An Introduction To Writing TDI Filter Drivers

Viviane Zwanger [1]

International Secure Systems Lab[2], *Institute Eurecom*[3]
University Tuebingen[4]

March 18, 2010

[1]Viviane@iseclab.org
[2]http://iseclab.org
[3]http://www.eurecom.fr
[4]http://www.uni-tuebingen.de

Contents

1 Preface **2**

2 Introduction to TDI **4**
2.1 IRPs . . . 6
2.2 The concept of a device . . . 6
2.3 Compiling and loading a driver . . . 8
2.4 Major functions . . . 10

3 IRP Handling **13**
3.1 Completion routines . . . 16
3.2 Cancel routines . . . 19

4 TDI **21**
4.1 A simple Forward-and-Forget filter driver . . . 21
4.2 Overview of the essential TDI structs . . . 28
4.3 An improved Forward-and-Forget filter driver . . . 31
4.4 A redirecting TDI filter driver . . . 35
4.5 Countermeasures against IRP manipulation in kernelspace . . . 39
4.6 Freeing IRPs . . . 43

5 Summary and Conclusion **45**
5.1 Acknowledgments . . . 47

Chapter 1

Preface

```
     """,""   """" "*
   *"   ""*\\'"" (
  ""'"" """"""*" ""' """"
     '""*"")/' """"""**"
 *""""""'  '"**"   ""*   "
   ""  " \ ((
          \\,) '*"" "'
         "')/.)/   ""
          /;/
   _"___,//#)-_.___,______V
```

An Introduction To
Writing TDI Filter Drivers

[1] Documentation and examples about the TDI (Transport Driver Interface) architecture are very rare. At the time of writing, no official documentation exists on the field of TDI, besides some small hints given at MSDN[2] and OSR[3], which may only be found by having previous experience. The NDIS layer, even though being settled at a much more deeper layer and thus not being suited too well, is much better documented than the higher TDI layer. As a consequence, some people and companies profit by selling

[1] The ASCII artwork depicts a tree, which sometimes holds as a symbol of the operating system. Although not living, an operating system resembles a tree in the respect to starting from the root and expanding into many branches and leaves containing the filesystem as well as having vital organs, called drivers.

[2] http://MSDN.microsoft.com

[3] http://www.osronline.com

knowledge and code for surprising amounts of money. While this is a legitimate and inevitable consequence of the situation, the author is of the opinion that it is dangerous to keep such knowledge undocumented in the long term. It is an annoying obstacle for security researcher and network programmers. This white paper aims to finally shed some light upon this undocumented field.

The purpose of this paper is to give a detailed introduction to writing kernelmode filter drivers for the Transport Driver Interface (TDI). It describes the fundamentals of WDM-based drivers and introduces IRP handling. Correct IRP-handling is one of the most important issues in writing drivers for Microsoft Windows. In the main chapter, several simple TDI filter drivers are described, which are enhanced in following sections of the chapter. Furthermore, different possibilities and techniques of how to use TDI are shown, which will be accompanied by practical code examples.

Chapter 2

Introduction to TDI

TDI, the Transport Driver Interface, is an abstraction layer for easier use of common protocol stacks, such as TCP/IP, IPX and NBF and others. Kernelmode drivers can access this interface for using basic network-functions and to provide their own network-related functions to other programs.

The Transport Driver Interface is settled completely in kernelmode. Above TDI, Windows offers a more abstract layer, which can be used in userspace. A well-known example is the Winsock API. In most cases, using the Winsock API will be sufficient, but in rare circumstances, for example in IT-Security, or when high performance is needed, kernelmode network-operations are the right choice. Rather than accessing the hardware device-drivers directly, it is easier to use TDI in these cases.

The following list gives a quick overview about the more known network-protocols supported by TDI:

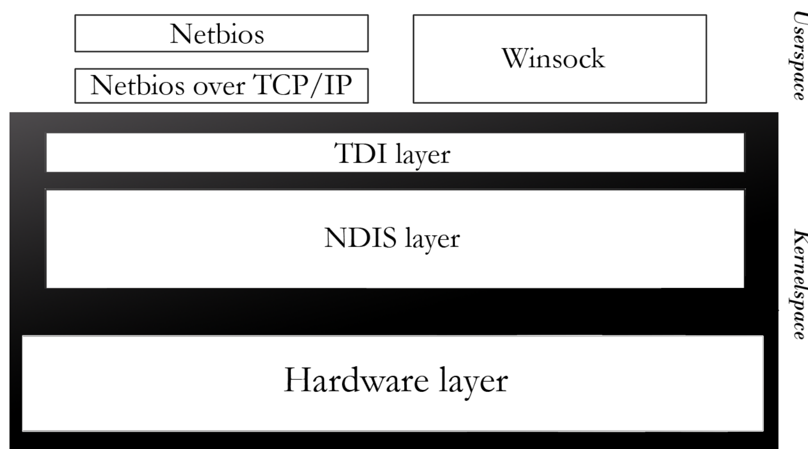
- TCP/IP is the most known and used network protocol, it is routable and therefore suitable for communication over the Internet.
- IPv6 is the successor of the IP protocol. There is TDI support provided for both Windows 2000 as well as Windows XP and

upwards.

- IPX is an older architecture, quite fast and simple to use, but only usable within one network. It is not routable. Since the upcoming of the Internet, it has lost its importance.
- NBF is a protocol developed by Microsoft. Like IPX, it is not routable and only applies within one network.
- APPLETALK is a network protocol developed by Apple for communication between computers, file sharing and printing. It is deprecated now, since Apple switched over to the TCP/IP protocol suite.

Further supported protocols such as VNS can be found in *tdi.h* localized in the Windows DDK directory.
The listed protocol stacks are fully implemented by Microsoft Windows on top of the NDIS layer, the Network Driver Interface Specification, and unified under TDI. This allows a driver developer to use a standardized API for these Windows-supported protocol stacks.

The following figure illustrates the relationship of the different network layer used by Windows:



In this first chapter, we introduce the basics of writing kernelmode drivers and provide several practical examples.

2.1 IRPs

IRPs are the way Drivers communicate with each other or with user mode programs. An IRP is a complex packet of data, which contains information about the request that was made. Drivers can ”register” for a certain device they are interested in and receive all IRPs concerning this specific device.
Drivers are chained to each other, with the lowest driver being the hardware driver. The hardware driver is responsible for the actual reading and writing to a piece of hardware. A low-level Hardware driver does not do complex operations, it usually works as simple as possible. A higher driver in the chain will use the provided (simple) functions to offer more complex functions to an even more higher driver. Highest drivers usually have the most complex operations.

2.2 The concept of a device

A device might be a piece of hardware, for example a usb stick or the hard disk. In this case, the driver would probably be a low-level driver. However, most drivers do not work with actual hardware devices. They use their own devices or devices created by other drivers. These devices have more similarity to a file: Drivers can create, open, read or write to those devices and share a device with other drivers.
This is how a filter driver works: it attaches to an existing device, for example, the TCP device, and receives all IRPs concerning this

device. Now it is able to observe all data concerning TCP. [1]

We previously discussed devices and IRPs. In the following, an introductory example of how to write a simple driver is presented. This driver has no device and, therefore, cannot receive and process IRPs concerning this device.

Example: a simple hello-world driver.

```
1 // For wdm-based drivers, wdm.h is obligatory to include.
2 #include <wdm.h>
3
4 VOID UnloadDriver(PDRIVER_OBJECT  DriverObject);
5 NTSTATUS DefaultFunction(PDEVICE_OBJECT DeviceObject, PIRP Irp);
6
7 /****************
8  *  DriverEntry
9  ****************/
10 NTSTATUS DriverEntry(
11 PDRIVER_OBJECT  pDriverObject,
12 PUNICODE_STRING  pRegistryPath)
13 {
14    NTSTATUS status = STATUS_SUCCESS;
15    unsigned int i = 0;
16
17    DbgPrint("Hello World! \r\n");
18
19    if(status == STATUS_SUCCESS) {
20
21    for(i = 0; i < IRP_MJ_MAXIMUM_FUNCTION; i++)
22       { pDriverObject->MajorFunction[i] = DefaultFunction; }
23
24    pDriverObject->DriverUnload = UnloadDriver;
25    }
26
27 return status;
28 }
29
30 VOID UnloadDriver(PDRIVER_OBJECT  DriverObject)
31 {
32    DbgPrint("Bye! \r\n");
33 }
34
35 NTSTATUS DefaultFunction(PDEVICE_OBJECT DeviceObject, PIRP Irp)
36 {
37     NTSTATUS status = STATUS_SUCCESS;
38     return status;
39 }
```

[1]Note: Of course, another driver might implement a whole custom TCP/IP protocol stack for itself and offer own custom TCP or UDP functions, unseen by TDI.

The code snippet depicts a very simple, minimalistic driver. It is important to note that variables must *always* be declared at the beginning of a function, not later in the code. By using build.exe of the Windows DDK the code can be compiled. The DDK[2] is freely available at Microsoft.

2.3 Compiling and loading a driver

In order to compile any code with the DDK, it is necessary to create a directory containing the source. For example:
”C:\winDDK\myFolder\myFirstDriver”
Besides the file containing the source code, at least two other files are needed: MAKEFILE and SOURCES;

The MAKEFILE only needs one line:

```
1 !INCLUDE $(NTMAKEENV)\makefile.def
```

Whereas a SOURCES file for a driver called MyFirstDriver would look like:

```
1 TARGETNAME=MyFirstDriver
2 TARGETPATH=OBJ
3 TARGETTYPE=DRIVER
4 INCLUDES = $(INCLUDES);
5
6 SOURCES=MyFirstDriver.c
```

After selecting the correct Windows DDK environment and moving into the MyFirstDriver directory, the code is compiled by using the default command ”build -ceZ”.

The compiled driver file has the extension ”.sys”. It must be loaded into memory either by a userspace program or via an .inf file. Creating an .inf file is a complex topic and will not be discussed here. Since much documentation already exists on how to

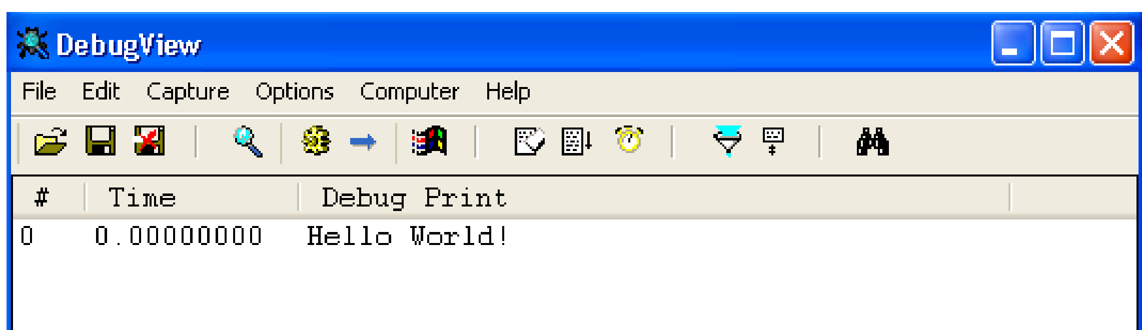
[2] The name DDK changed to WDK since the upcoming of Windows 7. Still, the WDK supports writing drivers for all NT-kernels.

correctly write a driver-loading application, it will not be explained here, either.
For the purpose of learning, a driver-loading application with graphical interface is provided by OSR, a well-known site dedicated to Windows driver development [3], although registering to OSR is needed. Another driver-loading application with provided source code is available at the TU Chemnitz. [4] As a last choice, there exists an application with a graphical interface, which is easy to use. Unfortunately it is only available from a site dedicated to rootkit development. [5]

These tools have in common to load and unload the driver dynamically per default, which means that in case of any serious error, a reboot is sufficient to get rid of the driver. Therefore, the driver must be reloaded explicitly after a reboot by one of these tools. It will not load automatically.

The debug print output can be observed by using DbgView, a freely available tool provided by Microsoft Sysinternals[6].

An example output using Sysinternals DbgView:



[3]http://www.osronline.com/article.cfm?article=157
[4]http://www-user.tu-chemnitz.de/~heha/hs_freeware/gerald.zip
[5]http://www.rootkit.com/vault/hoglund/InstDvr.zip
[6]http://technet.microsoft.com/en-us/sysinternals/bb896647.aspx

Certain particularities exist when writing code for Windows drivers. The main routine is not called *main()*, but *DriverEntry()*. The return value must be a NTSTATUS value, which has the value ”STATUS_SUCCESS” if no problems occurs. A full list of possible STATUS values is available on MSDN. Since the driver operates in kernelmode, there is no *printf* function. All access to the normal Windows API is strictly forbidden and will result in a definite crash. Access to normal userspace memory is forbidden in most cases, too.

It is possible to request direct access to userspace memory though, but this option is a bit risky and should be used with great care.[7]

Now that we presented the structure of a simplistic driver, we can continue with further development. As explained, the driver does not have any device. This is why the default-function, to which the major functions point to, is empty: the driver will not receive any IRPs containing an I/O request.

2.4 Major functions

We previously used the term ’major functions’. The major functions are different functions that will be invoked by other programs.[8] A userspace program can load the driver and demand from Windows to start it. The execution of code will start right in the *DriverEntry*.

[7]By using Neither Buffered Nor Direct I/O method, it is possible to directly access a userspace address, with the limitation that it must be done within the thread-context of the calling user mode thread. Please refer to the MSDN library for further details: `http://MSDN.microsoft.com/en-us/library/cc264614.aspx`

[8]a list of common major functions is listed at MSDN: `http://MSDN.microsoft.com/en-us/library/ms806157.aspx` Note that the list is far from complete and contains only the most used major functions.

After the driver has registered the DriverUnload major function (in the previous code example called ”UnloadDriver”), the user mode program will be able to request from the driver to unload itself. Of course, there are much more major functions, the most obvious being IRP_MJ_READ and IRP_MJ_WRITE. These functions will be executed, when a program requests reading or writing to the driver’s own device. It is important to assign all major functions to corresponding custom functions.

In this most simple example, all major functions (with the exception of the unload function) point the same function: a function that does nothing.

In the next step, we create a simple driver with a custom device. Thus, IRPs containing I/O requests can be sent to this driver in order to invoke the corresponding major function.

Example: A simple driver with a device.

```
1 /*************
2  *  DriverEntry
3  **************/
4 NTSTATUS DriverEntry(
5 PDRIVER_OBJECT  pDriverObject,
6 PUNICODE_STRING  pRegistryPath)
7 {
8    NTSTATUS status = STATUS_SUCCESS;
9    int i = 0;
10   PDEVICE_OBJECT pDeviceObject = NULL;
11   UNICODE_STRING DriverName, DosDeviceName;
12
13    DbgPrint("DriverEntry invoked!\r\n");
14
15    RtlInitUnicodeString(&DriverName, L"\\Device\\MyTestDevice");
16    RtlInitUnicodeString(&DosDeviceName, L"\\DosDevices\\MyTestDevice");
17
18   status = IoCreateDevice(
19               pDriverObject,
20               0,
21               &DriverName,
22               FILE_DEVICE_UNKNOWN,
23               FILE_DEVICE_SECURE_OPEN,
24               FALSE,
25               &pDeviceObject);
26
27   if(status == STATUS_SUCCESS) {
28
```

```
29        for(i = 0; i < IRP_MJ_MAXIMUM_FUNCTION; i++)
30        { pDriverObject->MajorFunction[i] = nothing; }
31
32        pDriverObject->MajorFunction[IRP_MJ_CLOSE] = MyCloseFunction;
33        pDriverObject->MajorFunction[IRP_MJ_CREATE] = MyCreateFunction;
34        pDriverObject->MajorFunction[IRP_MJ_READ] = MyReadFunction;
35        pDriverObject->MajorFunction[IRP_MJ_WRITE] = MyWriteFunction;
36        pDriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = MyIOCTLFunction;
37        pDriverObject->DriverUnload = UnloadDriver;
38
39        IoCreateSymbolicLink(&DosDeviceName, &DriverName);
40    }
41    return status;
42 }
```

We created a device with the name ”MyTestDevice” at Line 18-25. We also created a symbolic link at Line 39, *”\DosDevices\MyTestdevice”*, which is a shortcut that programs can use for accessing the device. It is used as *”\\.\MyTestDevice”*.[9]

As one can see, some major functions are pointing now to different, hopefully useful functions, which will be executed in case of a corresponding IRP-request.

[9] Not: ”\DosDevices\MyTestDevice”

Chapter 3

IRP Handling

IRP handling is of utmost importance when writing kernelmode drivers.

There are two modes for handling an IRP: synchronous and asynchronous.

- **Synchronous** IRPs are processed in a sequential, single-threaded way: Driver A receives the IRP, processes it and sends it down to the next Driver B, which processes it and sends it further down to Driver C. If Driver A has to do a post-processing after Driver C has completed, then Driver A will receive the IRP *after* Driver C has finished.
- **Asynchronous** IRPs can be processed out of order. Sometimes, they are also referred to as ”Nonthreaded IRPs”, because they are not associated to any thread. A driver can start processing an IRP, stop at a later point and begin processing another IRP. To enable other drivers to continue while retaining ownership of the IRP, the driver marks this IRP as pending. A driver receiving a pending IRP must wait until the IRP was completed and the pending bit removed. However the driver is free to continue processing other IRPs.

Depending on whether the IRP is handled synchronously or asynchronously, it is necessary to treat the IRP differently. The probably most important difference is that synchronous IRPs are freed by the I/O manager.[1] Thus, synchronous IRPs should not be freed by a driver.

Synchronous IRPs are created by the I/O manager. If a user mode application requests a certain action to be done, the I/O manager creates a corresponding synchronous IRP.

On the other side, a driver can and should create asynchronous IRPs for performance reasons[2], if possible. The I/O manager does not free such an IRP. The driver *must* free the IRP at the end. Asynchronous IRPs usually increase performance, but they are generally more difficult to handle.

When a driver receives an IRP, there are several possibilities:

- **The IRP requests an operation concerning the drivers own device**. The driver must accept the IRP and process it. When the driver has processed the IRP, it can either free the IRP directly or pass it down. The decision mostly depends on whether the IRP is synchronous or asynchronous.
- **The IRP is interesting and the driver needs to see 'what it contains', but it is destined for other drivers**. This would be a typical forward-and-forget filter driver. Since this article is about TDI, we could imagine an IRP requesting to receive a TCP-packet. Maybe the driver needs to check the ip-address?

[1]This is explained in "Handling IRPs: What Every Driver Writer Needs to Know", section "Nonthreaded IRPs", available at `http://www.microsoft.com/whdc/driver/kernel/irps.mspx`

[2]See "Handling IRPs: What Every Driver Writer Needs to Know", section "IRP as a Thread-Independent Call Stack", available at `http://www.microsoft.com/whdc/driver/kernel/irps.mspx`

Hence, the driver has to take a quick look at the IRP, without changing or modifying the content. Then, the driver passes the unmodified IRP to the next driver in the chain. Afterwards, the driver cannot access the IRP anymore.

- **The driver needs not only to look into the IRP, but also needs to modify it. However, it is still destined for other drivers, too**. If we stay with the TCP-example, the driver might need to modify the IP-address, if certain conditions are met. If the driver needs to process the IRP after all other drivers completed, the driver has to set up a completion routine. In the completion routine, the driver specifies the code that should be executed after the lower drivers completed with processing.

  Generally, the steps in this case are:

  1. Getting a pointer to the IRP stack by invoking *IoGetCurrentIrpStackLocation*
  2. Executing *IoCopyCurrentIrpStackLocationToNext* to copy the IRP data to the stack location of the next lower driver.
  3. Pre-processing the IRP (optional, if needed)
  4. Setting a completion routine (only needed in case of post-processing)
  5. Forwarding the IRP down by invoking *IoCallDriver* (which causes the completion routine to be finally called by the next-lower driver after all lower drivers had the chance to complete.)
  6. Doing post-processing
  7. Finally calling *IoCompleteRequest*

Attention: never access any IRP-related data after having passed the IRP to another driver! The driver loses permission to access an IRP

after having passed it down to the next-lower driver. Trying to access the IRP in such a case will result in a system crash.

3.1 Completion routines

As mentioned in the previous section, the completion routine is registered before forwarding the IRP down to the next-lower drivers. Registering a completion routine is usually done only in case that post-processing is needed. In such a case, the completion routine simply returns STATUS_MORE_PROCESSING_REQUIRED:

An example of a completion routine:

```
1 NTSTATUS ASimpleCompletionRoutine(
2     PDEVICE_OBJECT   DeviceObject,
3     PIRP             Irp,
4     PVOID            Context
5     )
6 {
7     return STATUS_MORE_PROCESSING_REQUIRED;
8 }
```

A comprehensive example that shows how to implement a completion routine is provided by Microsoft: `http://support.microsoft.com/kb/320275/en-us`

Using *IoForwardIrpSynchronously*, the code listed in the Microsoft provided example can be simplified. If possible, *IoForwardIrpSynchronously* should be preferred. An example, how a filter driver should correctly use *IoForwardIrpSynchronously* for post-processing, is shown in the following dispatch routine.

Example dispatch routine for post-processing of an IRP:

```
1  // The full dispatch routine to which
2  // a major function of a filter driver might point to.
3  NTSTATUS DispatchRoutineWithPostProcessing (
4                PDEVICE_OBJECT DeviceObject, PIRP Irp) {
5
6      PIO_STACK_LOCATION StackIrpPointer = NULL;
7      NTSTATUS ntStatus=STATUS_SUCCESS;
8      PEXTENSION_OBJECT DeviceContext =
9            (PEXTENSION_OBJECT)DeviceObject->DeviceExtension;
10
11     DbgPrint("Main dispatch routine called \r\n");
12     StackIrpPointer = IoGetCurrentIrpStackLocation (Irp);
13
14     // Sanity check: StackIrpPointer is never zero!
15     if (StackIrpPointer) {
16
17         // Check if this is the correct device.
18         // In this example, the driver is only interested
19         // in the device \Device\DeviceToBeFiltered.
20         if (DeviceObject == L"\\Device\\DeviceToBeFiltered") {
21
22             // For synchronous forwarding of IRPs,
23             // IoForwardIrpSynchronously is the better choice!
24             if (IoForwardIrpSynchronously(
25                         DeviceContext->TopOfDeviceStack,
26                         Irp)) {
27
28                 DbgPrint("forwarding done! \r\n");
29
30                 // -------------------------
31                 // This is the place for doing post-processing.
32                 // -------------------------
33
34                 // We simply return STATUS_SUCCESS here.
35                 // Instead of returning STATUS_SUCCESS, the post-processing
36                 // code should assign the adequate value.
37
38                 ntStatus=STATUS_SUCCESS;
39
40                 // Here, we signalize to the I/O manager that
41                 // we finished processing.
42                 // Thus, the I/O manager can continue with invoking
43                 // the completion routines of higher drivers
44                 // to complete processing.
45                 IoCompleteRequest(Irp,IO_NO_INCREMENT);
46             }
47         }
48     }
49
50     return ntStatus;
51 }
```

With the definition of the device extension being:

```
1 // This globally declared struct is used for remembering
2 // the location of the device in the stack.
3 typedef struct _EXTENSION {
4     PDEVICE_OBJECT TopOfDeviceStack;
5 } EXTENSION_OBJECT, *PEXTENSION_OBJECT;
```

Using *IoForwardIrpSynchronously* is equivalent to following steps:

- Copying the IRP data to the next-lower driver by invoking *IoCopyCurrentIrpStackLocationToNext*
- Setting a completion routine
- Forwarding the IRP down by invoking *IoCallDriver* and waiting for callback in order to do post-processing.

Thus, *IoForwardIrpSynchronously* simplifies the code while being very functional and effective. The only problem is that it applies only to synchronous IRPs, such as IRPs generated by the I/O manager on behalf of a user-application.

In some cases, the completion routine also contains code to free the IRP. Freeing an IRP is usually done in the completion routine.[3] It is obligatory, if the IRP was created by using *IoAllocateIrp* or *IoBuildAsynchronousFsdRequest*. Such an IRP is handled asynchronously.[4]

An example completion routine for freeing an IRP is as follows:

```
1 NTSTATUS CompletionRoutineWithFreeIRP (
2 PDEVICE_OBJECT     DeviceObject,
3 PIRP          Irp,
4 PVOID         Context)
5 {
6     DbgPrint("Freeing IRP!\r\n");
7
8     IoFreeIrp(Irp);
9     return STATUS_MORE_PROCESSING_REQUIRED;
10 }
```

[3] `http://msdn.microsoft.com/en-us/library/ms801629.aspx`

[4] It might be contrary to any official documentation, but it is actually possible to free any IRP in an absolutely arbitrary way without having to call lower drivers. It also works for synchronous handled IRPs. A practical implementation of this trick is presented in the last chapter ”TDI”, section ”Freeing IRPs”.

In this example completion routine, the IRP is freed at Line 8. The completion routine returns STATUS_MORE_PROCESSING_REQUIRED to continue in the main dispatch routine afterwards. At the end of the dispatch routine, the driver must not call *IoCompleteRequest* here since it already freed the IRP.

It is strongly recommended to read the following comprehensive code examples provided by Microsoft for understanding how to treat synchronous and asynchronous IRPs:
`http://support.microsoft.com/kb/326315/en-us/`

When a driver invokes *IoCompleteRequest*, the completion routine of each higher driver in the chain that registered a completion routine is called. If such a higher driver returns STATUS_MORE_PROCESSING_REQUIRED, completion is stopped and the associated driver gets the chance to do further processing in its dispatch routine. At the end, this driver must call *IoCompleteRequest*, because it interrupted the process of completion. Calling *IoCompleteRequest* triggers continuation of the next-higher driver completion routines. The completion continues until the highest driver completed. A driver's own completion routine is invoked by the lower driver in the chain.[5]

3.2 Cancel routines

Another option can be to cancel the IRP by invoking *IoCancelIrp*. There exist several reasons for doing so. An example would be an IRP, that was queued by a driver and was left in a pending state, while the user mode program, that initially raised the request,

[5] The lowest driver at the bottom of the chain does not have a completion routine.

terminated unexpectedly.

A cancelled IRP will stop being processed by other drivers. Not all IRPs are cancellable, though. A device driver can specify a cancel routine for IRPs concerning its own device to make them cancellable. Without a cancel routine, the cancelled-flag in the IRP is ignored. If any driver wants to cancel the IRP of another driver, it should first check if this IRP is cancellable. Typically, cancelling an IRP is done by the driver to which the IRP belongs or by the I/O manager. However, cancelling an IRP is a legitimate action for any driver and can be invoked at any place. The originating driver must be prepared for reacting in an adequate way. This can be very difficult, especially for asynchronous handled IRPs. Furthermore, a cancelled IRP is still a valid IRP. As a consequence, a cancelled IRP must still be completed. For writing TDI filter driver, setting up an own cancel routine option has not much importance.[6] Also, most TDI requests are issued by the I/O manager and cancelling them seems to have no effect.

This was a short overview of the possibilities concerning the question what to do with an IRP. We will use these possibilities more detailed in several practical TDI examples.

[6]A detailed description is provided by Microsoft at http://download.microsoft.com/download/5/7/7/577a5684-8a83-43ae-9272-ff260a9c20e2/cancel_logic.doc

Chapter 4

TDI

Since this paper is about writing a TDI filter driver, we normally do not wish to process and finish IRPs ourselves. This is already done by the actual low-level drivers. In our first approach, we just wish to look at the IRPs without modifying them, and pass them to the next lower driver. There is not much need for concern here, since we do not really interact with the IRP yet.

Before going deeper into the details of the various TDI data structs, we begin with a small, but functional example. It follows the ”Forward and Forget” principle. The IRP is never accessed.

4.1 A simple Forward-and-Forget filter driver

Since TCP is probably the most common protocol used in practice, we will attach to the TCP device and catch all attempts to connect via the TCP-protocol using TDI. Note that attaching to a device is a very common procedure. It is not restricted to TDI. Any device is possible.

TDI_CONNECT is not the only possible TDI function, but probably the most easiest to comprehend. It is the reason why we chose this function for demonstration purposes. The TDI API

provides several types of requests. They are defined in the *tdikrnl.h*, as listed in the following:

```
1  #define TDI_ASSOCIATE_ADDRESS     (0x01)
2  #define TDI_DISASSOCIATE_ADDRESS  (0x02)
3  #define TDI_CONNECT               (0x03)
4  #define TDI_LISTEN                (0x04)
5  #define TDI_ACCEPT                (0x05)
6  #define TDI_DISCONNECT            (0x06)
7  #define TDI_SEND                  (0x07)
8  #define TDI_RECEIVE               (0x08)
9  #define TDI_SEND_DATAGRAM         (0x09)
10 #define TDI_RECEIVE_DATAGRAM      (0x0A)
11 #define TDI_SET_EVENT_HANDLER     (0x0B)
12 #define TDI_QUERY_INFORMATION     (0x0C)
13 #define TDI_SET_INFORMATION       (0x0D)
14 #define TDI_ACTION                (0x0E)
15 // There is space left at this position,
16 // the function numbers continue at 0x27.
17
18 #define TDI_DIRECT_SEND           (0x27)
19 #define TDI_DIRECT_SEND_DATAGRAM  (0x29)
20 #define TDI_DIRECT_ACCEPT         (0x2A)
```

For the moment, we are mainly interested in function 0x3, TDI_CONNECT.

First, we need to attach to the device we wish to filter. This should be done in the *DriverEntry*:

```
1 // Again, the global definition of our Device Extension struct:
2 typedef struct _EXTENSION {
3     PDEVICE_OBJECT TopOfDeviceStack;
4 } EXTENSION_OBJECT, *PEXTENSION_OBJECT;
```

```
1  NTSTATUS DriverEntry(
2        PDRIVER_OBJECT DriverObject,
3        PUNICODE_STRING RegistryPath)
4  {
5     NTSTATUS ntStatus = STATUS_SUCCESS;
6     PEXTENSION_OBJECT DeviceExtension;
7     UNICODE_STRING FilteredDeviceName;
8     unsigned int i = 0;
9
10    DbgPrint("DriverEntry was called!\n");
11
12    // Initialize a device
13    ntStatus = IoCreateDevice(
```

```
14          DriverObject,
15          sizeof(EXTENSION_OBJECT),
16          NULL,
17          FILE_DEVICE_NETWORK,
18          FILE_DEVICE_SECURE_OPEN,
19          FALSE,
20          &DeviceToBeFiltered);
21
22    // Check for failure
23    if(!NT_SUCCESS(ntStatus)) {
24    // We did not attach yet. Trying to detach
25    // would result in a system crash.
26          IoDeleteDevice(DeviceToBeFiltered);
27          DbgPrint("Attaching to TCP device failed! \r\n");
28       }
```

As shown at Line 13-20, first a device with the same device characteristics as the TCP device must be initialized, before we can attach to the device.

```
1    // We can proceed with attaching to the device.
2    else {
3
4    // The device extension is used for identifying the device in the stack.
5    // We use the same struct listed in the previously shown example.
6    DeviceExtension =
7          (PEXTENSION_OBJECT)DeviceToBeFiltered->DeviceExtension;
```

Before actually attaching to the TCP device via *IoAttachDevice*, the major functions should be initialized first. The reason for this arrangement is that two major functions are invoked by *IoAttachDevice*: *MJ Close* and *MJ Cleanup*. They correspond to the actions of the I/O manager, since the I/O manager has to access the TCP device and attach it to the filter driver. It does not seem to be absolutely necessary, though. It is just a convention and in our example, both functions have no actual code, anyway. Here is a good point to initialize the major functions:

```
1    // This simple example driver uses only one major function.
2    for(i = 0; i < IRP_MJ_MAXIMUM_FUNCTION; i++)
3       { DriverObject->MajorFunction[i] = NotSupported; }
4
5    // TDI uses the IRP_MJ_INTERNAL_DEVICE_CONTROL major function.
6    DriverObject->MajorFunction[IRP_MJ_INTERNAL_DEVICE_CONTROL] = Dispatch;
7
8    DriverObject->DriverUnload = UnloadDriver;
```

After initialization of the major functions, the TCP device is attached to our device. In order to do this, a unicodestring containing the path of the TCP device must be passed to *IoAttachDevice*.

```
1    // RtlInitUnicodeString is a very commonly used function
2    // to convert Strings to unicode format.
3     RtlInitUnicodeString(&FilteredDeviceName, L"\\Device\\Tcp");
```

Finally, we can attach to the TCP device.

```
1    // Attach to device
2     ntStatus = IoAttachDevice(
3                DeviceToBeFiltered,
4                &FilteredDeviceName,
5                &DeviceExtension->TopOfDeviceStack);
6
7    // Again, never forget to check for failure
8     if(!NT_SUCCESS(ntStatus)) {
9            UnloadDriver(DriverObject);
10         DbgPrint("Attaching to TCP device failed! \r\n");
11        }
12    }
13
14  return ntStatus;
15 }
```

This is all that has to be done in the *DriverEntry* routine.

When we invoked the instruction '*DeviceExtension = (PEXTENSION_OBJECT)DeviceToBeFiltered->DeviceExtension;*', we created a global pointer to the device object. This allows to access the TCP device object from within any function. Therefore, the global device object is declared *outside* of the *DriverEntry*:

```
1 // A global pointer to the filtered \Device\Tcp
2 PDEVICE_OBJECT DeviceToBeFiltered;
```

Since dispatch routines will access the filtered device regularly, this is very practical. Another important point is to continually check the NTSTATUS, since an error can always occur at any place.

A very important yet short function is the unload routine. Two steps are necessary:

- Detach the device from the filtered device
- Delete the device

The following unload routine realizes both instructions:

```
1  VOID UnloadDriver(PDRIVER_OBJECT  DriverObject)
2  {
3
4      PEXTENSION_OBJECT DeviceExtension =
5           (PEXTENSION_OBJECT)DriverObject->DeviceObject->DeviceExtension;
6
7      DbgPrint("Unloading Driver! \r\n");
8
9      // Detach the device before deleting it.
10     IoDetachDevice(DeviceExtension->TopOfDeviceStack);
11     IoDeleteDevice(DriverObject->DeviceObject);
12 }
```

In this code example, detaching is done at Line 10, whereas deleting the device occurs at Line 11. The Unload routine is a void function, it does not return any NTSTATUS-value.

A dummy routine will be assigned to unused major functions. It will forward all IRPs to the next-lower driver:

```
1  NTSTATUS NotSupported(PDEVICE_OBJECT DeviceObject, PIRP Irp)
2  {
3  // This function does nothing but forwarding IRPs to other drivers.
4      NTSTATUS ntStatus = STATUS_SUCCESS;
5      PEXTENSION_OBJECT DeviceExtension =
6        (PEXTENSION_OBJECT)DeviceObject->DeviceExtension;
7
8      IoSkipCurrentIrpStackLocation(Irp);
9      ntStatus = IoCallDriver(DeviceExtension->TopOfDeviceStack, Irp);
10
11     return ntStatus;
12 }
```

The routine simply forwards all IRPs using any other major functions than IRP_MJ_INTERNAL_DEVICE_CONTROL to the next-lower driver.

We will now instruct the dispatch routine, our main function, to simply fish for any TDI_CONNECT requests and to print them via debug print. Furthermore, we will print the corresponding

function codes of all other invoked TDI requests. Looking at the list of TDI functions, it is trivial to get the corresponding TDI function.

A dispatch routine of a minimalistic TDI filter driver:

```
1 NTSTATUS Dispatch(PDEVICE_OBJECT DeviceObject,PIRP Irp) {
2
3    NTSTATUS ntStatus=STATUS_NOT_SUPPORTED;
4    PEXTENSION_OBJECT DeviceExtension =
5          (PEXTENSION_OBJECT)DeviceObject->DeviceExtension;
6    PIO_STACK_LOCATION StackIrpPointer = NULL;
7
8    // Obtain a pointer to the corresponding stack in the IRP.
9    StackIrpPointer = IoGetCurrentIrpStackLocation (Irp);
10
11   // Sanity check
12   // StackIrpPointer is never zero!
13   if (!StackIrpPointer) {
14         DbgPrint("Fatal Error: IRP stack pointer is NULL! \r\n");
15         return STATUS_UNSUCCESSFUL;
16      }
17
18   // Sanity check
19   // Since we attached to the TCP device, we can only
20   // receive IRPs concerning this device.
21   if (DeviceObject == DeviceToBeFiltered) {
22
23      // We are interested in the TDI_CONNECT Function
24      if (StackIrpPointer->MinorFunction == TDI_CONNECT) {
25         DbgPrint("A TDI connect (Function 3) via TCP was invoked!\r\n");
26      }
27
28      // case: it is another TDI function...
29      else {
30      // ..so we simply forward the IRP down.
31
32      // However, before forwarding the IRP, we print the observed
33      // TDI function as hexadecimal number (as defined in tdikrnl.h):
34      DbgPrint("TDI Request observed: function code %x \r\n",
35               StackIrpPointer->MinorFunction);
36
37      // Additionally, we can print the size of TCP data packet sent by
38      // TDI-using applications.
39      // The according TDI function code is code 7.
40      if (StackIrpPointer->MinorFunction == TDI_SEND) {
41         DbgPrint("Application sent a TCP-Data packet of size %d.\r\n",
42                  StackIrpPointer->
43                     Parameters.DeviceIoControl.OutputBufferLength);
44      }}
45
46      // Since we do not wish to modify the IRP, we finally forward it
47      // down to the next-lower driver.
48      IoSkipCurrentIrpStackLocation(Irp);
49      ntStatus=IoCallDriver(DeviceExtension->TopOfDeviceStack, Irp);
50   }
```

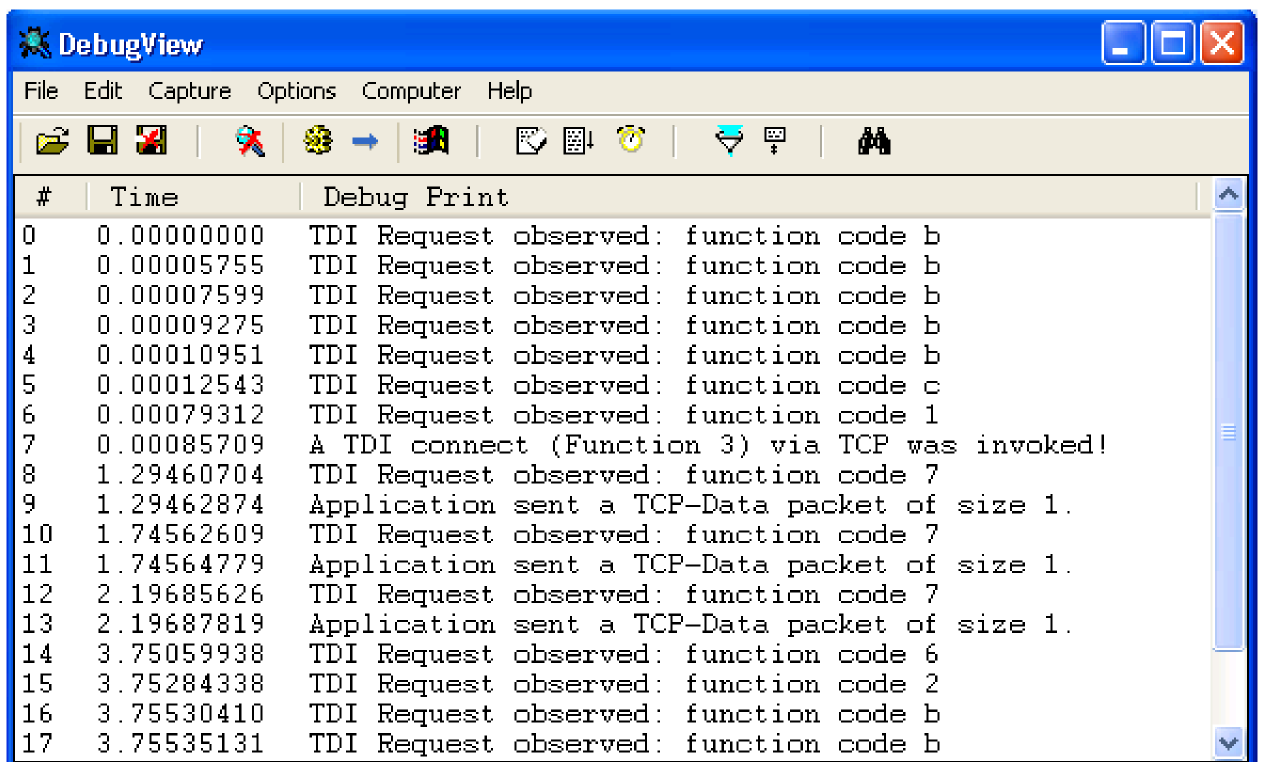
```
51     return ntStatus;
52 }
```

As for the headers, a TDI filter driver usually needs the following:

```
1 #include <wdm.h>   // Needed for any wdm-based driver!
2 #include <tdikrnl.h>
3 #include <tdi.h>
```

We now have a full, working filter driver. It detects all attempts to access the TCP device via TDI and allows to determine the exact TDI function.

An example output produced by the simple TDI filter in Debug View:



For testing the filterdriver, a very simple test method is to use the native hyperterminal (hypertrm.exe), which provides a

comfortable graphical interface, or telnet (telnet.exe). The advantage of using those native tools is the simplicity. Especially, the connection can be controlled very well: when issuing a 'disconnect', the disconnect command is executed *at once* and without delay. This is not trivial. When using Firefox or the Internet Explorer, many connections will open automatically and these connections will remain for several minutes, even if the application was completely closed.

4.2 Overview of the essential TDI structs

The next interesting point is to actually access the TDI data and extract the information. However, there are various structs used by TDI, which we will inevitably encounter when writing a TDI filter driver. Obtaining an overview of all these structs is somewhat difficult and may be confusing at first. This is why a complete list is provided in the following. However, probably only the last struct contains useful information for a filter driver.

When receiving an IRP with TDI data, the parameters at the IRP stack location point to a TDI_REQUEST_KERNEL_CONNECT struct.

- **struct TDI_REQUEST_KERNEL_CONNECT**

  Actually, this struct uses a generic TDI_REQUEST_KERNEL struct. It is defined in tdikrnl.h as:

```
1 typedef struct _TDI_REQUEST_KERNEL {
2     ULONG_PTR RequestFlags;
3     PTDI_CONNECTION_INFORMATION RequestConnectionInformation;
4     PTDI_CONNECTION_INFORMATION ReturnConnectionInformation;
5     PVOID RequestSpecific;
6 } TDI_REQUEST_KERNEL, *PTDI_REQUEST_KERNEL;
```

- **struct TDI_CONNECTION_INFORMATION**

As listed above, the TDI_REQUEST_KERNEL struct contains a field named ”RequestConnectionInformation” pointing to a TDI_CONNECTION_INFORMATION struct.

The TDI_CONNECTION_INFORMATION is defined in tdi.h as:

```
1 typedef struct _TDI_CONNECTION_INFORMATION {
2    // length of user data buffer
3     LONG UserDataLength;
4     // pointer to user data buffer
5     __field_bcount(UserDataLength) PVOID UserData;
6     // length of following buffer
7     LONG OptionsLength;
8     // pointer to buffer containing options
9     __field_bcount(OptionsLength) PVOID Options;
10    // length of following buffer
11    LONG RemoteAddressLength;
12    // buffer containing the remote address
13    __field_bcount(RemoteAddressLength) PVOID RemoteAddress;
14 } TDI_CONNECTION_INFORMATION, *PTDI_CONNECTION_INFORMATION;
```

The void pointer ”RemoteAddress” actually contains a buffer which is convertible to the struct TRANSPORT_ADDRESS.

- **struct TRANSPORT_ADDRESS**

  The TRANSPORT_ADDRESS is defined as:

```
1 typedef struct _TRANSPORT_ADDRESS {
2    // number of addresses following
3     LONG TAAddressCount;
4     // actually TAAddressCount elements long
5    TA_ADDRESS Address[1];
6 } TRANSPORT_ADDRESS, *PTRANSPORT_ADDRESS;
```

  Therefore, it contains a pointer to a TA_ADDRESS struct.

- **struct TA_ADDRESS**

  The TA_ADDRESS is defined as:

```
1 typedef UNALIGNED struct _TA_ADDRESS {
2    // length in bytes of Address[]
3     USHORT AddressLength;
4     // type of this address
5     USHORT AddressType;
```

```
6     // address information
7     UCHAR Address[1];
8 } TA_ADDRESS, *PTA_ADDRESS;
```

The field "Address" contains a pointer to a buffer with actual address information. In case of the TCP/IP protocol, it is convertible to a TDI_ADDRESS_IP struct, which contains IP address and port.

- **struct TDI_ADDRESS_IP**

  Finally, this is the last struct, and it is the struct we are interested in. As explained, this struct contains the IP address and the port when TCP/IP protocol is used. (Do not forget that TDI also supports other protocol stacks, such as IPX.)

  The TDI_ADDRESS_IP is defined as:

```
1 typedef struct _TDI_ADDRESS_IP {
2     USHORT sin_port;
3     ULONG  in_addr;
4     UCHAR  sin_zero[8];
5 } TDI_ADDRESS_IP, *PTDI_ADDRESS_IP;
```

  Further TDI_ADDRESS_XXX structs for other protocol stacks are defined in tdi.h. For example, IPv6 is likewise supported by TDI:

```
1 typedef struct _TDI_ADDRESS_IP6_XP {
2     USHORT sin6_port;
3     ULONG  sin6_flowinfo;
4     USHORT sin6_addr[8];
5     ULONG  sin6_scope_id;
6 } TDI_ADDRESS_IP6_XP, *PTDI_ADDRESS_IP6_XP;
```

Pay attention to the fact that for Windows 2000, another TDI IPv6 struct is used:

```
1 typedef struct _TDI_ADDRESS_IP6_WIN2K {
2     USHORT sin6_port;
3     ULONG  sin6_flowinfo;
4     UCHAR  sin6_addr[16];
5 } TDI_ADDRESS_IP6_WIN2K, *PTDI_ADDRESS_IP6_WIN2K;
```

Therefore, Windows 2000 is missing the sin6_scope_id field.

Using this knowledge, we can construct an improved filter driver.

4.3 An improved Forward-and-Forget filter driver

In our next version of a TDI filter driver, we wish to access the actual data contained in the IRP. It still follows the forward-and-forget principle, though. We do not modify any data in the IRP.

It is only the dispatch routine that has to be modified. Especially, we need to access the TDI_ADDRESS_IP struct. As shown in the last section, it contains the remote IP address and port of the TCP connect that was initiated by a TDI-using application.

The first step is to access the IRP stack parameters and convert the data it contains into a pointer to a TDI_REQUEST_KERNEL_CONNECT struct.

Since TDI_CONNECTION_INFORMATION contains a void pointer instead of a pointer to a TRANSPORT_ADDRESS struct, an explicit typecast to PTRANSPORT_ADDRESS is necessary. It cannot be accessed directly. Furthermore, TA_ADDRESS contains a pointer to UCHAR. This buffer can be typecasted into TDI_ADDRESS_IP format.

In the following example, we will do these three typecasting steps explicitly. Of course, it should be noted that it is neither necessary nor should someone create three different pointer to structs containing similar data. However, in order to provide a understandable example, too much typecasting in one instruction should be avoided.

The improved code of the dispatch routine is as follows:

```
1  NTSTATUS Dispatch(PDEVICE_OBJECT DeviceObject,PIRP Irp) {
2
3     NTSTATUS ntStatus=STATUS_NOT_SUPPORTED;
4     PEXTENSION_OBJECT DeviceExtension =
5           (PEXTENSION_OBJECT)DeviceObject->DeviceExtension;
6     PIO_STACK_LOCATION StackIrpPointer = NULL;
7
8     // The first TDI related struct we encounter...
9     PTDI_REQUEST_KERNEL_CONNECT TDI_connectRequest;
10
11    // ...followed by further structs
12    PTA_ADDRESS TA_Address_data;
13
14    // We are mainly interested in this final struct.
15    PTDI_ADDRESS_IP TDI_data;
```

We will use a custom struct, in which we store relevant data. IP address and port will be sufficient in this example. In a real example, the struct should be declared as a global object to provide fast access from within other routines. Planning adequate memory structures and considering performance is of great importance when writing kernelmode drivers, since a driver does not stop running in kernelspace, whereas an application usually ends after some time.

```
1     // A struct for storing data.
2     typedef struct _NETWORK_ADDRESS
3     {
4        unsigned char address[4];
5        unsigned char port[2];
6     } NETWORK_ADDRESS;
7
8     NETWORK_ADDRESS data;
9
10    unsigned short Port=0;
11    unsigned long Address=0;
```

```
12
13    // Obtain a pointer to the corresponding stack in the IRP.
14    StackIrpPointer = IoGetCurrentIrpStackLocation (Irp);
15
16    // Sanity check
17    // StackIrpPointer is never zero!
18    if (!StackIrpPointer) {
19          DbgPrint("Fatal Error: IRP stack pointer is NULL! \r\n");
20          return STATUS_UNSUCCESSFUL;
21       }
22
23    // Sanity check
24    // Since we attached to the TCP device, we can only
25    // receive IRPs concerning this device.
26    if (DeviceObject == DeviceToBeFiltered) {
27
28       // For the moment we are only interested in the TDI_CONNECT Function.
29       if (StackIrpPointer->MinorFunction == TDI_CONNECT) {
30          DbgPrint("A TDI connect (Function 3) via TCP was invoked!\r\n");
```

Since this is supposed to be a comprehensive example, we split the process of typecasting and extracting the final TDI_ADDRESS_IP struct into several steps. We only use the last struct, the TDI_ADDRESS_IP.

```
1 // Step I: Fill the PTDI_REQUEST_KERNEL_CONNECT with data.
2       TDI_connectRequest =
3             (PTDI_REQUEST_KERNEL_CONNECT)
4                (&StackIrpPointer->Parameters);
5
6 // Step II: Fill the PTA_ADDRESS with the data
7 // contained in the TDI Connect Request.
8       TA_Address_data =
9       // An explicit typecast is necessary here,
10      // since RemoteAddress is a void pointer!
11         ( (PTRANSPORT_ADDRESS)
12           (TDI_connectRequest->RequestConnectionInformation->RemoteAddress))
13            ->Address;
14
15 // Step III: Finally copying the data to a PTDI_ADDRESS_IP.
16 // Again, the field Address in TA_ADDRESS is a pointer to an UCHAR.
17 // It must be typecasted to a TDI_ADDRESS_IP pointer explicitly.
18      TDI_data = (PTDI_ADDRESS_IP) (TA_Address_data->Address);
```

We extracted and stored all data at this point. Concerning the processing of an IRP, the general rule is: be as quick as possible!

Thus, the processing of data can wait.

```
1       // Forwarding the IRP down to the next-lower driver:
2       IoSkipCurrentIrpStackLocation(Irp);
3       ntStatus=IoCallDriver(DeviceExtension->TopOfDeviceStack, Irp);
```

```
4
5       // The IRP is gone, but we stored all data.
6
7       Address = TDI_data->in_addr;
8       Port = TDI_data->sin_port;
9
10      data.address[0] = ((char *)&Address)[0];
11      data.address[1] = ((char *)&Address)[1];
12      data.address[2] = ((char *)&Address)[2];
13      data.address[3] = ((char *)&Address)[3];
14
15      data.port[0] = ((char *)&Port)[0];
16      data.port[1] = ((char *)&Port)[1];
17      Port = data.port[0] + data.port[1];
18
19      DbgPrint("TCP address is %d.%d.%d.%d:%d \r\n",
20         data.address[0],data.address[1],data.address[2],data.address[3],
21         Port);
22      }
23
24      // case: it is another TDI function.
25      else {
26      // After printing the function code and data size,
27      // we simply forward the IRP down.
28      DbgPrint("TDI Request observed: function code %x \r\n",
29               StackIrpPointer->MinorFunction);
30      if (StackIrpPointer->MinorFunction == TDI_SEND) {
31         DbgPrint("Application sent a TCP-Data packet of size %d.\r\n",
32                  StackIrpPointer->
33                     Parameters.DeviceIoControl.OutputBufferLength);
34      }
35      IoSkipCurrentIrpStackLocation(Irp);
36      ntStatus=IoCallDriver(DeviceExtension->TopOfDeviceStack, Irp);
37      }
38    }
39    return ntStatus;
40 }
```

The improved filter driver now prints the IP address and the port of the TCP connect initialized by a TDI-using application. It also prints the data of the TCP packet each time TCP data is sent to a remote host. Likewise, the content of a sent TCP packet could be read easily.

We discussed Forward-and-Forget filter drivers by now. The reader should be able to write a fully working TDI filter driver. In the next section, we will discuss how to write a TDI filter driver, which not only reads the TDI data, but also modifies the data contained in the IRP.

4.4 A redirecting TDI filter driver

When modifying TCP data from within the kernel, the problem is often to see if the modification was successful. Therefore, a way to observe the effect caused by the filter driver is needed.

To create a scenario, in which the effect of having successfully modified the data can be observed, we imagine that an application tries to connect to a remote server in the network. Furthermore, we imagine certain important IP addresses to be not available (due to technical problems, for example).
Since it would not make any sense to connect to one of these IP addresses, we will write a TDI filter driver, which is able to redirect the attempt of the application to connect via TCP on the fly, if it matches certain IP addresses. Instead of connecting to the original IP addresses, the application will connect to another IP address provided by our filter driver.

For the following example filter driver, the concerned IP addresses are 10.0.0.88 and 10.0.0.99. To provide a more realistic scenario, we might imagine both IP addresses being important servers (a main server and a secondary server for load distribution). Furthermore, we imagine that a serious error occurred and caused both servers to be suddenly down and unavailable. Since the clients heavily rely on the operational availability of these servers, a new alternative server is needed urgently. In this scenario, a manual reconfiguration of the clients to an alternative server would take too much time.

In order to provide the possibility to automatically switch to an emergency server in case of any failure, a mechanism to automatically redirect connection attempts to an emergency server would be needed. A TDI filter driver offers this possibility.

We will present such a redirecting emergency filter driver in a code example. It will simply redirect any connection attempts to the main servers to an auxiliary emergency system. Hence, to see any effect, a corresponding, responding server with IP address 10.0.0.62 should be set up. Of course, there is no need to set up a server with an IP address matching 10.0.0.88 or 10.0.0.99. This is possible, because the IRP requesting the TCP connect occurs *before* an actual tcp packet is sent, since it is the task of the network drivers to process the IRP and finally send data to the network. Thus, when launching a connect to 10.0.0.88, a fast redirection to 10.0.0.62 is guaranteed.

Again, we only modify our main dispatch routine. Besides small code additions, some rearrangement is needed.

Code of a modifying example TDI filter driver:

```
1 NTSTATUS Dispatch(PDEVICE_OBJECT DeviceObject,PIRP Irp) {
2
3    NTSTATUS ntStatus=STATUS_NOT_SUPPORTED;
4    PEXTENSION_OBJECT DeviceExtension =
5          (PEXTENSION_OBJECT)DeviceObject->DeviceExtension;
6    PIO_STACK_LOCATION StackIrpPointer = NULL;
7
8    // The TDI structs
9    PTDI_REQUEST_KERNEL_CONNECT TDI_connectRequest;
10   PTA_ADDRESS TA_Address_data;
11   PTDI_ADDRESS_IP TDI_data;
12
13   // A struct for storing data.
14   typedef struct _NETWORK_ADDRESS
15   {
16      unsigned char address[4];
17      unsigned char port[2];
18
19   } NETWORK_ADDRESS;
20
21   NETWORK_ADDRESS data;
22
23   unsigned short Port=0;
24   unsigned long Address=0;
25
26   // Obtain a pointer to the corresponding stack in the IRP.
27   StackIrpPointer = IoGetCurrentIrpStackLocation (Irp);
28
29   // Sanity check
30   // StackIrpPointer is never zero!
```

```
31    if (!StackIrpPointer) {
32          DbgPrint("Fatal Error: IRP stack pointer is NULL! \r\n");
33          return STATUS_UNSUCCESSFUL;
34       }
35
36    // Sanity check
37    // Since we attached to the TCP device, we can only
38    // receive IRPs concerning this device.
39    if (DeviceObject == DeviceToBeFiltered) {
40
41       // For the moment we are only interested in the TDI_CONNECT Function
42       if (StackIrpPointer->MinorFunction == TDI_CONNECT) {
43          DbgPrint("\r\nA TDI connect (Function 3) via TCP was invoked!\r\n");
44
45       // Step I: Fill the PTDI_REQUEST_KERNEL_CONNECT struct with data.
46       TDI_connectRequest =
47             (PTDI_REQUEST_KERNEL_CONNECT)
48                (&StackIrpPointer->Parameters);
49
50       // Step II: Fill the TA_ADDRESS struct with the data
51       // contained in the TDI Connect Request.
52       TA_Address_data =
53          ( (PTRANSPORT_ADDRESS)
54            (TDI_connectRequest->RequestConnectionInformation->RemoteAddress))
55             ->Address;
56
57       // Step III: Finally copying the data to the TDI_ADDRESS_IP struct.
58       TDI_data = (PTDI_ADDRESS_IP) (TA_Address_data->Address);
```

All data was extracted and stored at this point. Since we need to know the IP address *now*, we must process the information without delay. Depending on whether it is one of the two IP addresses, the code must react differently.

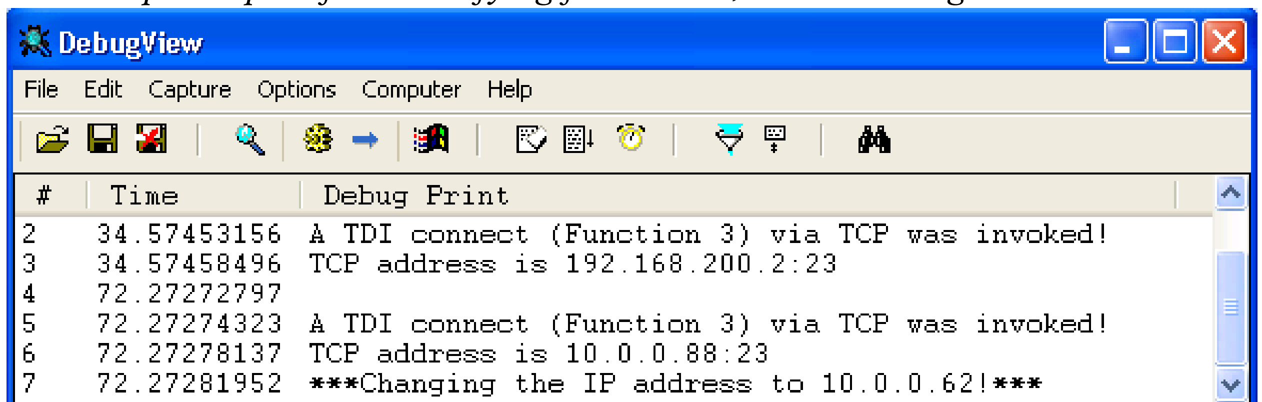
```
1        Address = TDI_data->in_addr;
2        Port = TDI_data->sin_port;
3
4        data.address[0] = ((char *)&Address)[0];
5        data.address[1] = ((char *)&Address)[1];
6        data.address[2] = ((char *)&Address)[2];
7        data.address[3] = ((char *)&Address)[3];
8
9        data.port[0] = ((char *)&Port)[0];
10       data.port[1] = ((char *)&Port)[1];
11       Port = data.port[0] + data.port[1];
12
13       DbgPrint("TCP address is %d.%d.%d.%d:%d \r\n",
14          data.address[0],data.address[1],data.address[2],data.address[3],
15          Port);
```

The driver should only redirect the connect if the IP address matches one of the failing servers. Hence, a check is issued:

```
1        if ((data.address[0]==10) &&
2           (data.address[1]==0) &&
3           (data.address[2]==0)) {
4           // The last number decides:
5           if ((data.address[3]==88) || (data.address[3]==99)) {
6              DbgPrint("***Changing the IP address to 10.0.0.62!***\r\n");
7
8              // Change the IP address!
9              ( (PTDI_ADDRESS_IP)
10             (( (PTRANSPORT_ADDRESS)
11            (TDI_connectRequest->RequestConnectionInformation->RemoteAddress)
12                )->Address->Address) )->in_addr=0x3e00000a; // little-endian
13
14          }
15       }
16
17       }
18
19       // Forward the IRP down to next-lower driver.
20       IoSkipCurrentIrpStackLocation(Irp);
21       ntStatus=IoCallDriver(DeviceExtension->TopOfDeviceStack, Irp);
22    }
23    return ntStatus;
24 }
```

This code example provides a filter driver able to automatically redirect a connect, if certain conditions are met. As soon as a server with IP address 10.0.0.62 is available, *any* application trying to connect to either 10.0.0.88 or 10.0.0.99 will correctly connect to IP address 10.0.0.62 instead. This will be the case even if no server with IP address 10.0.0.88 or 10.0.0.99 exists.

An example output of the modifying filter driver, as seen in Dbgview:



Of course, the redirection can additionally be restricted to certain port numbers, or a port redirection could be implemented. There are much possibilities regarding manipulating IRP data. The same applies to UDP. The corresponding device would be *"\Devices\Udp"*.

4.5 Countermeasures against IRP manipulation in kernelspace

Manipulation of IRP data might appear useful for bridging connection-related technical problems, but it could be misused for malicious purposes. Fortunately, such an attempt can be detected using the same TDI techniques.

Until now, the presented examples always pre-processed the IRP and forwarded it down to the next-lower driver. This means: any other driver could have changed the IP address likewise. Since the scenario of a malicious filter driver changing known IP addresses of online banking sites to a custom phishing site is not unthinkable, a verifying IP address filter driver would be useful.

Such a verifying filter driver has to check the IP address as soon as it receives the IRP. It then forwards it down to the other drivers either by setting up a completion routine or by using *IoForwardIrpSynchronously*. After all drivers finished processing the IRP, the IRP returns back to the verifying filter driver. Thus, it can now check the current IP address contained in the IRP and compare it to the previously seen IP address. *If* a filter driver is manipulating the IP addresses, the verifying filter driver sees the discrepancy.

This is possible with following steps:

- Pre-processing the IRP and noting down its IP address
- Sending the IRP down by using *IoForwardIrpSynchronously*, which has been previously discussed in chapter ”IRP Handling”.
- Post-processing the IRP, when control returns, thus checking if the IP address changed.

The following code realizes these steps. Note that the procedure of typecasting TDI related structs has been shortened.

```
1 NTSTATUS Dispatch(PDEVICE_OBJECT DeviceObject,PIRP Irp) {
2
3    NTSTATUS ntStatus=STATUS_NOT_SUPPORTED;
4    PEXTENSION_OBJECT DeviceExtension =
5          (PEXTENSION_OBJECT)DeviceObject->DeviceExtension;
6    PIO_STACK_LOCATION StackIrpPointer = NULL;
7    PTDI_REQUEST_KERNEL_CONNECT TDI_connectRequest;
8
9    // The IP address is noted here.
10   unsigned long Address;
11
12   unsigned long Address2; // After returning
13
14   // Obtain a pointer to the corresponding stack in the IRP.
15   StackIrpPointer = IoGetCurrentIrpStackLocation (Irp);
16
17   // Sanity check
18   // StackIrpPointer is never zero!
19   if (!StackIrpPointer) {
20         DbgPrint("IP verifier: Fatal Error: IRP stack pointer is NULL! \r\n");
21         return STATUS_UNSUCCESSFUL;
22      }
23
24   // Sanity check
25   // Since we attached to the TCP device, we can only
26   // receive IRPs concerning this device.
27   if (DeviceObject == DeviceToBeFiltered) {
28
29   Address=0; Address2=0;
30
31      // For the moment we are only interested in the TDI_CONNECT Function
32      if (StackIrpPointer->MinorFunction == TDI_CONNECT) {
33
34      TDI_connectRequest =
35            (PTDI_REQUEST_KERNEL_CONNECT)
36               (&StackIrpPointer->Parameters);
```

When the IRP arrives, the driver must first note down the IP address. This allows to compare the IP address later, when the IRP returns.

```
1        // Note the IP address down.
2        Address=( (PTDI_ADDRESS_IP)
3              (( (PTRANSPORT_ADDRESS)
4             (TDI_connectRequest->RequestConnectionInformation->RemoteAddress))
5                 ->Address->Address) )->in_addr;
6
7        DbgPrint("\r\nIP verifier: Noted down: IP address was 0x%x! \r\n",
8                             Address);
9
10       // Send the IRP down to the next-lower driver
11       if (IoForwardIrpSynchronously(DeviceExtension
12                                   ->TopOfDeviceStack,Irp)) {
13          // The IRP returned here.
14          Address2=( (PTDI_ADDRESS_IP)
15             (( (PTRANSPORT_ADDRESS)
16            (TDI_connectRequest->RequestConnectionInformation->RemoteAddress))
17                ->Address->Address) )->in_addr;
```

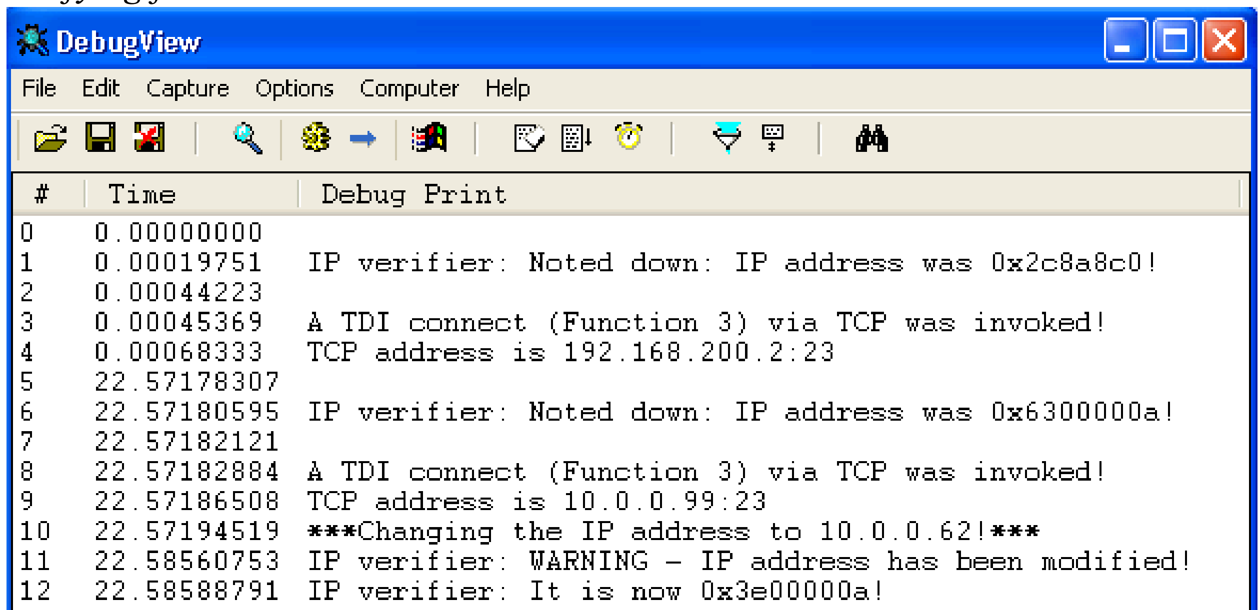
Now we can check if the IP address has been modified. We simply compare both IP addresses:

```
1           if (Address2 != Address) {
2           DbgPrint("IP verifier: WARNING -
3                    the IP address has been modified!\r\n");
4           DbgPrint("IP verifier:
5                    It is now 0x%x (hexadecimal)\r\n",Address2);
6           }
7           ntStatus = Irp->IoStatus.Status; // return the correct value!
8           IoCompleteRequest(Irp,IO_NO_INCREMENT);
9           return ntStatus;
10       }}
11
12       IoSkipCurrentIrpStackLocation(Irp);
13       ntStatus=IoCallDriver(DeviceExtension->TopOfDeviceStack, Irp);
14    }
15    return ntStatus;
16 }
```

In the listed code example, the IP address is noted down, before the IRP is sent to the next-lower driver. On return of the IRP, the IP address is checked again and compared with the previous value. In case of a mismatch, a warning is printed as consequence.

The provided example of an IP verifying TDI filter driver can be installed after the example of the IP address redirecting filter driver has been installed. To see the IP verifying filter driver working, it is recommended to first let the computer connect to any unrelated IP address, for example 10.0.0.42 or even 10.0.0.62. Since the IP address will not be changed in this case, the verifying TDI filter driver has nothing to announce. However, as soon as an application tries to connect to the IP address 10.0.0.88 or 10.0.0.99, the IP address manipulating filter driver will react and redirect the application to 10.0.0.62. The verifying filter driver will detect the redirection and print out a corresponding warning.

An example output of both the modifying filter driver and the verifying filter driver:



DebugView

File Edit Capture Options Computer Help

#	Time	Debug Print
0	0.00000000	
1	0.00019751	IP verifier: Noted down: IP address was 0x2c8a8c0!
2	0.00044223	
3	0.00045369	A TDI connect (Function 3) via TCP was invoked!
4	0.00068333	TCP address is 192.168.200.2:23
5	22.57178307	
6	22.57180595	IP verifier: Noted down: IP address was 0x6300000a!
7	22.57182121	
8	22.57182884	A TDI connect (Function 3) via TCP was invoked!
9	22.57186508	TCP address is 10.0.0.99:23
10	22.57194519	***Changing the IP address to 10.0.0.62!***
11	22.58560753	IP verifier: WARNING - IP address has been modified!
12	22.58588791	IP verifier: It is now 0x3e00000a!

A mere warning printed out via debug print might not be a sufficient action. In the next section, several ways to drop IRPs containing an unwanted request are shown.

4.6 Freeing IRPs

A very interesting option would be to arbitrarily drop a certain attempt to connect. As mentioned earlier, cancelling the IRP is no option. The I/O manager does not seem to provide any cancel routine and cancelling does not have any effect. Obviously, Microsoft did not consider dropping IRPs arbitrarily as legitimate action. Still, there exist several ways to realize the dropping of IRPs.

The first way is to keep the IRP for itself instead of forwarding it down. Thus, no network driver will ever have the chance to process the request. It is a very simple and rude method, but effective. The only needed action is to set an adequate NTSTATUS in the IRP status field and returning this ntstatus by the dispatch routine. STATUS_TIMEOUT proved to be useful in such a case, since STATUS_TIMEOUT can naturally happen to a network-related IRP.

```
1 Irp->IoStatus.Status = STATUS_TIMEOUT;
2 ntStatus=STATUS_TIMEOUT;
```

It signalizes the I/O manager that something went wrong. However, a negative side effect will occur on side of the userspace application, which tries to connect and eventually gives up because of a timeout.

The second way is actually even more unusual.[1] It allows to free an IRP arbitrarily.

[1]The author discovered it by pure random and still wonders why it actually works.

The code is as follows:

```
NTSTATUS destroyIRP(PDEVICE_OBJECT DeviceObject,PIRP Irp) {

   NTSTATUS ntStatus=STATUS_NOT_SUPPORTED;
   PEXTENSION_OBJECT DeviceExtension =
         (PEXTENSION_OBJECT)DeviceObject->DeviceExtension;
   PIO_STACK_LOCATION StackIrpPointer = NULL;

   DbgPrint("Freeing IRP NOW!\r\n");
         StackIrpPointer = IoGetCurrentIrpStackLocation (Irp);
         IoCopyCurrentIrpStackLocationToNext(Irp);
         IoSetCompletionRoutine(
               Irp,
               (PIO_COMPLETION_ROUTINE) destroyingRoutine,
               NULL,
               TRUE,
               TRUE,
               TRUE);
         Irp->IoStatus.Status = STATUS_SUCCESS;
         IoCompleteRequest(Irp, IO_NO_INCREMENT);
         return ntStatus;
   }
```

With the completion routine being

```
NTSTATUS destroyingRoutine(
      PDEVICE_OBJECT DeviceObject,
      PIRP Irp, PVOID Context)
{
   Irp->IoStatus.Status = STATUS_SUCCESS;
    IoFreeIrp(Irp);
    DbgPrint("IRP was freed! \r\n");
    return STATUS_SUCCESS;
}
```

Unfortunately, besides these possibilities, no 'correct' solution exists, which a TDI filter driver could use to arbitrarily drop an unwanted IRP. Yet, there may exist further possibilities not shown here.

Chapter 5

Summary and Conclusion

We described the very basics of writing a TDI filter driver. We began by introducing how to write and compile a wdm-based driver and explained the fundamentals of WDM driver concepts, such as IRP handling, devices, major functions as well as different types of routines:

- Dispatch routines, which are assigned to major functions.
- Completion routines, which enable the caller to receive control over an IRP once all lower drivers finished processing.
- Cancel routines, which must be provided by the original driver and allow any driver to arbitrarily cancel IRPs.

In the main chapter, we explained in detail how to write TDI filter driver, first by creating a simple Forward-and-Forget filter driver and, after an overview of the different structs used by TDI, we continued with more elaborated examples and techniques. We quickly pictured the practical use of a filter driver able to automatically redirect network connections and provided a typical scenario.
In the last part, we then moved attention towards techniques of defense against unwanted manipulation of IRP data. With each example, new methods and possibilities of how to use TDI were

introduced while the code examples generally increased in complexity.

TDI filter drivers can serve numerous possibilities and purposes. Two main functions are:

1. Security: as a firewall or Intrusion Detection System. This probably applies to the majority of TDI filter drivers.
2. Use in network operations and server-oriented solutions due to optimal performance as well as fast and reliable actions.

A TDI filter driver offers much more options and flexibility than a userspace application. It has access to any data, since it operates in kernelmode. It is faster and very reliable, compared to userspace code. For example, userspace applications usually depend on many libraries, which might contain errors and bugs as well as vulnerabilities. On the contrary, bugs and vulnerabilities are rare in the Windows Kernel. Also, an unintentional termination is very unlikely.

This document was an introduction to writing TDI filter drivers. It was written to finally provide free documentation and encourage the development of new TDI-based applications.

5.1 Acknowledgments

I would like to thank my mentor, Engin Kirda of the Institute Eurecom and International Secure Systems Lab, for providing all technical support and advices I could have wished for, for encouraging to write about TDI, and for his great patience. I also would like to thank my remote mentor Pavel Laskov from the University of Tuebingen and Fraunhofer Institute FIRST Berlin for having laid the foundation, without his presence this document would undoubtedly not exist. Finally, I would like to thank unnamed friends for moral support in times of doubts.