OS-9 System Extension Modules

Boisy G. Pitre

The technical information, especially in the OS-9 Level Two manuals, is brimming with details and information that can unlock a wealth of understanding about how OS-9 works. Unfortunately, some of this information can be hard to digest without proper background and some help along the way. This series of articles is intended to take a close look at the internals of OS-9/6809, both Level One and Level Two. So along with this article, grab your OS-9 Technical Manual, sit down in a comfortable chair or recliner, grab a beverage, relax and let's delve into the deep waters!

Assemble Your Gear

For successful comprehension of the topics presented in this and future articles, I recommend that you have the following items handy:

- OS-9 Level Two Technical Reference Manual OR
- OS-9 Level One Technical Information Manual (light blue book) and the OS-9 Addendum Upgrade to Version 02.00.00 Manual
- A printout of the `os9defs` file for your respective operating system. This file can be found in the DEFS directory of the OS-9 Level One Version 02.00.00 System Master (OS-9 Level One) or the DEFS directory of the OS-9 Development System (OS-9 Level Two).

In this article, we will look at a rarely explored, yet intriguing OS-9 topic: system extensions, a.k.a. P2 modules. When performing an `mdir` command, you have no doubt seen modules with names like `OS9p1` and `OS9p2` in OS-9 Level Two (or `OS9` and `OS9p2` in OS-9 Level One). These modules are essentially the OS-9 operating system itself; they contain the code for the system calls that are documented in the OS-9 Technical Reference documentation. In the case of OS-9 Level One, the modules `OS9` and `OS9p2` are located in the boot track of your boot disk (track 34). In OS-9 Level Two, `OS9p1` (equivalent to the `OS9` module in Level One) is found in the boot track while `OS9p2` is located in the bootfile. Both of the modules are of module type *Systm* and define the basic behavior and structure of OS-9. Even the module `IOMan` is a system extension, containing code for the I/O calls in the operating system.

While drivers and file managers have been the most common area to expand the capabilities of OS-9, they are pretty much limited to expanding the functionality of I/O. What system extensions allow you to do is even more powerful: they can add new system calls or even replace existing ones. Such functionality allows you to change the behavior of OS-9 in a very fundamental way. Of course, with such power, caution must be exercised. It is not wise to radically modify the behavior of an existing system call; such an action could break compatibility with existing applications.

What we aim to do in this article is not to replace an existing system call, but rather to add a new system call by looking at the example provided in Tandy's OS-9 Level Two documentation. Although the example is written for OS-9 Level Two, we will look at how it can be changed to run under OS-9 Level One as well. But first, let's get a little background on system calls and how they are constructed in OS-9.

The System Call

As an operating system, OS-9 provides system level functions, or system calls to applications. These system calls give applications a base by which they can operate consistently and without fear of incompatibility from one OS-9 system to the next. The system call in OS-9/6809 evaluates to an SWI2 instruction on the 6809, which is a software interrupt. Suffice it to say that when this instruction is encountered by the CPU, control is routed to OS-9, which interprets and performs the system call on behalf of the calling process.

While system calls are generally hidden by wrapper functions or procedures in high-level languages such as Basic09 and C, we can see the system call in its native form by looking at 6809 assembly language. Consider the following assembly source fragment:

```
        lda   #1
        leax  mess,pcr
        ldy   #5
        os9   I$Write
        rts
mess    fcc   "Hello"
```

In the middle of what appears to be normal 6809 assembly language source code is a mnemonic called `os9`. This is a pseudo mnemonic, since Motorola did not place an `os9` instruction in the 6809 instruction set. The OS-9 assembler actually recognizes this pseudo mnemonic as a special case, along with the `I$Write` string, and translates the above piece of code into:

```
        lda   #1
        leax  mess,pcr
        ldy   #5
        swi2
        fcb   $8A
        rts
mess    fcc   "Hello"
```

The $8A which follows the `swi2` instruction is the constant representation of the I/O system call `I$Write`. Since the `swi2` instruction calls into the OS-9 kernel, the code in the kernel looks for the byte following the `swi2` instruction in the module (the $8A) and interprets that as the system call code. Using that code, OS-9 jumps to the appropriate routine in order to execute the `I$Write`.

Since the system call code following the `swi2` instruction is a byte, in theory this would allow OS-9 to have up to 256 different system calls that can be executed on behalf of an application. Under OS-9 Level Two, this is the case; however under OS-9 Level One there are restrictions placed on exactly which codes are available. The following tables show the range of system call codes.

Table 1 – OS-9 Level One System Call Ranges

System call range	*Function*
$00-$27	User mode system call codes
$29-$34	Privileged system mode call codes
$80-$8F	I/O system call codes

Table 2 – OS-9 Level Two System Call Ranges

System call range	*Function*
$00-$7F	User mode system call codes
$80-$8F	I/O system call codes
$90-$FF	Privileged mode system call codes

The idea behind *User mode* vs. *System mode* is to allow two different points of execution for the same system call, depending on whether the calling process is running in user state or system state. OS-9 controls this by maintaining two system call tables: one for user state and one for system state. When installing a system call, as we'll soon see, we can specify whether our system call should only be called from system state (hence only updating the system table) or from both user and system state (updating both the user and system tables).

An example of a system call that can be executed in both user and privileged modes is the `F$Load` function code (pp. 8-25 in the OS-9 Level Two Technical Reference manual; pp. 106 in the OS-9 Level

One Technical Information manual). Since F$Load can be called from a user state process as well as from a driver or other module running in system state, OS-9 installs this system call in both the user and system tables. On the other hand, a privileged mode system call such as F$AProc (Level Two: pp. 8-74; Level One: pp. 141) can only be called from system state and therefore a user state process attempting to call it will receive an error.

Notice that in both OS-9 Level One and OS-9 Level Two, codes $80-$8F are reserved for I/O system call codes. When the OS-9 kernel receives one of these codes, it passes the code along to IOMan for processing. I/O system calls cannot be added since they are under the control of IOMan.

Installing a new system call involves selecting a free system call code, determining whether the call will be accessible from both user/system state or from system state only, and building a table in assembly language that will be used to install the system call. Interestingly enough, the method of installing a system call is by calling a system call! It's called F$SSvc and is documented in your respective OS-9 Technical manual.

Installing a System Call in OS-9 Level Two

The source code in Listing 1 is the system extension module, os9p3.a, which contains the code to install the system call, as well as the system call code itself. Incidentally, this is virtually the same code that is found in the OS-9 Level Two Technical Reference Manual on pp. 2-2 to 2-4. I've eliminated the comments for brevity since they are already in your manual, as well as changed the use directive. Instead of including /dd/defs/os9defs, I include /dd/defs/os9defs.l2. The reason for this is that I do compiling of both OS-9 Level One and OS-9 Level Two modules on my CoCo 3 development system. Since the OS-9 definitions are different for each operating system, I have renamed their respective os9defs files with an extension indicating which operating system they belong to. Even if you just develop for one operating system or the other, I strongly suggest following the same naming convention; it will save you headaches in the long run.

This module, called OS9p3, installs the F$SAYHI system call. A process making this call can either pass a pointer to a string of up to 40 bytes (carriage return terminated) in register X, or set X to 0, in which case the system call will print a default message. In either case, the message goes to the calling process' standard error path. While not very useful, this system call is a good example of how to write a system extension.

The asm program is used to assemble this source code file. Notice that the entry point for the module is the label Cold, where Y is set to the address of the service table, SvcTbl. Each entry in this table contains three bytes. The first is the system call code that we have selected from a range that Microware says is safe to use for new system calls, and the remaining two are the address of the first instruction of the system call. The table, which can contain any number of entries, is terminated by byte $80. After setting Y to the address of the service table, a system call to F$SSvc is made, which takes the table pointed to by Y and installs the system calls.

The code for the F$SAYHI system call in listing 1 is for OS-9 Level Two only. It determines whether or not a valid string pointer has been passed in register X. If indeed the caller has passed a valid pointer, then control is routed to the label SayHi6 where Y is loaded with the maximum byte count and the process descriptor of the calling process is used to obtain the system path number of the process' standard error in register A. The separation of user and system state paths is an important concept to understand; however, we will discuss it in detail in another article. For now, let's continue analyzing the code.

The I$WritLn system call then prints the string at register X to the caller's standard error path. If on the other hand, register X contains a zero, then room is made on the caller's stack for the default message, which is then copied into the caller's address space using the F$Move system call. The moving of the default message from the system address space to the caller's address space is necessary due to the separation of a process' address space in OS-9 Level Two.

Once the module has been compiled, it should be included in your OS-9 Level Two bootfile. Reboot with the new bootfile, and the `OS9p2` module will find `OS9p3` then jump into the execution offset (the `Cold` label in this case). This will install the `F$SAYHI` system call and make it available for programs immediately.

Installing a System Call in OS-9 Level One

Listing 2 is similar to the code in Listing 1, except that the code to move the default message from system space to the caller's address space has been removed. Also, the code to install the system call has changed, and the module type is not of type `Systm`, but instead of type `Prgrm`. This is due to the lack of separation of address space in Level One, which makes writing system extension modules much easier than in Level Two.

The common address space between the system and all processes in OS-9 Level One also makes the `F$SSvc` system call available from user state as well as from system state. Unlike OS-9 Level Two, where the system extension module must be placed in the bootfile, installing a system extension in OS-9 Level One takes a different approach. Placing a module called `OS9p3` in an OS-9 Level One bootfile will NOT cause the system extension to be called because there are no provisions for that in the kernel. Instead, system extensions are installed by creating a module of type `Prog` that contains both code to install the system call and the system call itself. Installing the system call entails executing the module from the command line.

Besides the `sayhi.a` source in listing 2, another example of the this is the `Printerr` command that comes with OS-9 Level One. This is a program that actually installs a newer version of the `F$PErr` system call. To install the new system call, you simply run `Printerr` from the command line. It then installs the call and exits. There is an advantage to OS-9 Level One's approach to installing system calls: it can be done at run-time without making a new bootfile and rebooting the system. However, additional care must be taken not to unlink the `Printerr` module from memory. Why? Because the code for the replacement `F$PErr` call is in that module, and if the module is unlinked, the memory it occupied is made available subsequent reallocation and at some point, a system crash will ensue.

Exercising Our New System Call

Listing 3 is a small assembly language program, `tsayhi`, which calls the F$SAYHI routine. It will work fine under both OS-9 Level One and Level Two. If you fork the `tsayhi` program without any parameters, then the `F$SAYHI` system call is called with register X set to $0000, which will cause the system call to print the default message. Otherwise, you can pass a message on the command line as a parameter and up to 40 of the message's characters will be printed to the standard error path.

Summary

Extension modules give us an effective way of altering the behavior of OS-9 by allowing us to add a new system call or modify the behavior of an existing one. Writing extension modules requires an extremely good understanding of the internals of OS-9. The particulars of writing a system extension vary under OS-9 Level One and Level Two primarily due to the differences between memory addressing.

Listing 1 - Source for os9p3.a for OS-9 Level Two

```
Type     set   Systm+Objct
Revs     set   ReEnt+1
         mod   OS9End, OS9Name,Type,Revs,Cold,256
OS9Name  fcs   "OS9p3"

         fcb   1                        edition number

         ifp1
         use   /dd/defs/os9defs.l2
         endc

level    equ   2
         opt   -c
         opt   f

* routine cold
Cold     leay  SvcTbl,pcr
         os9   F$SSvc
         rts

F$SAYHI  equ   $25

SvcTbl   equ   *
         fcb   F$SAYHI
         fdb   SayHi-*-2
         fcb   $80


SayHi    ldx   R$X,u
         bne   SayHi6
         ldy   D.Proc
         ldu   P$SP,y
         leau  -40,u
         lda   D.SysTsk
         ldb   P$TASK,y
         ldy   #40
         leax  Hello,pcr
         os9   F$Move
         leax  0,u
SayHi6   ldy   #40
         ldu   D.Proc
         lda   P$PATH+2,u
         os9   I$WritLn
         rts

Hello    fcc   "Hello there user."
         fcb   $0D

         emod

OS9End   equ   *
```

```
end
```

Listing 2 - Source for sayhi.a for OS-9 Level One

```
Type     set   Prgrm+Objct
Revs     set   ReEnt+1
         mod   OS9End, OS9Name,Type,Revs,Cold,256
OS9Name  fcs   "SayHi"

         fcb   1                        edition number

         ifp1
         use   /dd/defs/os9defs.l1
         endc

level    equ   1
         opt   -c
         opt   f

* routine cold
Cold     equ   *
* The following three instructions are important.  They cause the link
* count of this module to increase by 1.  This insures that the module
* stays in memory, even if forked from disk.
         leax  OS9Name,pcr
         clra
         os9   F$Link

         leay  SvcTbl,pcr
         os9   F$SSvc
         bcs   Exit
         clrb
Exit     os9   F$Exit

F$SAYHI  equ   $25

SvcTbl   equ   *
         fcb   F$SAYHI
         fdb   SayHi-*-2
         fcb   $80

* Entry point to F$SAYHI system call
SayHi    ldx   R$X,u
         bne   SayHi6
         leax  Hello,pcr
SayHi6   ldy   #40
         ldu   D.Proc
         lda   P$PATH+2,u
         os9   I$WritLn
         rts

Hello    fcc   "Hello there user."
         fcb   $0D

         emod
```

```
OS9End   equ   *
         end
```

Listing 3 - Source for tsayhi.a

```
Type     set   Prgrm+Objct
Revs     set   ReEnt+1
         mod   OS9End, OS9Name,Type,Revs,start,256
OS9Name  fcs   "TSayHi"

         fcb   1                        edition number

         ifp1
         use   /dd/defs/os9defs
         endc

level    equ   2
         opt   -c
         opt   f

F$SAYHI  equ   $25

* routine cold
start    equ   *
         lda   ,x
         cmpa  #$0D
         bne   SayHi
         ldx   #$0000
SayHi    os9   F$SAYHI
         bcs   error
         clrb
error    os9   F$Exit

         emod

OS9End   equ   *
         end
```