Copy - #3

Inter-Office Memorandum

To	Hal Lazar	Date	December 17, 1974
From	P. Heinrich, W. Shultz [illegible]	Location	A3-17/Ext. 1571
Subject	Selection of a System Programming Language for OIS	Organization	ITG IPD SRAP - WS27

XEROX

Introduction

Although there is not much question that we need a higher level Systems Programming Language for OIS software development, let us restate the chief reasons for this need:

1. It permits more modular, machine independent implementation of software systems.

2. It reduces development and maintenance costs for large software projects (as opposed to assembly language development techniques).

3. It makes it easier for a total system (both hardware and software) to grow and evolve over time.

Given, then, that we want to use a higher level language for development of OIS software, we have several choices:

1. We can invent a new language.

2. We can adopt an industry standard.

3. We can adopt or adapt an existing Xerox language.

We rejected alternative #1--the world does not need another systems language. We really can't select alternative #2--there really is no industry standard for systems work, although Algol and PL/1 derivatives are both common and successful. We choose instead alternative #3. Of the possible Xerox programming languages, only three seem to be appropriate candidates for OIS software development. These are:

1. SPL - Structured Programming Language developed by DSD in El Segundo.

2. BCPL - Basic CPL, used at PARC for Nova and some Alto work.

3. MPL - Modular Programming Language, developed at PARC (also called Mesa).

We rejected BCPL as being too basic and too primitive. (For example, it has no data structure except 16-bit integers.) This left only SPL and MPL.

For the past several months, as time permitted, Jim Frandeen and the authors have looked into the relative advantages of SPL and MPL. We have traveled to PARC, and Jim Mitchell has spent time with us in El Segundo. We have looked into SPL work here in El Segundo. We have also come to understand the needs and system architecture of OIS better than before. From this study we have come to a conclusion regarding OIS software development. As noted below, there would be company-wide advantages at standardizing on a single Systems Programming Language for all Xerox software development (for computers and copiers), but we make no recommendation in this regard. We do discuss the implications of this later, however.

Conclusion

We have concluded that the best language for developing all OIS software is MPL--the Modular Programming Language developed at PARC. A detailed analysis of the reasons for this decision are presented below. Also, a preliminary action plan to implement this decision is presented later in this paper.

A detailed comparison of SPL and MPL is given below. In fact, SPL would be a very adequate language for OIS software development, but MPL is a better language, in our opinion. This made the evaluation more difficult. But we were fortunate that Xerox had two languages that were worthy of consideration, since this helps insure that the one OIS language can benefit from the experience of both.

We now recommend that other people in the company who are interested in Systems Programming Languages look at MPL and at the implications of standardization.

Technical Evaluation

Quality or Feature	SPL	MPL
Adequate for OIS system software development	Yes	Yes
Compatibility with OIS Processor Architecture	Adequate	Excellent (Oriented to base register and recursive use)
Features present but not required for OIS	Floating point (not yet implemented)	None
Easy to learn and to program	Very good	Very good
Easy to build and debug complex systems	Barely adequate (weak debugging in current version)	Excellent (data typing and parameter checking combined with complete debug system)
Compiler operating efficiently on development machine (Sigma)	Reasonable (not great, but could be improved)	Expected to be reasonable. (May be slow due to lack of base registers)
Produces efficient object code (for OIS)	Too early to determine	Too early to determine
Permits symbolic, packed data structures	Yes	Yes
Permit explicit control of registers and basic hardware instructions	By assembly level procedures	By in-line assembly level code
Optional limits and trace (debug)	Not now	Yes.
Permit modular program construction and debugging	Yes (adequate)	Yes (better, because of data typing and checking)
Permit open and closed subprogram use	Yes (by use of "compile time" variables)	No (only closed)
Permit user control of synchronous error processing	Not yet. Plans to add with ON statement	Use of SIGNAL feature
Permit user control of asynchronous events (for I/O end action or inter-task communication)	Not yet. Might be able to add with part of ON feature.	Variation of SIGNAL feature
Machine independence	Compiler not completely independent; also, I/O calls are strictly dependent on CP-V; source could be if user is careful	Easier to be machine independent since data typing specifies value range not bit range
Structured programming made easier	Adequate; good block structured language	Better, since more disciplined data typing
I/O facilities (for user)	At source level, but for CP-V-type I/O only	By use of special built-in procedures
Self-documenting	Good	Better, again due to data typing
Desirable advanced features (see attached memo by J. Frandeen)	None	Co-routine and recursion
Basic extensibility of compiler and language	Compiler code is very marginal. Little experience on language itself.	Compiler coding practices are excellent. Language is very general.

Other Considerations

Language Maturity: We are concerned about language maturity (or actual use) because this determines its current stability and quality. No language is born perfect--it must grow and develop and mature over a period of time.

SPL is derived from PL/1 with many changes, mostly specified by Mike Kuppin, with inputs from others (including Evergreen Associates). The first version is just now being completed. Thus, there is little real experience with using SPL for systems work. The only significant program to date written in SPL is the compiler itself, and it is not a very good implementation.

MPL version A was derived from QSPL (on the 940, by Lampson and Deutsch), an SPL at BCC (by Lampson and Mitchell), BLISS (Carnegie-Mellon) and L10 (SRI). MPL version B (the current version) was derived from MPL-A as modified by ideas from ECL (at Harvard, by Wegbreit), Simula 67, Pascal, and minor influence from BCPL. It has been used for several months by many people at PARC, and will be used by several other projects at PARC in 1975. Therefore, although not as clear a derivation from as major a language as PL/1, it is felt that the language is well-tested by a group of very sophisticated system programmers and is therefore currently more mature than SPL.

Timeliness: Basically, will the language be ready when we need it for OIS? We will not begin coding for OIS operating software before early 1976, although we need to know what the language is in 1975 to permit software specifications to be developed and to permit hardware/software optimization to begin. Also, OIS software development tools can be started by mid-1975, if funds permit.

SPL will be available in a form usable for this work by 4Q75. Some features not in the current language (such as the ON statement) need to be added, and the support system needs to be built up extensively. A higher efficiency production version would probably not be available before 4Q76.

MPL will be usable at PARC in early 1975, and could be converted to Sigma 7 by 4Q75 for development work. A higher efficiency production version could probably be ready on Sigma by 2Q76.

Support Software: The compiler is only one element of a large software development system. Without a good total system, the compiler is not very valuable.

SPL essentially has no support system of its own. It uses the CP-V loader, debugger, source editor, file editor, and file management. This results in rather poor symbolic debugging and updating.

MPL has a complete support system, written in MPL, including:

- o A loader (link editor)
- o A file editor
- o A source editor
- o A debugger

MPL uses the MAXC file management services for basic file support, but could be converted to CP-V.

Under CP-V, either MPL or SPL programs could be created and debugged interactively.

<u>Personnel</u>: We assume the responsibility for the final development of the OIS System Programming Language would rest in El Segundo under the OIS budget center. There are three people on the current SPL project in DSD. There is one person in OIS development (Jim Frandeen) who could work on either SPL or MPL. There are about three people at PARC on MPL development, but they could do little more than consult with us in 1975.

<u>Documentation</u>: A compiler is no good without adequate documentation.

SPL will have a preliminary reference manual out in January, 1975. There is no user guide or specific support documentation.

MPL has very brief language and support software reference documentation available now. We would have to work with the people at PARC to help increase the availability of user documentation in early 1975.

<u>Cost</u>: A precise costing has not been performed, due to the uncertainties in OIS requirements and the schedules and manpower involved. An action item below calls for a more detailed plan in this regard. However, our estimates are that there is no significant difference in the final cost to OIS, for these two alternatives. That is, although SPL is already on Sigma 7 under CP-V, it definitely requires some extensions, some more support software, and significant performance rework. MPL is a good implementation, but must be converted from MAXC to Sigma 7. Both require a new object code generation module, to prepare output for OIS hardware.

MPL Development for OIS

The required modifications to MPS for OIS fall into two categories: conversion to Sigma/CP-V and enhancement for OIS. (The conversion to Sigma/CP-V is to permit development work under CP-V.)

Conversion:

1. The I/O interface between MPL and the operating system must be converted to CP-V service calls.

2. The object code generation phase must be changed to produce Sigma object code and special provision must be made for dynamic reference to variable data.*

3. An option to allow static procedures (no recursion) should be implemented to permit more efficient code to be generated for Sigma.

Enhancement:

1. An object code generation phase will be written for the OIS processor. This will be an interactive process which involves testing the object code, redefining the instruction set, changing the compiler and microcode, then testing again.

2. Automatic instrumentation for program performance analysis.

3. Development of other language related development tools such as program analyzer that checks for adherence to standards; also simulators and utilities.

4. Investigation of other enhancements in the area of OIS control program/MPL interface (for I/O and interrupts).

---

*MPS takes advantage of the base register architecture of MAXC and ALTO to allocate and access dynamically local variables. The absence of base registers on Sigma will require an object code modification to index registers before they are used to reference local dynamic variables.

Action Plan

An OIS/MPL development team of two to three people must be identified and brought together by the middle of January. Initially, the team will spend time at PARC familiarizing themselves with MPL and support software, and with other related materials.

The Sigma conversion effort should begin no later than March 1, 1975. Thereafter, the first order of business for the team is a development plan for the enhancement of MPL for OIS. This development plan will detail features, schedule and manpower required. By that time, OIS architecture will be firm and control software will be better understood.

It is expected that the OIS/MPL team will help the MPL staff at PARC produce user-level reference documentation during the first half of 1975.

A preliminary schedule goal is to have MPL operating on Sigma under CP-V by 4Q75 for the development of other OIS development software.

As part of the support plan, an alternative to the Sigma development work should be explored; namely, to investigate using MPL on ALTO for some early development and learning. (The PARC MPL staff is currently converting MPL to ALTO.)

Open Issues

There are several issues which remain open following our recommendation of MPL for OIS development.

o OIS Hardware Architecture

Although a preliminary version is proposed, the hardware architecture is not yet fixed. A radical change in hardware design could impact the recommendation we have made. To a great extent, the hardware design being proposed by the architecture team has been influenced by and oriented toward the MPL language, because of the interaction between Alto and MPL at PARC. The intention was and is to produce an optimal hardware/software product.

o Computer Division Higher Level Language

Can the computer Division use MPL? Should SPL development be halted and joint development of MPL started? What new or revised computer products require, or can use, a higher level language? This question is significant because if joint development does not occur, additional funds will have to be generated in OIS to staff MPL development.

o Standardization vs. the Tower of Babel

Can and should other products or programs such as DPS, PECOS, RASCAL, etc., use MPL? For PECOS III the answer is certainly yes. All OIS products must use a common language. It is highly desirable to have a company-wide standard higher-level language system. It avoids duplication, increases the pool of trained personnel, and focuses our efforts in one direction, and it helps permit us to build unified systems out of modular, stand alone products.

lg

Attachment: memo by J. Frandeen on OIS/MPL

c: OIS Architecture Board
B. Beeson
K. Campbell
J. Elkind
S. Klee
W. Klein
A. Kopito
B. Lampson
A. Lipton
C. Martin
J. Mitchell
R. Spinrad
E. Vance

To Peter J. [illegible] Date 10 December 197[illegible]

From Jim Frandeen Location A1-63/Ext.1541

Subject Reentrancy, Recursion and Coroutines for OIS Programming Language

Organization Development Programming CS-74-7111

XEROX

Consideration of Mesa as the language used to implement OIS has led to discussion of the following language features: coroutines, recursion, and reentrant procedures. Since these features are implemented in Mesa but not is SPL, it is important to understand how they are used and whether or not we need them for OIS.

REENTRANT PROCEDURES

A procedure is said to be reentrant if more than one activation of the procedure can exist at any one time.

Reentrancy is usually associated with multiprogramming and task switching. Multiprogramming is the use of an operating system to execute a number of tasks concurrently. A task is an activation of a program. A program is a collection of one or more procedures. It is important to distinguish between a procedure and a task. A procedure is a module of executable code. A task is an activation of a procedure, including its context and local variables. If a procedure is reentrant, it can be shared by several tasks. Consider the following multiprogramming example. The system I/O supervisor is executing for task A. The I/O supervisor is building an I/O control block for this task. Task A is interrupted because task B, which has a higher priority, becomes ready to run. Task B invokes the I/O supervisor, and it begins building an I/O control block for task B. Note that the I/O supervisor never finished building the I/O control block for task A; it must be able to continue where it left off when task A is reactivated. With this example, we can understand the requirements of a reentrant procedure:

1. The code must not modify itself.

2. Variables that are local to the procedure must be unique for each activation of the procedure -- otherwise, a second activation of the procedure would destroy variables from the previous activation.

For the OIS system, it will be necessary for some procedures to be reentrant. Without reentrant procedures, a multiprogramming type of environment is not possible. There are two possible methods for implementing reentrant procedures.

In the first method, reentrancy is handled by the code of the procedure whenever the procedure is activated. In Mesa, when a procedure is activated, the initialization code of the procedure calls a system routine to allocate a frame and then sets a base register to point to the frame. A frame is a record that contains the context of the procedure -- the return address, parameters passed to it, a place to save the registers if the procedure is interrupted, and all local variables. The code addresses all local variables as an offset from the beginning of the frame pointed to by the base register. In the example above, the I/O supervisor would begin building an I/O control block in the frame of task A. When interrupted and reentered, a new frame would be allocated, the base register would be set to point to the frame of task B, and the I/O supervisor would begin building a different I/O control block for task B. Later, when task A is reactivated, the base register is restored to point to the frame of task A, and the code continues execution where it left off.

In the second method, reentrancy must be handled by the operating system. Executable code and local variables are placed in separate pages of memory. If a procedure is to be reentered, the operating system must first save the local storage of the active task. This can be cone by swapping out local storage or, in a virtual machine such as the Sigma 7, by changing the memory map. In SPL, code and local storage are placed in separate control sections so that reentrancy could be implemented by the operating system.

It seems more desirable to have reentrancy handled automatically by the code of the reentrant procedure. First of all, the language will be more machine independent if reentrancy is handled by the language. If reentrancy is handled by the operating system, we must depend on a virtual machine, and this will make it very difficult to adapt the language to different machines. It is not yet clear what the architecture of the OIS machine will be. Even if we choose a virtual machine for OIS, we hope that the language we choose for OIS will be used for other applications throughout Xerox on a variety of machines. Secondly, if local storage is allocated by the procedure, this storage uses core space only while the procedure is active. For static procedures, local storage must be permanently allocated. For small machines, the savings in core storage provided by dynamic local storage can be considerable.

RECURSION

An active procedure that can be reactivated from within itself or from within another procedure is said to be recursive; such reactivation is called recursion. This characteristic is extremely important because some kinds of problems require this kind of capability, and others are stated most easily by using recursion. To clarify what is meant by recursion, the classic example of the factorial is most easily understood:

```
    FACTORIAL:  PROCEDURE(N);

          IF N = 1, THEN ANSWER = 1;

          ELSE ANSWER = N*FACTORIAL(N-1);

    END FACTORIAL;
```

In this case, the FACTORIAL procedure is repeatedly called from within itself to find the value of N!

Recursive routines are especially useful in parsing. For example, consider a simple grammar:

```
     variable      ::= A|B|C|D|E

     expression    ::= <variable>|<variable> + <expression>

     statement     ::= <variable> = <expression>
```

The symbol '::=' can be read as 'is defined as'. The vertical bar represents 'or'. Examples of statements according to this grammar are:

A = A

B = C

A = B + C + E

Now, suppose we want to write a routine that will test to see if a character string is a valid expression. Since the definition of an expression is recursive, the most natural way to design the routine is to make it recursive. When the routine is scanning 'B+C+E' it gets to 'B+' and then it wants to know if the rest of the string is an expression, so it calls itself.

Other problems cannot be easily solved without recursion. The classic example is the asynchronous error routine which is called whenever an I/O error is detected. Suppose a tape read error occurs during processing, and the error routine is activated. The error routine wishes to send a message to the operator and wait for a reply, but an I/O error occurs during this interaction. Different systems handle this problem with varying degrees of sophistication.

- The error routine is reactivated, but since the routine is not recursive, the results are disastrous.

- The error routine checks to see if it is being reactivated, and if so, it causes the program to abort.

- The operating system forbids the program to execute I/O operations in the error routine.

Of course, if the error routine is a recursive procedure, there is really no problem.

As might be expected, recursion is related to reentrancy. Any procedure that is recursive must be reentrant. Reentrant procedures are not always recursive, depending on the implementation. However, if a language is being designed to handle reentrancy, it is not difficult to provide recursion as well. Mesa handles recursion and reentrancy automatically for all procedures. Since we will need reentrant procedures for OIS, we should provide recursion as well.

Efficiency of Recursive and Reentrant Code

Recursive and reentrant procedures are desirable features, but these facilities are not free. The user pays for them with increased overhead - call/return overhead and local storage access overhead. Each time a recursive/reentrant procedure is invoked, a great deal of overhead is required to allocate a frame and initialize the storage in the frame. Similarly, when the procedure terminates, the frame space must be freed.

Besides call/return overhead, additional overhead may be necessary in order to access variables in local storage. In a recursive/reentrant procedure, local variables reside in a dynamic frame pointed to by a register. The compiler generates code that addresses each variable as a displacement D from the beginning of the frame, plus the contents of a base register B. Every local storage access must specify a two-part address -- a displacement D and a base register B. On base register machines such as the 360, this works out very well because most instructions that access memory have a three-part address -- a displacement D, a base register B, and an index register I.

On the 32-bit Sigma machines, memory access instructions have a two-part address -- a 17-bit displacement D (a word address) and an index register I. It is possible, of course, to access variables in a frame by using the Sigma index register as a base register. This works perfectly well for accessing a fullword variable:

```
LW,R  D,B
```

This instruction loads into register R the fullword specified by the two-part address field D,B. The hardware computes the address by adding the displacement D to the base address contained in register B. The problem occurs when the access requires indexing. Suppose you want to access the ith occurrence of the variable, and index register I contains the index. You would like to write something like:

```
LW,R  D(I),B
```

On a base register machine, this is exactly what you would do, and the instruction would fit in one 32-bit word. The hardware computes the address of the word to be accessed by adding the contents of base register B and index register I to the displacement D. On Sigma hardware, this same memory access would require three words:

```
LW,C  I

AW,C  B

LW,R  D,C
```

The first two instructions compute a new base register C by adding the contents of the base register B and the index register. This is in effect simulating what base register machines such as the 360 do automatically.

The language we choose for OIS must be implemented to run on the Sigma 7 under CP-V. This will permit development work on OIS to begin before OIS hardware has been designed. The compiler can subsequently be adapted to generate object code for the OIS machine. In addition to use for OIS, it is hoped that the language chosen for OIS can be used for other applications throughout Xerox.

The requirements for recursion, reentrancy and implementation on the Sigma 7 seem to be contradictory. No matter what language we choose, recursion and reentrancy cannot be implemented efficiently on the Sigma 7 because the hardware does not provide base registers and instructions to facilitate the call/return sequence. This problem can be resolved rather easily if recursion and reentrancy are optional features of the language. Not all procedures need to be recursive and reentrant. For static routines, the compiler would simply allocate frame space with the object code, making all variables directly addressable. This is what SPL does for all procedures.

There is no question whether or not Mesa can be adapted to run on the Sigma 7. The question is "how efficient will it be?". The Mesa language provides recursion and reentrancy automatically for all routines. Would Mesa compilations take an inordinate amount of time? Would all Mesa programs run inefficiently on Sigma hardware?

We could study these questions at great length, but this would still not solve the problem of additional overhead when recursion and reentrancy are not really needed. A simple solution to this problem would be to add a facility to Mesa that would permit routines to be declared static. Implementing static procedures in Mesa would be easier that implementing recursion and reentrancy in SPL.

COROUTINES

Coroutines provide a very useful control structure. Coroutines are closely related to subroutines. The main program and a subroutine operate in a master/slave relationship -- the main program calls the subroutine, the subroutine begins execution at its beginning, runs to completion, and returns to the main program. The main program then continues to execute at the instruction following the subroutine call.

In contrast to this master/slave relationship between the main program and a subroutine, the relationship between coroutines is completely symmetrical. Coroutines call each other, and it is impossible to tell which is the subroutine of the other. A good example of a coroutine structure would be two chess playing programs. We will call one program Black and one program White and and activate one program (say White) first. White computes its move and calls Black. When Black is activated, it computes its move and calls White. Each time a coroutine is activated, it continues at the place where it last terminated. Coroutines are used most naturally for input/output routines. For example, consider the following coroutine written in Mesa.

```
nextchar:  COROUTINE RETURNS [CHARACTER] =
-- this routine returns the next input character.
-- after every card it returns a blank as the next character
BEGIN
card:  CHARACTER [80];            -- input card is character arr
i:  INTEGER;                      -- i indexes next character
DO                                -- DO forever
  readcard [card];                -- fill card buffer
  FOR i = 1 TO 80 DO
    RETURN [card [i] ];           -- return next character
  END                             -- continue loop next activat
  RETURN [SP];                    -- return blank at end of card
END                               -- repeat DO loop
END
```

This routine acts very much like a subroutine. The statement

```
c ←— nextchar[ ];
```

would activate the coroutine nextchar and assign the next character to c. Internally, there are some important differences between a coroutine and a subroutine.

1. Like a subroutine, local storage is allocated the first time a coroutine is activated. However, this local storage is not freed by the RETURN statement. In the chess example, this would be like upsetting the board after every move. The coroutine wants to remember what it was doing the next time it is entered.

2. When a subroutine is called, it always starts at the beginning. When a coroutine is called, it always continues at the point where it last terminated.

On the Need for Recursion and Coroutines

Recursion and Coroutines are two features provided by Mesa that SPL does not support. The question is "do we really need these features in the language we choose for OIS?". The answer is probably "no, we don't really need them -- these features are fairly new in the programming world, and we can continue to do things the way we have always done them". However, these are very useful tools, and many problems are most naturally solved by using coroutines and recursion. These tools do not fall into the category of "a better mousetrap". A tool of the better mousetrap variety is the literal. The literal constant is certainly a convenience because the programmer doesn't have to think of a name for the constant and declare it somewhere in the program; he can keep his mind on the problem and let the compiler worry about such routine matters. The literal provides a more convenient way to do something. Recursion and coroutines, however, really provide new ways to think about problem solving. With regard to programming tools and thinking habits, the following quote from Dykstra's "The Humble Programmer" is very relevant.

"I observe a cultural tradition, which in all probability has its roots in the Renaissance, to regard the human mind as the supreme and automonous master of its artifacts. But if I start to analyze the thinking habits of myself and my fellow human beings, I come, whether I like it or not, to a completely different conclusion, viz. that the tools we are trying to use and the language we are using to express or record our thoughts are the major factors determining what we can think or express at all! The analysis of the influence that programming languages have on the thinking habits of their users, and the recognition that, by now, brainpower is by far our scarcest resource, these together give us a collection of yardsticks for comparing the relative merits of various programming languages."

<u>SUMMARY</u>

The language we select for OIS must provide reentrant procedures. Without reentrant procedures, a multiprogramming type of environment is not possible. If the language handles reentrancy, then it should be able to provide recursion at little extra cost. The language we choose for OIS must be implemented on Sigma 7 under CP-V. This will permit development work on OIS to begin before the hardware has been developed, and it will permit the language to be used for other applications throughout Xerox. Since reentrancy and recursion cannot be efficiently implemented on Sigma hardware, recursion and reentrancy must be optional features of the language. The compiler will produce static procedures when these features are not needed.

Recursion and coroutines are very powerful tools that provide new ways to think about problem solving. If we are considering a language that will be used into the 1980s, the language should include these features.

Jim Frandeen

Jim Frandeen
OIS Project

cs

c: L. Cozza, S. Klee, A. Kopito, C. Martin, J. Mendelson, W. Shu

Inter-Office Memorandum

To	Hal Lazar	Date	2 January 1975
From	Jerry Elkind, Butler Lampson, Jim Mitchell	Location	Palo Alto
Subject	Comments on the Selection of Mesa for the OIS System Programming Language	Organization	CSL

XEROX

In a memo written December 17, 1974 Peter Heinrich and Wendell Shultz recommended that Mesa (MPL) be adopted as the System Programming Language for OIS, and they proposed a plan to transfer the system from PARC to OIS Development. In the following we comment on the recommendation and discuss alternative paths for implementing it.

1. We concur with the recommendation to adopt Mesa

We concur and endorse the fundamental recommendation that Mesa be adopted as the OIS implementation language. OIS needs a high level language for implementation. It appears to us that Mesa is more complete and more powerful than the other alternatives. Its use by both OIS and PARC would facilitate exchange and communication of software among these two communities. We are prepared to assist where we sensibly can in helping to transfer knowledge about the system to the OIS group so that they can use our implementation of the system or develop their own, whichever seems most appropriate.

2. The technical evaluation of Mesa is essentially correct

The technical evaluation of Mesa contained in the memo is essentially correct, but there are a few instances where the capabilities of Mesa were overstated and a few where they were understated. Jim Mitchell has already spoken to Wendell about these, and there should be some additional conversation between them on this subject. The principal overstatements were the suggestion that Mesa has a file editor and a source editor integrated into the Mesa system. We have been using the standard MAXC and Alto editors, but they are not written in Mesa. The principal understatements are that Mesa does not have an automatic instrumentation system (it does) and an implication that there will be an opportunity for signicant enhancement of the instruction set being developed for the Mesa Alto implementation. The current effort to design an instruction set for Alto will produce code that is reasonably close to the minimum size as determined by entropy measures. We believe that it will be hard to improve on it and that the principal task of the "object code generation phase" (page 6) should be to adapt this instruction set to the OIS processor.

3. Comments on the Action Plan

We are concerned about parts of the proposed action plan for implementing Mesa on the Sigma under CP-V.

First, the MAXC version of Mesa is not the appropriate take-off point for such an implementation. The Alto version is the one to start with. We are now concentrating on the Alto version, making considerable improvements in it, and are not intending to upgrade the Maxc version to maintain compatibility between it and the Alto one.

Second, it is obviously essential for OIS to have a Mesa system on which to do their development. There are three ways in which this can reasonably be done:

1. Obtain Altos and run the version of Mesa being developed at PARC.
2. Implement an interpeter on the Sigma that will be able to execute the Alto version of Mesa.
3. Implement a version of Mesa on the Sigma in the manner described in the memo.

The last of these alternatives is by far the hardest to accomplish, will take much longer to complete, and is the most risky. It would cause continuing compatibility problems between the PARC and OIS versions of the system, which would represent a considerable hazard especially during the period when the system is still in active evolution. We would argue strongly for either of the first two alternatives.

Third is the problem of compatibility. Obviously, PARC and the OIS Group must have complete and independent control over the versions of Mesa that each is using. However, there is much to be gained by keeping these two versions as compatible as possible for as long as we can. This is especially important in the near term when the systems will be actively evolving. We have had considerable success with two large systems, Interlisp and Tenex, in maintaining compatibility among a number of very independent organizations. It should certainly be possible to do as well with Mesa and the PARC and OIS organizations. The key to achieving compatibility is to define a standard interface, that is, a specification for a Mesa Virtual Machine that all implementations must satisfy. The virtual machine should be defined at as low a level as possible. If we adopt this approach, then we can be reasonably confident that Mesa programs will run correctly provided that they don't use facilities not in the virtual machine. Input/output almost certainly will have to be handled in a machine-specific manner at least to some extent, but this seems unavoidable. A Mesa Virtual Machine has not yet been specified and needs to be.

Finally, we concur in the suggestion that two or three OIS programmers spend some time at PARC to learn about the system. The right number seems to be two programmers. The time period should be about six months. We believe that the best and fastest way of learning is for them to spend this time as part of the Alto Mesa project team, working on documentation, converting parts of the system from MAXC to Alto, and criticizing and contributing to the Virtual Machine definition. Our resources at PARC are few, and it is important that we find a way of transferring the knowledge about the system without interfering greatly with the very crucial work of moving Mesa to Alto. The only way we can see to do this is by having these OIS team members participate in that transfer.

c: OIS Architecture Board

B Beeson	C Geshke
K Campbell	E Satterthwaite
S Klee	P Deutsch
W Klein	B Wegbreit
A Kopito	J Morris
A Lipton	C Simonyi
C Martin	W Teitelman
R Spinrad	J Moore
E Vance	D Bobrow
G Pake	C Thacker
H Hall	
W English	
R Taylor	