INVESTIGATING 65C816 INTERRUPTS

Published by BCS Technology Limited

TABLE OF CONTENTS

INTRODUCTION

The **Western Design Center**'s **W65C816S microprocessor** (hereafter referred to as the 65C816) extends the 65C02 eight bit microprocessor to a 16 bit capable device with added instructions and features. Among those features are interrupt processing capabilities that have no analog in the eight bit 6502 family. These capabilities can be harnessed in concert with other 65C816-unique features, such as stack pointer relative addressing, to create advanced software routines that can assist in complex operating system and device driver development. When applied to existing interrupt-driven functions, execution speed improvements and reductions in code size are possible with a modicum of effort.

Before you start reading please understand that ***this article is not a general 6502 interrupt primer***, nor does it discuss operation of the 65C816 in its 65C02 emulation mode. Everything herein pertains to native mode operation only, although some of the discussion on coding style is certainly applicable to any member of the 6502 family.

Although this article will present some material on the electrical characteristics of the 65C816's interrupt inputs and the circuits to which they may be connected, hardware design will be largely ignored. Much of this discussion will also be applicable to the W65C802, which is an obsolete W65C02S plug-compatible form of the 65C816. As the 65C802 was designed to replace a W65C02S in-circuit, some hardware signals described herein are not present.

In writing this article, we have assumed that you are a reasonably proficient programmer who knows the 65C816 assembly language, has developed a sound coding style, knows what an interrupt is and understands why computers use interrupts. If these assumptions don't describe you then you need to seek out other references before continuing. One such reference is Garth Wilson's **6502 interrupt primer**, which starts with the basics and explains in detail how the 6502 family behaves when interrupted. Code examples of common interrupt-driven tasks that would be performed in a typical 6502-powered system are also presented and in many cases, can be pasted directly into your assembly language programs. Enjoy the 1980s-style cartoons as well!

We caution that if you do not understand the concepts presented in Garth's primer then you will experience difficulty in understanding the material presented herein. This article has not been written to the level of a novice. An enormous amount of information about the 6502 family can be found both in printed form and on-line, as it is arguably the most documented microprocessor family ever developed. Two good places at which to start looking on-line are **6502.org** and **wilsonminesco.com**, the latter which is Garth Wilson's extensive 6502-oriented website. Or enter "65C816" into your favorite search engine.

For a general reference to the 6502 family assembly language, we recommend the Western Design Center's **programming reference manual**. Encompassing some 450 pages, this manual has a wealth of information for the beginning 6502 assembly language programmer, as well considerable detail for those who already know the 6502 assembly language and want to get the most out of the 65C816.

All 65C816 assembly language examples will use MOS Technology standard assembler syntax, which has been carried forward with suitable modifications to account for 24 bit operands and new addressing modes by WDC in their syntax standard for the 65C816. If your assembler conforms to this standard, great! If not, you may have to do some adjusting to code examples.

Table of Contents

ACRONYMS, SYMBOLS AND NOTES

Throughout this article, various acronyms and symbols will be used to refer to 65C816 features, frequently-used programming elements and other concepts:

Ø2	System clock signal, pronounced *fee 2*
IRQ	Maskable interrupt, pronounced *eye are cue*
LSB	Least-significant bit or byte; obvious from context
MSB	Most-significant bit or byte; obvious from context
NMI	Non-maskable interrupt, pronounced *en em eye*

Text references to the 65C816's registers are as follows:

Symbol	Register Description	Size in Bits
.A	accumulator LSB (m=1)	8
.B	accumulator MSB (m=1)	8
.C	accumulator (m=0)	16
.X	X-index register (affected by x)	8/16
.Y	Y-index register (affected by x)	8/16
DB	Data bank	8
DP	Direct page pointer	16
PB	Program bank	8
PC	Program counter	16
SP	Stack pointer	16
SR	Status	8
m	Accumulator/memory size flag	1
x	Index register size flag	1

A brief digression on the DB and PB registers:

> During program execution, the effective program address (EPA) seen by other hardware in the system is the 16 bit content in PC catenated with the eight bit content in PB, resulting in a 24 bit address in which PB represents bits 16-23 and PC represents bits 0-15. The EPA is where the 65C816 will fetch an instruction opcode and its operand, if any. Note that there is no direct programmatic means by which the content of PB can be changed.
>
> Similarly, the effective data address (EDA) is the concatenation of the value in DB, which represents bits 16-23, with the 16 bit address derived from a 16 bit address operand, or from a 16 bit address stored in a pair of contiguous direct page locations, resulting in a 24 bit address from which data will be read or written. EDA and EPA are always $00xxxx in a system in which no bank latching hardware is present, regardless of the actual values in DB and PB. Furthermore, the value in DB will be ignored if an instruction has a 24 bit operand or if the addressing mode involves direct page, for example, LDA $12. Direct page and hardware stack accesses are always addressed to bank $00.

Table of Contents

65C816 INTERRUPTS

As is typical of almost all microprocessors, the 65C816's interrupt types broadly fall into two classes: hardware and software.

- **A hardware interrupt occurs when a specific electrical input on the 65C816 is asserted by other hardware in the system**, such as an input/output (I/O) device. ***Asserted*** means

the input has been brought to state that is opposite of the normal or quiescent operating state. 6502 family interrupt inputs are quiescent when at logic one, which is approximately five volts in many applications, and thus are brought low (logic zero, which is near zero volts) when asserted, a type of logic referred to as "low-true." The manner in which the 65C816 reacts to a hardware interrupt is determined by which input has been asserted and in some cases, by when the input has been asserted relative to the instruction cycle. Hardware interrupts are asynchronous in nature, meaning they must be expected to occur at any time with no advance warning to a running program (the "foreground task").

- **A software interrupt is caused by the 65C816 executing a program instruction that triggers an interrupt-like sequence in the microprocessor's internal circuits**. As software interrupts are generated under program control, they are synchronous, which means the executing program will always "know" when one is going to occur.

Hardware Interrupts

The 65C816 has four hardware interrupt inputs, summarized as follows:

Common Name	Interrupt Mnemonic	Input Name	Input Type	Microprocessor Action	Hardware Vector
Abort	ABT	ABORTB	Level	"Abort" current instruction	$00FFE8
Interrupt request	IRQ	IRQB	Level	Service maskable interrupt	$00FFEE
Non-maskable interrupt	NMI	NMIB	Edge	Service non-maskable interrupt	$00FFEA
Reset	RST	RESB	Level	Reset state	$00FFFC

Some key points related to the above follow:

- ***Interrupt mnemonic*** is a convenient way to refer to the interrupt in a textual context. Saying "NMI" or "IRQ" is more convenient than "non-maskable interrupt" or "interrupt request." An interrupt mnemonic is also useful in hardware electrical schematics. In such cases, it is customary to add a symbol that indicates the low-true nature of the signal, such as /NMI or *NMI. Overlining, for example, $\overline{\text{NMI}}$, is often found in typeset literature, but rarely seen in schematics.

- ***Input type*** refers to the nature of the electrical signal that is required for the 65C816 to recognize that it is being interrupted:

  - ***Level*** means the 65C816 responds to the interrupt input any time the signal level is logic zero or low.

  - ***Edge*** means the 65C816 responds to the interrupt input only when the signal level makes a transition from logic one (high) to logic zero.

  While no attempt will be made here to delve into the more arcane aspects of interrupt circuit design, understanding the characteristics of level versus edge sensitive interrupt inputs, as well as other factors, is essential to writing a trouble-free interrupt service routine, especially for a system that will run at a high Ø2 rate (the 65C816 has been successfully operated at speeds as high as 20 megahertz (MHz) in commercial products, and much higher speeds are attainable in FPGA designs). Some discussion on this will be presented later on.

- ***Input name*** is the official designation in the **65C816 data sheet** of the chip pin corresponding to the interrupt input. In WDC data sheets, a B on the end of the input name, for example, IRQB, signifies that it is a low-true input.

- Each interrupt input is associated with a specific hardware vector from which the 65C816 will get the address of the corresponding interrupt service routine. ***The listed hardware vectors are for native mode operation only***, with one exception: the reset vector is the same for both native and emulation modes, as the 65C816 is reverted to emulation mode when RESB is asserted.

- In the event two or more interrupt inputs are simultaneously asserted, the 65C816 will respond in a defined order, that is, it will give interrupts a response priority:

  1. RESB
  2. ABORTB
  3. NMIB
  4. IRQB

  This list simply means that if, for example, ABORTB and NMIB are simultaneously asserted, the 65C816 will respond to ABORTB and then upon completion of the abort interrupt service routine, will respond to NMIB, assuming that NMIB is still asserted.

- Normally, the 65C816 doesn't "look" at its interrupt inputs until the currently executing instruction has been completed. When response to an interrupt does occur, the 65C816 has to save its state on the stack before loading the interrupt vector and proceeding. If the 65C816 is operating in native mode, eight Ø2 clock cycles will be consumed in responding to the interrupt and saving state. If the interrupt happens to occur just as the current instruction's opcode is being fetched, as many as eight additional Ø2 cycles may elapse before interrupt recognition. Therefore, an **interrupt latency** of as many as 16 Ø2 cycles is possible, depending on the instruction being executed and when during the instruction sequence the interrupt input is asserted. The effects of variable interrupt latency will be discussed later on.

A more detailed explanation of each interrupt type follows. In the lists that describe the sequence of actions for each interrupt type, the event numbers are merely for list purposes, and do not imply how many Ø2 clock cycles have elapsed at any given point in the sequence.

- **Abort**

  The abort interrupt is used in systems that have specialized memory management logic to cope with unusual hardware conditions that may arise during program execution. In such systems, an abort interrupt might be triggered if a running program accesses memory that either doesn't exist—a "page fault" in a virtual memory system, or is privileged—an "access violation" or "memory fault" in a multitasking protected memory environment. Few homebrew computers are likely to be built with such features—the required hardware logic is quite complicated. Hence discussion on processing an abort interrupt will be limited to describing how the 65C816 reacts when ABORTB is asserted. If you are sufficiently hardware-savvy to be able to design a system that can take advantage of this interrupt then it is a sure bet you won't need any assistance in determining how to process it when it occurs.

  When the 65C816 receives an abort interrupt, the following actions occur:

  1. All steps of the current instruction are completed but ***no changes are made to the registers or memory***.
  2. PB is pushed to the hardware stack.
  3. The aborted instruction's address is pushed to the stack, MSB first followed by LSB.
  4. SR is pushed to the stack.

5. The I (IRQ disable) bit in SR is set.
6. The D (decimal mode) bit in SR is cleared.
7. PB is loaded with $00.
8. PC is loaded with the contents of the abort hardware vector at $00FFE8 (LSB) and $00FFE9 (MSB).
9. Execution is transferred to the abort interrupt service routine.

Note that the 65C816 does not automatically save .A, .B, .X, .Y, DB and DP, nor does it change any bits in SR except D and I. Upon executing an RTI instruction, the above sequence will be logically reversed to return the 65C816 to the state it was in at the time of the interrupt, and unless the address that was pushed to the stack in steps 2 and 3 is altered within the interrupt service routine, the 65C816 will return to and again execute the aborted instruction in the interrupted program.

It should be noted that despite the interrupt's name, ***an "aborted" instruction isn't actually aborted***—all steps of the instruction will be completed before the 65C816 reacts to the interrupt. What is aborted are computational changes to a register and/or memory that the instruction would have made had it not been "aborted." ABORTB has strict timing requirements relative to the instruction sequence that must be satisfied in order to assure that the above behavior will actually occur. Understanding these requirements and the character of an abort interrupt is crucial to being able to design a system that can support hardware memory protection and/or instruction execution trapping.

- **Interrupt request**

An interrupt request (IRQ) is also referred to as a maskable interrupt, which means the microprocessor can be made to ignore an IRQ. As the IRQB input is level-sensitive, it is practical to connect multiple interrupt sources to it in a configuration referred to as "wired-OR." In a typical wired-OR configuration, the interrupt service routine has to determine which devices are interrupting by using a procedure referred to as "polling." More advanced systems may include hardware that can tell the 65C816 which device has interrupted, which helps to reduce software-induced interrupt latency by eliminating the necessity of polling each possible IRQ source. If IRQB is still low after an interrupt source has been serviced and cleared, a new IRQ will occur.

In most systems, an IRQ is the primary means by which input/output (I/O) devices get the 65C816's attention when they need service. For example, a disk controller would assert IRQB to indicate that it is ready for some data. Or serial interface hardware, such as a UART (**U**niversal **A**synchronous **R**eceiver/**T**ransmitter), may interrupt when a user types at a terminal. In many systems, a hardware timer running at a constant rate generates a **jiffy IRQ** that is used for general timekeeping, process scheduling, etc. Therefore, an IRQ service routine may be quite complex and lengthy, and could involve considerable hardware interaction, depending on the number and nature of the I/O devices in the system.

When the 65C816 receives an interrupt request, the following actions occur:

1. All steps of the current instruction are completed and memory and/or registers are updated as required.
2. The I bit in SR is tested and if set, the IRQ is ignored—none of the following steps will be executed.
3. If the I bit in SR is clear the 65C816 will process the interrupt and start by pushing PB to the stack.
4. PC, which is pointing to the next instruction to be executed, is pushed to the stack, MSB first followed by LSB.
5. SR is pushed to the stack.
6. The I bit in SR is set.

7. The D bit in SR is cleared.
8. PB is loaded with $00.
9. PC is loaded with the contents of the IRQ hardware vector at $00FFEE (LSB) and $00FFEF (MSB).
10. Execution is transferred to the IRQ service routine.

Note that the 65C816 does not automatically save .A, .B, .X, .Y, DB and DP, nor does it change any bits in SR except D and I. Upon executing an RTI instruction, the above sequence will be logically reversed to return the 65C816 to the state it was in at the time of the interrupt, and unless the address that was pushed to the stack in steps 3 and 4 is altered within the interrupt service routine, the 65C816 will execute the next instruction in the interrupted program.

- **Non-maskable interrupt**

A non-maskable interrupt (NMI) is similar to an IRQ in effect, except there is no programmatic means by which an NMI can be blocked. Typically, an NMI would be used to interrupt the microprocessor in response to one very high priority event. For example, NMIB may be driven by a timekeeper so as to guarantee that the device is immediately serviced when it interrupts. Although multiple interrupt sources may be connected to NMIB in a wired-OR configuration, such an arrangement will be problematic, as NMIB is edge-sensitive. Unless all NMI sources are checked and cleared by the NMI service routine, the 65C816 will not respond to another NMI, since NMIB will continue to be held at logic zero by the device that wasn't serviced. In many 6502-based homebrew computers, NMIB is not used at all or is wired to a push button circuit so the user can interrupt a runaway program and regain control.

When the 65C816 receives a non-maskable interrupt, the following actions occur:

1. All steps of the current instruction are completed and memory or registers are updated as required.
2. PB is pushed to the hardware stack.
3. PC, which is pointing to the next instruction to be executed, is pushed to the stack, MSB first followed by LSB.
4. SR is pushed to the stack.
5. The I bit in SR is set.
6. The D bit in SR is cleared.
7. PB is loaded with $00.
8. PC is loaded with the contents of the NMI hardware vector at $00FFEA (LSB) and $00FFEB (MSB).
9. Execution is transferred to the NMI service routine.

Note that the 65C816 does not automatically save .A, .B, .X, .Y, DB and DP, nor does it change any bits in SR except D and I. Upon executing an RTI instruction, the above sequence will be logically reversed to return the 65C816 to the state it was in at the time of the NMI, and unless the address that was pushed to the stack in steps 2 and 3 is altered within the interrupt service routine, the 65C816 will execute the next instruction in the interrupted program.

- **Reset**

Although you may not consider a reset to be an interrupt, RESB is an interrupt input and triggers some internal actions that are like those of other interrupt types. In the overwhelming majority of applications, RESB is wired to a circuit that includes a push-button for manual restarting of the system. The same circuit is usually designed to hold RESB low for a short period of time after power is applied so that all voltages and other circuit conditions will have time to stabilize before the 65C816 commences code execution. Embedded controller applications may use RESB as an actual interrupt in

cases where the controller idles for long periods of time awaiting activity and the 65C816 has been stopped to conserve power while waiting. Some controllers may have a watchdog timer wired to RESB to force a restart if system fatality occurs.

When RESB is brought low the 65C816 will immediately halt whatever it is doing and remain in a halted state as long as RESB is held low. Upon release of RESB, the following actions will occur:

1. The internal clock, which is driven by the Ø2 clock generator circuit, will be restarted if it had previously been stopped by an STP or WAI instruction (described in the **high speed interrupt response subsection**).
2. The I bit in SR will be set.
3. The D bit in SR will be cleared.
4. The hidden E (emulation) bit in SR will be set, causing the 65C816 to revert to W65C02S emulation mode.
5. The m and x bits in SR will be set and made inaccessible, thus forcing the accumulator and index register sizes to eight bits.
6. DB and PB will be set to $00, thus "hard wiring" all accesses to bank $00 and limiting the highest accessible address to $00FFFF.
7. DP will be set to $0000, thus "hard wiring" direct page program references to the physical zero page in RAM.
8. The MSB of SP will be set to $01, thus "hard wiring" the stack to $000100-$0001FF in RAM.
9. PC will be loaded with the contents of the reset hardware vector at $00FFFC (LSB) and $00FFFD (MSB).
10. Execution will be commence at the system reset handler.

The LSB of SP is undefined following a reset and must be explicitly set in the reset handler code, typically to $FF, since stack growth is downward. Also, the C, N, V and Z bits in SR will be in undefined states.

Software Interrupts

The 65C816 has two software interrupt instructions, summarized as follows:

Common Name	Instruction Mnemonic	Hardware Vector
Break	BRK	$00FFE6
Co-Processor	COP	$00FFE4

Again, note that the listed hardware vectors apply only to native mode operation. Key points are as follows:

- Both BRK and COP cause hardware interrupt-like sequences to occur in the 65C816, as will be shortly described.

- BRK and COP are treated as two byte instructions by the 65C816. However, standard assembly language syntax for BRK usually doesn't accept an operand, although one may be added by the programmer using an appropriate assembler pseudo-op. COP, on the other hand, must be assembled with an operand. The byte that follows BRK or COP is customarily referred to as a "signature byte."

- Unlike the eight bit 6502 family members, the 65C816's BRK instruction has its own hardware vector when operating in native mode. This features eliminates the need to examine the stack copy of the status register to differentiate between an interrupt caused by BRK and one

caused by asserting IRQB.

- Software interrupts are preempted by hardware interrupts if one of the latter occurs during the opcode fetch part of an instruction cycle. For example, if IRQB is asserted at the same time the 65C816 is fetching a COP opcode, the IRQ will be processed first. Only after the IRQ service routine has exited with RTI will the COP instruction be executed.

A more detailed explanation of COP and BRK follows.

- **Break**

  BRK is the "traditional" software interrupt with which all 6502 assembly language programmers are, or should be, familiar. BRK is most commonly used during software debugging to stop the program undergoing testing and start a machine language monitor to inspect memory and/or the microprocessor's registers. In the past, BRK was used to patch PROMs when program bugs were discovered, a practice that was obsoleted when EPROMs became readily available. In some cases, BRK has been used as a supervisor call instruction to invoke operating system services.

  Upon executing BRK the following actions occur:

  1. PB is pushed to the stack.
  2. PC is double incremented and then pushed to the stack, MSB first, followed by LSB.
  3. SR is pushed to the stack.
  4. The I bit in SR is set.
  5. The D bit in SR is cleared.
  6. PB is loaded with $00.
  7. PC is loaded with the contents of the BRK hardware vector at $00FFE6 (LSB) and $00FFE7 (MSB).
  8. Execution is transferred to the BRK service routine.

  Note that the 65C816 does not automatically save .A, .B, .X, .Y, DB and DP, nor does it change any bits in SR except D and I. Upon executing an RTI instruction, the above sequence will be logically reversed to return the 65C816 to the state it was in at the time of the interrupt, and unless the address that was pushed to the stack in steps 1 and 2 is altered within the interrupt service routine, the 65C816 will execute the next instruction in the interrupted program. As PC is twice incremented before being pushed to the stack, the "next instruction" will be at the address of the BRK instruction plus two. The interceding signature byte is ignored by the 65C816 and can be anything—a NOP ($EA) is customary during debugging, unless the BRK handler refers to the signature.

  It is important to note that IRQs will be masked by executing BRK, which means that if BRK is intercepted by a machine language monitor it is essential that IRQs be re-enabled. Otherwise, all interrupt-driven I/O operations will cease and the system will most likely be unresponsive.

- **Co-Processor**

  The COP instruction is unique to the 65C816 and is described in the data sheet as "...support[ing] co-processor configurations, i.e., floating point processors." Despite that statement, COP is a just another software interrupt, and it is up to the imagination of the system designer (you) to decide how to use it. As no floating point hardware that is bus-compatible with the 65C816 is known to exist at this time, COP may be (mis)used in a number of ways, one being as a operating system service call instruction—more on that to follow.

  Upon executing COP the following actions occur:

1. PB is pushed to the stack.
2. PC is double incremented and then pushed to the stack, MSB first, followed by LSB.
3. SR is pushed to the stack.
4. The I bit in SR is set.
5. The D bit in SR is cleared.
6. PB is loaded with $00.
7. PC is loaded with the contents of the COP hardware vector at $00FFE4 (LSB) and $00FFE5 (MSB)
8. Execution is transferred to the COP handler.

Note that the 65C816 does not automatically save .A, .B, .X, .Y, DB and DP, nor does it change any bits in SR except D and I. Upon executing an RTI instruction, the above sequence will be logically reversed to return the 65C816 to the state it was in at the time of the interrupt, and unless the address that was pushed to the stack in steps 1 and 2 is altered within the interrupt service routine, the 65C816 will execute the next instruction in the interrupted program. As PC is twice incremented before being pushed to the stack, the "next instruction" will be at the address of the COP instruction plus two. The interceding signature byte, which is required by the WDC assembly language syntax for COP, is ignored by the 65C816 and can be anything. However, WDC recommends that user signature bytes be confined to the range $00-$7F, as bytes $80-$FF are listed as "reserved" in the data sheet.

As with the BRK instruction, IRQs will be masked by executing COP.

Interrupt Vectoring

As previously noted, this article only superficially treats hardware. That said, brief mention will be made of the 65C816's VPB (vector pull) output signal. VPB is normally held at logic one by the 65C816. However, during cycles seven and eight of the microprocessor's interrupt response sequence VPB will go to logic zero to indicate that the 65C816 is loading PC with the appropriate interrupt vector, the LSB during cycle seven and the MSB during cycle eight. System logic can monitor VPB and when it goes to logic zero, modify the interrupt vector "on the fly" to reduce software-induced latency, as well as change the execution environment to suit operating system requirements.

Table of Contents

SOFTWARE ENGINEERING

A well-designed interrupt service routine represents a significant challenge for many programmers —as well as an occasionally rude lesson on the value of disciplined coding habits. In addition to meeting the obvious requirement of being able to correctly service and clear interrupts, an interrupt service routine should be:

- Transparent
- Reentrant
- Succinct

Other characteristics may well be required, but the above three are the most important in most systems. Let's take a closer look at this.

Transparency

An interrupt service routine is said to be "transparent" if it does not affect the environment of the interrupted foreground task in any way. In order for the foreground task to be able to be restarted without error following an interrupt, the interrupt service routine must preserve the state of the microprocessor at the time of the interrupt and must restore that state when interrupt processing has been completed. Also, transparency requires that the interrupt service routine use no memory other than the hardware stack, except in well-defined cases that are acceptable to interrupted foreground tasks. Otherwise, memory locations being used in the foreground may randomly change for no apparent reason, creating a potential debugging nightmare.

Getting back to preservation of the microprocessor's state, the interrupt service routine must make sure that whatever values were in the registers at the time of the interrupt are there when execution of the foreground task resumes. Some of this preservation process is automatically handled by the 65C816 when it acknowledges an interrupt, as it will push PB, PC, and SR to the stack prior to executing the interrupt service routine. However, as already noted, the 65C816 doesn't preserve any of its other registers, which means the interrupt service routine must see to that chore. Which registers must be preserved and restored will be implementation-dependent.

As a fairly rigid rule, any register that will be "touched" (changed) within an interrupt service routine must be preserved to assure transparency in all cases. Preservation is accomplished by pushing the registers to the stack before being touched and pulling them from the stack when the interrupt service routine has completed its work. ***Note that it is not necessary to preserve a register that the interrupt service routine does not touch***. For example, if your interrupt service routine only uses .X and .Y, there's no good reason to preserve the accumulator—doing so would simply waste valuable clock cycles and stack space. If you can guarantee that your interrupt service routine will not touch any of the registers, a possibility in some tightly-written embedded applications, do not waste time preserving them.

The 65C816 slightly complicates register preservation because the accumulator and index registers may be set to either eight or 16 bits at the time of the interrupt. Therefore, the interrupt service routine has to be careful to not make any assumptions in that regard, lest data be lost. This is especially true when the accumulator is considered, as it is really two registers designated .A and .B. Pushing the accumulator when it has been set to eight bits (m=1 in SR) will not preserve .B, which could result in a loss of transparency should the interrupt service routine touch .B. Therefore, it is essential that the accumulator be set to 16 bits before preservation and restoration if the interrupt service routine will be using .B. Also, beware of changes to .B via the TDC, TSC and XBA instructions. TDC and TSC are particularly sneaky, in that they result in a 16 bit transfer that overwrites .B, regardless of the status of the m bit in SR.

A non-obvious problem that confronts the 65C816 assembly language programmer is the fact that there is no way to conveniently determine the register widths in the interrupt service routine except by examining the m and x bits in SR. However, doing so requires that SR be pushed to the stack and then retrieved in the (eight bit) accumulator for analysis, causing the value in .A to be lost before it is preserved. This problem is best circumvented by assuming that the registers are set to 16 bits at the time of the interrupt, which simply means that the m and x bits in SR should be cleared before pushing the registers, and again cleared before pulling them at the end of the interrupt service routine. Doing so adds extra instructions and some clock cycles to the interrupt service routine, but does guarantee full register preservation.

For most applications, the following code will completely preserve the 65C816's state at the beginning of an interrupt service routine:

```
phb                 ;save DB
phd                 ;save DP
rep #%00110000      ;select 16 bit registers
pha                 ;save .C
phx                 ;save .X
phy                 ;save .Y
```

If the interrupt service routine has a stack separate from other stacks, preservation of SP must occur in memory after the above pushes have been completed. The following code, added to the above sequence, would handle this requirement:

```
tsc                 ;copy SP to .C &...
sta sp_fgnd         ;save somewhere in safe RAM
lda sp_isr          ;get ISR's stack pointer &...
tcs                 ;set new stack location
```

There are several items to consider:

- As previously noted, ***you should preserve only the registers that your interrupt service routine will be touching***.

- In a system with more than 64 kilobytes of RAM that is running multiple processes, it is quite possible that the interrupt service routine may need to load a different bank into DB in order to access a different process' data, hence the preservation of the data bank with the PHB instruction. Otherwise, PHB can be omitted if a change to DB isn't necessary during interrupt processing, or if the system has no more than 64 kilobytes of RAM.

- Preservation of the direct page pointer (the PHD instruction) is necessary if the interrupt service routine has been assigned its own direct page. However, use of direct page in this fashion, while preserving transparency, may prevent your code from **being reentrant**.

- A 65C816-powered system without benefit of special hardware logic will direct all stack accesses to RAM bank $00, regardless of the amount of RAM actually present. It may be advantageous in some cases to change the stack pointer after register preservation has been accomplished to prevent interrupt service routine stack accesses from inadvertently affecting the foreground task(s) stack(s). However, doing so may compromise transparency and most likely will prevent **reentrancy**—use caution.

- If you do assign a separate stack area to the interrupt service routine ***you must preserve the old stack pointer in RAM, not on the stack***. The following code will work but will also create an intractable problem:

```
rep #%00100000      ;16 bit accumulator
tsc                 ;copy SP to .C &...
pha                 ;save on stack
lda sp_isr          ;get ISR's stack pointer &...
tcs                 ;set new stack location
```

  As the TCS instruction will set a new stack pointer, how would you reverse the PHA instruction that pushed the foreground task's stack pointer to the foreground task's stack?

At the completion of the interrupt service routine, the above steps would be reversed as follows:

```
rep #%00110000      ;16 bit registers
lda sp_fgnd         ;get foreground task's SP &...
tcs                 ;set it
ply                 ;restore .Y
plx                 ;restore .X
pla                 ;restore .C
pld                 ;restore DP
plb                 ;restore DB
rti                 ;resume foreground task
```

Again, omit any steps that involve registers that weren't touched by the interrupt service routine. Note that upon executing RTI, the 65C816 will automatically restore the correct register sizes, since pulling SR restores the state of the m and x bits to what existed at the time of the interrupt.

For programming convenience, you may wish to write a single interrupt service routine exit point, which would encompass the above instructions, except for the stack pointer restoration:

```
;crti: COMMON INTERRUPT RETURN
;
crti     rep #%00110000        ;16 bit registers
         ply                   ;restore .Y
         plx                   ;restore .X
         pla                   ;restore .C
         pld                   ;restore DP
         plb                   ;restore DB
         rti                   ;resume foreground task
```

As the IRQ handler typically sees much more activity than the other interrupt service routines in most systems, `CRTI` (**C**ommon **R**e**T**urn from **I**nterrupt) should be in-line with the IRQ handler to reduce execution time—that is, the IRQ handler should be able to "fall through" to `CRTI` to avoid the time penalty of a branch or jump instruction. Other interrupt service routines should use `BRA`, `BRL` or `JMP` to get to `CRTI`, with `BRA` being preferred. Avoid use of `BRL` in interrupt service routines unless you are writing relocatable code. `BRL` uses four Ø2 cycles, whereas `BRA` and `JMP` use three.

Reentrancy

An interrupt service routine is said to be "reentrant" if it can be interrupted and made to process a new interrupt without disturbing any work that was in progress on behalf of the most recent interrupt. Depending on how interrupt processing has been arranged, such "nested interrupts" may occur even in small systems.

For example, consider a system in which a 65C51 UART is communicating with a modem, while a 65C22 VIA (**V**ersatile **I**nterface **A**dapter) is responsible for generating a jiffy IRQ to maintain system timekeeping. Let's suppose the VIA generates a timer underflow IRQ. As soon as the VIA's interrupt status register has been examined in the interrupt service routine and the timer IRQ has been cleared, let's also suppose IRQs are re-enabled and immediately thereafter, the UART interrupts because it has received a byte from the modem. If the interrupt service routine is fully reentrant, the UART interrupt will be serviced without delay and then the 65C816 will pick up where it left off while servicing the VIA interrupt (incidentally, this scenario implies that the UART has a higher interrupt priority than the VIA, which given the limitations of the 65C51, represents a sound software engineering decision). The amount of this sort of interrupt nesting that is possible is primarily limited by stack space, which is far less a concern with the 65C816 than with its eight bit cousins.

Reentrancy can only be achieved by fully satisfying the goal of transparency, especially the requirement that no memory except the hardware stack be used for storing temporary data. The 65C816's stack pointer relative addressing instructions, such as `LDA (<offset>,S),Y`, perform both direct and indirect loads and stores on the stack, with indirection facilitating the use of the stack as a fugacious direct page. Stack addressing in the context of interrupt processing will be extensively covered in the **advanced software interrupt programming** section.

Succinctness

Interrupt service routines need to execute as quickly as possible, as the time required to process interrupts is time that is not available to run foreground tasks. During the design of an interrupt service routine, consideration must be given to how often the routine will be executed in a given period of time. For example, in most systems, the NMI handler may not see much use, but the IRQ handler might be executed thousands or even tens of thousands of times per second. Clearly, any effort expended on improving the execution speed of the NMI handler would be better applied to the IRQ handler's code.

Unfortunately, the goal of succinctness can be elusive—there is usually a tradeoff between code

size and speed. However, with the 65C816 there are often ways to improve speed and reduce code size at the same time:

- **If you need to increment or decrement a value, avoid loading that value into a register**:

  Most experienced assembly language programmers already know this tip but often forget about it:

  ```
          inc counter
  ```

  is smaller and much faster than:

  ```
          ldx counter
          inx
          stx counter
  ```

  unless, of course, you need the new value in COUNTER loaded into .X for later operations.

  > **Caution**: Using any read-modify-write (R-M-W) instruction, such as ASL or INC, on I/O device registers may cause unexpected behavior.

- **Use BIT to test a hardware register if the actual register value isn't needed**:

  Many 6502 family hardware devices, such as the 65C22, indicate that they are interrupting by setting bit 7 in a flag register—bit 7 is logically wired to the device's $\overline{\text{IRQ}}$ output. Owing to how the flag bits are arranged in the register, it may be possible to determine the reason why the device is interrupting solely by the effect of a BIT instruction, eliminating the need to load the register into .A and apply Boolean operations.

  > **Caution**: The register contents may be cleared by the BIT operation, clearing the interrupting condition as well.

- **Arrange code so a branch is not taken in the most common case**:

  If a branch doesn't have to be taken then only two clock cycles will be consumed to execute the branch instruction. An additional clock cycle will be consumed if the branch has to be taken.

- **Take advantage of 16 bit operations when possible**:

  ```
          rep #%00100000
          inc counter
  ```

  is much faster and more succinct than:

  ```
          sep #%00100000
          inc counter
          bne next
          inc counter+1
  next    ...program continues...
  ```

  The first example increments all 16 bits at COUNTER and COUNTER+1 in a single instruction, using fewer bytes of code and fewer total clock cycles than the second example. The eight bit equivalent uses nearly twice as much time just to increment the 16 bit COUNTER and adding insult to injury, suffers an additional performance penalty due to the branch instruction, since the branch will be taken during 255 consecutive passes through the code.

- **Don't use 16 bit operations unless necessary**:

Yes, this advice seems to contradict the previous bit of wisdom. However, if a sequence of operations can be performed with eight bit memory accesses, there's nothing to be gained by employing 16 bit loads and stores. All 16 bit operations on memory incur a one clock cycle penalty due to the extra access required to load or store the MSB. Also, any 16 bit immediate mode instruction will obviously require a 16 bit operand—even if the operand's MSB is `$00`, increasing the size of the instruction, as well as the time required to decode and execute it.

- **Avoid multiple successive 24 bit loads and stores**:

  Any 24 bit access, such as `LDA $AB1234,X`, will incur a one clock cycle penalty as compared to the same instruction using a 16 bit access, such as `LDA $1234,X`. If it is necessary to perform multiple successive "long" operations, a performance gain can usually be realized by temporarily setting `DB` to the target bank, using 16 bit accesses on the target locations and then restoring `DB`. For example, consider code that increments five bytes in bank `$AB`. The first routine uses 24 bit loads and stores:

```
        sep #%00100000      ;8 bit accumulator
        ldx #4              ;modifying 5 locations
;
loop    lda $ab1234,x       ;load
        inc a               ;increment
        sta $ab1234,x       ;store
        dex
        bpl loop            ;next
;
        ...program continues...
```

  Performance suffers where performance matters the most: in the read-modify-write loop. Two 24 bit accesses plus the `INC A` instruction are required to make up for the lack of an equivalent 24 bit read-modify-write operation—unfortunately, `INC $AB1234,X` isn't in the 65C816's instruction set.

  Now consider the following code, which temporarily changes `DB` to accomplish the same task:

```
        phb                 ;save current data bank
        sep #%00110000      ;8 bit registers
        lda #$ab            ;target bank
        pha                 ;push it to the stack & pull it...
        plb                 ;into DB, making it the default bank
        ldx #4              ;modifying 5 locations
;
loop    inc $1234,x         ;effectively INC $AB1234,X
        dex
        bpl loop            ;next
;
        plb                 ;restore previous bank
        ...program continues...
```

  Although the above version looks larger and slower than the previous version, it is slightly smaller and substantially faster in the loop because a single R-M-W instruction with 16 bit addressing accomplishes what three separate instructions accomplish with 24 bit addressing in the first version. Consider that the extra clock cycle penalty of a 24 bit access is avoided twice per loop iteration. Even though some additional instructions are needed to change and restore `DB`, overall execution time is significantly shorter.

- **Don't use the `BRL` instruction unless you are writing relocatable code**:

  As mentioned earlier, `BRA` and `JMP` take three clock cycles to complete, whereas `BRL` consumes four cycles. `BRL` confers no advantages in a system where the interrupt service routines are loaded to fixed addresses. While `BRA` is no faster than `JMP` it does

require one less byte of code, which may be important if available code space is real tight.

- **Use linear code**:

  The use of subroutines in an interrupt service routine can substantially hurt performance, as each `JSR – RTS` pair will consume 12 clock cycles, or 14 cycles if using `JSL – RTL`. If your interrupt handler includes three calls to the same subroutine and is processing a 100 Hz jiffy interrupt, 3600 clock cycles will be consumed per second just in executing `JSR` and `RTS` instructions. A lot of foreground processing can be completed in 3600 cycles! Only use subroutines if you have to squeeze every last byte out of the available address space.

- **Avoid multiple device register accesses**:

  Operating the 65C816 at Ø2 rates over 8 MHz may necessitate the use of hardware **wait-states** when I/O devices must be accessed. A wait-state halts the microprocessor for one or more Ø2 cycles, during which time it will be doing absolutely nothing. If your interrupt service routine accesses the same I/O device register multiple times and access to that device requires a wait-state, the microprocessor will be doing absolutely nothing multiple times. If possible, access a device register only once and if the register content is needed later on, push it to the stack.

Spurious Interrupts

A spurious interrupt, also referred to as a phantom or ghost interrupt, is a hardware interrupt that does not have any apparent cause. The microprocessor responds to what appears to be a logic zero state at one of its interrupt inputs, but during the execution of the interrupt service routine none of the devices connected to that input indicate that they were interrupting. Depending on how the interrupt service routine has been written, nothing untoward will happen, or the microprocessor may do something completely bizarre trying to process an interrupt that never existed.

Spurious interrupts are occasionally caused by a number of factors related to chip timing (or more rarely, chip errata), but are most often due to interrupt circuit electrical characteristics. As earlier stated, this article isn't about hardware design. However, knowing something about the way in which wired-OR interrupt circuits behave can assist in avoiding spurious interrupts.

A wired-OR interrupt circuit connects the **open collector** interrupt outputs of multiple chips to an interrupt input on the microprocessor. "Open collector" means that unless a chip is actively interrupting, its interrupt output appears to be an open circuit, causing it to have no measurable effect on the system. This arrangement allows multiple interrupting devices to control a common interrupt input, reducing the parts count in the circuit. Any or all of the chips can simultaneously interrupt the microprocessor with no mutual interference.

As earlier explained, the microprocessor expects each of its interrupt inputs to be at a logic one voltage level when no interrupt is pending. As an open-collector device cannot actively drive a circuit to logic one, a **pull-up resistor** that connects the interrupt circuit to the computer's voltage source (Vcc) is used to maintain a logic one state when no interrupt is pending. When a chip does interrupt, it will pull the circuit down to logic zero, with the pull-up resistor limiting the current flow to a safe level. The microprocessor will recognize this state as an interrupt pending.

In theory, the transition from logic zero back to logic one that occurs when an wired-OR interrupt source is cleared is instantaneous. In practice, a phenomenon referred to as **parasitic or stray capacitance** will cause some delay before logic one is attained. Parasitic capacitance has to be charged up to Vcc through the pull-up resistor, which takes a measurable amount of time, this time being defined as the circuit's **R-C time-constant** (R-C means "resistance-capacitance"). The R-C time-constant sets a hard limit on how fast the circuit can change state from logic zero to logic one, which is what sets the stage for a spurious interrupt.

Although a careful circuit design that uses short and direct connections, as well as an appropriate value for the pull-up resistor, can minimize the R-C time-constant, it can never be reduced to zero. Therefore, your interrupt service routine must be written with the understanding that when an interrupt source is cleared there will be a delay before the microprocessor will actually "see" the transition from logic zero to logic one at its interrupt input. If logic one has not been attained by the time the interrupt service routine has completed its work and returned control to the interrupted foreground task, the microprocessor will start another interrupt sequence, even though no device is interrupting—a spurious interrupt.

In general, your interrupt service routine should poll and clear all interrupt sources as soon as possible after preliminary steps (for example, saving the 65C816's state) have been completed. The goal is to give the interrupt circuit as much time as possible to make the transition back to logic one before the interrupt service routine finishes. The longer your interrupt service routine waits before clearing interrupt sources, the greater the likelihood of a spurious interrupt.

In many chip designs, an interrupt status register has to be read to determine if the device is interrupting and if so, which event(s) caused the interrupt. Oftentimes, reading the interrupt status register will automatically clear the interrupt—which implies that the register value may have to be preserved for later processing if the device has multiple interrupt events (push it to the stack if necessary). In other cases, explicit action will be required to clear an interrupt, such as writing a mask value into a flag register. In either case, failing to take proper action can result in a device endlessly interrupting the microprocessor, which may eventually cause system fatality due to the rapid consumption of stack space. Be sure to carefully read the data sheet for each device in your system that is able to trigger an interrupt and understand exactly what must be done to clear the interrupt's cause.

Edge- vs. Level-Sensitive Interrupts

As earlier stated, some of the microprocessor's interrupt inputs are level-sensitive and others are edge-sensitive. It's important to be aware of these distinctions in your interrupt service routine, as seemingly random problems may otherwise occur, occasionally giving the impression that the system has crashed.

In most 6502-family designs, multiple interrupt sources are connected to `IRQB`, which is a level-sensitive input. Any one or all devices attached to `IRQB` can interrupt and assuming proper interrupt service routine design, the microprocessor will service all such devices, even if one interrupts while the microprocessor is already executing the interrupt service routine code in response to a previous interrupt.

The situation is different when multiple interrupt sources are connected to `NMIB`, which is an edge-sensitive input. As you may recall, the microprocessor responds to an edge-sensitive input only when a transition from high to low occurs. Therefore, once any device has asserted `NMIB` the microprocessor will not recognize that another NMI has occurred until `NMIB` has been deasserted and then asserted once more.

For example, if two devices are connected to `NMIB`, one interrupts and gets serviced and while the NMI handler is executing, the second device interrupts, the microprocessor will have no way of knowing that the second device is requesting service. When the NMI handler executes `RTI` and returns to the foreground task, the unserviced device will be ignored, as `NMIB` will remain low. If servicing that second device is critical to maintaining system activity (consider a jiffy timer that schedules tasks) and the microprocessor ignores it, the system may eventually **deadlock**. While it is possible to accommodate such a scenario with a carefully crafted NMI handler (it would have to poll devices several times before exiting to make sure that all have been serviced), a better arrangement is to connect only one device to `NMIB` and completely avoid the problem.

High Speed Interrupt Response

The 65C816 responds to hardware interrupts with alacrity. In fact, the 65C816's hardware interrupt latency is much shorter than comparable designs, and can be reduced even more by using the somewhat-arcane WAI (**WAI**t for interrupt) instruction. Let's review something that was earlier presented about hardware interrupts:

> Normally, the 65C816 doesn't "look" at its interrupt inputs until the currently executing instruction has been completed. When response to an interrupt does occur, the 65C816 has to save its state on the stack before loading the interrupt vector and proceeding. If the 65C816 is operating in native mode, eight Ø2 clock cycles will be consumed in responding to the interrupt and saving state. If the interrupt happens to occur just as the current instruction's opcode is being fetched, as many as eight additional Ø2 cycles may elapse before interrupt recognition. Therefore, an interrupt latency of as many as 16 Ø2 cycles is possible, depending on the instruction being executed and when during the instruction sequence the interrupt input is asserted.

In a general purpose computer running interactive software, hardware interrupt latency is important but is usually not a major issue, and is not alterable by ordinary means. However, in real-time applications, processing deadlines may make multiple cycle latency unacceptable. Furthermore, **jitter** that is a byproduct of variations in latency that occur from one interrupt to the next may adversely affect the performance of a system in which a high volume of interrupts must be serviced within strict time limits. This is where the WAI instruction gets interesting.

Consider the following code:

```
sei                 ;IRQs off
wai                 ;wait for interrupt
lda via001          ;start of interrupt handler
```

The above sequence disables IRQs with SEI and then stalls the microprocessor with WAI. WAI actually stops the 65C816's internal clock in the Ø2 high state, putting the microprocessor into a sort of catatonia, reducing its power consumption to micro-amperes and halting all processing (hardware note: executing WAI also causes the 65C816's bi-directional RDY pin to go low—knowing that is a clue to what is going on inside while the 65C816 is WAIting). The system will appear to have gone completely dead.

However, as soon as any hardware interrupt other than a reset occurs the microprocessor will restart and exactly one Ø2 cycle after the interrupt was received, the LDA VIA001 instruction will be executed. In other words, interrupt latency in this scenario will always equal exactly one Ø2 cycle —70 nanoseconds at the 65C816's maximum officially-rated Ø2 frequency of 14 MHz. Unlike the usual behavior when a hardware interrupt input is asserted, there is no delay while the current instruction finishes execution (there is no "current instruction" while WAIting) and the 65C816 performs no stack operations upon awakening.

This method of handling interrupts obviously isn't practical in a general purpose computer that has to process foreground tasks along with interrupts—all foreground processing will cease upon execution of WAI. It is, however, a technique that is eminently suited to any system where all processing is interrupt-driven, such as might be the case in a high speed data acquisition unit. This programming technique is also useful in specialized types of hardware, such as **implanted heart defibrillators**, in which long periods may elapse without activity and minimal power consumption is desired, but prompt response is required in the event of a "situation."

Note that the STP (**ST**o**P**) instruction could be used in place of WAI, as STP's internal effect on the microprocessor is essentially the same. However, the only interrupt input to which the microprocessor will respond following execution of STP is RESB, which means that single-cycle response isn't possible—three Ø2 cycles will elapse following the assertion and subsequent release of RESB before the 65C816 starts the actual reset sequence. Naturally, a reset will cause the 65C816 to switch back to emulation mode and execute the code pointed to by the reset vector.

Table of Contents

ADVANCED SOFTWARE INTERRUPT PROGRAMMING

One of the exciting possibilities with the 65C816 is that of being able to implement an execution environment that prevents user programs from affecting each other or an **operating system kernel**. When the 65C816's interrupt capabilities are combined with stack pointer relative addressing and suitable hardware logic, the development of a protected environment not unlike that of a commercial multiuser system can become a reality. While this section does not delve into operating system design or complex hardware logic (thick tomes regarding both subjects have already been written), it does discuss the basics of using a **kernel trap** to implement an operating system ***application programming interface*** or ***API*** on a 65C816 system.

This section gets into more esoteric concepts than previously presented, so please be sure you thoroughly understand what has already been discussed before proceeding.

API Calling Theory

Found at the core of almost all computer operating systems is the kernel, which is at the most basic level, a body of software responsible for mediating access to the computer's hardware. The kernel may also be responsible for scheduling processes, maintaining timekeeping, gathering statistics on system usage, and other activities. Some kernel functions are strictly for internal use, for example, servicing jiffy IRQs, while others are intended to be utilized by user programs to do such things as read and write files or get data from a keyboard. The formalized means by which a user program is given access to such kernel functions is the system's API. The code that transfers execution from a user program to a kernel function is often referred to as an ***API call***.

The theory behind providing a kernel API is a user program doesn't need to know how to, for example, access a disk drive or transmit a byte from the computer's serial port to a printer, as these are tasks that are handled by internal functions of the kernel. The program only needs to know how to call a kernel API that will handle the desired operation. In this way, the user program's design can concentrate on accomplishing the task it was intended to accomplish and avoid having to include the complex instructions needed to deal with the arcaneness of, say, interacting with a disk drive controller or driving a video display.

Over the years, various methods that implement API calls have been devised, the two most common being that of treating a kernel function as a conventional subroutine or treating a kernel function as a specialized form of an interrupt service routine.

In the former case, a **static jump table** provides access to the internal kernel functions. Perhaps the best known example of a kernel jump table is the one present in the Commodore 64's "kernal" ROM, with which any Commodore eight bit assembly language programmer will be familiar. User programs access kernel functions by treating them as subroutines and pass parameters via the microprocessor registers. Each kernel function has a unique entry point in the jump table, which as the "static" adjective implies, appears at a fixed address, with entries in an immutable order. The result is that assembly language programs that use only the jump table to access the kernel are portable to any eight bit Commodore computer in which the required kernel functions are present.

In the interrupt service routine method, APIs are called via a kernel trap, which is a machine-dependent code sequence that transfers execution from the user program to the kernel. Each API call is assigned an immutable index number that tells the kernel what code must be executed to complete the desired function. Along with the API index number, any parameters to be passed to the kernel are loaded into the microprocessor's registers and/or pushed to the hardware stack before the call. Any parameters returned by the API are likewise loaded in the registers and/or placed on the stack. Implied is that the microprocessor has a large number of general purpose

registers, has instructions to address the stack by means other than just pushes and pulls, or both capabilities.

Naturally, both API calling methods have their strong and weak points. Use of a jump table makes for simple user application programming and a generally less complicated kernel. Applications merely `JSR` to access the API and the kernel exits with `RTS`. The required kernel code can be very small and fast-executing, which was an important consideration in early home computers. However, once a system has been developed with a specific jump table layout, the design is essentially cast in concrete, even if future hardware and/or operating system revisions would be better served with a relocated kernel and/or rearranged jump table. The fact that applications must know where in memory the kernel is loaded and must be able to access that memory makes the kernel non-portable and if running in RAM instead of ROM, vulnerable to corrupting wild writes caused by program errors and/or malicious coding.

Calling APIs via a kernel trap offers the advantages of portability and isolation. User programs don't need to know specific addresses to access the kernel API—applications only need to know API index numbers. If a new kernel is released with a new API-accessible function, the lowest unused API index number is assigned to the new function, which will not affect any applications that were written prior to the kernel update. As a user-accessible jump table is not used for calling APIs, the kernel can be loaded anywhere in memory that is convenient.

Isolation offers the kernel some protection from misbehaving user applications, reducing the likelihood of random instructions or wild address pointers accidentally accessing and/or overwriting kernel space and causing system fatality. In most systems, a kernel trap causes a hardware context switch that may be used to modify the memory map, alter memory protection rules, and/or change instruction execution privileges, all of which can be used to tightly control what user programs can and cannot do.

The principal downsides to a kernel trap API calling method are greater code complexity, heavy stack usage and slower execution. As will be seen, a kernel trap API ultimately involves a software interrupt to switch execution from user mode to kernel mode. Therefore, code in both the API "front end" and "back end" is essentially a specialized form of an interrupt service routine, which adds complexity to the kernel. Also, since the API entry point is the same for all APIs, dispatch code is required to select the specific function that must be executed for a particular API, as well as determine how many parameters are expected by the API. That an API call culminates in an interrupt means that slower execution will occur due to hardware and software-induced latency.

Downsides notwithstanding, the flexibility and extensibility of a kernel trap API are features that are hard to ignore. Virtually all modern operating systems use this method to offer services to user programs.

Kernel Trap API Mechanics

Most API calls require that at least one parameter be passed to the kernel. The number and types of parameters that must be passed to an API will necessarily be dependent on what information is needed to implement the desired function. Use of the stack for parameter passing is common, primarily because the number of available general purpose registers may not be sufficient to handle all parameters in all cases. Therefore, prior to making an API call that requires parameters the calling function may have to generate a **stack frame**.

The term ***stack frame*** refers to a group of related parameters that are pushed to the stack in a defined order prior to the execution of a function. As the sequence of pushes and the sizes of the parameters are defined by the function being called, individual parameters are readily "cherry-picked" from the stack as needed to carry out the desired operation. The function may also modify one or more of the parameters to return data back to the calling function. The calling function could, in turn, modify the stack frame that was generated by its caller, and so forth, thus passing results back "up the line." Understanding the concept of a stack frame is essential and will be

expanded upon as discussion proceeds.

Turning to the mechanics of making a kernel trap API call, examining the assembly language code generated and linked by a language compiler sheds some light on how a stack frame and a kernel trap are used to invoke a kernel function. The way in which it is done with a Motorola 68000 microprocessor is a good example to follow in this regard, as that processor has some lineage to its eight bit counterpart (the MC6800) and thus indirectly to the 6502 family. As some of the first systems to make widespread use of the MC68000 ran the UNIX operating environment, a quick look at how a UNIX kernel API call would be coded in MC68000 assembly language can be instructive.

In the UNIX environment, where ANSI C is the predominant development language, the compiler outputs an intermediate assembly language program, and a linker generates the executable binary file containing the appropriate machine instructions that will perform the desired task. In sections of the binary where an API call is to be made, system-specific code in a **standard library** will be linked into and become part of the finished program. Generally speaking, standard library functions are assembly language subroutines that contain the instructions required to cause the kernel trap to occur.

Here's an example of how this would work on an MC68000-powered UNIX system, using a brief C program that creates and opens a file named `/usr/bdd/newfile` with `rw-rw-r--` permissions:

```
/* create & open a new file in ANSI C */

char fname[] = "/usr/bdd/newfile"; /* pathname */

int main() {
    int fd;                         /* file descriptor */
    fd = creat(fname,0664);         /* create & open file */
    return(fd);                     /* return file descriptor to caller */
}
```

`creat()` is a function in the standard C library that is a machine language interface to a UNIX kernel API, also named `creat`, that creates and opens a new file. The `creat` kernel API call requires that two parameters be passed to it: a pointer to the file's pathname, which is the variable `fname` in the C source code, and a file permissions mode value, which is the literal `0664` octal number. The `creat()` library code passes these values and an API index number to the kernel on behalf of the user program. If the `creat` API call is successful, a small positive integer value called a **file descriptor** will be returned in the variable `fd`. Alternatively, `fd` will return `-1` if `creat` fails for any reason. A separate variable called `errno` would be conditioned to describe the nature of the failure (error-handling code will be omitted for clarity).

Here's the MC68000 assembly language that the C compiler might generate for the above program on a UNIX machine running the System V kernel:

```
* machine code generated in main()...
*
         move #$01b4,(sp)       * push mode to stack
         move #$41d7,-(sp)      * push pathname pointer to stack
         jsr creat              * call creat API library code
*
*
* creat() kernel API call library machine code...
*
creat    moveq #$08,d0          * load register D0 with creat API index ($08)
         trap #$00              * invoke kernel API
         bcs _error_            * if error, branch w/error code in D0
*
         rts                    * file created & opened, file descriptor in D0
*
_error_  ...handle error processing...
```

As in 6502 assembly language, $ indicates a hexadecimal value and # means the operand is the data (immediate mode addressing). Comments are started with *.

The switch from user mode to kernel mode occurs in the creat() library subroutine, where register D0, one of the MC68000's general purpose registers, is loaded with the eight bit API index number for the creat API call. Next, a TRAP instruction causes execution to be transferred to the kernel.

However, before the library subroutine is called, main() generates a stack frame with two parameters: the pathname pointer, $41D7 (the pointer is a made-up value), and the new file's permissions mode value $01B4, equal to the octal constant 0664 in the C source code. In both instructions, SP refers to the user stack pointer. Within the creat part of the kernel, code will read the user stack parameters and act upon them in various ways, the details of which the user program need not know. Immediately prior to exit, the kernel will load the file descriptor into register D0 if the call was successful, that is, if the file was created and opened, and clear the carry bit in the MC68000's condition code register (CCR) to indicate that the file was created and opened. If carry were set in the CCR it would mean that D0 contains an error code instead of a file descriptor. As an aside, in the UNIX environment it is the caller, main() in this case, that has to clean up the stack following a function or API call.

In the MC68000, executing TRAP causes a software interrupt and a hardware context change from user mode to supervisor (kernel) mode, the latter action having important implications in a multitasking environment like UNIX. The context change aspect of TRAP will be ignored here, as the 65C816 hardware has no such behavior. However, it is worth noting that in other respects, the MC68000's software interrupt behavior is very similar to that of the 65C816. In both microprocessors, an internal interrupt-like sequence will occur. Also, the MC68000 will jump through a defined vector when TRAP is executed, just as the 65C816 will jump through a defined vector when it executes BRK or COP.

6502 Software Interrupt API

Implementing a kernel trap API on 6502 or 65C02 hardware will unavoidably involve the use of the BRK instruction, as it is the lone software interrupt in the instruction set. Due to the use of BRK, as well as general microprocessor limitations, three significant programming problems must be considered:

1. The same hardware vector ($FFFE-$FFFF) is used by BRK and IRQ, which means one interrupt type must be distinguished from the other in software. This unavoidable step increases execution time and usually "clobbers" two registers (.A and .X).

2. The 6502's registers can only handle eight bit values, which substantially complicates the passing of more than one 16 bit value to the API, a common procedure required with many I/O operations.

3. There are no 6502 instructions that facilitate the use of the stack for parameter passing or temporary indexed storage. As stack frame elements cannot be directly addressed by using the stack pointer as the relative index, considerable code may be required to implement indexed storage and retrieval.

The 65C816's enhanced stack addressing capabilities, 16 bit registers and separately vectored software interrupts when operating in native mode completely circumvent all of the above problems. Consequently, implementing a kernel trap API with the 65C816 is possible with relatively succinct code.

Unlike the MC68000 and other microprocessors that usually support preemptive multitasking environments, the 65C816 has no wired-in means of differentiating between "user mode" and "kernel mode" when interrupted and thus in itself cannot support any kind of protected environment. However, the 65C816 does provide an output signal (VPB) that could be harnessed in

conjunction with complex logic to simulate user and kernel modes. How to go about doing so is well outside the scope of this article, but should be food for future thought.

65C816 Kernel Trap API Call Model

Although the MC68000 is a more sophisticated microprocessor than the 65C816 and has a more complex behavior when executing a software interrupt instruction, an analog to the earlier UNIX API call procedure can be modeled in 65C816 assembly language without much difficulty. In fact, the principles are virtually identical; only the methodology and machine instructions differ. In all of the following code, it will be assumed that the 65C816's stack pointer has been initialized to **$CFFF** prior to any calls being made. If an instruction affects the stack pointer the new SP value following the execution of that instruction will be noted in **boldface red**.

Here is the 65C816 analog of the above API code:

```
;machine code generated in main()...
;
         pea #$01b4            ;push file mode to stack       $CFFD
         pea #$41d7            ;push pathname pointer         $CFFB
         jsr creat             ;call creat() library function $CFF9
;
;
;creat() kernel API call library machine code (SP = $CFF9)...
;
creat    sep #%00100000        ;select 8 bit accumulator
         lda #$08              ;create() API index
         cop $00               ;transfer execution to kernel  $CFF5
         bcs _error_           ;kernel API returned an error
;
         rts                   ;file created & opened         $CFFF
;
_error_  ...error processing...
```

The PEA instruction, which incidentally also exists in similar form in the MC68000, pushes its 16 bit operand to the stack. Despite the mnemonic's purported meaning, the operand can be anything that is known at assembly time, or could be altered at run time via self-modifying code. In this example, both the pathname pointer (address) and file mode have been statically assembled into the program and pushed, mode first and then the pathname address, the same order as shown in the MC68000 API call. The same procedure could be achieved by loading a 16 bit register and pushing it—the choice is implementation-dependent. Ultimately, all that has to be accomplished is placing parameters of the correct size on the stack in the correct order.

The 65C816's COP instruction stands in for the MC68000's TRAP instruction, with the eight bit API index number loaded into .A. The rationale behind using COP instead of BRK is that the latter instruction is traditionally associated with setting debugging breakpoints in programs, and in our opinion, its use should be limited to that purpose. On the other hand, COP is intended to be used to change operating context in some undetermined way (recall that the instruction means **CO**processor), so its use as a kernel trap instruction is more appropriate. Also, although our focus is on native mode operation, the 65C816 has a unique vector for COP even when operating in emulation mode.

While it is possible to use COP's signature byte as the API index, doing so isn't a straightforward process, and in our opinion, there is no good reason for the effort. Utilizing the stack for parameter passing leaves the registers unencumbered, therefore loading the API index into .A is quick and efficient. If the signature were to be used as the API index, additional code would have to compute its location in RAM and fetch it. More significantly, complications would arise unless the bank in DB was the bank in which the user program that called the API is running. Bits 16-23 of the 24 bit address that will be propagated to the hardware when the 65C816 loads the signature will be determined by what is in DB, not what was in PB before COP was executed. Hence the 65C816 could conceivably load a garbage byte, not the signature. Circumventing this possibility would require

that the value of PB that was pushed when COP was executed be copied from the stack and temporarily loaded into DB to assure that the correct bank is being accessed. Following the signature byte retrieval, DB would have to be restored so that parameter fetching would be from the correct location. That is a lot of hoop-jumping just to get one byte from RAM.

When the above code has executed the resulting "stack picture" following the COP software interrupt will be:

Stack Index	Absolute Stack Address	Data	Data Description
SP+$0A	$00CFFF	$01	file creation mode MSB
SP+$09	$00CFFE	$B4	file creation mode LSB
SP+$08	$00CFFD	$41	pathname pointer MSB
SP+$07	$00CFFC	$D7	pathname pointer LSB
SP+$05	$00CFFA	????	library RTS address
SP+$04	$00CFF9	??	PB
SP+$02	$00CFF7	????	PC
SP+$01	$00CFF6	??	SR

Data entries marked with ?? or ???? will vary during program execution.

The stack picture is something that we will refer to a number of times, as it gives insight on how to write kernel trap API front and back end code. First, the front end that will be invoked when COP is executed:

```
;KERNEL API FRONT END — EXECUTED IN RESPONSE TO A COP INSTRUCTION
;
;    ——————————————————————————————————————————————————————————
;    .A must be loaded with the 8 bit API index prior to executing COP.
;    ——————————————————————————————————————————————————————————
;
icop     rep #%00110000        ;16 bit registers
         pha                   ;save .A for return access     $CFF3
         phx                   ;preserve .X &...              $CFF1
         phy                   ;.Y if necessary               $CFEF
         cli                   ;restart IRQs
         and #$00FF            ;mask garbage in .B (16 bit mask)
         beq icop01            ;API index cannot be zero†
;
         dec a                 ;zero-align API index
         cmp #maxapi           ;index in range (16 bit comparison)?
         bcs icop01            ;no, error†
;
         asl a                 ;double API index for...
         tax                   ;API dispatch table offset
         sta apioff            ;save offset &...
         jmp (apidptab,x)      ;run appropriate code
;
;
;    invalid API index error processing...
;
icop01   ...handle invalid API index...
```

† Although system-dependent, a typical UNIX kernel reaction to an invalid API index is a core

dump, followed by forcible process termination. The only likely cause of an invalid index is a bug in the standard library code that was linked into the executable binary.

After pushing the registers, the stack picture will be as follows:

Stack Index	Absolute Stack Address	Data	Data Description
SP+$10	$00CFFF	$01	file creation mode MSB
SP+$0F	$00CFFE	$B4	file creation mode LSB
SP+$0E	$00CFFD	$41	pathname pointer MSB
SP+$0D	$00CFFC	$D7	pathname pointer LSB
SP+$0B	$00CFFA	????	library RTS address
SP+$0A	$00CFF9	??	PB
SP+$08	$00CFF7	????	PC
SP+$07	$00CFF6	??	SR
SP+$05	$00CFF4	$??08	.C
SP+$03	$00CFF2	????	.X
SP+$01	$00CFF0	????	.Y

Note the following:

- `.C` is pushed so a return value can be passed back to the calling function by overwriting the stack copy. The nature of the return value, which may be data or an error code, would be determined by the particular API that was invoked.

- Similarly, `.X` and `.Y` are pushed so they may be preserved or modified for return to the calling function. The need to do so would be implementation-dependent.

- Recall that IRQs are disabled when the 65C816 executes a software interrupt instruction. Hence interrupts must be re-enabled as soon as practical so the system doesn't inadvertently go into deadlock.

- After the API front end has pushed the registers, the above stack picture can be defined as three frames, the user stack frame, which starts at `SP+$0D`, the library stack frame, which starts at `SP+$0B` and the register stack frame, which starts at `SP+$01`. The user stack frame contains four bytes in total, the library stack frame contains two bytes and the register stack frame contains ten bytes. For programming convenience, each stack frame can be symbolically represented as follows:

```
;    register stack frame...
;
reg_y    =1                  ;16 bit .Y
reg_x    =reg_y+2            ;16 bit .X
reg_a    =reg_x+2            ;16 bit .A
reg_sr   =reg_a+2            ;8 bit SR
reg_pc   =reg_sr+1           ;16 bit PC
reg_pb   =reg_pc+2           ;8 bit PB
s_regsf  =reg_pb+1-reg_y     ;register stack frame size in bytes
;
;
```

```
;    library stack frame...
;
lib_rts  =reg_pb+1             ;library RTS address
s_libsf  =lib_rts+2-lib_rts    ;library stack frame size in bytes
;
;
;    user stack frame...
;
fmode    =lib_rts+2            ;file creation mode
pnptr    =fmode+2              ;pathname pointer
```

Note that the register and library stack frame definitions include an assembly-time value (`s_regsf` and `s_libsf`, respectively) that defines the size of each frame in bytes, which is practical because these sizes are fixed. The size of the user stack frame will vary according to the API being called.

Creating definitions in this fashion makes it easier to symbolically reference any stack frame element without having to know the specific offset, thus eliminating a potential source of program errors. Also, these definitions simplify the process of realigning the stack when the API returns to the caller, as will soon become evident.

- Prior to use, the API index number is masked to prevent the content of `.B` from affecting the following instructions. After masking, the index is tested for range (API index `$00` is usually deemed to be illegal for a user API call) and if the range is acceptable, the index is doubled to create the `apidptab` dispatch jump table offset. This API dispatch method supports a maximum of 255 callable API functions—more could be supported by passing a 16 bit index.

- Another look-up table, `sparmtab`, is consulted to find out how many user stack frame bytes are expected by each API function. The use of this table is not demonstrated in the above code but will be demonstrated in the next series of code fragments.

Post-API Processing

After the API code has completed its task and has placed return values on the stack, processing can be switched back to user mode. Prior to doing so, arrangements must be made to take care of stack housekeeping. Otherwise, the stack will be out of balance and when `RTI` executes to return control to the `creat()` library code, the 65C816 will pull an incorrect address from the stack, surely resulting in a major malfunction.

Stack housekeeping consists of three steps:

1. Disposing of temporary workspace that was created within the called API function; handled within the function, since only it would "know" how much workspace was used.

2. Disposing of the register stack frame; handled within the API back end code.

3. Disposing of the user stack frame; handled by the user function that called the API or handled within the API back end code.

Stack cleanup is a relatively painless process with the 65C816. In this subsection, we will demonstrate how to perform stack cleanup in the kernel API back end code, clearing the user stack frame as well as the register stack frame. We are not advocating that user stack frame disposal occur within the kernel—use what is best for your application.

In general, the stack housekeeping process is one of shifting the register and library stack frames up the stack by the total number of bytes in the user stack frame, and then adjusting the stack pointer so it points at the location immediately below the relocated register stack frame. With that done, pulling the 16 bit registers will dispose of most of the register stack frame, hence incrementing the stack pointer until it is pointing one byte below the stack copy of the status

register. Upon execution of RTI at the end of the procedure the 65C816 will pull SR, PC and PB in that order, disposing of the remainder of the register stack frame, exiting the kernel and resuming execution at the bcs _error_ instruction in the creat() library code. When the creat() library code finally executes RTS to return to the calling function, SP will again be $CFFF, which is where it started before main() called creat().

The most convenient way to shift stack frames is by using one of the 65C816's block copy instructions. As the stack grows toward lower addresses, the shift is upward in memory and some overlap is likely to occur, which means use of the MVP instruction is the correct choice for this procedure. In the following code, 16 bit operations are used throughout and a sneaky little trick will be used to set the stack pointer to the correct location following the register and library stack frames shift. To assist you in understanding what is going on, the code will be interspersed with explanatory text:

```
rep #%00110000        ;select 16 bit registers
clc
tsc                   ;get SP (currently $CFEF)
adc #s_regsf+s_libsf  ;add bytes in register & library stack frames
tax                   ;now is "from" address for stack frame shift
```

Following the above steps, .C contains $CFFB, since the register stack frame occupies ten bytes and the library stack frame occupies two. $00CFFB is the absolute address where the library RTS address MSB was stored when it was pushed by the jsr creat instruction in main(). It is necessary to compute this address because the MVP instruction works "backwards" to lower memory. Therefore, MVP has to start at the highest address from which bytes are to be copied, which would be that of the library RTS address MSB. As the MVP instruction treats the value in .X as the copy source address, .C is transferred to .X.

Continuing:

```
ldy apioff            ;API dispatch offset
adc sparmtab,y        ;add bytes in user stack frame
tay                   ;now is "to" address for stack frame shift
```

Now, .C contains $CFFF, since the user stack frame occupied four bytes, information that was gotten from the sparmtab parameter size look-up table. $00CFFF will be the absolute address of the library RTS address MSB after the register and library stack frames have been shifted, and is currently the address that is occupied by the MSB of the file creation mode parameter that was pushed by the calling function. As the MVP instruction treats the value in .Y as the copy destination address, .C is transferred to .Y. Again, keep in mind that MVP copies in reverse.

Continuing:

```
lda #s_regsf+s_libsf-1
mvp 0,0               ;shift stack frames
```

MVP uses .C as a down-counter to keep track of the number of bytes copied. Copying stops when .C has been decremented below zero. Therefore, the count that must loaded into .C is the size of the register stack frame plus the size of the library stack frame ***minus one***. Also, copying must occur in bank $00 because that is where all stack references are directed. Hence zero is hard-coded for the two MVP operands.

When MVP has finished, the registers will be as follows:

.C = $FFFF
.X = $CFEF
.Y = $CFF3

and the stack picture will now be:

Stack Index	Absolute Stack Address	Data Description
SP+$0F	$00CFFE	library RTS address
SP+$0E	$00CFFD	PB
SP+$0C	$00CFFB	PC
SP+$0B	$00CFFA	SR
SP+$09	$00CFF8	.C
SP+$07	$00CFF6	.X
SP+$05	$00CFF4	.Y

You may well be wondering how the stack index for .Y ended up being $05—it previously was $01. The register and library stack frames were shifted upward by the size of the user stack frame, which was $04 bytes. However, SP was not disturbed by any preceding instructions and thus is still $CFEF. Therefore, $01+$04=$05 and $CFF4-$05=$CFEF.

With the stack frame shifting out of the way, all that's left in the housekeeping process is to adjust SP. Recall above where we said "...a sneaky little trick is used..."? Take a good look at the ending values in the registers after MVP finished, think about what was going on inside the microprocessor as it was copying (read page 40 of the data sheet if you're not familiar with how MVP works) and then see if you can figure out why the next two instructions correctly set SP (no fair peeking at the following explanation):

```
tya                 ;adjust...
tcs                 ;stack pointer              $CFF3
```

It may not be immediately obvious why this even works. After all, no one ever uses .Y to set a stack pointer, right? Well, here is an exception!

Consider that as MVP executes the microprocessor repeatedly copies a byte and then decrements all three registers. Hence when MVP has finished, .C will be $FFFF, .X will be pointing to a location one byte below the address where the 65C816 got the final byte and .Y will be pointing to a location one byte below the address where the 65C816 put that final byte. As the register and library stack frames have been relocated higher on the stack by the number of bytes in the user stack frame, the final address in .Y is now the first unused location on the stack, ***which is by definition the address to which the stack pointer points***. So, adjusting the stack pointer merely involves copying whatever is in .Y to SP!

The final steps are to restore the registers and then exit to the creat() library code:

```
ply                 ;restore registers          $CFF5
plx                                             $CFF6
pla                                             $CFF9
rti                 ;exit to creat() library code  $CFFD
```

As RTI causes the microprocessor to pull SR from the stack, any changes that were made to the stack copy of SR, such as setting the carry bit to flag an error, will immediately take effect and the creat() library code can act upon them. Similarly, if the stack copies of any of the registers were altered, those changes will be propagated back to the creat() library code as well.

Accessing Stack Frame Elements

Nothing in this subsection has anything to do with interrupt processing *per se*. However, everything that has preceded has made frequent reference to the stack. So in the spirit of expanding your knowledge about the 65C816, consider this to be a bonus section.

As the API handler code executes it will need to be able to access both the register and user stack frame elements, the former to write values that will be returned to the caller, and the latter to read the parameters that were pushed by the user application. The 65C816's stack addressing instructions greatly simplify the process, as they index relative to the current stack pointer, eliminating the need for tedious address calculations.

First, a recapitulation of the stack frame definitions:

```
;    register stack frame...
;
reg_y    =1                    ;16 bit .Y
reg_x    =reg_y+2              ;16 bit .X
reg_a    =reg_x+2              ;16 bit .A
reg_sr   =reg_a+2              ;8 bit SR
reg_pc   =reg_sr+1             ;16 bit PC
reg_pb   =reg_pc+2             ;8 bit PB
s_regsf  =reg_pb+1-reg_y       ;register stack frame size in bytes
;
;
;    library stack frame...
;
lib_rts  =reg_pb+1             ;library RTS address
s_libsf  =lib_rts+2-lib_rts    ;library stack frame size in bytes
;
;
;    user stack frame...
;
fmode    =lib_rts+2            ;file creation mode
pnptr    =fmode+2              ;pathname pointer
```

Using the above definitions, here are some examples of how to read and write stack frame elements.

First, read the file creation mode from the user stack frame:

```
         rep #%00100000        ;16 bit accumulator
         lda fmode,s           ;get mode
```

Note the use of the `fmode` stack frame definition and `,S` (stack pointer relative) addressing. Assuming that `SP` hasn't changed since the API entry point, the `fmode,S` operand is interpreted by the microprocessor to mean `$CFEF+$0F` or `$CFFE`, since `fmode=$0F` and `SP=$CFEF`. Therefore, the instruction is effectively `LDA $CFFE`.

Next, copy the pathname to a buffer. The pathname is a character string of arbitrary length that has been terminated by a null byte (`$00`):

```
         sep #%00100000        ;select 8 bit accumulator
         rep #%00010001        ;select 16 bit index & clear carry
         ldy #0                ;pathname index (16 bit load)
;
.0000010 lda (pnptr,s),y       ;get pathname byte-by-byte &...
         sta buffer,y          ;store in work buffer
         beq .0000020          ;done
;
         iny
         cpy #PATH_MAX         ;check pathname length
         bcc .0000010          ;okay so far
;
```

```
            lda #ETOOLONG       ;pathname too long: error
            bra error           ;goto error handler
    ;
    .0000020 ...program continues...
```

Here, use is made of stack pointer relative indirect addressing to copy the pathname from user space to the work buffer. Again, the `pnptr,S` operand is translated by the microprocessor at run-time to `$CFEF+$0D`, effectively making the instruction `LDA ($CFFC),Y`, although such an addressing mode doesn't exist in reality. Note that a check is made for an excessively long pathname, the maximum permissible length being defined by `PATH_MAX`. Labels such as `.0000010` and `.0000020` are local labels—use whatever syntax is implemented in your assembler.

Return the open file descriptor to the calling function via the eight bit accumulator by overwriting the appropriate register stack frame element:

```
            sep #%00100000      ;select 8 bit accumulator
            lda #0              ;clear...
            xba                 ;.B
            lda filedes         ;get file descriptor, ...
            rep #%00100000      ;select 16 bit accumulator &...
            sta reg_a,s         ;overwrite .C's stack copy
```

When the accumulator is pulled it will contain the value that was in `filedes`.

Flag an error by setting the carry bit in `SR`:

```
            sep #%00100000      ;select 8 bit accumulator
            lda reg_sr,s        ;stack copy of SR
            ora #%00000001      ;set carry bit &...
            sta reg_sr,s        ;rewrite
```

Flag a successful operation by clearing the carry bit in `SR`:

```
            sep #%00100000      ;select 8 bit accumulator
            lda reg_sr,s        ;stack copy of SR
            and #%11111110      ;clear carry bit &...
            sta reg_sr,s        ;rewrite
```

The above code snippets should give you a basis on which to expand your programming activities.

Table of Contents

CONCLUSION

It is hoped this article has been of value to you as you explore the capabilities of the 65C816 microprocessor. While every effort has been made to assure accuracy, errors may have crept in during the editing process. Suing us over any such errors will be a complete waste of your time, so don't bother trying. Also, if you encounter any instances of dubious grammar or sloppy spelling, we profusely apologize and ask that you consider that we are computer geeks, not English professors. Please contact BCS Technology Limited to report any errors and/or omissions, or to suggest edits.

2013/11/01 — BDD (revised 2014/05/29)

The POC W65C816S Single-Board Computer Website
