Inter-Office Memorandum

To	Distribution	Date	November 15, 1978
From	John Wick	Location	Palo Alto
Subject	**Binding variables in Mesa**	Organization	SDD/SS/DE

XEROX

Filed on: [Iris] < Wick > Design > BindVar.bravo

DRAFT

This memo is part of a series addressing questions of implementation detail relevant to Mesa 5.0 (see also the minutes of the Mesa Language Working Group). It describes a scheme that allows variables (in addition to procedures, signals, and programs) to be bound between modules.

The following is taken from Satterthwaite, *Inline Procedures*, August 8, 1978, page 4:

We have assumed that the environment of an inline procedure with free variables (other than interface items) is specified by supplying a pointer to a global frame. Peter Bishop has suggested an alternative approach based upon generalizing the concept of an interface item: (pointers to) objects with arbitrary types can be importable and exportable components of interfaces. In several examples of interest to him, this reduces cases 4 [inline defined in PROGRAM module, called in DEFINITIONS module] and 6 [defined in PROGRAM module, called in PROGRAM module] to cases 2 [defined in DEFINITIONS module, called in DEFINITIONS module] and 3 [defined in DEFINITIONS module, called in PROGRAM module] respectively. His proposal would also decouple modules more effectively. On the other hand, this extension has rather far-reaching implications for the notion of interface, BCD formats, etc. Example:

```
Defs: DEFINITIONS =
  ...
  a: ARRAY Index OF T;
  Proc: PROCEDURE [...] = INLINE
    BEGIN
    ... a[i] ...
    END;

Implementer: PROGRAM EXPORTS Defs =
  ...
  a: PUBLIC ARRAY Index OF T;    -- the storage for a

Client: PROGRAM IMPORTS Defs =
  BEGIN
  ... Defs.Proc[...]; ...    -- a referenced through binder-supplied link
  END.
```

XEROX SDD ARCHIVES
I have read and understood
Pages_____To_____
Reviewer_____Date_____
# of Pages_____Ref. 78SDD-216

While the proposal does have "far-reaching implications for the notion of [an] interface", it turns out to be rather easy to implement; with suitable restictions, no change in BCD format is required, and only the compiler and loader (but not the binder) are affected.

Note: Several religious arguments might be made against this proposal (e.g., "global variables considered harmful"). However, it solves so many problems related to inline procedures that we can't hardly pass it up. The manual should contain some advice on when *(not)* to use exported variables.

Binding Variables

First, we expand the notion of an interface to include all binding-time variables, not just procedures and signals. Each such variable is assigned a slot in the interface record. When a program module claims to be exporting the interface, any public variable which matches a name in the interface is being exported, and an entry is made in the module's export record for that interface. Except for type matching, this processing is exactly as for procedures, signals, and programs in the current scheme.

For a procedure, the relevant entry in the export record, in addition to the *gfi* of the implementor, is an entry point number (*epn*), which points (indirectly, through the module's entry vector) to the corresponding procedure body. For a variable, the relavent contents of the export record is an external variable number (*evn*), which similarly points (perhaps indirectly) to the appropriate variable in the implementor's global frame. The two cases are distinguished (if need be) by the tag of the entry: zero for programs and frame variables, one for procedures and signals.

You might think of this scheme as a simple extension of the current mechanism for binding module instances (implemented as global frame pointers). Think of an imported module as an importation of the global frame header (corresponding to *evn* zero) of that module; now expand the concept to include frame addresses (*evn*s) other than zero. It is for this reason that the scheme can be implemented without changes to the binder. The loader need only be extended to recognize non-zero *evn*s attached to imported frames, and treat them as frame offsets.

External Variable Descriptors

The external variable number will actually index a table appended to the implementing module's BCD which gives the offset in the global frame of the exported variable. Alternately, the *evn* might actually be the offset in the global frame. Using the current procedure descriptor encoding, this would restrict exported variables to be within the first 128 words of the frame (the same restriction as the number of entry points per module). Although the compiler could provide some help in placing these variables, this restriction seems pretty unreasonable for all concerned. Since the number of exported variables per module should be small, the extra table won't add much to BCD sizes.

The format of the external variable table (*evt*) is straightforward: each entry (an *EVEntry*) is an array of frame offsets, indexed by *evn*. The entries are pointed to by a field in each module table record (an *EVIndex*, which is a relative pointer). Multiple instances of a module can all point to the same *evt* entry. The Binder has only to merge all the entries into a single table, updating the *EVIndicies* in the module table. It is unclear at this point whether the length of an *EVEntry* is required; if so, it could easily be a sequence rather than an array.

There is some question about what the format of an external variable descriptor should be. If we use the current procedure descriptor format, including the tag bits, then modules which export more than thirty-two variables will be allocated extra GFT slots, even though they won't be used for anything at runtime; this seems unpleasant. Alternately, the low-order seven bits of the descriptor can be used for the *evn*. This means that the type of an exported item can not be determined from the export record, and must be got from the importers (where a tag of zero in a frame fragment item indicates an imported variable). As far as we know, this effects only the BCD lister.

Now all that is needed is a representation for unbound variables; NIL is the obvoius choice. It is already used for unbound PROGRAMs, and it will cause a trap if NIL checking is turned on.

Open Issues

In comparing this scheme to the current POINTER TO FRAME mechanism for accessing global variables, two deficiecies come to mind. First, you need one pointer to get at each external variable, instead of a sigle (frame) pointer for each exporter. The advantages of decoupling the importer and exporter (with respect to compilation dependencies) probably outweigh this, but a more complicated scheme that required only one pointer per exporter could probably be worked out (but not for Mesa 5.0).

A related issue is that the code for accessing external variables won't be very good, as the only way to get at them will be with the load link (LLKB) instruction. That is, the magic global locations and instructions will never be used, and all references will be done starting with the pointer on the stack. One might think that, if the external were referenced often enough, the compiler could generate code to copy the link into a magic global, but the compiler has no idea when the link will be bound, so it doesn't know when to make the copy.

The current thought is to live with these problems until we have a better idea of how (often) exported variables are used.

Distribution:
Geschke Johnsson Koalkin Lampson Levin Malasky Mitchell
Sandman Satterthwaite Sweet