CHAPTER 4

# Writing a TPM Device Driver

The Trusted Platform Module (TPM) is in essence a passive storage device that is hard mounted on the motherboard. The TPM is attached to the Low Pin Count (LPC) bus, a low-pin count bus, which is also used for attaching the system BIOS Flash memory. This ensures that the TPM device is available during the early system bootstrap before any other device is initialized.

Communication with the TPM is typically handled by the TCG device driver library (TDDL), and its interface is defined by the TSS (TCG Software Stack) specification. This library typically communicates with a device driver inside the kernel, and this device driver communicates with the actual TPM device.

There are currently two different TPM device programming interfaces. The TPM 1.1b-compliant devices use a device interface that is vendor specific. Because of the lack of a standardized device interface, each vendor is forced to define its own interface. This interface is typically very simple because the TPM 1.1b specification places only modest requirement on the device interface. To curtail the myriad of incompatible device interfaces, TCG decided to standardize the PC client interface as part of its 1.2 TPM specification. This device interface supports dynamic root of trust and consequently requires a more sophisticated interface that supports locality and special LPC bus cycles.

In this chapter, we will examine how to write a TPM device driver. We start with the TDDL interface and illustrate how to program the TPM 1.1b and 1.2 low-level device interfaces to implement part of the TDDL interface. The code examples have been developed on Linux, but in the interest of staying operating system-neutral, we have removed all the Linux-specific device driver boilerplates. Instead we focused on the core device driver functions that are similar for any operating system.

In particular, this chapter covers the following:

- The TCG device driver library
- The 1.1 device driver interface
- The 1.2 device driver interface
- The device programming interface (both 1.1 and 1.2)

## TCG Device Driver Library

The TDDL interface is the standard interface that applications use for communicating directly with the TPM. This interface is part of the TSS library and is in fact used by the TSS stack to talk to the TPM. Although this interface is available for use by applications, its direct use is typically best avoided. The TDDL interface is a single-threaded synchronous interface, and it assumes that all TPM commands are serialized by the caller of the interface. If this interface is used together with TSS library calls, then the results may be undefined.

For those applications that do need direct device access, the TDDL interface specification is listed next:

```
TSS_RESULT Tddli_Open( );
TSS_RESULT Tddli_Close( );
TSS_RESULT Tddli_Cancel( );
TSS_RESULT Tddli_GetCapability(UINT32 CapArea, UINT32 SubCap,
                        BYTE* pCapBuf,  UINT32* pCapBufLen);
TSS_RESULT Tddli_SetCapability(UINT32 CapArea, UINT32 SubCap,
                        BYTE* pSetCapBuf, UINT32
                        SetCapBufLen);
TSS_RESULT Tddli_GetStatus(UINT32 ReqStatusType, UINT32*
                        pStatus);
TSS_RESULT Tddli_TransmitData(BYTE* pTransmitBuf, UINT32
                        TransmitBufLen,
                        BYTE *pReceiveBuf, UINT32*
                        pReceiveBufLen);
```

The TDDL interface is straightforward. It includes calls to open and close access to the TPM device, cancel any outstanding commands, get and set vendor-specific device capabilities, get the device status, and transmit a command blob to the TPM and receive a response. Unlike the actual device interfaces that are discussed later in the chapter, the TDDL interface is the same for TPMs that follow the 1.1b or the 1.2 specification.

In the next sections, we are going to examine how to implement the device driver equivalent of the `Tddli_TransmitData` function with a TPM 1.1b- and a TPM 1.2-compliant device.

## TPM 1.1b Specification Device Interface

The TCG 1.1b specification was the first TCG TPM standard that was widely available, and many vendors have developed products around that standard. Unfortunately, that standard did not define a mechanism by which to communicate with the actual TPM device. Consequently, each vendor was forced to define a low-level communication interface that was unique to that vendor.

In the following example, we describe the interface to the Atmel 1.1b TPM chip. This TPM device was one of the earliest available and is often used in desktop and notebook computers.

### Technical Details

The Atmel 1.1b TPM uses port I/O for communication and does not support interrupts. The port I/O addresses that the device uses are 0x4E, 0x4F, 0x400, and 0x401. The first two ports, 0x4E and 0x4F, are used to query the chip for details such as version number and manufacturer. They are used in an index/data pair configuration and are read as:

```
int rdx(int index)
{
        outb(index, 0x4E);
        return inb(0x4F) & 0xFF;
}
```

When read, port address 0x401 acts as the TPM status register. It signals that the TPM is busy (bit 1), or when data is available to be read (bit 2). Writing a value to port 0x401 with the first bit set will cause the TPM to abort the current command that it is executing. Port 0x400 is the data port; it is used to read and write TPM commands from and to the device.

A typical sequence for a TPM device driver is to send a command to the device, wait for a response, and then read the response. Because the Atmel TPM device does not support interrupts, the device driver has to poll the status register to determine whether a response is available. This is a rather inefficient way to wait for a device since, when done incorrectly, it can make the host system unresponsive for a long period of time. This is especially true for a TPM device. The wait time between a send command and a receive response may literally take 60 seconds or more for some operations, such as key generation.

For convenience, we use the following definitions in the code to make it easier to follow:

```
#define ATMEL_DATA_PORT          0x400   /* PIO data port */
#define ATMEL_STATUS_PORT        0x401   /* PIO status port */

#define ATMEL_STATUS_ABORT       0x01    /* ABORT command (W) */
#define ATMEL_STATUS_BUSY        0x01    /* device BUSY (R) */
#define ATMEL_STATUS_DATA_AVAIL  0x02    /* Data available (R) */
```

These definitions provide specifics for a particular Atmel TPM. Having that settled, we can now write cleaner code.

## Device Programming Interface

The Atmel TPM requires no special initialization for it to work correctly. The LPC bus, on which it resides, is typically already initialized and set up by the system BIOS. To check for the presence of the Atmel TPM, the following check suffices:

```
int init(void)
{
        /* verify that it is an Atmel part */
        if (rdx(4) != 'A' || rdx(5) != 'T' ||
            rdx(6) != 'M' || rdx(7) != 'L')
                return 0;
        return 1;
}
```

The Atmel TPM uses the signature "ATML" to signal that the device is present and operating correctly.

The `Tddli_TransmitData` function consists of two major components: a function to send a command to the TPM device and a function to receive the response from the device. The send command is illustrated next:

```
int send(unsigned char *buf, int count)
{
        int i;

        /* send abort and check for busy bit to go away */
        outb(ATMEL_STATUS_ABORT, ATMEL_STATUS_PORT);
        if (!wait_for_status(ATMEL_STATUS_BUSY, 0))
                return 0;

        /* write n bytes */
        for (i = 0; i < count; i++)
                outb(buf[i], ATMEL_DATA_PORT);

        /* wait for TPM to go BUSY or have data available */
        if (!wait_for_not_status(ATMEL_STATUS_BUSY|
                                 ATMEL_STATUS_DATA_AVAIL, 0))
                return 0;

        return count;
}
```

The first step of the send function is to ensure that the TPM device is in a well-defined state where it is ready to accept a command. This is achieved by aborting any outstanding command

and waiting for the device to become ready. The `wait_for_status` function does exactly that: It checks the status register for a state change using the condition `(status & mask) == value`, and if it did not evaluate to true, then it waits for a brief period and tries again. After a while, the function times out and will return 0. If the wait for condition is met, then the function returns 1. In this particular instance, `wait_for_status` waits for the busy bit to turn off.

Once the TPM device is ready to accept a command, it is simply written to the data port one byte at a time. The command is a TPM blob as defined by the TCG TPM specification. As soon as the command is written, we need to check for either of two conditions: 1) the TPM is processing the command, or 2) the command has already finished and the response is available. We check for these conditions by calling the `wait_for_not_status` function and test for the busy and available bits. This function is analogous to `wait_for_status` except that the test condition tests for inequality: `(status & mask) != value`.

The receive function follows similar concepts but is more complicated because it needs to handle more conditions:

```
int recv(unsigned char *buf, int count)
{
        unsigned char *hdr = buf;
        unsigned long size;
        int i;
        /* wait while the TPM to respond */
        if(!wait_for_status(ATMEL_STATUS_DATA_AVAIL|
                        ATMEL_STATUS_BUSY,
                        ATMEL_STATUS_DATA_AVAIL))
                return 0;
        /* start reading header */
        for (i = 0; i < 6; i++) {
                if ((inb(ATMEL_STATUS_PORT) &
                     ATMEL_STATUS_DATA_AVAIL)
                                                        == 0)
                        return 0;
                *buf++ = inb(ATMEL_DATA_PORT);
        }
        /* size of the data received */
        size = decode_u32(hdr + 2);
        if (count < size)
                return 0;
        /* read all the data available */
        for ( ; i < size; i++) {
                if ((inb(ATMEL_STATUS_PORT) &
                     ATMEL_STATUS_DATA_AVAIL)
                                                        == 0)
```

```
                        return 0;
                *buf++ = inb(ATMEL_DATA_PORT);
        }

        /* sanity check: make sure data available is gone */
        if (inb(ATMEL_STATUS_PORT) & ATMEL_STATUS_DATA_AVAIL)
                return 0;

        return size;
}
```

Before calling the preceding receive function, you need to make sure that the device is not still busy working on the command itself. You can check this by examining the busy bit in the status command and schedule another task if it is still busy. Some commands can take up to more than a minute to process, so spinning on the busy bit may lead to an unresponsive system.

The first step in the receive function is to ensure that there is a response available and that the TPM is no longer busy processing a request. The next step is to read the response buffer.

Like most 1.1b TPMs, the Atmel TPM does not allow you to query the length of the response without reading the actual data. For this reason, receiving a response from the TPM consists of two parts. During the first part, we read the TPM blob header from the device. This header is always 6 bytes long and contains the total length of the response in a big endian byte order. After the length has been converted to native byte order, we use it in the second part to read the remainder of the response packet.

Although the device interface we described in this section is specific to the Atmel TPM implementation, the concepts and the way to interact with the device are similar for other 1.1b TPMs. Still, each 1.1b TPM requires its own device driver, which is quite a burden for firmware and operating system developers. To eliminate the need for multiple device drivers, TCG defined a single device interface standard for TPMs that follow the TPM 1.2 specification. We will discuss that next.

## TPM 1.2 Specification Device Interface

To reduce the myriad of TPM device interfaces and device drivers, TCG decided to standardize the device interface as part of its TPM 1.2 specification. This standard is called the *TCG PC Client Specific TPM Interface Specification*, or TIS for short.

As a result of this standard, firmware and operating system vendors need to implement only one device driver to support all the available TIS-compliant devices. It is exactly for this reason that Microsoft decided to only support TPM 1.2-compliant devices in its Microsoft Vista operating system, even though Vista currently uses only TPM 1.1b features.

The TIS defines two different kinds of device interfaces. The first is the legacy port I/O interface and the second is the memory mapped interface. Because the port I/O interface is similar to the one discussed in the previous section, we will only concentrate on the memory mapped interface in the next section.

## Technical Details

A TPM 1.2-compliant device uses memory mapped I/O. It reserves a range of physical memory that the host operating system can map into its virtual address range. Instead of executing explicit I/O instructions to communicate to the device, it is sufficient to use traditional memory accesses.

The memory mapped interface for the TPM is shown in Figure 4.1. It consists of five 4KB pages that each presents an approximately similar register set. Each page corresponds to a locality—that is, commands that are sent to the TPM through the register set at address FED4.0000 are implicitly associated with locality 0. Similarly, commands sent using the register set at FED4.2000 use locality 2. The reason for the duplication of register sets and the alignment on page boundaries is to enable the host operating system to assign different localities to different levels of the system and regulate their access by controlling their virtual memory mappings. For example, a security kernel would typically only have access to the register set associated with locality 1, while the applications would be associated with the register set belonging to locality 3. Locality 0 is considered legacy and is used by systems that use the static root of trust measurements.

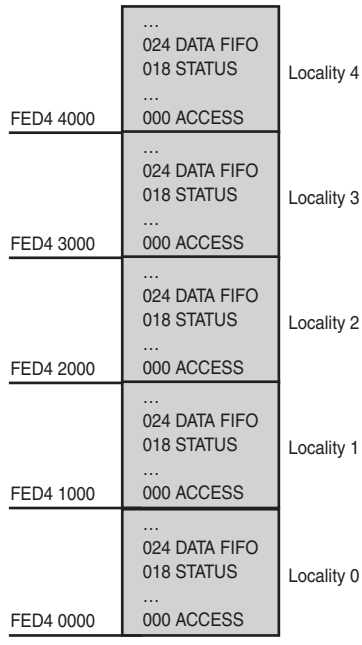


**Figure 4.1** TPM memory map

Locality 4 is special in the sense that it is used by the processor to store the measurement of a secure loader that was started as part of the dynamic root of trust mechanism. The secure loader measurement is stored in PCR17 and can only be set by using special LPC bus cycles that cannot be generated by software. This ensures that the secure loader measurement originated from the processor itself.

Unlike most devices that follow the TPM 1.1b specification, the TIS standard specifies that a TPM device should be able to generate interrupts. That is, whenever data becomes available or the device is ready for its next command, it generates an interrupt. However, sometimes interrupt-driven device drivers are inappropriate, especially during early bootstrap. In these cases, the TPM device can still be used in polled I/O mode.

An interesting aspect of the memory mapped interface is that multiple consumers can access the TPM device at the same time, leading to potential concurrency issues. For example, a trusted operating system and an application can concurrently issue commands to the TPM to update a particular PCR. To ensure the consistency of the internal TPM state, the TIS designers developed a locking protocol. Before a consumer at a certain locality can use the TPM, it needs to request access to the TPM by setting the RequestUse bit in the access register. If no locality is currently using the TPM, then access is granted. If the TPM is being used, then the access request remains pending until the owning locality relinquishes control. The highest-level locality is then given access first. If a locality has crashed or is malicious and does not relinquish control, a higher-level locality can take over the TPM by setting the Seize bit.

In the next section, we will describe how to implement a function that is similar to the `Tddli_TransmitData` function. Because the TIS interface is a lot more complicated than the TPM 1.1b driver, we will focus on a simplified driver that uses the memory mapped interface and only uses the legacy locality 0. A driver with these restrictions uses only a core set of the TIS interface, which we will describe later in the chapter.

Among the set of registers the TPM provides are the access, status, and data FIFO registers. These registers reside at offsets 000, 018, and 024 (hexadecimal), respectively. The access register is used to gain access to a particular locality and is used by the locking protocol described previously. When written, bit 2 signals a request to use and bit 6 relinquishes control. When reading the access register, bit 6 signals that this locality is currently active.

The status register indicates that the device is ready for a command (bit 6), has a response available (bit 5), or is expecting more data (bit 4). The high-order bit of the status byte (bit 8) indicates that the status is valid, and this bit can be used by a driver that uses polled I/O to wait for a particular status to occur. When writing the status register, bit 6 indicates that the recently written command blob should be executed. The status register is an 8-bit register. The remaining bytes of the status word (byte 1 and 2) form a burst count. That is essentially the size of the data FIFO and specifies the number of bytes that can be written or read consecutively. The TPM's data FIFO is available through the register at offset 024.

Some register sets, such as the set for locality 0, contain additional registers. An example of that is the device and vendor ID register at offset F00. This register is typically used to determine

the presence of the TPM and, if one is present, the vendor. The assigned vendor IDs currently follow the PCI-Express vendor designations.

In the following code examples, we use this shorthand to read and write memory locations: The address argument is relative to the base of the memory range where physical address FED4.0000 is mapped and is only used to read and write the TPM register sets. The routines also include the necessary memory barrier operations to ensure coherency.

```
unsigned char read8(unsigned long addr);
void write8(unsigned char val, unsigned long addr);
unsigned long read32(unsigned long addr);
void write32(unsigned long val, unsigned long addr);
```

For convenience, we use the following definitions in the code to make it easier to follow:

```
/* macros to access registers at locality ''l'' */
#define ACCESS(l)                       (0x0000 | ((l) << 12))
#define STS(l)                          (0x0018 | ((l) << 12))
#define DATA_FIFO(l)                    (0x0024 | ((l) << 12))
#define DID_VID(l)                      (0x0F00 | ((l) << 12))
/* access bits */
#define ACCESS_ACTIVE_LOCALITY          0x20 /* (R)*/
#define ACCESS_RELINQUISH_LOCALITY      0x20 /* (W) */
#define ACCESS_REQUEST_USE              0x02 /* (W) */
/* status bits */
#define STS_VALID                       0x80 /* (R) */
#define STS_COMMAND_READY               0x40 /* (R) */
#define STS_DATA_AVAIL                  0x10 /* (R) */
#define STS_DATA_EXPECT                 0x08 /* (R) */
#define STS_GO                          0x20 /* (W) */
```

## Device Programming Interface

Before we can use the TPM, we need to initialize the device. The following routine performs a trivial initialization and presence check:

```
int init(void)
{
        unsigned vendor;
        int i;

        for (i = 0 ; i < 5 ; i++)
                write8(ACCESS_RELINQUISH_LOCALITY, ACCESS(i));
```

```
        if (request_locality(0) < 0)
                return 0;

        vendor = read32(DID_VID(0));
        if ((vendor & 0xFFFF) == 0xFFFF)
                return 0;

        return 1;
}
```

The initialization routine is executed only once when the driver is instantiated. It first forces all localities to relinquish their access and then requests the use of the legacy locality 0. When access is granted, it checks for the presence of a valid vendor ID. When this check succeeds, the TPM is ready to accept commands. If you were writing a driver that uses interrupts, this would be the place where you would check for the supported interrupts and enable them.

```
int request_locality(int l)
{
        write8(ACCESS_RELINQUISH_LOCALITY, ACCESS(locality));

        write8(ACCESS_REQUEST_USE, ACCESS(l));
        /* wait for locality to be granted */
        if (read8(ACCESS(l) & ACCESS_ACTIVE_LOCALITY))
                return locality = l;

        return -1;
}
```

To request a locality that is specified by parameter `l`, we relinquish the current locality (stored in the global variable `locality`) and request the use of the specified locality. For a TPM driver that only provides access to the legacy 0 locality, this access should be granted immediately with the locality active bit set in the status register. If this routine is used for the non-legacy localities, it would have to wait until the access is granted. This wait can be done by either polling the active bit or by waiting for the locality changed interrupt.

To send a command to the TPM, we use the following function:

```
int send(unsigned char *buf, int len)
{
        int status, burstcnt = 0;
        int count = 0;

        if (request_locality(locality) == -1)
                return -1;
```

```
        write8(STS_COMMAND_READY, STS(locality));

        while (count < len - 1) {
                burstcnt = read8(STS(locality) + 1);
                burstcnt += read8(STS(locality) + 2) << 8;

                if (burstcnt == 0){
                        delay(); /* wait for FIFO to drain */
                } else {
                        for (; burstcnt > 0 && count < len - 1;
                                                burstcnt--) {
                                write8(buf[count],
                                          DATA_FIFO(locality));
                                count++;
                        }

                        /* check for overflow */
                        for (status = 0; (status & STS_VALID)
                                          == 0; )
                                status = read8(STS(locality));
                        if ((status & STS_DATA_EXPECT) == 0)
                                return -1;
                }
        }

        /* write last byte */
        write8(buf[count], DATA_FIFO(locality));

        /* make sure it stuck */
        for (status = 0; (status & STS_VALID) == 0; )
                status = read8(STS(locality));
        if ((status & STS_DATA_EXPECT) != 0)
                return -1;

        /* go and do it */
        write8(STS_GO, STS(locality));

        return len;
}
```

In order for us to send a command to the TPM, we first have to ensure that the appropriate locality has access to the device. This is especially important when the driver supports multiple localities with concurrent access. Once we get access to the requested locality, we tell the TPM that we are ready to send a command and then proceed to write the command into the DATA FIFO at burst count chunks at the time. After each chunk, we check the status register for an overflow condition to make sure the TPM kept up with us filling the FIFO. As an added safety measure, we write the last byte separately and check whether the TPM is no longer expecting more data bytes. If it is not, then the TPM is ready to execute the command blob we just transferred to it. This is done by setting the status to GO.

Receiving a response from the TPM is slightly more complicated, and its code is divided into two parts: a helper function to get the data from the TPM and a receiver function to reconstruct the response.

The helper function is listed next:

```
int recv_data(unsigned char *buf, int count)
{
        int size = 0, burstcnt = 0, status;

        status = read8(STS(locality));
        while ((status & (STS_DATA_AVAIL | STS_VALID))
                        == (STS_DATA_AVAIL | STS_VALID)
                        && size < count) {
                if (burstcnt == 0){
                        burstcnt = read8(STS(locality) + 1);
                        burstcnt += read8(STS(locality) + 2) << 8;
                }

                if (burstcnt == 0) {
                        delay(); /* wait for the FIFO to fill */
                } else {
                        for (; burstcnt > 0 && size < count;
                                                burstcnt--) {
                             buf[size] = read8(DATA_FIFO
                             (locality));
                             size++;
                        }
                }
                status = read8(STS(locality));
        }

        return size;
}
```

Receiving a response from the TPM follows the same principles as writing a command to it. As long as there is data available from the TPM and the buffer count has not yet been exhausted, we continue to read up to burst count bytes from the DATA FIFO.

It is up to the following receive function to reconstruct and validate the actual TPM response:

```
int recv(unsigned char *buf, int count)
{
        int expected, status;
        int size = 0;

        if (count < 6)
                return 0;

        /* ensure that there is data available */
        status = read8(STS(locality));
        if ((status & (STS_DATA_AVAIL | STS_VALID))
                        != (STS_DATA_AVAIL | STS_VALID))
                return 0;

        /* read first 6 bytes, including tag and paramsize */
        if ((size = recv_data(buf, 6)) < 6)
                return -1;
        expected = be32_to_cpu(*(unsigned *) (buf + 2));

        if (expected > count)
                return -1;

        /* read all data, except last byte */
        if ((size += recv_data(&buf[6], expected - 6 - 1))
                                            < expected - 1)
                return -1;

        /* check for receive underflow */
        status = read8(STS(locality));
        if ((status & (STS_DATA_AVAIL | STS_VALID))
                          != (STS_DATA_AVAIL | STS_VALID))
                return -1;

        /* read last byte */
        if ((size += recv_data(&buf[size], 1)) != expected)
                return -1;
```

```
        /* make sure we read everything */
        status = read8(STS(locality));
        if ((status & (STS_DATA_AVAIL | STS_VALID))
                        == (STS_DATA_AVAIL | STS_VALID)) {
                return -1;
        }

        write8(STS_COMMAND_READY, STS(locality));

        return expected;
}
```

The receive function is called either after a data available interrupt occurs or, when the driver uses polling, the status bits valid and data available were set. The first step for the receive function is to ensure that the TPM has a response that is ready to be read. It then continues to read the first 6 bytes of the response. This corresponds to the TPM blob header and contains the length of the total response in big endian byte order. After converting the response size to native byte order, we continue to read the rest of the response. As with the send function, we treat the last byte separately so that we can detect FIFO underflow conditions. Once the entire response is read, the TPM is ready to accept its next command. This is achieved by setting the command ready bit in the status register.

## Summary

In this chapter, we have demonstrated how to talk to a TPM (both 1.1 and 1.2) at the lowest level. This will be useful for programmers who want to talk to the TPM directly—either in BIOS or a device driver. When developing new hardware that will make use of a TPM, this will be useful for determining if the TPM is working as designed. Although all commands can talk to the TPM directly, application-level programs should use the TCG Software Stack (TSS) application programming interface.