The Tool Locator
External Reference Specification
Steven E. Glass
February 19, 1986

Revised 11/21/85 (by Steven Glass)
Combines previous documents.
Changes calling mechanism and parameter passing.
Calls must be made from full native mode!

Revised 2/19/86 (by Steven Glass)
Adds new required calls.

Preface

This document replaces two previous documents: "A Framework for Implementing ROM- and RAM-Based Tools in Cortland" and "Writing and Calling Cortland Tools." I have combined the information in the two, into this single ERS. This ERS has a little more background information in it than most ERS but I provide the information so that the reader may understand why certain design decisions have been made.

Introduction

In order to make Cortland as attractive as possible, we will include a number of software tools in ROM. This approach makes the tools available to all programs without using disk space and without the need to link tool libraries to applications.

Because there is never enough ROM to go around and because there will be bugs in the ROM, some tools will end up in RAM from day one, and others will have to migrate there between ROM revisions in the form of RAM based ROM patches. This suggests we develop a "ROM/RAM tools standard" rather than a "ROM tools standard."

This document gives a framework for implementing ROM/RAM based tools on Cortland. It defines an elementary unit of software packaging called a **tool set** and an elementary unit of service called a **function**. Roughly, a tool set, which consists of related functions, is a complete software tool or a major subset of a software tool. Each function is an "entry point" of the tool set that performs a fundamental operation. Additionaly, this ERS describes the tool set whose job it is to allow tools and applications to communicate amongst themselves; this tool set is called the **Tool Locator**.

DEFINITIONS

A **function** is a fundamental operation that converts zero or more inputs to zero or more outputs and side effects. For example, **allocate memory** and **multiply single precision floating point** are functions.

A **tool set** is a group of logically related functions. For example, a **memory manager** is a tool set that implements such related functions as **allocate memory**, **free memory**, etc. A tool set is to be implemented as a single code module.

The **Tool Locator** is the particular tool set which allows tools and applications to communicate.

A **tool configuration** is a collection of tool sets, some of which may be in ROM and some of which may be in RAM.

A **level of support** is a tool configuration that is appropriate to support a given class of application.

An **operating system** is a program that manages use of some or all of the resources of a computer system. The set of Apple // operating systems includes **DOS**, **ProDOS**, and **UCSD Pascal**.

An **operating environment** is a specific configuration of hardware, operational modes (e.g., "native mode," "emulation mode"), operating system, and tool configuration defined by Apple. For example, one operating environment might be "ProDOS running on a 256Kby Cortland with one 3.5" floppy drive and the standard set of ROM based tools." In most cases, the level of detail needed to describe an operating environment will be much greater than this.

An **application** is a program that provides a set of services directly relevant to a user's task. Examples include spreadsheets, compilers, assemblers, database managers, text editors, word processors, file management utilities, etc.

An **environment switcher** is a program that can establish any of the defined operating environments on a computer system.

A **desktop manager** is a program that identifies available applications, provides a way for users to choose an application to run, establishes the operating environment by calling the environment switcher, and makes various desk accessories (such as calculator, clock, calendar, notepad) available to users.

Emulation mode is the operational mode of the 65816 when the E control bit is set to 1. In this mode, the 65816 behaves much like a 6502.

Full Native mode is the operational mode of the 65816 when the E, M, and X control bits are set to 0.

Mixed Native mode is the operational mode of the 65816 when the E control bit is set to zero but either M or X or both are set to one.

Assembly Language Conventions

BYTE expr assembles a byte containing the given expression value.
WORD expr assembles a 2-byte word containing the given value.
LONG expr assembles a 4-byte location containing the given value.
BLOCK expr reserves a block of storage consisting of **expr** bytes.

SUGGESTED CHARACTERISTICS OF TOOL SETS

This section lists characteristics intended to maximize the efficiency, usefulness, flexibility, and implementability of ROM and RAM based tool sets without unnecessary constraints. It is not suggested that we rewrite any existing code along these lines, although, in some cases, it may be desirable to do so.

[1] Full Native mode. Whenever possible (which will be almost always) new tool sets should use native mode. In general, this increases speed and decreases code size compared to emulation mode. Since there is a long-term desire to migrate all code to native mode, writing tool sets this way will minimize mode switching over the long term.

[2] "ROMability." All tool sets should be written assuming they will be placed in ROM. This might not happen in early versions of Cortland, but the machine has 1Mby of ROM address space, so things could change.

[3] Position independent code. All tool sets should be written in position independent code—code that executes properly *without relocation* no matter where it is placed in memory (assuming it does not straddle a bank boundary). This simplifies the use of loadable tool sets with acceptable impact on performance.

[4] Standard interface. From the caller's viewpoint, there should be one standard protocol for calls to functions in tool sets. In particular, the caller should not need to know if the called function is in ROM or RAM or if it is a RAM based ROM patch of an entire tool set or only a single function.

[5] New tool sets should be accessible to programs in a straightforward manner. While it would be nice to allow both native mode and emulation mode programs to use new tools, the tools need to be efficient when in native mode even at the expense of emulation mode programs.

[6] Dynamically assigned workspace. New tool sets should not use any fixed RAM locations for work space. All work space must be obtained from the Memory Manager. This avoids memory conflicts such as those caused by fixed usage of "screen holes." A limited set of exceptions to this rule will be discussed later.

[7] Simple interrupt environment. All new functions must either be reentrant or must disable interrupts during execution. Because each approach has significant costs, the designer must consider this decision very carefully. Most functions, especially those that execute in less than 500µs, will probably choose to disable interrupts. More time consuming functions should probably also choose to disable interrupts, especially if they are executed rarely.

[8] Few fixed ROM addresses. In order to minimize the impact of ROM updates and RAM based ROM patches, the interface to ROM based functions must avoid having lots of fixed ROM entry points. It must be possible for the system to construct a RAM-resident table of all function entry points at system initialization time.

[9] Functions must restore the caller's execution environment before returning control to the caller.

[10] O. S. independence. Functions may not assume the presence of any operating system.

OUTLINE OF AN IMPLEMENTATION

This section describes the essential features of a tool locator system that has most of the above characteristics.

Addressing Tool Sets and Functions

Each tool set is assigned a *permanent* tool number. Assignment starts at one and continues with each successive integer. Each function within a tool set is assigned a *permanent* function number. For the functions within each tool set, assignment starts at one and continues with each successive integer. Thus, each function has a unique, permanent identifier of the form (TSNum,FuncNum).

Both the TSNum and FuncNum are 8 bit numbers.

So far, the following are assigned.

Tool Set Number	Descriptions
1	Tool Locator
2	Memory Manager
3	Misc. Tools
4	QuickDraw II
5	Desk Manager
6	Event Manger
7	Scheduler
8	Sound Manager
9	FDB Tools
10	SANE

For each Tool Set, the following calls must be present:

FuncNum	Descriptions
1	boot initialization function for each tool set
2	application startup function for each tool set
3	application shutdown function for each tool set
4	version information
5	Reset
6	Reserved
7	Reserved
8	Reserved

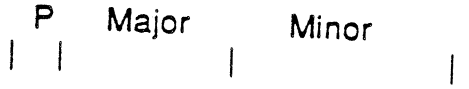
The boot initialization function is executed at boot time either by the ROM startup code or when the tool is installed in the system.

The application startup function is called by the application before using the tool. A tool uses this call to obtain initialization information from the application that is going to use it. For example, an tool may require space in bank zero for zero page and it gets it from the application on this call. An initialization function needs to check if the tool is already active. If it is active it should return an "already initialized" error and do nothing. Multiple initializations are No-ops.

The application shutdown function is called by the application before it terminates. A tool should release any memory it has obtained in the name of the application at this time.

All tool's will return version information in the same form: a word. The high byte of the word will indicate the major release number (starting with 1). The low byte of the word will indicate the minor release number (starting with 0). The most

significant bit of the of the word indicates whether the code is an official release or a prototype (set implies prototype). There is no distinction between alpha, beta or other prototype releases other than whether or not it is a prototype.



The reset function is called whenever reset occurs. If a tool is active, it needs to do whatever makes sense for reset. If a tool cannot continue after reset, it should return a "cannot reset" error. If any tool returns an error on reset, the tool locator will force a system death after the last tool has been reset.

Structure of Tool Sets

The Tool Locator System proposed here requires no fixed ROM locations and a few fixed RAM locations.

How do we perform this magic? All functions are accessed through the tool locator via their tool set number and function number. The Tool Locator uses the tool set number to find an entry in the Tool Pointer Table (TPT). This table contains pointers to Function Pointer Tables (FPT). Each tool set has an FPT containing pointers to the individual functions in the tool set. The Tool Locator uses the function number to find the address of the function being called.

Each tool in ROM has an FPT in ROM. There is also a TPT in ROM pointing to all the FPT's in ROM. One fixed RAM location is used to point to this TPT in ROM. This location is initialized at power up and warm boot by the firmware. In this way the address of the TPT in ROM does not ever have to be fixed. We can change it every time we revise the firmware as long as the firmware initializes the RAM based TPT Pointer.

The TPT has the following form:

Count (4 bytes)
Pointer to TS 1 (4 bytes)
Pointer to TS 2 (4 bytes)
...

An FPT has the following form:

Count (4 bytes)
(Pointer to F1) - 1 (4 bytes)
(Pointer to F2) - 1 (4 bytes)
...

In both tables, the count is the number of entries plus 1.

Tools are to obtain any memory they need dynamically (using as little fixed memory as possible). To use memory obtained through a memory manager, a tool needs some way to find out where its data structures are. The tool locator system maintains a table of work area pointers for the individual tools. The Work Area Pointer Table (WAPT) is a table of pointers to the work areas of individual tools. Each tool will have an entry in the WAPT for its own use. Entries are assigned by tool number (tool four has entry four and so on). A pointer to the WAPT must be kept in RAM at a fixed memory location so that space for the table can be allocated dynamically. At firmware initialization time, the pointer to the WAPT is set to zero.

Summary of Terms

Tool Pointer Table (TPT)
This is the table of pointers to indivdual Function Pointer Tables.

Function Pointer Table (FPT)
This is the table of pointers to the functions in an individual tool set.

Work Area Pointer Table (WAPT)
This is the place a tool keeps a pointer to its work area.

User Tools and System Tools

The Tool Locator System proposed here is so flexible that individual application writers may want to write their own tool sets to use in their applications. The problem comes up when we have to assign tool set numbers. Rather than trying to reserve tool set numbers for tools we have not yet written (to reserve space in the TPT), the Tool Locator System will support both system tools and user tools.

Permanently Reserved RAM

The tool locator system permanently reserves some space in bank $E1). It is used as follows:

(4 by) Pointer to the active TPT. This will point to the ROM based TPT if there are no RAM based tool sets and no RAM based ROM patches. Otherwise, it will point to a RAM based TPT.

(4 by) Pointer to the active user's TPT. This will be zero initially, indicating that no user tools are present.

(4 by) Pointer to the Work Area Pointer Table (WAPT). The WAPT parallels the TPT. Each WAPT entry is a pointer to a work area assigned to the corresponding tool set. At startup time, each WAPT entry is set to zero, indicating no assigned work area.

(4 by) Pointer to the user's Work Area Pointer Table (WAPT).

(16 by) Entry points to the dispatcher

This is the only RAM permanently reserved by the tool locator system.

Tool Locator System Initialization

Each tool set must be initialized before use by application programs. Two types of initialization are needed: boot initialization and application initialization. Boot initialization occurs at system startup time (boot time); regardless of the applications to be executed, the system calls the boot initialization function of every tool set. Thus, each tool set must have a boot initialization routine (FuncNum = 1), even if it does nothing. This function has no input or output parameters.

Application initialization occurs during application execution. The application calls the application startup function (FuncNum=2) of each tool set that it will use. The application startup function performs the chores needed to startup the tool set so the application can use it. This function may have inputs and outputs. Each tool set will define what they are. A common input will be the address of space in bank zero that the tool can use.

The application shutdown function (FuncNum=3) should be executed as soon as the application no longer needs to use the tool set because it releases the resources used by the tool set. As a precaution against applications that forget to execute the shutdown function, the startup function should either execute the shutdown function itself or do something else to assure a reasonable startup

state. This function may have inputs and outputs as well. Again they are defined by the individual tool sets.

The provision of two initialization times reflects the needs of currently envisioned tool sets. For example, the Memory Manager will require boot time initialization because it must operate properly even before any application has been loaded. On the other hand, SANE only needs to be initialized if the system executes some application or desk accessory that uses it. Initializing only the tool sets that will be used saves resources, particularly RAM.

System startup code must copy the ROM based pointer to the TPT to the fixed RAM location and then call the the tool locator boot initialization routine (TSNum=1, FuncNum=1).

The firmware initialization routine will

1. Initialize the four RAM pointers described above.

2. Call the Tool Locator boot initialization function (TSNum=1, FuncNum=1), which will do the following:

    a. Call the memory manager boot initialization function (TSNum=2, FuncNum=1), which will initialize its private workspace as well as any other workspace it needs to indicate the initial reservation status of all of memory.

    b. Determine the number of tool sets.

    c. Call the memory manager to allocate space for the WAPT, and initialize all WAPT entries to zero.

    d. Successively call the boot initialization function of every tool set starting with TSNum=3.

There are several points to keep in mind:

The Initialization Function does not load RAM based tool sets.

The memory manager boot initialization function needs reserved private workspace because it has no other way to find workspace. This function must find and catalog all available RAM.

Boot Initialization functions and Application Initialization functions operate in the standard execution environment for functions. This is described later.

Each tool set designer must determine how to split initialization tasks between the module's boot initialization function and its application initialization function.

Disk and RAM Structure of Tool Sets

This section discusses additional details of dynamically loaded, RAM based tool sets and of RAM based ROM patches. The exact form of tool set on disk is undecided at this time. Our goal is that Tool Sets will be kept in simple load modules which can be dynamically loaded into memory whenever they are needed. Still unresolved is

1. Naming conventions. Will the tools be in single file with a specific name (e.g. system.tools) or will we use a file type and/or suffix (e.g. graphics.tools, math.tools, window.tools, desk.tools)

2. Loading conventions. How will an application cause tools to be loaded? What if they are already in memory (used by the finder which launched the application)? What if they were on the finder disk and the application disk?

3. Forcing a particular version. Can an application force a particular version of a tool to be used?

Handling RAM Based Tool Sets

The routine which causes a tool to be loaded will be responsible for calling the boot initialization function of each RAM based tool set after it loads it and installs it in the TPT. This raises the possibility of double execution of a tool set's boot initialization function—ROM based boot initialization function at boot time which is called again when a RAM based patch is loaded and initialized. Thus, each RAM based boot initialization function must be able to 1) undo the effect of its corresponding ROM based boot initialization function (if any) and 2) perform its own initialization processing.

The main uses of RAM based tool sets are to accommodate tools that could not fit in ROM and to allow patches of erroneous ROM code. The first usage is simple because an entire tool set is loaded into RAM from disk. The second case is more complex because we may want to patch only a few of the functions in a given tool set.

We can add RAM based tool sets or patches by building a new TPT. The new TPT contains all the entries in the old TPT except for the newly added RAM based

tools and/or patches. The Tool Locator provides a single call to handle this in a sensible way.

The biggest problem with patches of this kind is significant restrictions on how things are implemented. For example, a patch has no simple way to access local subroutines and constant data in the main body of the original tool set without prior agreement on some convention that makes access to these items independent of their ROM addresses. For example, the boot initialization function of a tool set could put a pointer to a subroutine or data address table into its work area, thus making it accessible to the RAM based code. These conventions need to be worked out in detail by each tool set designer.

INTERFACE BETWEEN APPLICATIONS AND FUNCTION CALL DISPATCHER

The Goals. In developing a call mechanism we are trying to find one that is fast, compact and easialy callable from a high level language. We do not want calls to tools to take so long that no program could afford the time to make them; we do not want calls to take up so much space that a program could not afford the space to make them; we do not want high level languages to use so much glue to make tool calls that high level language use will be discouraged.

Past History. We have two precedents to look at in the Apple II (6502) world. ProDOS and all the Apple II tool kits are called with the following ensemble:

```
jsr EntryPoint
byte CallNumber
word ParameterTable
```

On the Apple ///, SOS was called with the following slightly different ensemble:

```
brk
byte CallNumber
word ParameterTable
```

Both schemes lead to a relatively small amount of code in line and relatively quick execution. But they do not allow for easy high level language interface. (See the appendix on the path not taken for how we could have used this kind of scheme on Cortland.)

A high level language would like to call tools just like it calls any other subroutines. A compiler wants to generate the following code for a procedure X.

```
procedure   X      ( p1, p2,..., pn)
```

```
        push p1
        push p2
        ...
        push pn
        jsl X
```

A compiler also wants to generate similar code for a function Y.

```
function    Y    ( p1, p2,..., pn) : value

        push ValueSpace
        push p1
        push p2
        ...
        push pn
        jsl Y
```

ISSUING THE CALL THROUGH THE DISPATCHER

Our dispatching scheme looks very similar to the code generated by the compiler.

```
        push inputs
        ldx #TSNum+FuncNum*256
        jsl Dispatch
        bcs HandleError                 (optional, usually not required)
```

The inputs look the same on the stack but the call number and error information are passed in registers. A high level language will have to use a small glue routine to handle this. The calling code and glue will look like

```
        Push Input1
        Push Input2
        ...
        Push Inputn
        jsl Glue
        ...

    Glue
        ldx #TSNum+FuncNum*256
        jsl Dispatch2
        bcs HandleError                 (optional, usually not required)
        rtl
```

The glue calls a different entry point because there is an extra three bytes of return address information on the stack. The two different entry points make the

stack look the same to the fucntion being called. (It would not do to have the inputs be at different depths on the stack depending on how a function is called).

What About Speed? The scheme presented so far fulfills most of the requirements outlined earlier. Unfortunately, there is overhead associated with making a function call. Current estimates suggest that call dispatch will take about 118 micro seconds. For most calls, this is fast enough, but not for all calls. Individual tool sets may set up conventions for calling some of their functions directly.

Return from the Call

Upon completion of the call, the function call returns control directly back to the calling routine. Some tool sets will support returning errors on some functions. If they do, the convention is as follows:

C Flag indicates error
A-register contains error code

The state of all flags and registers is summarized as follows:

N flag	As set by function
V	As set by function
m	Unchanged (must be 0)
x	Unchanged (must be 0)
D	Set to 0
I	Unchanged
Z	As set by function
C	As set by fucntion or error flag
E	Unchanged (must be 0)
A register	As set by fucntion or A=0 successful call, A≠0 error code
X	As Set by function
Y	As Set by function
S	Parameters have been removed from stack
D	Unchanged
P	See list of flags above
DB	Unchanged
PB	Unchanged
PC	Address following call

Note that "unchanged" means "same as value just before function call."

Error Codes

Tool sets should return error codes in the a register that have the tool set number in the high byte and the "message" in the low byte. The dispatcher will return two errors and have a high byte of zero.

The following error code values are reserved for exclusive use by the function dispatcher:

Error code=	$0001	Value of <TSNum> does not make sense
	$0002	Value of <FuncNum> does not make sense

Every tool set may have to return an already initialized error and a cannot reset error. These error codes are defined as follows:

Error Code	Meaning
XX01	Already initialized
XX02	Cannot reset

where XX is the tool set number.

Remaining error codes are defined by individual tool set designers.

Parameter Passing Details

Generally, there are several ways to pass parameters:

1) in the stack.
2) in a parameter block.
3) in the A, X, and Y registers.

Method 1 is the most common method used by high level languages. Method 2 is also very flexible, since the parameter block may be anywhere in memory and may contain additional pointers to anywhere in memory. Method 3 is also useful for small or few parameters but since the tool dispatcher does not preserve the registers going into a function, it is only useful for one way communication.

The parameters and parameter passing method are defined by each function.

Passing Control to the Function

The application will set up its parameters and make the function call. Before handing control to the function, the tool dispatcher checks the machine state. If the call was not made from full native mode, an error is returned (or system death).

If the call is made in the right mode, the D bit is cleared. (The I bit remains unchanged and the remaining status and control bits are undefined.) Next, the tool dispatcher manipulates the stack so that any inputs are at the right depth no matter how a call is made. Finally, the tool dispatcher puts the low word of the work area pointer in the A register and the high word of the work area pointer in the Y register. With the registers set up this way, it will call the function using a **JSL** instruction.

The following table summarizes stack contents on function entry:

Offset from S	Contents
?	Function Value area if any
7-?	Parameters
4-6	Return address in Glue
1-3	Return address in calling code
0	Top of Stack

Register contents will be as follows on function entry:

A register	Low word of pointer to work area
Y	High word of pointer to work area
X	Undefined
S	Current top of stack (i.e. one byte below lowest used location)
D	As left by application program
P	See previous paragraph
B	Data bank is as left by application
K	Program bank is bank of function.
PC	Address of function.

Return from the Function

The function itself defines its handling of all parameters. The most common case will be with stack parameters handled by pulling off any input parameters and leaving any function return on the stack for the calling program to handle.

WRITING A FUNCTION

Tools need memory for their data. They will be more efficient if they store their data on zero page, but how will they know what part of bank zero they can use for zero page? The only sensible answer is to have the application tell the tool what it can use (this would be done with the application initialization function).

A function may not use any memory that has not been assigned to it by the application or reserved via the memory manager unless it saves and restores the memory. If it uses memory in this way it must turn interrupts off while unreserved memory is changed.

Is Bank Zero Required?

Does an application have to provide bank zero space for each tool it will use? If it did not, the tool would have to save and restore bank zero on each call. We can design this ability into our tools but should we? I think not; if the application cannot afford a little bank zero space, it should save and restore the area itself.

Re-Entrant Tools

A tool needs either to be re-entrant or to turn off interrupts while it runs. The first option requires careful coding and the second may be undesirable.

For a tool to be re-entrant, it must keep state information in such a way that one can interrupt a call, make a new call, and return from the interrupt in such a way that the original call is not disturbed. If we assume that a tool does not have any self-modifying code, its entire state is characterized by its data.

A Tool's State

A tool has two kinds of data: permanent data and temporary data. The permanent data are variables which need to be maintained from call to call; temparary data are variables that are initialized and used only when a function in a tool is running.

We can make rules which guarantee preservation of state information. A possible rule is that all global variables must be kept in a known place (which can be saved and restored by an interrupt) and that temporary variables must be kept on the stack. Unfortunately, our processor does not lend itself well to using the stack for variables.

The Rule

A more workable rule is to keep all variables in a known place, and make the interrupting code responsible for preserving them.

What is a known place? Tools do not use fixed memory locations for their data. They get bank zero memory from the application, and get additional memory from the memory manager. The location of one of these is kept in the work area pointer table (WAPT) and the address of any other (if there is any other) is kept in the work area. Thus the entire state of a tool is characterized by the pointer in the WAPT. If we save this pointer, restart a tool, do something and restore the pointer we preserve the tool's state as if nothing was done.

Two Tool Locator calls will take care of this for us: GetWAP and SetWAP (for get and set work area pointer).

If a tool requires that the application give it some zero bank for its global area, data access can be very fast. Two examples follow.

Example 1: A Function to Set Global Value on Zero Page. In Pascal SetAValue would look like:

procedure SetAValue (value : integer);

Code to call the function

```
lda TheValue
pha
ldx #FuncNum*256+TSNum
jsl Dispatch
```

The Function itself

```
MyFunction
OrigDirect equ 1
RTL1       equ OrigDirect+2
RTL2       equ RTL1+3
TheValue   equ RTL2+3

           phd                  ; save current direct register
           tcd                  ; make my zero page active

           jsr StartCheck       ; find out if I was initialized
           beq GoOn             ; all is well
           brl ErrorOut2        ; an error occured
GoOn       lda TheValue,s       ; get the new value
           sta Global           ; put it on zero page
           jmp EndCall2         ; use quit routine
           end
```

```
; Returns with no error in a register if a or y are non-zero

StartCheck
           cmp #0
           bne OK
           cpy #0
           bne OK
           lda #NotInitialized
           rts
OK         lda #0
           rts
           end
```

```
; EndCall2 and ErrorOut2 move the return addresses up two
; bytes on the stack.  Similar closing routines would be necessary for
; each way a tool is called.

EndCall2
           lda #0               ; set result code
ErrorOut2
           tax                  ; save result in x
           pld                  ; restore original direct register
```

```
lda 5,s                 ; move rtl adr's up
sta 7,s
lda 3,s
sta 5,s
lda 1,s
sta 3,s
pla                     ; Remove extra word
txa                     ; get back result
cmp #1                  ; set carry right
rtl                     ; all done
```

Example 2: A Function with more than one inputs. In Pascal the SetRect procedure has five inputs. A pointer to a rectangle record and four integers that are to be put into the rectangle record.

```
procedure  SetRect   ( var  TheRect : rect;
                            Top : integer;
                            Left : integer;
                            Bottom : integer;
                            Right : integer)
```

Code to call the function

```
pea ^TheRect            ; push high word of rect ptr on stack
pea TheRect             ; push low word of rect ptr on stack
lda Top                 ; push the value of top
pha
lda Left                ; push the value of left
pha
lda Bottom              ; push the value of bottom
pha
lda Right               ; push the value of right
pha
ldx #FuncNum*256+TSNum
jsl Dispatch
```

The Function itself

```
MyFunction

origDirect equ 1
Ret1       equ OrigDirect+2
Ret2       equ Ret1+3
Right      equ Ret2+3
Bottom     equ Right+2
Left       equ Bottom+2
Top        equ Left+2
TheRect    equ Top+2
```

```
        phd                     ; save current direct register
        tsc
        tcd                     ; get stack pointer

        lda Top
        sta [TheRect]
        lda Left
        ldy #2
        sta [TheRect],y
        lda Bottom
        ldy #4
        sta [TheRect],y
        lda Right
        ldy #6
        sta [TheRect],y

        jmp EndCall12
```

; EndCall12 and ErrorOut12 move the return addresses up twelve
; bytes on the stack.

```
EndCall12
        lda #0
ErrorOut12                      ; set result code
        tax
        pld                     ; save result in x
                                ; restore original direct register

        lda 5,s
        sta 17,s                ; move rtl adr's up
        lda 3,s
        sta 15,s
        lda 1,s
        sta 13,s

        tsc
        clc                     ; get stack pointer
        adc #12                 ; cut back stack by 12
        tcs
        txa
        cmp #1                  ; get back result
        rtl                     ; set carry right
                                ; all done
```

Tool Locator Calls

TLBootInit

Call 1

Does boot initialization for the Tool Locator including initializing all other ROM based Tool Sets. No application should ever call this routine.

TLStartup Call 2

Called by every application before any other tool calls.

TLShutDown Call 3

Called by every application just before quitting.

TLVersion Call 4

output version word

Returns version information about the Tool Locator.

TLReset Call 5

This call is made whenever reset occurs. It calls the reset function of every tool set in the system.

Reserved Call 6

Reserved Call 7

Reserved Call 8

GetTSPtr Call 9

input	UserOrSystem
input	TSNum
output	Pointer

Returns pointer to the Function Pointer Table of the specified tool set.

SetTSPtr Call 10

input	UserOrSystem
input	TSNum
input	Pointer

Installs the pointer to a Function Pointer Table in the appropriate TPT. (0 will be the system while $8000 will be for the user.) If the TPT is not yet in RAM, it copies the TPT to RAM. (Memory for the TPT is obtained from the memory manager.) If there is not enough room in the TPT for the new entry, the TPT is moved to a bigger chunk of memory. Likewise, the WAP Table is expanded. (Memory for these expansions is obtained from the memory manager.) If the new pointer table has any zero entries, old entries are moved from the old pointer table to the new pointer table. (This is the call that will allow us to patch a subset of a Tool Set without replacing the whole thing.)

GetFuncPtr Call 11

input	UserOrSystem
input	TSNum
input	FuncNum
output	Pointer

Returns pointer to the specified function in the specified Tool Set.

There is no SetFuncPtr in this specification. Does anyone think we need it? The SetTSPtr call should do all we need for patching an individual routine.

GetWAP Call 12

input	UserOrSystem
input	TSNum

	output	Pointer

Gets the pointer to the work area for the specified module.

SetWAP		Call 13
	input	UserOrSystem
	input	TSNum
	input	Pointer

Sets the pointer to the work area for the specified module.

Summary of changes since previous version.

The reset call and three reserved calls are now required of every tool set. The first non required function number is 9.

Appendix A
The Path Not Taken

Earlier versions of this document used a different mechanism for calling the tool dispatcher. This appendix explains what that mechanism was and why we choose not to use it.

Calling the Dispatcher. The dispatcher was called using the new 65816 instruction **COP**.

The **COP** is a two byte instruction: an opcode followed by a signature byte. Western Design Center has reserved all the signature values from 128 through 255. This leaves us with 0 through 127.

Call Information. The bytes in memory immediately following the **COP** contained call information. The first byte is the tool set number (TSNum), the second byte is the function number (FuncNum).

The pointer to a parameter table comes next. Since the parameter table can be anywhere in memory, the pointer must be at least three bytes long. But by convention we use four bytes to represent addresses in the 65816 memory range. This is much more efficient than three byte pointers for passing a pointer (although less efficient for memory use).

Sample Call. A sample call looked something like:

```
cop 0
dfb TSNum
dfb FuncNum
long Params
```

So what's wrong with it? We had a version of the dispatcher which used this mechanism working before we discarded it. We discarded it for two reasons: first it was not easy for high level languages to use this scheme. Second, the best dispatch time we could get was 170 microseconds. The scheme we choose takes only 110 microseconds.