XEROX
BUSINESS SYSTEMS
System Development Division
November 9, 1977

To: Distribution

From: Dick Sweet

Subject: Mesa Language Working Group minutes

The Language Working Group met on 4 November, 1977. The items considered were inline procedures and monitors.

Inline Procedures

For inline procedures, syntax is needed in three cases: at the declaration of the procedure in a program, at the declaration of a procedure in a defs module, in an implementing module that provides an out-of-line body for a procedure declared in a defs module, and at the point of call.

All declarations of *inline-ness* goes on the body of the procedure. The choices of syntax and their meanings are:

```
foo: PROCEDURE [...] = INLINE BEGIN ... END;
```

> The default method of calling is inline, no body is generated. In a program module, this implies that the only way of calling is inline.

```
foo: PROCEDURE [...] = USUALLY INLINE BEGIN ... END;
```

> The default method of calling is inline, but a body is also generated. This is presumably illegal in a definitions module, although it could determine whether or not the procedure goes into the interface.

```
foo: PROCEDURE [...] = OPTIONALLY INLINE BEGIN ... END;
```

> The default method of calling is out-of-line. In a definitions module, this certainly generates a slot in the interface, which some implementing module must provide an instance of the body to fill.

If a procedure is declared USUALLY or OPTIONALLY INLINE in a defs module, an implementing module can instantiate a body by the declaration:

```
foo: PROCEDURE [...] = BODY;
```

At the call site, the default method of calling may be overridden by the statements:

```
INLINE foo[...];     and     OUTOFLINE foo[...];
```

This has ramifications of the current syntax for MACHINE CODES. The old and new syntax are as follows:

```
baz: MACHINE CODE [...] = INLINE [byte, byte];   -- old

baz: PROCEDURE [...] = MACHINE CODE BEGIN byte; byte END;   -- new
```

Monitors

The following topics were listed for consideration

1. Independent FORK
2. Return type of a FORK
3. Questions of scope
4. Initialization of Monitors and condition variables
5. Monitor priority
6. Condition timeout
7. Aborts
8. Interaction with SIGNALS

Items 1 and 2 could probably be handled by having the FORK construct return a procedure that is called when one wishes to JOIN the process. There are potential troubles with item 7 from this proposal, although a system routine could figure out from the procedure who to kill.

Item 3, scope, was considered at some length. First, three styles of monitor declaration were given.

Basic Style

```
M: MONITOR [args] =
   BEGIN
     .
     .
     .
   p: ENTRY PROCEDURE [...] =


   END.
```

Pack Style

```
M: MONITOR [args] LOCKS arg=
   BEGIN
     .          where arg is a POINTER TO MONITORED RECORD
     .
     .
   p: ENTRY PROCEDURE [...] =


   END.
```

Object Style

```
M: MONITOR [...] LOCKS f(obj) =
   BEGIN
     .        the function, f, would be expanded in the context
     .        of the entry procedure
     .
   p: ENTRY PROCEDURE [obj, ...] =


   END.
```

There was a proposed alternative form:

```
M: MONITOR [...] =
   BEGIN
     .
     .
     .
   p: ENTRY PROCEDURE [obj, ...] LOCKS f(obj) =


   END.
```

The remaining discussion concerned allowing multiple monitors within a single module. First the current uses of modules were given

Scope
Global Frame
Source code - compilation unit
Object code - swap unit

Some of these uses are being changed with other changes for Mesa 4.0, such as swap unit.

Pros and cons of sharing global frames by monitors were considered

Pro	Con
Saves 3 wds/monitor	Worse code generated
Allows local calls	Bad for structure with current language
Saves gft entry	

The concensus seemed to be that it was not worth the trouble.