading binary zeros in the *value.* A *scale* is used to adjust the value for an optimal fit into the number of buckets. There is a storage pool of 256 words that is shared among all histograms to hold buckets and histogram information.

Node Histogram

A histogram associated with a node. The value of the node is the first variable specified in the conditional breakpoint that determines the node. (See Section 3 of *Mesa Debugger Documentation.*) The value is treated as a 16 bit unsigned quantity. For a simple node histogram, the value is adjusted by subtracting the base (if any) and dividing by the scale factor; the resulting quotient is recorded. A logarithmic node histogram has a maximum of 16 buckets because the value is a 16 bit quantity.

Leg Histogram

A histogram associated with a leg. The value of the leg is the 32 bit time of the leg in units of ticks. The value is adjusted by shifting the value to the right by the scale. A logarithmic leg histogram has a maximum of 32 buckets because the value is a 32 bit quantity.

Components

PerfTool is the component of the measurement system that lives with user programs built on top of Alto/Mesa. This configuration contains two modules: PerfMonitor and PerfBreakHandler. PerfMonitor initializes the PerfTool. PerfBreakHandler contains a breakpoint handler that intercepts all conditional breakpoints and accumulates statistical information about nodes and legs. PerfTool must be loaded and started in the system it will monitor. This may be done by including PerfTool in the client configuration whose control module starts PerfDefs.PerfMonitor or by executing the following command to the Alto Executive:

```
>Mesa PerfTool Client
```

PilotPerfTool is the component of the measurement system that lives with user programs built on top of Pilot. This configuration contains two modules: PilotPerfMonitor and PilotPerfBreakHandler. These modules perform the same functions as PerfMonitor and PerfBreakHandler, respectively. Since there is no loader in Pilot 2.0, PilotPerfTool must be included in the client configuration whose control module starts PerfDefs.PilotPerfMonitor. In addition, the code for PilotPerfBreakHandler must be made resident. Use the StartPilot command `ResidentCodeModule` by executing the following command to the Alto Executive:

```
>StartPilot ResidentCodeModule["PilotPerfTool>PilotPerfBreakHandler"]
 Build["Client"]
```

PerfPackage is the component that lives as a userproc in the Mesa Debugger. It implements the basic commands required to manipulate the node table and the leg table and to output measurement results. PerfPackage must be loaded into the Debugger before its commands can be executed. The easiest way is to load it when installing the Debugger by executing the following command to the Alto Executive:

```
>XDebug WindEx/l PerfPackage/li
```

The command interpreter for the PerfPackage is invoked by calling the userproc `PerfMonitor`. The userproc dispatcher is invoked by the `↑UserProc` command (`control-U`). If only one userproc is loaded, it is automatically called, otherwise some unique prefix must be typed when the dispatcher prompts for a procedure name. See the *Mesa Debugger Documentation: Debugger - Extended Features* for details.

Operation

When the break handler intercepts a breakpoint, it checks to see if the breakpoint is a conditional breakpoint. If so, it finds the node corresponding to the breakpoint, and increments its counters and processes its histogram if one exists. If tracking of legs is enabled, the leg table is searched for the legs of which this node is a part. Otherwise, the break point is resumed.

In the simple case, a leg is tracked as follows. The break handler intercepts a conditional breakpoint that is the *from* node of the leg (from) and some time later it intercepts a conditional breakpoint that is the *to* node of the leg (to). At this point, the leg's time is recorded, its count is incremented, and its histogram (if any) is processed.

This simple model of tracking a leg is complicated by recursion, signals, and multiple processes. With recursion, from may be encountered several times before to is encountered. With signals, a process may be unwound after it encounters from but before it encounters to. With multiple processes, one process may encounter from and then another immediately encounter to.

To deal with the complication of multiple processes, there is the concept of the *tracked process.* If the tracked process is not NIL then only those conditional breakpoints that are encountered by the tracked process are treated as nodes. All others are simply resumed as if they did not exist. If the tracked process is NIL, then all processes are tracked.

To deal with these complications, there is a *leg owner.* A leg owner is the process that last encountered from. When to is encountered and the current process is its owner, then the leg is recorded and the leg owner is cleared. If the current process is not the owner, the leg is ignored. As a result of ignoring legs, from and to may be counted more times than the leg between them is counted.

Normally, when a node is encountered all legs of which that node is a part are tracked. Alternatively only the leg defined by the last node encountered and the current node is tracked.

Commands

The command interpreter completes commands like the Debugger command interpreter. The capitalized characters are all that must be typed to specify a command.

General Commands

`Collect nodes`

enters conditional breakpoints as nodes into the Node Table.

`Initialize tables [Confirm]`

completely reinitializes all tables and counters. Both the node table and the leg table and all histograms are cleared.

`List Tables`

displays all the summary statistics gathered so far and the complete contents of the node table and the leg table. May be aborted by typing ↑DEL.

`Monitor on`

turns on performance monitoring. All conditional breakpoints will now be monitored.

`Monitor off`

turns off performance monitoring. All conditional breakpoints will now behave like normal conditional breakpoints.

`Quit [Confirm]`

exits the PerfPackage. and returns to the Mesa Debugger.

`Zero tables [Confirm]`

zeros out all counts and sums from the tables (including the total time spent measuring) but will leaves all other information in the tables unchanged. This command is useful for preserving the measurement environment but just zeroing out the counts and sums collected so far.

Leg and Node Commands

`Add Legs [Node pair, Node pair, ...]`

adds the legs specified by the node pairs to the leg table. If a designated leg entry is already in the leg table, the leg is not affected.

`Delete Legs [LegID, LegID, ...]`

deletes the specified legs from the leg table.

`List Leg table`

displays the contents of the leg table. A `LegID` followed by an asterisk has a histogram associated with it. May be aborted by typing `↑DEL`.

`List Node table`

displays the contents of the node table. A NodeID followed by an asterisk has a histogram associated with it. May be aborted by typing `↑DEL`.

Mode Control Commands

`Add Immediate successors`

enables the PerfBreakHandler to add legs that it encounters that are not in the table. These legs may be deleted if there is no room in the leg table when legs are being added by the `Add Legs` command.

`Add No legs`

prevents the PerfBreakHandler from adding legs that are not in the table. This is the default mode for adding automatic legs.

`Track All Legs`

tells the PerfBreakHandler to track all legs in the table. This is the default mode for tracking legs

`Track All Processes`

tells the PerfBreakHandler to track all processes. All processes are tracked in the default case.

`Track Immediate successors`

tells the PerfBreakHandler to track only the leg defined by the last node encountered and the current node.

`Track No legs`

tells the PerfBreakHandler to disable tracking of legs.

`Track Process [process]`

tells the PerfBreakHandler to track only those legs that are executed by the specified process. Nodes encountered by other processes will not be recorded. An octal ProcessHandle as obtained from the Debugger's `List Processes` command is acceptable as input to this command. A carriage return will set the process to all processes.

Histogram Commands

`Add Histogram for Leg [LegID]`

adds a histogram and associates it with the specified leg. The command prompts for number of buckets, type (simple or logarithmic), scale, and base if the type is simple. Note that since scaling of a leg histogram is done by shifting instead of dividing, the scale is entered as a power of two.

`Add Histogram for Node [NodeID]`

adds a histogram and associates it with the specified node. The command prompts for number of buckets, type (simple or logarithmic), scale, and base if the type is simple.

`Delete Histogram for Leg [LegID]`

deletes the histogram associated with the specified leg.

`Delete Histogram for Node [NodeID]`

deletes the histogram associated with the specified node.

`List Histogram for Leg [LegID]`

displays the histogram associated with the specified leg. May be aborted by typing `↑DEL`.

```
List Histogram for Node [NodeID]
```

displays the histogram associated with the specified node. May be aborted by typing ↑DEL.

Command Tree

This is the command tree structure for the PerfPackage. It is formatted like the command tree for the Mesa Debugger (see *Mesa Debugger Documentation*).

```
Add Histogram for Node
                  Leg
    Immediate successors
    Legs
    No legs

Collect nodes

Delete Histogram for Leg
                     Node
       Legs

Initialize tables

List Histogram for Leg
                   Node
     Leg table
     Node table
     Tables

Monitor on
        off

Quit

Track All Legs
          Processes
      Immediate successors
      No legs
      Process

Zero tables
```

Limitations

1. Time Base: The time base available on the Alto is a 26-bit counter, where the basic unit of time is 38 microseconds. Thus the counter turns over every 40 minutes, and no individual time greater than 40 minutes is meaningful on the Alto. Total times are 32-bit numbers and will overflow after 340 minutes.

2. Overhead Calculation: Due to implementation restrictions and timer granularity, some of the overhead of processing a breakpoint is incorrectly assigned to the client program instead of the PerfTool. As a result, leg times will be about ten microseconds high for each node that was enountered while processing that leg. Elapsed time is similarily affected.

3. Counter Sizes: In a long measurement session, the counters on nodes, leg and histograms may overflow. Node and leg counters are 22-bit numbers, while histogram counters are 16-bit numbers. If a node or leg counter overflows, a "*" follows the count when the field is listed.

4. Recursive Procedure Calls, UNWINDs, multiple processes: As mentioned in the section on operation, the above interfer with the simple start to end concept of a leg. With recursion and multiple processes, the start node of a leg may be tripped several times before the end node is tripped. With unwinding, the start node of a leg may be tripped and the end node never reached. If any of these cause a leg to be ignored, the referenced field in the Leg Table has a "~" following it when the table is listed.

5. Table Sizes: The node table contains 20 entries. (Note that the `PerfBreakHandler` automatically extends the number of conditional breakpoints that can be set in the debugger from 5 to 20.) The leg table currently has 40 entries. Note that this number is small when compared to the 20*20 possible legs. For this reason, there exist a number of commands to give the user control over exactly what legs are in the table.

6. Memory Requirements: The PerfTool requires seven pages of resident memory; three for `PerfBreakHandler`'s code, and four for PerfTool's frames. This may affect the performance of some systems that use a lot of memory, especially on the Alto.

7. `PerfBreakHandler` acts like a worry mode breakpoint and as a consequence, you may find you cannot `Quit` from the Debugger after your session. Use the `Kill` Debugger command instead.

Getting Started

Outlined below are the steps required for using the measurement tool.

1. obtain the bcd's for PerfTool and PerfPackage.

2. install the PerfPackage in the Mesa Debugger (version 4.1 or later).

3. start your program executing with the PerfTool included.

4. enter the Debugger and set conditional breakpoints as desired.

6. turn measurements on via the `Monitor on` command.

7. manipulate the leg table as desired.

8. proceed with program execution.

9. return to the debugger via control-swat or an unconditional breakpoint.

10. display results with the `List` commands.

Sample Session

The following annotated listing of a DEBUG.TYPESCRIPT session should give a fair idea of the use of the measurement tool.

```
Alto/Mesa Debugger 4.1 of  6-Sep-78 18:47
13-Nov-78 11:49

>SEt Module context: Segments
>Interpret Call Procedure: AllocatePages 0: 160
  (anon)=26000B↑              -- Allocate most of memory to cause swapping for example
>SEt Module context: Swapper
>Break Entry Procedure: AllocVM, Condition: AllocVM.pages = 1
-- a histogram will be attached to this breakpoint and the local variable pages will be counted.
>Break Xit Procedure: AllocVM, Condition: 1
>Break Entry Procedure: MakeSwappedIn, Condition: 1
>Break Xit Procedure: MakeSwappedIn, Condition: 1
>userProc [confirm]
Proc: PerfMonitor
@Monitor on                   -- Now conditional breakpoints activate
@Collect nodes
@List Node table
 - - - - - - N O D E   T A B L E   C O N T E N T S - - - - - - - - - - -
Node Global  Program  Number of  Config   Module   Proc     Source
 Id  Frame   Counter  References Name     Name     Name     Line
---- ------- -------  ---------- -------- -------- -------- ----------
  0  173314    3314            0 Mesa     Swapper  AllocVM  @Entry
  1  173314    3651            0 Mesa     Swapper  AllocVM  @Exit
  2  173314    2251            0 Mesa     Swapper  TryCodeS @Entry
  3  173314    2775            0 Mesa     Swapper  TryCodeS @Exit

@Add Legs: 0-1,2-3
@List Leg table
- - - - - - - L E G   T A B L E   C O N T E N T S - - - - - - - - - -
Leg  From  To    # of Times  Total Time      Average Time   % of
Id   Node  Node  Referenced  sec.msec:usec   sec.msec:usec  Time
---  ----  ----  ----------  -------------   -------------  -----
  0    0 ->  1            0              0               0    .00
  1    2 ->  3            0              0               0    .00
@Add Histogram for Node: 0
Type of Histogram: Simple
Number of Buckets [1..246]:12
Scale Factor [1..65,535]:1
Base: 0
@Add Histogram for Leg: 0
Type of Histogram: Logarithmic
Number of Buckets [1..32]:12
Scale Factor (2↑n) [0..31]:0
@Add Histogram for Leg: 1
Type of Histogram: Simple
Number of Buckets [1..204]:12
Scale Factor (2↑n) [0..31]:5
Base: 32
@Quit [Confirm]
>Proceed [confirm]
>userProc [confirm]
Proc: PerfMonitor
@List Tables
```

```
Total Elapsed Time of Measurements =          16.567:544
Elapsed Time less PerfMonitor Overhead =      14.884:980
Total Overhead of PerfMonitor Breaks =         1.682:564
Total number of Perf Breaks handled =                382
Average Overhead per Perf Break =                  4:404
% of Total Time spent in PerfMonitor =             10.15

- - - - - - N O D E   T A B L E   C O N T E N T S - - - - - - - - - -
Node Global  Program  Number of  Config   Module   Proc     Source
 Id  Frame   Counter  References Name     Name     Name     Line
---- ------  -------  ---------- -------- -------- -------- ----------
  0* 173314     3314        102  Mesa     Swapper  AllocVM  @Entry
  1  173314     3651        102  Mesa     Swapper  AllocVM  @Exit
  2  173314     2251         89  Mesa     Swapper  TryCodeS @Entry
  3  173314     2775         89  Mesa     Swapper  TryCodeS @Exit

- - - - - - - L E G   T A B L E   C O N T E N T S - - - - - - - - -
Leg  From  To    # of Times  Total Time     Average Time   % of
Id   Node  Node  Referenced  sec.msec:usec  sec.msec:usec  Time
---  ----  ----  ----------  -------------  -------------  -----
  0*   0 -> 1          102      4.328:390         42:435   26.12
  1*   2 -> 3           89        605:530          6:803    3.65
@List Histogram for Node: 0
Number of References               102
Sum of Values                      432
Average Value                        4
Scale Factor                         1
Base                                 0
     Value        Count
 -------------- -------
             0        0
             1        3
             2       11
             3        7
             4       19
             5       62
             6        0
             7        0
             8        0
             9        0
            10        0
            11        0
      Overflow        0
@List Histogram for Leg: 0
Number of References               102
Sum of Values                  113,905
Average Value                    1,116
Scale Factor (2↑n)                   0
     Value        Count
 -------------- -------
             1        0
             2        0
             4        0
             8        0
            16        1
            32        0
            64        6
           128        6
```

```
               256       0
               512       0
             1,024      89
             2,048       0
          Overflow       0
@List Histogram for Leg: 1
Number of References                89
Sum of Values                   15,935
Average Value                      179
Scale Factor (2↑n)                   5
Base                                32
     Value        Count
 -------------- -------
         Underflow      17
                32       2
                64      14
                96       7
               128       4
               160       2
               192       4
               224       6
               256       6
               288      20
               320       2
               352       3
               384       2
          Overflow       0
@Quit [Confirm]
>Kill session [confirm]
```

Inter-Office Memorandum

To	Mesa Users	Date	December 15, 1978
From	J. Sandman	Location	Palo Alto
Subject	Control Transfer Counting Tool	Organization	SDD/SS/DE

XEROX

Filed on: [IRIS]<AlphaMesa>Doc>XferCounter.bravo .press

DRAFT

A tool for studying behavior of Mesa programs is described below. It counts the number of control transfers (xfers) to a module and records the time spent executing in a module. An xfer is the general control transfer mechanism in Mesa. The following are all xfers: procedure call, return from a procedure, traps, and process switches.

The system is implemented as a set of commands that can be executed from the Mesa Debugger, a routine that intercepts all xfers and collects statistics about them, plus a routine that intercepts conditional breakpoints for turning the xfer monitoring on and off. Existing Debugger commands are used to specify where xfer monitoring is enabled, and additional commands are provided for controlling the counting of xfers and outputting the results. Both Alto/Mesa programs as well as Pilot programs may use this tool.

This tool is intended to provide a global view of the behavior of a system. With this tool, a user can identify modules that warrant closer study will other tools such as the Performance Tool.

Components

XferCounter is the component of the tool that lives with user programs built on top of Alto/Mesa. This configuration contains one module: Counter. It contains the xfer trap handler and a breakpoint handler. XferCounter must be loaded and started in the system it will monitor. This may be done by including XferCounter in the client configuration whose control module starts XferCountDefs.Counter or by executing the following command to the Alto Executive:

```
>Mesa XferCounter Client
```

PilotXferCounter is the component of the measurement system that lives with user programs built on top of Pilot. This configuration contains one module: PilotCounter, which performs the same functions as Counter. Since there is no loader in Pilot 2.0, PilotXferCounter must be included in the client configuration whose control module starts XferCountDefs.PilotCounter. In addition, the code for PilotCounter must be made resident. Use the StartPilot command `ResidentCodeModule` by executing the following command to the Alto Executive:

```
>StartPilot ResidentCodeModule["PilotXferCounter>PilotCounter"]
Build["Client"]
```

XferCountPackage is the component that lives as a userproc in the Mesa Debugger. It implements the basic commands required to enable xfer monitoring and to output measurement results. XferCountPackage must be loaded into the Debugger before its commands can be executed. The easiest way is to load it when installing the Debugger by executing the following command to the Alto Executive:

```
>XDebug WindEx/l XferCountPackage/li
```

The command interpreter for the XferCountPackage is invoked by calling the userproc `XferCounter`. The userproc dispatcher is invoked by the `↑UserProc` command (`control-U`). If only one userproc is loaded, it is automatically called, otherwise some unique prefix must be typed when the dispatcher prompts for a procedure name. See the *Mesa Debugger Documentation: Debugger - Extended Features* for details.

Operation

When xfer monitoring is enabled and a xfer occurs, the xfer trap handler calculates the time since the last xfer and adds that to the cumulative time for the current module. It then calculates which module is the destination of the xfer and makes that the current module, incrementing its count. The xfer handler then completes the xfer and the user program resumes execution.

The state of xfer monitoring can be controlled by two methods. The first is by setting a conditional breakpoint to be handled by the tool's break handler. The second is by calling the procedures XferCountDefs.StartCounting and XferCountDefs.StopCounting.

When the break handler intercepts a breakpoint, it checks to see if the breakpoint is a conditional breakpoint. If not, the breakpoint handler proceeds to the debugger. If so, the state of xfer monitoring is changed and program execution is resumed. A condition of 0 turns on xfer monitoring. A condition of 1 toggles the state of xfer monitoring. A condition of 2 turns off xfer monitoring. Any other condition has no effect.

Since multiple processes may interfere with each other, there is the concept of the *tracked process.* If the tracked process is not NIL, only those xfers that are encountered by the tracked process are counted. All others are simply resumed. If the tracked process is NIL, then all processes are tracked.

Commands

The command interpreter completes commands like the Debugger command interpreter. The capitalized characters are all that must be typed to specify a command.

General Commands

`List Module`

displays the statistics for the specified module. The module may be specified by either its global frame table index (gfi), global frame address (g) or its module name if the current configuration contains the desired module.

`List Sorted by Time`

displays all the statistics for each module in order of decreasing time. May be aborted by typing `↑DEL`.

`List Sorted by Xfers`

displays all the statistics for each module in order of decreasing number of xfers. May be aborted by typing `↑DEL`.

`List Table`

displays all the statistics for each module. May be aborted by typing `↑DEL`.

`Monitor on`

turns on the tool's breakpoint handler. All conditional breakpoints will now toggle the monitoring switch.

`Monitor off`

turns off the tool's breakpoint handler. All conditional breakpoints will now behave like normal conditional breakpoints.

`Quit [Confirm]`

exits the XferCountPackage and returns to the Mesa Debugger.

`Track All Processes`

tells the XferCounter to count the xfers of all processes. All processes are counted in the default case.

`Track Process [process]`

tells the XferCounter to count only those xfers that are executed by the specified process. An octal ProcessHandle as obtained from the Debugger's `List Processes` command is acceptable as input to this command. A carriage return will set the process to all processes.

`Zero tables [Confirm]`

zeros out all counts and times.

Command Tree

This is the command tree structure for the XferCountPackage. It is formatted like the command tree for the Mesa Debugger (see *Mesa Debugger Documentation*).

```
List Module
     Sorted by Time
```

```
                  Xfer
        Table

    Monitor on
           off

    Quit

    Track All Processes
              Process

    Zero tables
```

Limitations

1. Execution Speed: Xfer monitoring slows down the executions of a program since extra processing is done on every xfer. As a result, interrupt processes will run relatively more frequently.

2. Idle Loop Accounting: When no process is running, the Mesa Emulator runs in its idle loop waiting for a process to become ready. This idle time is charged to the process that was last running.

3. Time Base: The time base available on the Alto is a 26-bit counter, where the basic unit of time is 38 microseconds. Thus the counter turns over every 40 minutes, and no individual time greater than 40 minutes is meaningful on the Alto. Total times are 32-bit numbers and will overflow after 340 minutes.

4. Overhead Calculation: Due to implementation restrictions and timer granularity, some of the overhead of processing a xfer is incorrectly assigned to the client program instead of the XferCounter. As a result, times must be interpreted as only a relative measure of the time spent in a module.

5. Counter Sizes: Counts are 32-bit numbers. The maximum total count is 4,294,967,295 xfers.

6. Table Size: The XferCounter's tables hold the data for the first 256 global frame table slots. If the global frame table is larger, some xfers may be ignored.

7. Memory Requirements: The XferCounter requires seven pages of resident memory; Two for XferCountBreakHandler's code, and five for XferCounter's frames and tables. This may affect the performance of some systems that use a lot of memory, especially on the Alto.

8. XferCounter's break handler acts like a worry mode breakpoint and as a consequence, you may find you cannot `Quit` from the Debugger after your session. Use the `Kill` Debugger command instead.

Getting Started

Outlined below are the steps required for using the measurement tool.

1. obtain the bcd's for XferCounter and XferCountPackage.
2. install the XferCountPackage in the Mesa Debugger (version 4.1 or later).
3. start your program executing with the XferCounter included.
4. enter the Debugger and set conditional breakpoints to enable monitoring as desired.
5. turn the break handler on via the `Monitor on` command.
6. proceed with program execution.
7. return to the debugger via control-swat or an unconditional breakpoint.
8. display results with the `List` commands.

Sample Session

The following annotated listing of a DEBUG.TYPESCRIPT session should give a fair idea of the use of the measurement tool.

```
Alto/Mesa Debugger 4.1 of  6-Sep-78 18:47
 4-Dec-78 11:17

*** interrupt ***
>SEt Module context: Loader  -- Count xfers involved in loading a configuration
>Break Entry Procedure: New, Condition: 0   -- Start monitoring when hit this break
>Break Xit Procedure: New, Condition: 2     -- Stop monitoring
>userProc [confirm]
Proc: XferCounter
@Monitor on
@Track Process: 2770  -- Track only the main process.  Ignore the keyboard process
@Quit [Confirm]
>Proceed [confirm]
*** interrupt ***
>userProc [confirm]
Proc: XferCounter
@List Table
Total Xfers      5,884
Total Time     950:152
 Gfi Frame   Module Name                # Xfers % Xfers        Time % Time
 --- ------- ----------------- ----------- ------- ----------- ------
  1B 173760B Resident                     4     .06         152    .01
  3B 173740B DiskIO                     335    5.69     311:752  32.81
  4B 173314B Swapper                    794   13.49      95:874  10.09
 10B 173040B LoaderUtilities            245    4.16      19:380   2.03
 11B 173024B LoadState                   95    1.61      87:020   9.15
 12B 173020B LoaderBcdUtilities         973   16.53      69:160   7.27
 13B 173014B Loader                   2,207   37.50     264:556  27.84
 21B 172740B NonResident                 59    1.00      15:732   1.65
 22B 172730B Segments                   432    7.34      26:182   2.75
 24B 172724B Strings                    635   10.79      51:300   5.39
 25B 172714B Files                       48     .81       4:636    .48
 31B 172700B FSP                         57     .96       4:408    .46

@List Sorted by Xfers
```

```
Total Xfers       5,884
Total Time      950:152
 Gfi Frame   Module Name          # Xfers % Xfers        Time % Time
 --- ------- ------------------ ----------- ------- ----------- ------
 13B 173014B Loader                 2,207   37.50     264:556  27.84
 12B 173020B LoaderBcdUtilities       973   16.53      69:160   7.27
  4B 173314B Swapper                  794   13.49      95:874  10.09
 24B 172724B Strings                  635   10.79      51:300   5.39
 22B 172730B Segments                 432    7.34      26:182   2.75
  3B 173740B DiskIO                   335    5.69     311:752  32.81
 10B 173040B LoaderUtilities          245    4.16      19:380   2.03
 11B 173024B LoadState                 95    1.61      87:020   9.15
 21B 172740B NonResident               59    1.00      15:732   1.65
 31B 172700B FSP                       57     .96       4:408    .46
 25B 172714B Files                     48     .81       4:636    .48
  1B 173760B Resident                   4     .06         152    .01

@List Sorted by Time
Total Xfers       5,884
Total Time      950:152
 Gfi Frame   Module Name          # Xfers % Xfers        Time % Time
 --- ------- ------------------ ----------- ------- ----------- ------
  3B 173740B DiskIO                   335    5.69     311:752  32.81
 13B 173014B Loader                 2,207   37.50     264:556  27.84
  4B 173314B Swapper                  794   13.49      95:874  10.09
 11B 173024B LoadState                 95    1.61      87:020   9.15
 12B 173020B LoaderBcdUtilities       973   16.53      69:160   7.27
 24B 172724B Strings                  635   10.79      51:300   5.39
 22B 172730B Segments                 432    7.34      26:182   2.75
 10B 173040B LoaderUtilities          245    4.16      19:380   2.03
 21B 172740B NonResident               59    1.00      15:732   1.65
 25B 172714B Files                     48     .81       4:636    .48
 31B 172700B FSP                       57     .96       4:408    .46
  1B 173760B Resident                   4     .06         152    .01

@Quit [Confirm]
>Kill session [confirm]
```