Modifying a process *File Descriptor Table* (*FDT*) at runtime

T. Castillo Girona, F. Verdugo Arias & J. Hornos Arias

<toni.castillo@fa.upc.edu>
<fernando.verdugo@upc.edu>
<josep.hornos@uab.es>

Abstract

As long as a certain process p *is running, there's no possibility of altering its* FDT *externally. Our project tries to solve this issue using GNU/Linux Kernel aids, developing a Linux Kernel Module (LKM) which can alter any process* FDT *at runtime, with no need of killing the process nor modifying its source code. This way, our LKM can insert a new File Descriptor, say* fd'*, inside the process* FDT*, exchanging* <FDT[fd],FDT[fd']> *whenever a non-root user decides.*

1 Introduction

A number of functions commonly used to manage files can be easily found simply by browsing the Linux Kernel sources. These functions were originally designed to allow each process to only manage its own opened files. That is, as long as a certain process *p* is running, a different process *p'* will not be able to access the files opened by *p*.

All I/O functions directly related with the major part of kernel tasks involving system calls were originally written not using `task_struct *` parameter. As a result, an updated set of C functions should be written and added to the Linux Kernel mainstream.

So we will develop an *LKM (Linux Kernel Module)* containing these C functions. This article is split into two main sections: the first one explains how the Linux Kernel deals with the *File Descriptor Table*, and the second one shows how *MODEST* has been designed and what it can do.

2 MODEST's design

The design of *MODEST* is based on the classic Linux device driver development techniques. In order to avoid using this LKM directly as a root user (through *insmod/modprobe/rmmod*), we decided to write a char device driver [3]. This way as far as the user is concerned, there's no need to deal directly with modutils.

Thus, *MODEST* has been developed using Kernel's API facilities and, at the same time, adding new functions to improve some aspects of *FDT* management.

MODEST components are:

- An LKM, *kmodest.ko*, which is in charge of doing the most difficult tasks over external processes.
- A user-land utility, *umodest*, which interacts with users and *kmodest.ko*.
- Finally, a char device driver on `/dev/modest` - or `/dev/.static/dev/modest` -, depending

on whether the UDEV [1] subsystem is present or not.

3 task_struct* and kernel spinlocks

First of all, we must consider two important aspects of the Linux Kernel mechanisms. This way we can understand how *FDT* is managed and, simultaneously, how *MODEST* has been designed: what is `task_struct` * and why kernel needs spinlocks.

3.1 `task_struct *` in a nutshell

None of the C functions implemented in the Linux Kernel use a parameter to choose which process will be affected by their actions. In other words, each function call only affects the current process, defined in the `<include/asm-i386/current.h>`.

So it is not feasible to pass a parameter that will allow the choice of a given process. In order to solve this problem we will implement a new set of C functions in the frame of *MODEST* source. The right parameter to transfer to each C function is, precisely, the `task_struct *` pointer. We will develop this matter in section *Dealing with the FDT*.

3.2 Kernel control-paths and spinlocks

Kernel Control-Paths are a group of actions performed from one initial point called *a* until an endpoint placed at *b* on an immediate future [2]. So, all tasks occurring at the time interval *[a,b]* must be protected from illegal access, race-conditions and dead-locks. *Spinlocks* have been implemented by the Linux Kernel in order to achieve that.

[1] Location of special device file modest can be set explicitly editing `umodest.h` header file.

Spinlock is defined in the `<include/linux/spinlock_types.h>` header file. A process which attempts to acquire access to a shared critical data structure calls `spin_lock(&struct->spinlock_var);` on time `t`. If the lock is open, the process acquires it and continues its execution until calling `spin_unlock(&struct->spinlock_var);`, on time *t'*.

Any process which is trying to access the same data structure on *(t,t']*, will "spin" around executing a tight instruction loop.

4 Dealing with the *FDT*

First of all, we must have a clear picture about how the Linux Kernel deals with the *FDT*. We have to study a few C functions and data structures involved with it. Few of these functions are written as exported symbols using the `EXPORT_SYMBOL();` macro.

4.1 The `task_struct` structure

Each process on the system is represented by the Linux Process Descriptor, a C data structure defined in the `<include/linux/sched.h>` header file. This way the Kernel can have a clear picture about what each process is doing. This C data structure has a lot of fields but only a few are interesting for our purposes. *MODEST* will be able to alter the *FDT* of any process previously chosen, so we only have to study fields directly related to the *File Descriptor Table*. These fields are shown in Figure 2.

- `long state;`: The process state is very important to determine when it's running, sleeping, or waiting for a I/O call to complete.
- `struct files_struct *files;`: This is the most significant part for *MODEST*

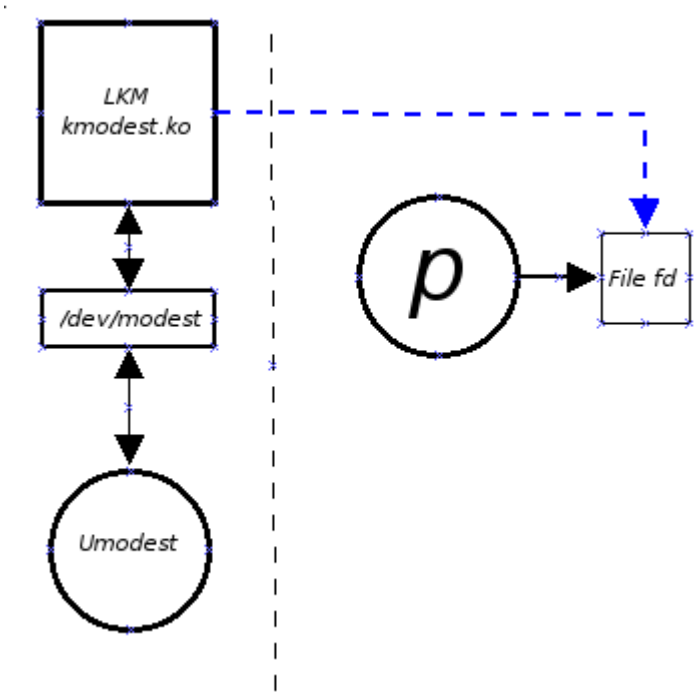

Figure 1: *MODEST*'s components relationship

implementation. It gives us access to all opened files information for any process.

4.2 The *File Descriptor Table*

It's defined in the <include/linux/file.h> header file, and it's precisely the most important data structure for the *MODEST* module. A brief list of the most important fields for our discussion are shown in Figure 3.

- `atomic_t count;`: Usually, each data structure present inside the kernel needs some kind of logical barriers to avoid concurrency problems. In order to accomplish this, the kernel implements atomic variables, defined in the <include/asm-i386/atomic.h> header file. In this case, it is feasible to share this data structure between a group of processes. Thus, this variable acts as a basic access counter. For example, if this variable stores the integer value of 7, 7 processes will share this data structure. Atomic variables are executed with no possibilities to suspend their execution until they end. In other words, there are no concurrency related problems. On X86 architectures, a disassembled C code instruction looks like [6]:

```
...
lock;
<mnemonic X86 instruction here>
...
```

- `struct file *fd_array[NR_OPEN_DEFAULT]`: This pointer is, in fact, the *File Descriptor Table*. Here, we find an array of `struct file*` typedefs, each of them pointing to a valid opened file by the process or NULL, defined in the <include/linux/fs.h> header file.

```
struct task_struct {
        /* -1 unrunnable, 0 runnable, >0 stopped */
        volatile long state;
        ...
        /* open file information */
        struct files_struct *files;
        ...
};
```

Figure 2: A summarized task_struct fields's view.

As shown in Figure 3, it seems that a process can have a maximum of `NR_OPEN_DEFAULT` files opened simultaneously. We agree up to appoint.
When a process calls the `open();` system call in order to open - or create - a file, the kernel does a lot of tasks to accommodate it into its *FDT*. There are far many of these kinds of tasks to mention in this article, so we won't go into too much detail. However, we can look at how the kernel uses the `next_fd` variable.
Each system call involved with the action of opening a file adds a new file pointer (represented by the `struct file*`) inside fd_array array, at the location pointed by `next_fd`. After this, the kernel increments this variable by one.
To conclude, each process has three opened files inside its *FDT*: stdin, stdout and stderr. These files are placed at: `fd_array[0]`, `fd_array[1]` and `fd_array[2]`.

4.3 Adding a new fd into the*FDT*

Each action over the *FDT* is made by a logical group of C functions whose parameters contain, at least, a `fdtable` pointer, defined in the <`include/linux/file.h`> header file. In order to add a new file descriptor inside this table, the Kernel calls `fd_install();` function, implemented in the <`fs/open.c`> file, as seen in Figure 4

It's very easy to understand what the Kernel does reading fd_install():

- First of all, the kernel gets an opened-files reference for the running process:
  `struct files_struct *files = current->files;`

- Then, it ensures that race-conditions are avoided by putting a lock over this structure:
  `spin_lock(&files->file_lock);`

- To conclude, the kernel gets a pointer reference to the process *FDT* table, assigns a new fd pointer to it and releases the lock obtained before:
  ```
  fdt = files_fdtable(files);
  rcu_assign_pointer(fdt->fd[fd],
  file);
  spin_unlock(&files->file_lock);
  ```

As a result, this new file descriptor is ready to use. *MODEST* implements its own `fd_install();` C function.

In addition, *fd* value is obtained after calling `get_unused_fd();` function which seeks the next available fd number and returns it,

```
struct files_struct {
       /*
       * read mostly part
       */
       atomic_t count;
       struct fdtable *fdt;
       ...
       /*
       * written part on a separate cache line in SMP
       */
       spinlock_t file_lock ___cacheline_aligned_in_smp;
       int next_fd;
       ...
       struct file * fd_array[NR_OPEN_DEFAULT];
};
```

Figure 3: A summarized files_struct's view.

incrementing the `next_fd` variable, introduced earlier in this section, by one.

4.4 Removing fd from the *FDT*

When any process closes a previous opened file, the Kernel has to free the file descriptor number - and, in fact, all referenced pointers associated with it -while simultaneously updating the next_fd variable accordingly. These tasks are done thanks to the `sys_close();` C function, implemented in the `fs/open.c` file. Basically, this function is the kernel counter-part of system call `close();` defined in the libc6 wrapper libraries.

Let's take a look at it:

- First of all, the Kernel unsets the reference pointed by *fd*, updating the table accordingly.
- Then, it updates the `next_fd` value by calling the `__put_unused_fd();` function, setting this file descriptor number as the new available file descriptor number to use on a future `fd_install();` function call:

```
...

if (fd < files->next_fd)
   files->next_fd = fd;
   ...
```

MODEST implements its own C function in order to uninstall a previous initialized file descriptor.

5 Syscalls

The Linux kernels prior to 2.6.X versions had only one method for entering inside *Ring 0* Intel processor execution mode: interrupt vector at offset `0x80`. Newer versions can accomplish this either by using 'new' Intel `sysenter` opcode - introduced by Pentium II+ processors - or interrupt vector *IDT*.

MODEST only has to catch any write over a given file descriptor number, so just we need

```
void fastcall fd_install(unsigned int fd, struct file * file){
     struct files_struct *files = current->files;
     struct fdtable *fdt;
     spin_lock(&files->file_lock);
     fdt = files_fdtable(files);
     BUG_ON(fdt->fd[fd] != NULL);
     rcu_assign_pointer(fdt->fd[fd], file);
     spin_unlock(&files->file_lock);
}
```

Figure 4: fd_install(); C function implemented by the Linux Kernel

```
asmlinkage long sys_close(unsigned int fd){
           ...
           rcu_assign_pointer(fdt->fd[fd], NULL);
           ...
           __put_unused_fd(files, fd);
           ...
}
```

Figure 5: sys_close(); syscall function implemented by the Linux Kernel

a tiny assembler code just inserted on *IDT* at offset `0x80`. This can be executed before the real operating system entry-point.

5.1 The syscall `sys_write();` by Interrupt Handler

Let's take a look at C code written for the fake process for testing purposes, showed in Figure 6.

At each iteration, fakeproc writes a pseudo-random integer value and puts into OUTPUTFILE on disk. The syscall `write();`, on Linux boxes, is implemented as a wrapper routine placed inside libc6 libraries. Depending on whether the binary is linked statically or dynamically, this function can be placed inside the text segment on the executable file.

For example, in the case of static linked binaries, such as fakeproc.c in our discussion:

```
# gcc -static fakeproc.c -o fakeproc
```

the complete libc6 `write()` routine will be written inside fakeproc's ELF text segment. In order to check that, we can disassemble fakeproc's text segment entirely typing:

```
# objdump -d ./fakeproc > fakeproc.S
```

As a result of the previous command, we'll have the complete assembler code written in fakeproc.S file. Reading quickly inside it, we'll find something like:

```
80482dc:  e8 bf bb 00 00 call 8053ea0
<__libc_write>
```

```
 ...
/* Open file for writting :  */
fd=open(_OUTPUTFILE_, O_WRONLY|O_APPEND|O_CREAT|O_SYNC, S_IRWXU);
if(fd==-1){ perror("Calling open()"); return errno; }



/* Iterate each second and write to file */
while(1){
     sprintf(buffer,"Entry:%d\n",rand_r(&pi));
     bytes_written=write(fd,buffer,strlen(buffer));
     /* Wait for a second */
     sleep(1);
} return 0;


...
```

Figure 6: C fakeproc code sample for sys_write() testing purposes

Clearly, at offset `80482dc` fakeproc calls `__libc6_write` symbol, located at offset `8053ea0`. We must examine this exact location in order to analyze its contents, as shown in Figure 7.

At `0x8053eaa`, `0x8053eae` and `0x8053eb2` offsets, mov instructions just copy the three parameters transfered to the `write();` system call getting their values from user-stack: `int fd` is stored at `ebx`, `char * buffer` at `ecx` and, finally, `size_t len` at `edx` register.

Before calling the *IDT* handler located at offset `0x80`, the wrapper stores the syscall number to call at `eax` register. In this case, the number 4 as defined in the `<include/asm-i386/unistd.h>` header file:

```
#define __NR_write 4
```

Finally, at offset `0x8053ebb` the instruction `int $0x80` calls the operating system entry-point implemented in `arch/i386/kernel/entry.S` assembler code file, `system_call();`, which finally executes the desired system call.

Thus, *MODEST* implements its own system_call entry-point that runs just before the real one. It does this in order to do a few tasks related to *MODEST*'s functionalities and, then, returns control to Linux's original system handler.

To conclude, *MODEST* only examines binaries linked in static way, and pays no attention to dynamic ones.

6 *MODEST*'s implementation

As long as a certain file descriptor called *fd* has been opened by process *p*, all subsequent `sys_write();` calls can be:

- ... redirected to another file descriptor called *fd'*.
  or...
- ... connected to another file descriptor called *fd'*.

```
 08053ea0 <__libc_write>:
8053ea0:  83 3d f8 17 0b 08 00 cmpl $0x0,0x80b17f8
8053ea7:  75 21 jne 8053eca <__libc_write+0x2a>
8053ea9:  53 push %ebx

8053eaa:  8b 54 24 10 mov 0x10(%esp),%edx
8053eae:  8b 4c 24 0c mov 0xc(%esp),%ecx
8053eb2:  8b 5c 24 08 mov 0x8(%esp),%ebx
8053eb6:  b8 04 00 00 00 mov $0x4,%eax

8053ebb:  cd 80 int $0x80
```

Figure 7: libc6 wrapper dissassembled write(); call

MODEST can do these tasks, no matter what kind of process is running on the system, in three steps:

- `kmodest.ko` pauses the selected process before manipulating its *FDT*.
- `kmodest.ko` inserts the new file descriptor *fd'* at `fd_array[next_fd]` slot.
- Finally, depending on what umodest command was sent to `kmodest.ko`, it redirects *fd* to *fd'* simultaneously or not.

We will discuss these steps in the next sections.

6.1 Pausing the process

Kmodest has to pause [2] the process before altering its *FDT* to avoid *Kernel OOPS* and inconsistency related problems. To do this, the LKM sends the signal `SIGSTOP` using the `sys_kill` system call. As soon as the process is paused, all buffers not written to disk yet will be synchronized calling the function `do_fsync()`, as shown in figure 8.

6.2 Inserting new *fd* into process *FDT*

While the process p is paused, the LKM can alter its *FDT* in secure way. Despite the fact that the Linux Kernel has a C function called `get_unused_fd`, kmodest must reimplement it by itself, as discussed earlier: `get_unused_fd_by_task()`. After a call to this function, the module obtains the next available file descriptor number to use reading `next_fd` variable.

Thus, kmodest initializes a new `file*` data structure and inserts it into the *FDT* slot pointed by `next_fd` value, calling its own implementation for `fd_install` kernel-function, `fd_install_by_task()`, as shown in Figure 9.

The current version of module kmodest.ko is not in charge of opening and closing this new file inserted into process *FDT* but *umodest*, the user-space utility.

The last step depends on how the module has been used, that is, kmodest will redirect all I/O calls from *fd* to *fd'* or will intercept them and will send them to *fd'* simultaneously.

[2] So there's a bit impact on process execution time.

```
if(sys_kill (process->pid, SIGSTOP)!=0)return -1;

/* Allow to run other processes ...  */
schedule();

rcu_read_lock();
file_to_sync = fcheck_files(process->files , user_args->oldfd);
if(file_to_sync)do_fsync(file_to_sync,0);
   rcu_read_unlock();
```

Figure 8: Pausing the process and syncing its buffers to disk

6.3 Remmaping all I/O syscalls from *fd* to *fd'*

Basically, the LKM kmodest.ko exchanges the file pointers stored at <`FDT->fd[fd]`, `FDT->fd[fd']`¿, calling `do_remap_fd()`, shown in Figure 10. Finally, it sends the `SIGCONT` signal to the process p in order to resume its execution.

So, from the process's point of view, next I/O operations - no matter what kind of system call can be called -, will affect directly to *fd'*. It's very clear that *fd* will never be used until kmodest.ko will restore the file descriptors calling `do_restore_tasks`, just before ending umodest user-space utility execution.

6.4 Examining each `sys_write();` / `sys_writev();` calls for *fd*

The *IDT* can be located quickly using an Intel X86 instruction code, `sidt` [7]. Starting from its base address, kmodest.ko inserts its own `system_call` handler at offset `0x80` in order to do some previous tasks just before returning the flow control to the original system handler implemented by The Linux Kernel.

As shown in Figure 11, all the code needed to do that is placed at module initialization phase, in the function `module_init()`:

As soon as the LKM unloads, the original system handler will be restored, simply doing the opposite thing at cleanup C function `modest_cleanup()`.

The new installed handler called `new_syscall_handler()` is written entirely in Assembler language, and can be found in the source file `syscall_entry.S`. As shown in figure 12, this handler do basically two main tasks: saves the processor regs values (`ebx, ecx and edx`) on C variables defined in the C source file `modest.c`, and then, depending on the system call called, decides whether it must call `modest_handler()` function or not.

As discussed earlier, all parameters passed to a system call function are stored in CPU registers, so kmodest.ko saves them - only thinking about the first three parameters for the system call function `sys_write() / sys_writev()`.- Then, calls a C function implemented in the source file `modest.c` in order to do additional tasks, as long as the system call called by process p had been `sys_write() / sys_writev()`.

Finally, this assembler piece of code restores the address of the original system handler saved previously on the stack, and returns the execution to it. This is the main reason for finding out a `ret` instruction instead of `iret`, because this handler is not, in fact, a real interrupt handler as defined and implemented in-

```
/* Get a reference to file opened by user-space utility program umodest:
/
modest = find_task_by_pid (user_args->umodest_pid);
if(modest){
   file_to_add = fcheck_files (modest->files , (modest->files->next_fd - 1));
   if(file_to_add){
      nfd = get_unused_fd_by_task(process);
      /* Store on variable to process later - see do_restore_task below -:  */
      krn_fd = nfd;
      /* Okay, now I have next_fd initialized rightly, so I can install
      * new file* inside fd array for outter process !  :  */
      fd_install_by_task(nfd , process , file_to_add);
   }else return -1; /* Unable to get the file opened !  */
}else return -1; /* Error getting umodest process descriptor */
```

Figure 9: Inserting a new file into the process *FDT*

side the Linux Kernel.

The code written in the C function `modest_handler` only needs to check for a write to *fd* by the process *p* and, in this case, to re-write all intercepted data to *fd'*. Since the parameter passed just before `int $0x80` into `ecx` CPU-register stores the address where the data to be written can be found - in user-space area, of course -, kmodest.ko implements its own `sys_write` function - `my_sys_write()`- which gets this data from the user's space memory area and writes over *fd'* file. The same idea can be applied to `sys_writev()` system call althought the user address passed by `ecx` CPU register stores the address of `struct iovec *` vector.

7 Conclusions

Despite the fact that there are other kind of tools available to catch some system calls - like `strace`, `dtrace` for Solaris Kernels, and so on-, *MODEST* could be a good solution to use in two particular cases, discussed below.

7.1 Case 1: "no space left on the drive"

While a process *p* is running, writting a lot of data to the hard drive, in spite of *LVM* existence and journaling filesystems allowed to grow up at runtime, there's an awful technical problem that could generate a process crash: free disk space.

Using *MODEST*, it is feasible to redirect for a while the new writes, with no need to kill the process by hand, to another file placed on another location with more available disk space. As soon as the original location will be able to save more disk space, *MODEST* can be used again to restore the original process *FDT*.

Apart from this, there's no need to modify the sources for the processes, and *MODEST* can be controlled by a non-root user account.

7.2 Case 2: Peeping process output

As soon as a certain process has data to be written over an opened file descriptor, the `sys_write() /sys_writev()` calls do not write their data immediately on the hard disk,

```
 spin_lock(&files->file_lock);
...
/* Get file table descriptor pointer :  */
fTable = files_fdtable(files);
tmp = fTable->fd[user_args->oldfd];
if(!tmp)return -EBADF;

/* Sync all cache to disk for this file before eXchange it :  */
do_fsync(tmp,0);

fTable->fd[user_args->oldfd] = fTable->fd[krn_fd];
fTable->fd[krn_fd] = tmp;
...
sys_kill(process->pid, SIGCONT);
```

Figure 10: Exchanging file descriptors <*fd,fd'¿*

at least while the flag `O_SYNC` is not set.

Thus, it is feasible to use *MODEST* in order to 'peep' their data and to write they, for example, on a file descriptor previously opened by umodest user-space utility with the `O_SYNC` flag set, to print it out or to show it up over a console session, even connecting they to a previously socket established connection.

8 License

This work is licensed under the **Creative Commons Attribution-No Derivative Works 2.5 Spain License**.
So as to view a copy of this license:

- (a) visit http://creativecommons.org/licenses/by-nd/2.5/es/deed.ca
- (b) send a letter to:
  Creative Commons
  171 2nd Street, Suite 300
  San Francisco, California 94105, USA

```
...
sm__ __volatile__("sidt syscall_gate=(struct interrupt_gate*)(idtr.base +
                              0x80*sizeof(struct interrupt_gate));
original_syscall_handler=
         ((syscall_gate->offset_high<<16)|syscall_gate->offset_low);

local_irq_disable();
syscall_gate->offset_low=(unsigned)(new_syscall_handler)&0xFFFF;
syscall_gate->offset_high=((unsigned)(new_syscall_handler)>>16)&0xFFFF;
local_irq_enable();
...
```

Figure 11: Installing its own `system_call` handler at offset `IDT+0x80`

```
 ...
movl %ebx , userfd
movl %ecx , memory
movl %edx , bytes_to_read
...


call modest_syshandler
...
pushl original_syscall_handler
ret
```

Figure 12: The new pseudo-system handler

```
...
if(pid_affected!=-1){
   if(current->pid == pid_affected && userfd == oldfd){
      switch(syscall_id){
          case __NR_write:  my_sys_write( krn_fd, memory , bytes_to_read);
          break;
          case __NR_writev:
            my_sys_writev(krn_fd,(const struct iovec __user*)memory,bytes_to_read);
          break;
      }
   }
}
...
```

Figure 13: Checking for a write to *fd* by the process *p* and writting to *fd'* simultaneously

References

[1] *MODEST project homepage*
http://kmodest.sf.net
http://www.sourceforge.net/projects/kmodest

[2] *Understanding the Linux Kernel, 3rd edition*
http://oreilly.com/catalog/9780596005658/?CMP=AFCak_book&ATT=Understanding+the+Linux+Kernel

[3] *Essential Linux Device Drivers*
http://www.pearson.ch/Informatik/PrenticeHall/1471/9780132396554/Essential-Linux-Device-Drivers.aspx

[4] *GCC Inline assembly HOW-TO*
http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html

[5] *ATT assembly language syntax reference*
http://sig9.com/articles/att-syntax

[6] *Intel X86 Instruction Set Reference A-M*
http://download.intel.com/design/processor/manuals/253666.pdf

[7] *Intel X86 Instruction Set Reference N-Z*
http://download.intel.com/design/processor/manuals/253667.pdf