In this note we discuss the memory and programming mechanism necessary to implement the generalized calling of subroutines, where a procedure at one level may call another procedure, which may call others, and so on—perhaps involving a call on itself recursively. We shall assume from the following set of assumptions:

1) Memory is addressed fundamentally by word numbers and the notion of a memory name and a word address. See Note 3-9 for an explanation of organization.

2) A processor has a small finite number of general registers for performing arithmetic and logical operations on quantities, and a small finite number of index registers that contain segment counters for checking the addresses of memory units efficiently.

3) Procedures segments are in pure procedure form, that is, their execution does not result in their modification.

4) We desire principles that will permit calling to an arbitrary depth without alteration of conventions or technique.
5. Procedures are to be established as a procedure does not need to know at what level in the hierarchy it will be called or with what procedure it might itself call (since all procedures might be supplied as parameters).

An Example:

As an example of what can happen, suppose we are given the procedures $F_1(x, G_1, Z)$, and $H_1(Y)$, having the structure shown in Figure. For simplicity we suppose that these procedures are not used as functions. That is, the names of the procedures do not appear in the left hand of assignment statements within their bodies.

We restrict the parameters of a procedure to be objects representable as a single word -- e.g., a single quantity, or the word name of a word that is either single quantity, the entry point of a procedure or the word name of another word. A parameter used by one procedure in calling another may be one of the parameters by which it itself was called.

In our example, procedure $F_1(x)$ has three parameters, $G_1$, and $Z$.

It contains a direct reference to or references to private variables $p$ and $x$, and calls a three parameter procedure $G_1(x)$ with the following actual parameters: (1) the parameter $G_1$, (2) the entry point of a private variable, and (3) the parameter $Z$. Procedure $G_1(p, x, Z)$ uses the parameter $Z$ as the name of a procedure that is called with the following actual parameters: (1) the parameter $p$, and (2) the parameter $x$.

A third procedure $H_1(p, x, Z)$ calls $F_1(x)$ as a two parameter procedure with the actual parameters $p$, and $x$. In our illustration of mechanisms for handling communication among procedures we consider the consequences of a call on $F_1(x)$ with the following as actual parameters: (1) the name of the word $Z$, (2) the name of the procedure $F_1(x)$, and (3) an issue of the procedure $H_1(x)$.
In order to discuss the nature of a subroutine, we shall assume that all procedures are divided into a subroutine and a calling procedure segment. We further assume that the calling segment is used as a push down stack for holding the addresses of data objects that are specific to a particular call upon a procedure. The subroutine is required to be in yet another segment.

The Stack Pointer

The push down stack must be viewed as the portion of program memory that contains the nested procedure. A stack pointer must be associated with the procedure to indicate which quantities in the stack are pertinent to the procedure whose instructions are currently being executed. We choose to have the stack pointer indicate the first quantity in the stack which is pertinent to the current procedure. Each time a new subroutine is entered, the stack pointer is increased to point to the first location in the stack that is not used by the calling procedure. Each time a subroutine returns, the stack pointer is restored to the value which it had immediately prior to that subroutine's call.

Conveyance of Parameter Information

In the remainder of this note, we distinguish between a subroutine's actual parameter information, and a subroutine's actual parameters themselves. Generally a subroutine must execute some sort of addressing algorithm in order to obtain its actual parameters. This addressing
an algorithm generally requires the subroutine to fetch certain integers at
linkage information in order to locate its actual parameter. The first
of these linkage words is that a subroutine examines which contains information
unique to a particular parameter (and to contain the parameter information
for that parameter).

We consider three methods by which parameter information can be
conveyed between a calling pure procedure subroutine and a called pure
procedure subroutine.

1) Stacking - The calling subroutine places parameter information
in the stack segment following the word that the stack pointer
will point to upon entry to the called subroutine.

2) Followers - Parameter information is placed within the calling
procedure segment immediately after the procedure call instruction.

3) Operand registers - Parameter information is placed in the
operand registers of the processor by the calling subroutine.

It is possible to use one, two, or all of these methods during a
given calculation. Our object here is not to describe an optimum method
of parameter linkage, but to characterize some of the possible methods.
We will therefore restrict our attention to schemes which rely solely
on one of the three methods mentioned above.

The stacking technique is straightforward and well known from its
use in interpreters for algebraic languages. The techniques of followers
and operand registers are in general use in conventional machine language
coding. To our knowledge, the provisions necessary for the general

After A.A. Smith
Certain characteristic of the stacking and linking stack are not apparent without such consideration of the parameter linkage. When stacked, then all forms of information can be transmitted in large quantities and not by the operation of procedures directly. The information is embedded in true procedure coding, the parameter linkage cannot be so modified. In particular, the information in the ordinary cannot be present in the stack, and the information in the ordinary cannot be present in the stack. On the other hand, the stacking of parameter information requires the calling procedure to move information into the stack prior to the call. This leads to extra procedure steps that increase execution times and require memory. Furthermore, the possibility of reducing the memory space required to transmit parameter information, provided a suitable addressing mechanism can be formulated.

The method of passing parameter information through special registers is conceptually the simplest of the three methods. Such subroutines can be written to pick up information about the first parameter in a particular operand register, information about the second parameter in another particular operand register, etc. If a certain subroutine's particular linkage registers are being used to hold other quantities at the time when it is called, then the calling subroutine must save those other linkage registers or in the stack, the contents of those linkage registers before loading them with appropriate parameter information. After the calling subroutine
return, the calling subroutine usually passed the address of their local register. Since the number of operands is generally on the order of 10, these subroutine operations occur quite frequently in a procedure call or subroutine return. This fact makes the method of operand registers more efficient than the stacking method. Moreover, information about only one parameter can be passed through operand registers.

In the remainder of this note we will examine the problems involved in the implementation of the stacking and follower techniques in the context of segmentation.

Addressing of Actual Parameters

Having listed three methods of transmitting parameter information, we now turn our attention to the nature of the information which is transmitted. We again consider, by coincidence, three addressing alternatives:

1) Immediate — the transmitted information is the parameter itself.

2) Direct — the transmitted information points to the parameter.

3) Indirect — the transmitted information points to a choice of indirect words which leads to the parameter.

In general, each of these three addressing techniques may be used with either the stacking or the follower transmission methods. However, immediate addressing and the follower method are incompatible in a pure procedure context, because under these circumstances the actual parameters of each procedure call cannot be changed by the calling procedure in accordance with its own actual parameters.
Addressing Segments Within a Procedure

We consider now a sub-problem of the problem of addressing actual parameters, namely the problem of enabling a procedure to access segments whose names are unknown to it prior to a call by another procedure.

This may be done by either of the following two methods:

1) Name passing - Segment names are passed as parameters.

2) Tag passing - Attachment tags are passed as parameters.

The name passing technique is perfectly general. In a procedure having n attachment registers, the tag passing technique is subject to the following restrictions:

1) The actual parameters of each procedure may reside in no more than n different segments.

2) In a computation involving more than n segments, subroutines may have to save and restore the contents of some of the attachment registers before calling other subroutines.

Summary

Figure 2 summarizes the various combinations of techniques which have been discussed thus far. The rest of this note will be devoted to detailed examples of three implementations of the procedure design diagrammed in Figure 1. These three implementations are: (1) stacking with direct addressing and name passing, (2) stacking with indirect addressing and tag passing, and (3) followers with direct addressing and name passing.
<table>
<thead>
<tr>
<th>Name</th>
<th>Stacking</th>
<th>Followers</th>
<th>Operating Regulator</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Immediate</td>
<td></td>
<td>Impossible for pure procedure</td>
</tr>
<tr>
<td></td>
<td>Direct</td>
<td>First detailed example</td>
<td>Third detailed example</td>
</tr>
<tr>
<td></td>
<td>Indirect</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tag</td>
<td>Immediate</td>
<td></td>
<td>Impossible for pure procedure</td>
</tr>
<tr>
<td></td>
<td>Direct</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Indirect</td>
<td>Second detailed example</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 2:** Summary of methods discussed
Operand Register Conventions

For generality we assume that each procedure can work in seven independent and without regard to the calling of the same procedure except according to well defined conventions. Conventions regarding the use of operand registers and attachment registers therefore must apply uniformly over all procedures that might be nested. The following conventions are made here and are examples of what is permissible.

1) Attachment register 0 contains the name of the stack segment.

2) Index register 0 contains the word address which is the stack pointer.

A further convention is made (although it is not essential as will be seen) regarding the use of processor registers for transmitting the return address (the word name of the procedure entry instruction) to the called procedure.

3) Attachment register 1 conveys the segment name of the calling procedure.

4) Index register 1 conveys the word address of the calling instruction in the calling procedure.

In the third detailed example, the following additional convention will be introduced.

5) Index register 1 conveys the word address which is the base stack address (defined below) of the calling procedure.

As with conventions 3) and 4), conventions 5) is not essential, but is introduced for convenience only.
We use upper case letters to represent segment names, i.e.
E, P, M - procedures E11, E12, and E13
S - push down stack
M - main procedure
X - segment containing the public variable X
Lower case letters such as a, b, y, z represent the word address
of objects. The notation
a/b
represents the word name consisting of the segment name a and the word
address b. The attachment tags are written as a0, a1, etc., while e.g.
x1, x2, ... represent the content of index registers. An address word
a/b
consists of an attachment tag a and a word address b. The notation
[a/b]
represents the word addressed by the address word. For convenience we
will sometimes write <a> to mean the attachment tag of an attachment
register containing the segment name a. We presume the existence of an
instruction for causes of the form
attach S, a0
that causes the segment name S to be placed in attachment register a.
If S is valid in the sphere of protection of the current process, the names
of private variables of a procedure (such as x and y in our example)
the letters represent the words themselves.

Certain procedure registers must be identified, namely those that
contain the address word of the current procedure step (current routine).

(17)
Formulated Scenario

Objectives of the Stack Frame

The stack frame is the interface between the calling procedure and the called procedure. It consists of:

1. The stack pointer and the base stack address for the procedure called.
2. The saved return address of the calling procedure.
3. The saved state of the calling procedure.

The stack frame is used to initialize the state of the called procedure. It provides a mechanism for passing parameters between procedures. The stack frame is also used to store data and local variables during the execution of the called procedure.

Execution Steps

The execution begins by initializing the stack frame.

1. The stack pointer is set to the base stack address.
2. The return address is saved.
3. The parameters of the called procedure are pushed onto the stack.
4. The state of the calling procedure is saved.

The called procedure then begins execution.

After the procedure completes, the following steps are taken:

1. The saved state of the calling procedure is restored.
2. The return address is popped from the stack.
3. The stack pointer is advanced to the next stack frame.
4. The stack frame is deallocated.

The execution of the calling procedure then continues from the saved return address.
Let one stack pointer be 
\[ s \]

Let one stack pointer be 
\[ \text{stack} \]

\[ (a) \rightarrow \text{stack} \]

\[ (\text{stack}) \rightarrow a \]

\[ a \rightarrow (\text{stack}) \]

\[ \text{stack} \rightarrow a \]

\[ a \rightarrow \text{stack} \]

\[ \text{stack} \rightarrow a \]

The reverse order of the calling sequence is
\[ (a) \rightarrow \text{stack} \]

\[ \text{stack} \rightarrow a \]

\[ a \rightarrow \text{stack} \]

The reverse order of the calling sequence is
\[ (a) \rightarrow \text{stack} \]

\[ \text{stack} \rightarrow a \]

\[ a \rightarrow \text{stack} \]

\[ \text{stack} \rightarrow a \]

The reverse order of the calling sequence is
\[ (a) \rightarrow \text{stack} \]

\[ \text{stack} \rightarrow a \]

\[ a \rightarrow \text{stack} \]

\[ \text{stack} \rightarrow a \]
\begin{align*}
\alpha(\alpha + 5) & \rightarrow (\alpha + 5) + 5 \tag{41} \\
\alpha(\alpha + 5) & \rightarrow (\alpha + 5) + 3 \tag{42} \\
\alpha \rightarrow (\alpha + 0) + 1 \tag{43} \\
\alpha \rightarrow (\alpha + 0) + 1 \tag{44} \\
\alpha \rightarrow (\alpha + 0) + 1 \tag{45}
\end{align*}

\text{attach } G, a1

\text{statement a1/s, 11}

\text{Figure 3 - continued}
Figure 3 - (continued)
\textbf{Figure 1 - (continued)}
We suppose that attachment 5 to a 80/800, as in Section 2, is obtained by use of the meta-instruction

\textit{obtain} 5 \to a

that loads an available attachment register with the operand whose tag has the corresponding attachment tag as its value. The attachment 5 then

\textit{release} 5

releases the attachment register associated with the operand address 5. We introduce one additional notation convention: assigning an address word by an asterisk

a0/(x0 + 4)^*

means that the addressed word is itself an address word - starred as unstarred - that is to be used to fetch an operand (indirect addressing).

\textbf{Organization of the Stack Segment}

With tag passing the stack is somewhat simplified. Instead of storing the word name of the entry instruction in registers 1 and 2 above the base stack address, we simply store the address word of the entry point in registers 1 above the base. The \textit{enter} and \textit{return} macros now become

\textbf{N/m},

\begin{align*}
& 10 \ x0 \to \ [a0/x0 + 4]^* \\
& 11 \ x0 \to \ x0 \\
& 111 \ \text{PAT/PHA} \to [x0/x0 + 11] \\
& 119 \ x0/x0 \to \ \text{PAT/PHA}
\end{align*}
Coding

The coding of the variable to assess cognitive impairment was done based on the condition of the patient just after the admission and up to the time.
Main procedure

X/1

X/2

X/3

X/4

release X

release Y

release F

quit

Figure 5 - Second detailed example
Figure 6 - Condition of the machine in the original layout.
Optimization of the stack segment

The stack segment is organized in this example as we were shown in the first example.

Subroutine Entry

In this example we will introduce for convenience, the subroutine convention that upon subroutine entry it contains the base word address of the calling subroutine. Also, in this example, it is convenient to use a return instruction which specifies a word address to be pashed on the return word address so that control can be returned to the function or block the following.

To accomplish these two objectives we let the return subroutine be identical to that used in the first example except for the inclusion of an additional step after step 13:

\[ x0 \rightarrow x1 \]

We postulate a new return instruction:

```
RETURN
```

which performs the following steps:

1) unless the return from the main:

\[ x0/x10 = [s][a][d] \]

\[ x0/x10 + x1 \rightarrow [s][x] \]

2) back up the stack pointer:

\[ x0/x00 + x1 \rightarrow [a] \]
Addressing of Parameters

In this example, thelookups specify where the parameters are located in the stack region or the called subroutine. Actual parameters relative to the base stack pointer in the calling context.

Comparison with the First Detailed Example

The first and third examples have shown parameter values being moved from one location to another in each call. However, in the third example, the calling function does not execute in the called subroutine, and hence exists only once in except for each subroutine.

In the first example, the calling to control movement may order a subroutine for as many times as the subroutine is called. Keeping it in another way, if a subroutine calls several other subroutines, it needs more data from its caller's region at the same time. It then delivers this information to other routines or controls, starting point of the followers of the calls. If these data followers are not necessary, not as such as the calling necessary to support the data movements that would be necessary on each call in the first example. Hence, we see that the method of the third example consumes less storage and the system...
of the flow elements by identifying the key parameter sets and

other procedures.

Figure 1: Example table showing the number of parameter sets considered for each

parameter under various criteria. The table illustrates a scenario where we see that the number of required parameters in the call is

two.

<table>
<thead>
<tr>
<th>Site 1</th>
<th>Site 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter 1</td>
<td>1</td>
</tr>
<tr>
<td>Parameter 2</td>
<td>2</td>
</tr>
</tbody>
</table>

**Figure 7** - Numbers of parameter cycles for parameter transmission to

example 1 and 2.
The coding of this example is given in Figure 4. Figure 2 shows the condition of the system just after the first step and before the second flow. Notice the similarity with the code in the examples 1 and 2.

**Main Procedure**

\[ M/S \]

\[ \begin{align*}
  C & = x_0 \\
  x & = \left[x_0 / x_0\right] \\
  x & = \left[x_0 / x_0 + 1\right] \\
  v & = \left[x_0 / x_0 + 2\right] \\
  z & = \left[la_0 / la_0 + 3\right] \\
  & \text{attach} \ V, \ a_z \\
  & \text{enter } x_2 / x_0 \end{align*} \]

**Figure 4 - Third detailed example**
Figure 2 (continued)
H/u, enter \[ a2 \frac{a0}{(x0 + 6)} \right] , 7

3
4
5
6
---
---
return 5

Figure 8 - (continued)
Figure 9 - Condition of the stack in the third detailed example.