Porting of an Ethernet driver to EtherCAT

Graduation Thesis

Cas de Rooij

June 18, 2013

Fontys Hogeschool ICT

TITELBLAD AFSTUDEERSCRIPTIE

Gegevens student:

Naam + Voorletters	C M de Rooij
Studentnummer	2162495
Afstudeerrichting	HBO ICT&Technology Voltijd
Afstudeerperiode	Van 18 februari 2013 t/m 17 juli 2013(95 werkdagen)

Gegevens bedrijf:

Naam Bedrijf	Prodrive B.V.
Afdeling	Development-Software
Plaats	Son
Bedrijfsbegeleider	M Nelissen

Gegevens docentbegeleid(st)er:

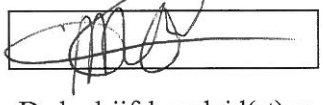
Naam + voorletters	C van Tilborg

Gegevens verslag:

Titel afstudeerverslag	Porting of an Ethernet to EtherCAT driver
Datum uitgifte afstudeerverslag	18 juni 2013
Vertrouwelijk	nee

Getekend voor gezien door bedrijfsbegeleid(st)er:

Datum: 18-06-2013



De bedrijfsbegeleid(st)er,

Preface

This Paper is written by me, Cas de Rooij, for graduating on the batchelor study ICT and Technology at Fontys University of Applied Sciences.

I've been researching on how to convert an Ethernet driver into an EtherCAT driver from February 2013 to June 2013. The project is aimed at making it possible to use a specific embedded system created by Prodrive to control the EtherCAT industrial field bus.

I would like to thank all my colleagues at Prodrive, who have been very helpful during the internship, and especially Micha Nelissen, my internship mentor, and Maik van Kranenburg.

Contents

Summary

EtherCAT is a real-time industrial field bus based on Ethernet. The bus master uses an Ethernet device to communicate with the slaves.
At Prodrive, it has been decided that the PPA8548 AMC, a PowerPC embedded system, should get support for the EtherCAT master functionality because the current EtherCAT master is more expensive and power consuming than the PPA8548 AMC, which can offer sufficient computing power in a lot of cases.

Usage of EtherCAT is possible without modification to the driver of the Ethernet device, but in order to communicate with real-time performance it's necessary to modify the driver of the Ethernet device.

This paper answers the question on how to convert a Linux Ethernet driver to an EtherCAT driver, and more specifically the "gianfar" driver, meant for the network interfaces of the PPA8548 AMC module.

The differences between Ethernet and EtherCAT are compared both on protocol level and in the drivers in order to answer this main question.

The most important, and most drastic changes are in the interrupts of the Ethernet driver. Normally an Ethernet device waits for interrupts to tell the driver to take action, but when this happens is non-deterministic and it can interrupt the flow of the driver, compromising real-time functionality. In order to stop the interrupts, they have to be replaced by a polling function.

A substantial part of the development is solving problems that occur because the driver source code is modified in a way it wasn't intended to be. This paper also discusses the problems that have occurred.

Samenvatting

EtherCAT is een real-time industriële veldbus die gebaseerd is op Ethernet; de bus master gebruikt een Ethernet device om te communiceren met de slaves.
Bij Prodrive is er besloten dat de PPA8548 AMC, een PowerPC embedded system, ondersteuning moet krijgen voor functionaliteit als EtherCAT master omdat de huidige EtherCAT master duurder is en meer energie verbruikt dan de PPA8548 AMC. De PPA8548 AMC heeft in veel toepassingen voldoende rekenkracht.

Gebruik van EtherCAT is mogelijk zonder de Ethernet driver aan te passen, maar om te kunnen communiceren met real-time performance is dit wel noodzakelijk.

Dit onderzoeksdocument beantwoordt de vraag hoe men een Linux Ethernet driver om kan bouwen tot een EtherCAT driver, en in het speciaal de "gianfar" driver die bedoeld is voor de netwerk interfaces van de PPA8548 AMC module.

Om de hoofdvraag te kunnen beantwoorden worden de verschillen tussen Ethernet en EtherCAT onderzocht, zowel op protocol niveau als in de drivers.

De belangrijkste en meest drastische veranderingen zijn aan de interrupts van de Ethernet driver. Normaal gesproken wacht een Ethernet device tot er een interrupt wordt gelanceerd om te vertellen dat de driver iets moet doen. Maar wanneer dit precies gebeurt is non-deterministisch en het kan de normale "flow" van de driver onderbreken waardoor het de real-time functionaliteit onmogelijk maakt. Om deze interrupts uit te schakelen moet er echter wel een "polling" functie worden gemaakt die de functies van de interrupts vervangt.

Een fors deel van de driver-ontwikkeling bestaat uit het oplossen van problemen die ontstaan omdat de driver code wordt aangepast op een wijze waarvoor het nooit was bedoeld. Dit onderzoek behandelt ook deze problemen.

List of Figures

List of Tables

Lexicon

Word or Abbreviation	Meaning
AMC	Advanced Mezzanine Card, a standard for connecting primarily communication hardware in a rack.
API	Application Programming Interface
Ethernet	A technology for computer networking through a coaxial cable, twisted pair cable or fibre optic cable
Kernel Oops	A non-fatal error in the linux kernel
MAC address	Media Access Control address
TCP	Transmission Control Protocol
UDP	User Datagram Protocol

Chapter 1

Introduction

There are several communication buses for real-time industrial communication. But these buses usually don't offer a high bandwidth and they require specialized hardware. EtherCAT was designed with these shortcomings in mind. It's an industrial communication bus which is based on the hardware layer of Ethernet. However, there are a number of differences:

- It uses a master with one or more slaves.
- The bus is capable of real-time communication
- Redundant communication is supported

The real-time capabilities of the EtherCAT bus make it possible to use it when exact timing is crucial. Redundant communication in EtherCAT means that the master is connected to both the beginning and the end of the chain with slaves, making it possible to still access all (working)slaves, even if a communication line or a slave fails.

Furthermore there are some properties that make EtherCAT stand out as an industrial communication bus. It's possible to run normal Ethernet communication over the EtherCAT bus.
Another interesting property is the flexibility in network topologies(figure 1.1). Most buses need to be connected in one long chain. With EtherCAT it's possible to add a node that splits off the bus adding a branch that's still capable of normal communication. It's even possible to add a node that splits off regular Ethernet.

Another feature is that the EtherCAT master stack doesn't require any custom hardware. It only requires an Ethernet card and a modified driver. The driver must be modified so that it communicates with the EtherCAT master stack software and that it functions in real-time. This research will be about how a normal Ethernet driver from the Linux kernel tree can be modified to run as an EtherCAT device.

This subject is only briefly discussed in the IgH EtherCAT Master Documentation[6], so this research will be of value to other programmers who need to modify Network drivers. Because of this, the target audience of this research paper is people with academic knowledge about IT, mainly in the technical domain.

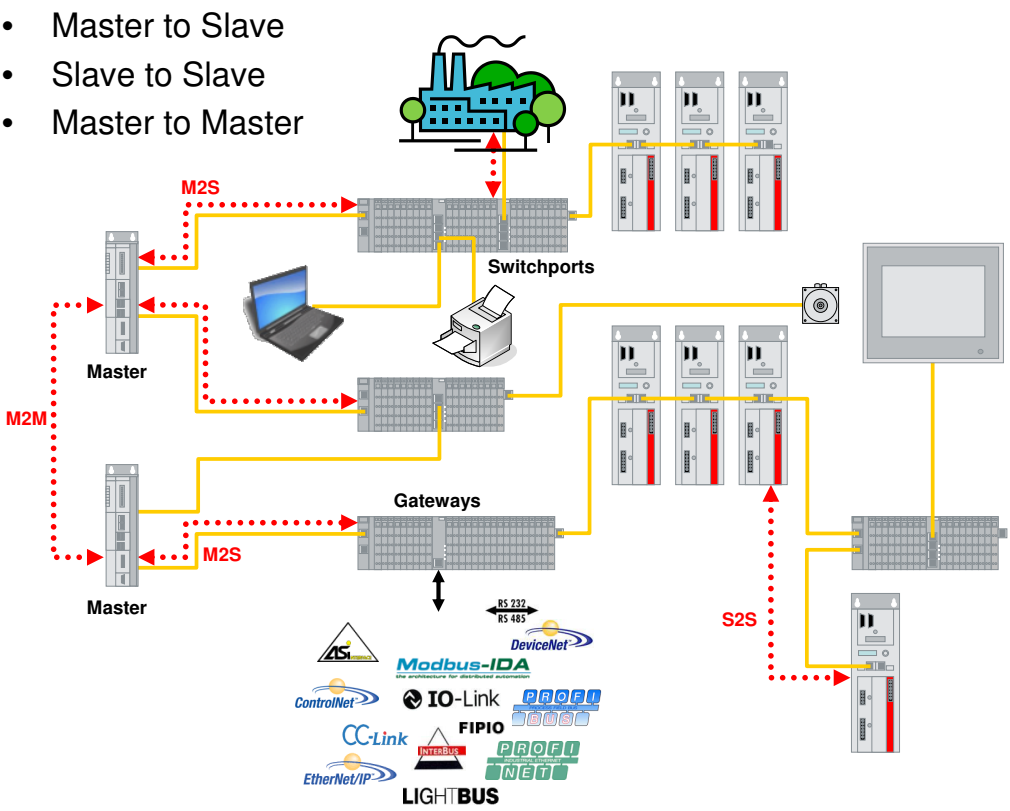


Figure 1.1: Possible EtherCAT topologies [4]

Chapter 2

The company

Prodrive is a company specialized in embedded processing, motion control and power electronics. Prodrive has been founded in 1993 by Hans Verhagen and Pieter Janssen as a company, based on the campus of Technische Universiteit Eindhoven, specialized in digital signal processing and motion control. In 1997 Prodrive moved to Science Park Eindhoven, by then it had 20 employees. In the following years the number of employees has grown increasingly faster and Prodrive had invested in its own manufacturing facilities. In 2010 Prodrive realized a new facility with several production halls, an own cleanroom and enough space to keep growing for the following years. At this date, 2013, Prodrive has about 600 employees.

Prodrive now produces several different products. For industrial embedded processing it produces PowerPC based Advanced Mezzanine Cards(AMC's). A notable company that uses these products in their machines is ASML. Other embedded systems Prodrive manufactures are Intel Atom based systems as generic computing platform and multimedia settop boxes.
Another category of products Prodrive sells is motion control and mechatronics. A good example is the EtherCAT motion platform. This platform offers everything necessary for an EtherCAT motion control system.

Prodrive also develops image processing hardware, for medical solutions and traffic cameras.
Prodrive not only manufactures embedded systems, but also power electronics; several specialized power converters ranging from low power to high power, above 150kW. One of these products is targeted for medical equipment.[7]

Chapter 3

The assignment

One of the products of Prodrive is the PPA8548 PowerPC AMC module. This module contains two network interfaces, and Prodrive wants to offer the possibility to use the module with the built-in network interface as an EtherCAT master device. There's already a complete software product to do this, but there are no drivers for the network interfaces on the PPA8548 AMC's. There is a "generic" EtherCAT driver that uses the normal Ethernet driver for the communication, but Ethernet drivers aren't designed for realtime performance. It is my task to research how to convert the normal Ethernet driver into an EtherCAT driver and how well it performs.

3.1 Research Method

The research is done by finding how to develop the driver and immediately applying the gathered knowledge. The products are the modified driver and this research paper.

3.2 Research questions

The main question is *How to to realize an EtherCAT Driver for the PPA8548 PowerPC platform and how to optimize the performance?*
This question assumes that there are several methods to do the job, but there is only one method with the best performance. Because of the time-critical nature of the EtherCAT bus, performance has a certain priority.

3.3 Subsidiary questions

To answer the main question the research will be divided into a number of subsidiary questions. Each question is answered in a seperate chapter.

- How does EtherCAT work?
- What are the differences between the PowerPC Ethernet drivers and existing EtherCAT drivers?
- What are the possible ways to convert Ethernet drivers of the PowerPC platform into EtherCAT drivers?
- What is the performance of the chosen solution(s)?

How does EtherCAT work? This question is important because it will give insight in what the IgH EtherCAT master stack actually does, and what the drivers do. Answering this question first will mean a good resource on background information when answering the other questions.

What are the differences between the PowerPC Ethernet drivers and existing EtherCAT drivers? This question is important to ask. You have to know what parts of the driver require modifications. Once this question is answered it is possible to go on to the next one.

What are the possible ways to convert Ethernet drivers of the PowerPC platform into EtherCAT drivers? Once you know what parts require modification, there might be multiple ways to implement them. It could also be possible there's only one viable method. In that case an explanation is required why it is so.

What is the performance of the chosen solution(s)? The performance of different chosen solutions might differ and it is important that the best solution will be chosen. If there's only one viable solution it's still important for the company to know what the performance is.

The performance is tested by measuring the execution time of several time-critical functions in the EtherCAT master stack. The jitter in the execution time is also measured, jitter will be explained in section 7.1. The results of the developed driver are compared with the results of the generic EtherCAT driver to verify whether there is performance increase and in what way the performance is increased.

Chapter 4

How EtherCAT works

Two aspects important to answer this question: How does the EtherCAT protocol work, and how does the implementation of the IgH EtherCAT master stack work?

4.1 EtherCAT in General

EtherCAT uses virtually the same frames as the Ethernet physical layer, or when IP routing is necessary it can use UDP frames. The only real difference is the payload.
In figure 4.1 is a diagram drawn of The Ethernet frames with the EtherCAT payload. The blue parts are part of the normal Ethernet frame. As a payload EtherCAT uses a 2 Byte header(the red part in the diagram) and after that one or more datagrams(the green part). There are two different datagrams. Process Data and Mailbox datagrams.

4.1.1 Process Datagrams

Process Datagrams contain Process Data Objects(PDOs), which are meant for sending commands to slaves in the network. These Process Datagrams can contain multiple PDOs as long as the total length does not exceed the maximum length of the frame.

4.1.2 Mailbox Datagram

The other datagram, Mailbox, is used for facilitating the possibility to send any other data over EtherCAT.
For instance, there are several (bus) protocols that can be "tunneled" over the ethercat bus like Ethernet over EtherCAT, CANopen over EtherCAT, Vendor specific over EtherCAT and Servo Profile over EtherCAT.[8] These protocols can still benefit from the real-time capabilities of EtherCAT.

4.1.3 Timing

EtherCAT frames can be sent every bus cycle, and are processed by any device on the EtherCAT bus, so slaves don't affect the performance of the EtherCAT bus.
This makes it possible that all slaves receive the frame almost simultaneously. The only delay created is the time it takes for the slave to pass the message from its input to the next slave or master. There is a best and a worst case timing scenario. This timing depends on The bus cycle and the processing time by the slaves(Figure 4.2). In the best case scenario the frame can be passed to the slave right away, in the worst case the frame would have to wait until the next EtherCAT cycle.

4.2 The IgH EtherCAT master stack

The IgH EtherCAT master stack is software designed for Linux systems to use the system as master of the EtherCAT bus. This is done by loading customized Ethernet drivers which are capable of EtherCAT

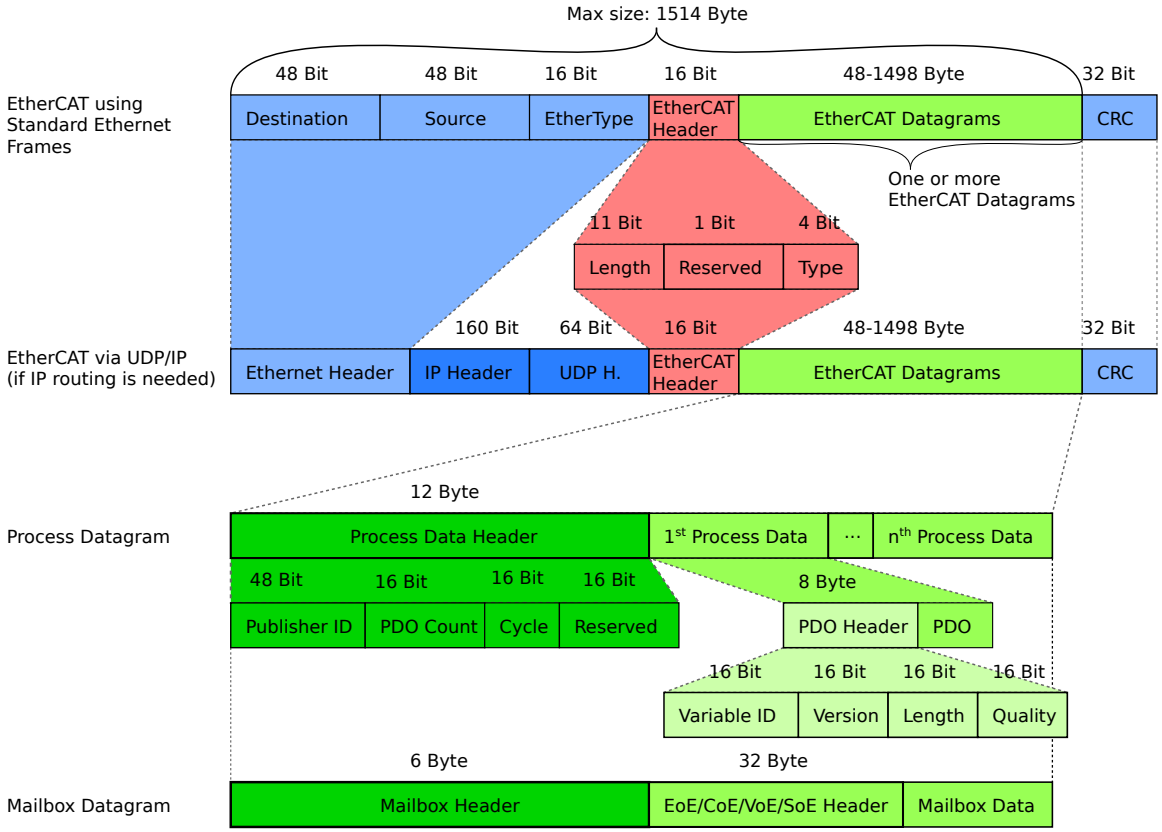


Figure 4.1: EtherCAT Frames [2]

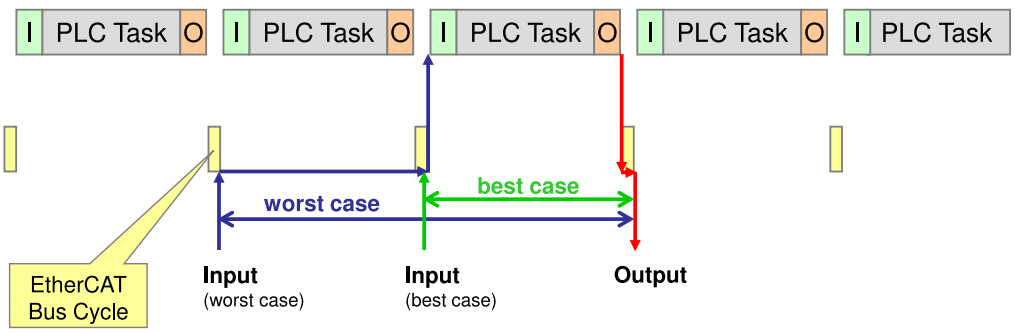


Figure 4.2: Reaction time of the EtherCAT bus [4]

communication.

The EtherCAT master stack consists of the following parts(Figure 4.3):

- EtherCAT master module
- EtherCAT capable Ethernet driver
- EtherCAT "generic" driver

4.2.1 EtherCAT master module

This is the core of the EtherCAT master stack. This module manages the character devices used for interfacing with the EtherCAT master hardware (like /dev/EtherCAT0) and passes through all communication between this character device and the actual EtherCAT driver. The master module also manages the current settings and keeps lisings of all slaves attached to each master device.

4.2.2 EtherCAT capable Ethernet driver

This is the Ethernet driver that has been modified to also be able to use an Ethernet interface as EtherCAT master interface. This kernel module is mainly managed by the EtherCAT master module. The EtherCAT master module decides which Ethernet interface must function as EtherCAT master device and it also decides when the driver must start polling.

4.2.3 EtherCAT "generic" driver

There's also an EtherCAT driver that uses Ethernet interfaces from unmodified Ethernet drivers as EtherCAT master. This makes it possible for any Ethernet device to work as EtherCAT master device but the real-time performance will be worse since interrupts (and possibly TCP offloading) aren't disabled. Usage of this driver is optional. It's advised to use the modified, EtherCAT capable Ethernet driver when possible.

4.2.4 Conclusion

EtherCAT is a lot like Ethernet, the real difference is the timing in communication. Ethernet Frames are almost identical to EtherCAT frames, but the payload is different. The payload of EtherCAT frames contains datagrams, there are 2 different types of datagrams, Process Datagrams and Mailbox Datagrams. The Process datagrams are meant for sending commands to the EtherCAT slaves, the Mailbox datagrams are more universal and can be used for all sorts of data. Both types offer real-time functionality.

In the EtherCAT master stack there is 1 module, the EtherCAT master module, that controls the entire EtherCAT communication. EtherCAT drivers can make connection with the EtherCAT master module, and the EtherCAT master module manages the interface, its settings and the slaves connected to it.

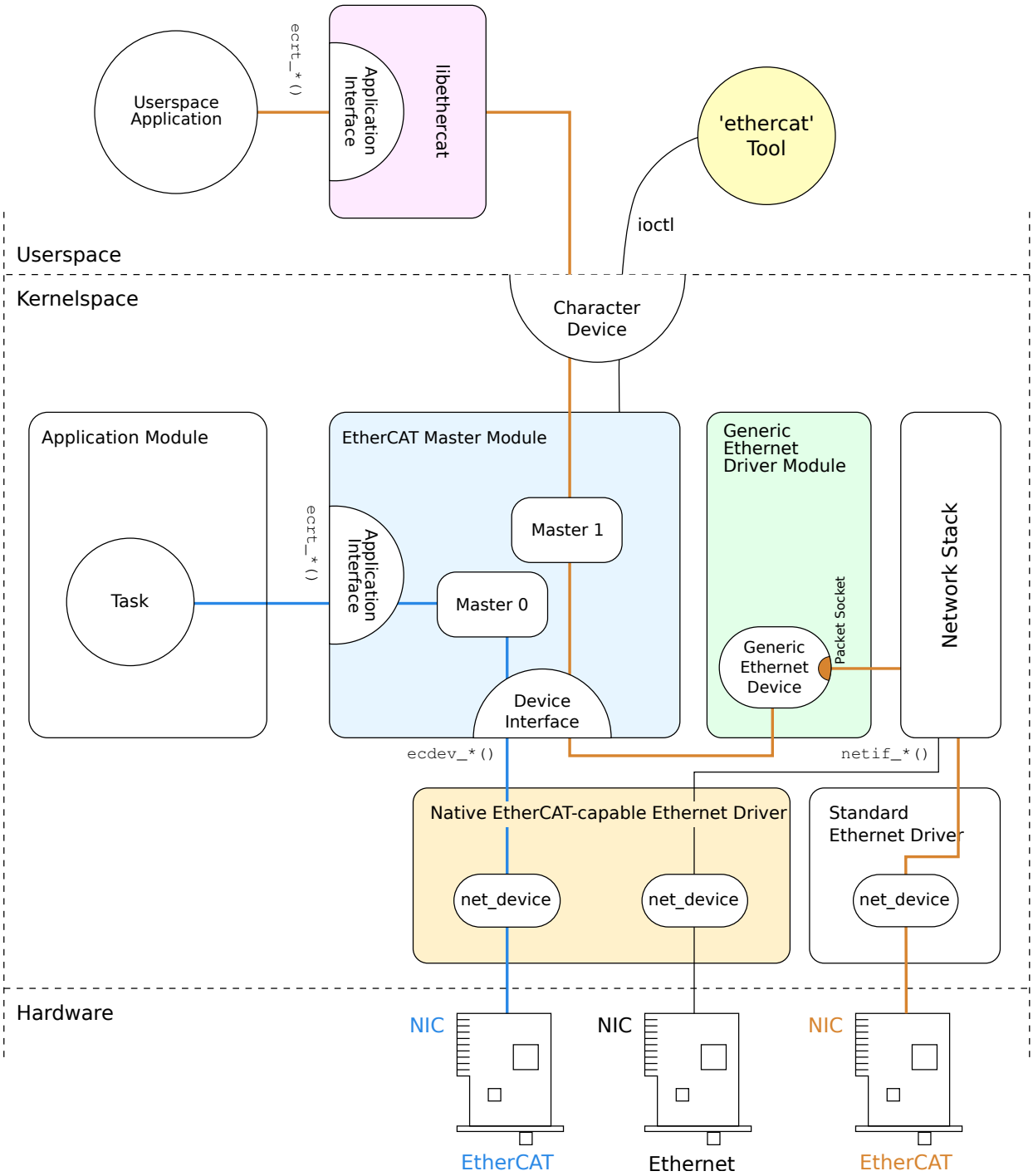


Figure 4.3: IgH EtherCAT master stack architecture [6]

Chapter 5

The differences between the PowerPC Ethernet drivers and existing EtherCAT drivers

5.1 Ethernet and EtherCAT driver general differences

A number of changes that must be done to any Ethernet driver to convert it to EtherCAT. I have gathered information on the differences by comparing the EtherCAT drivers of the Realtek 8139 with the Ethernet drivers(devices/8139too-2.6.37-ethercat.c and devices/8139too-2.6.37-orig.c in the IgH EtherCAT master stack tree). The Gianfar driver is not identical to the 8139 driver, but because they are both network drivers there are certain similarities. In this section I will discuss the similarities between the two drivers and how they relate to the changes that need to be done for it to be an EtherCAT driver.

5.1.1 Automatic loading

If the EtherCAT master stack is used with loadable kernel modules, then the drivers must not load automatically. This is because before the EtherCAT master stack starts, the normal Ethernet drivers should be loaded. When the init script of the EtherCAT master stack is started, it will unload the original drivers and load the EtherCAT versions.

Automatic loading is prevented by commenting out the MODULE_DEVICE_TABLE macro. This macro makes it possible for the kernel to hotplug devices. When the macro is not present the kernel will not know what to do with the module until it's loaded.

5.1.2 Function conversion

There are a number of functions, mostly of a diagnostic nature, will be replaced in the EtherCAT drivers. This is done because when the device is loaded as an EtherCAT device, it will not have a fully functional Ethernet device handler. The messages passed to the kernel still look the same when prepending "%s: " to the string and putting dev->name right after. These changes apply to the functions listed in table 5.1.
The changes are only necessary to be done when the message can occur both in EtherCAT mode and Ethernet mode.

An example:

```
1 netdev_dbg(dev, "Identified 8139 chip type '%s'\n",
2     rtl_chip_info[tp->chipset].name);
```

[1]
will become:

```
pr_debug("%s: Identified 8139 chip type '%s'\n",
  dev->name, rtl_chip_info[tp->chipset].name);
```

[5]

Original	EtherCAT
netdev_info	pr_info
netdev_warn	pr_warning
netdev_err	pr_err
netdev_dbg	pr_debug
netif_dbg	pr_debug

Table 5.1: All diagnostic message functions in the Ethernet version and their corresponding EtherCAT versions

5.1.3 Probing

When an Ethernet device is being probed, then the hardware will first be initialized to make it possible to retrieve the MAC address of the device. This address is then used by the EtherCAT master stack to determine whether this device must run in Ethernet or EtherCAT mode. This is done by calling the ecdev_offer() function. This function will return either NULL when the device is supposed to run in Ethernet mode or a pointer to an ec_device_t struct if the device is supposed to run in EtherCAT mode.

5.1.4 Polling

All operations to check on new messages, will be executed during polling. This replaces the interrupts of the Ethernet controller. A pointer of this polling function is passed to the EtherCAT master stack with ecdev_offer() so the EtherCAT master stack is able to call this function.

After offering the device, the EtherCAT master stack knows the function it has to call in order to poll. When the polling happens is decided by the EtherCAT master stack. The polling frequency can be adjusted by telling the EtherCAT master stack the new cycle frequency.

The polling function, usually called ec_poll() will call the function gfar_poll(). This is a good function to base the polling on, because it's already used for polling of the whole device by NAPI, an API meant for replacing interrupt functions[3]. The function can be used almost as-is.

5.1.5 Transmitting Frames

Transmission of frames by the EtherCAT master stack is done by using the existing function for queueing data that must be transmitted, gfar_start_xmit. But instead of waiting for an interrupt, the message will be sent as soon as polling is commenced.

5.1.6 ec_device_t

ec_device_t is the struct used as a handle for communicating with the EtherCAT stack. There is 1 instance of this struct per EtherCAT master device. A pointer to this struct must be part of the struct that represents the Ethernet device. See subsection "ec_device_t" in next chapter for more information. The ec_device_t struct does not replace the net_device struct, this struct is still necessary for some operations. The net_device struct is just not registered to the network stack. In table 5.2 are all functions for communicating with the EtherCAT master stack listed.

Function	Description
ec_device_t *ecdev_offer()	The master decides if it wants to use the device for EtherCAT operation or not.
void ecdev_withdraw()	The device is disconnected from the master and all device resources are freed.
int ecdev_open()	Opens the network device and makes the master enter idle phase(waiting for applications).
void ecdev_close()	Makes the master leave idle phase and closes the network device.
void ecdev_receive()	Forwards the received data to the master. The master will analyze the frame and dispatch the received commands to the sending instances.
void ecdev_set_link()	Sets a new link state. If the device notifies the master about the link being down, the master will not try to send frames using this device.
uint8_t ecdev_get_link()	Reads the link state.

Table 5.2: All functions to interface with the EtherCAT master stack module with descriptions[6]

Chapter 6

Possible ways to convert Ethernet drivers of the PowerPC platform into EtherCAT drivers

I have found 2 possible ways to convert the Ethernet driver into an EtherCAT driver. The approaches are quite different, and both have advantages and disadvantages.

6.1 Like the existing EtherCAT drivers

The Existing EtherCAT drivers use the same functions for handling both Ethernet and EtherCAT. To do this, the functions check whether the device is running EtherCAT to decide what to do. This saves a lot of time writing code because a lot of functions don't need any changing, or very marginal changes.

6.2 Splitting the driver up completely

This is a hard solution to implement. When a device is being probed, the driver will have 2 possibilities to choose from and "split up". This means that there's actually a complete copy of the Ethernet driver, modified to run as EtherCAT driver.

This method requires extensive code refactoring which takes more time than the other method. When the original Ethernet driver is updated, it also takes extra time to apply those patches to the EtherCAT driver.

6.3 Implementing

I have chosen to only implement the first of the two options because splitting up the driver code takes much more time, it makes the driver code harder to maintain and the driver is supposed to be published, so modifications according to the existing guidelines will make it easier for others to understand what's happening.

6.3.1 EtherCAT Master Stack Interface

The EtherCAT master stack has, as mentioned in the previous chapter, a number of functions which the drivers use to interface with it(table 5.2). I will explain how they are used in the Gianfar driver in this section.

ecdev_offer()

This function is part of the initialization of the device, and is placed in gfar_probe(). I've chosen to let this happen right after the queues are initialized. These queues are necessary for both Ethernet and EtherCAT, so no check is necessary. And it isn't possible to call ecdev_offer much later, because some interrupts would already have been started.

ecdev_open() and ecdev_withdraw()

When ecdev_offer() has decided the device will be in EtherCAT mode, then the device must be opened. That's what ecdev_open() is for. However, if this function fails this means there's something wrong with the EtherCAT master stack and usage of EtherCAT should not be further attempted. To be sure of this ecdev_withdraw() will disconnect the driver and free the resources used for the EtherCAT connection.

```
1 if ((priv->ecdev != NULL) && ecdev_open(priv->ecdev)) {
2   ecdev_withdraw(priv->ecdev);
3   goto register_fail;
4 }
```

(appendix B, line 175)

ecdev_receive

Ecdev_receive() is the function that's used for passing received frames to the EtherCAT master stack. The arguments are a pointer to the frame data and the length of the data. In the gianfar driver this data is in skb-¿data, and it has 16 bytes "padding", that don't belong to the frame data. So the real pointer passed through is skb-¿data + 16. And the acquired length, pkt_len, is 16 less.

ecdev_set_link

This function is used to tell the EtherCAT master stack whether the cable is connected or not. There is already a function in the driver for handling link changes, adjust_link(), so the only thing that needs to be done is to send the right variable with the function. I put this in the end of adjust_link(), after the link state has been retrieved:

```
1 if (new_state && netif_msg_link(priv))
2 {
3   phy_print_status(phydev);
4   /* Update the link status in the EtherCAT master stack as well */
5   if (priv->ecdev != NULL)
6     ecdev_set_link(priv->ecdev, priv->oldlink);
7 }
8 unlock_tx_qs(priv);
9 local_irq_restore_nort(flags);
```

(appendix B, line 523)

6.3.2 Driver Flow

The flow of an EtherCAT driver differs from an Ethernet driver. Ethernet drivers have one or more interrupts that are used to initiate any action, EtherCAT drivers don't use interrupts and use a periodic polling sequence.

ec_device_t

The ec_device_t pointer, called ecdev, is added to the gfar_private struct. The gfar_private struct represents the Ethernet device, and is used in almost any function as a handle to communicate with the correct device. It is possible for all those functions to talk to the EtherCAT side of the device because the ec_device_t pointer is added to the gfar_private struct.

Checking the driver mode

The driver mode is checked on numerous places in the code. This is always done by checking whether or not the EtherCAT master stack has assigned an ec_device_t pointer to the device that's being processed.

Disabling interrupts

One of the most important parts is to disable the interrupts when the device is in EtherCAT mode. This is necessary because interrupts can interrupt the rest of the driver code, compromising the real-time functionality.

Disabling the interrupts is done by skipping all functions that are meant for initializing or enabling the interrupt

The function register_grp_irqs() is completely dedicated to registering the interrupt requests of the device. This means it can be skipped as a whole when the device is in EtherCAT mode. It's possible to disable the use of register_grp_irqs() in EtherCAT mode, which can save cpu instructions.

6.3.3 Problems

While implementing the modifications, a number of problems occurred that have been solved.

Link state detection

One of the tasks of an Ethernet or EtherCAT driver is to detect whether or not the device is connected to a network. This was already implemented for the Ethernet driver, but for EtherCAT it works slightly different.

For the EtherCAT master stack to communicate it must first "know" a cable is connected. This is done by calling ecdev_set_link(). The function adjust_link() in the gianfar driver handles all link changes, so ecdev_set_link is called by that function after the current link state is determined.

Data transmission

Transmitting was a problem as well. The EtherCAT master stack mainly uses the existing functions for this task, but some parts must be skipped because they cause kernel "oops"-es because the driver tries to access memory that's normally used for Ethernet communication, but this isn't initialized in EtherCAT mode. This part in gfar_clean_tx_ring(), for example, uses netdev and netif functions.

```
1  /* If we freed a buffer, we can restart transmission, if necessary */
2  if (netif_tx_queue_stopped(txq) && tx_queue->num_txbdfree)
3    netif_wake_subqueue(dev, tqi);
4
5  /* Update dirty indicators */
6  tx_queue->skb_dirtytx = skb_dirtytx;
7  tx_queue->dirty_tx = bdp;
8
9  netdev_tx_completed_queue(txq, howmany, bytes_sent);
```

(appendix B, line 384)
These functions are only meant for communicating with the Ethernet stack, and they are useless in EtherCAT mode.

Transmission memory access

After that, there was another problem; in gfar_start_xmit() a sk_buff pointer, passed to the function by the EtherCAT master stack is freed while the EtherCAT master stack frees this pointer itself. This caused the kernel trying to access unused memory space.

Data reception

Receiving of the EtherCAT frames was another problem, and it has not been solved completely, but there's a workaround which is sufficient for testing purposes.
The Ethernet hardware must be initialized in such a way that it's receptive for Ethernet frames. This doesn't yet happen in the EtherCAT driver. However, it does happen when running the normal Ethernet drivers with the generic EtherCAT driver, and the Ethernet device retains this state even after rebooting the system. The workaround for time being is to first boot the system with the generic drivers and then reboot it with the real EtherCAT drivers.

6.3.4 Conclusion

To convert an Ethernet driver, it's necessary to disable the interrupts and replace the interrupt handlers by a polling function that's invoked by the EtherCAT master stack.
But replacing the absence of the interrupt handlers can cause certain functions to break, for example the data reception, which doesn't work as supposed.

Chapter 7

Performance

An user-space tool for measuring the time certain operations take has been written. This tool executes those operations 100 times a second and then calculates and outputs the statistics of the execution times. For example:

```
Average: 9554.409180ns, modus: 9631ns, median: 9601ns, min:9376ns, max: 9931ns
9376:1, 9391:9, 9405:2, 9406:12, 9421:13, 9436:5, 9450:1, 9451:1, 9466:1, 9526:1,
9556:3, 9601:2, 9616:12, 9631:15, 9646:8, 9661:4, 9676:3, 9691:2, 9751:1, 9796:1,
9856:1, 9886:1, 9931:1
statistics calculation time: 134414ns
```

The first line shows the average, modus, median, minimum and maximum of the 100 execution times gathered in the one second.

The second line doesn't speak so much for itself; it's a list of all execution times measured and how much that specific time has occurred in the set. So 9631:15 means that the execution time was 15 times 9631 nanoseconds. This line is used for creating graphs for inspection on the jitter. The last line is the total time it took to calculate these statistics. This isn't a performance indicator, but it's useful to check whether the calculation of the statistics has any impact on the process.
The all measurements used are in appendix A.

7.1 Jitter

Jitter is the variation in time it takes to execute an action. In this case the action is sending and receiving frames.
Jitter can be detected by checking how large the difference is between execution times. The quickest way to do that is to inspect a graph on how frequently each execution time occurs, like figure 7.2. All graphs are based on executing the operation 1000 times in 10 seconds, so 100 times per second. The spikes in the graph indicate the execution times that have been recorded most frequently. If the jitter is low, then the execution times in the graph are closer together making the spike is more narrow.

The timer used for testing the jitter doesn't have real nanosecond resolution; figure 7.1 is a graph of the timing without any operation between starting and stopping the timer. The graph shows that it takes at least 150ns, but it can just as likely take 165ns, but almost nothing in between. This could be caused by the operating system switching threads, and it should be taken into account when interpreting the graphs on the jitter.

7.2 The operations

The operations that have been measured on execution time are ecrt_master_receive(), ecrt_domain_process(), ecrt_domain_queue() and ecrt_master_send(). Ecrt_master_send() and ecrt_master_receive() are meant for sending and receiving, and these functions are directly affected by the performance of the driver.

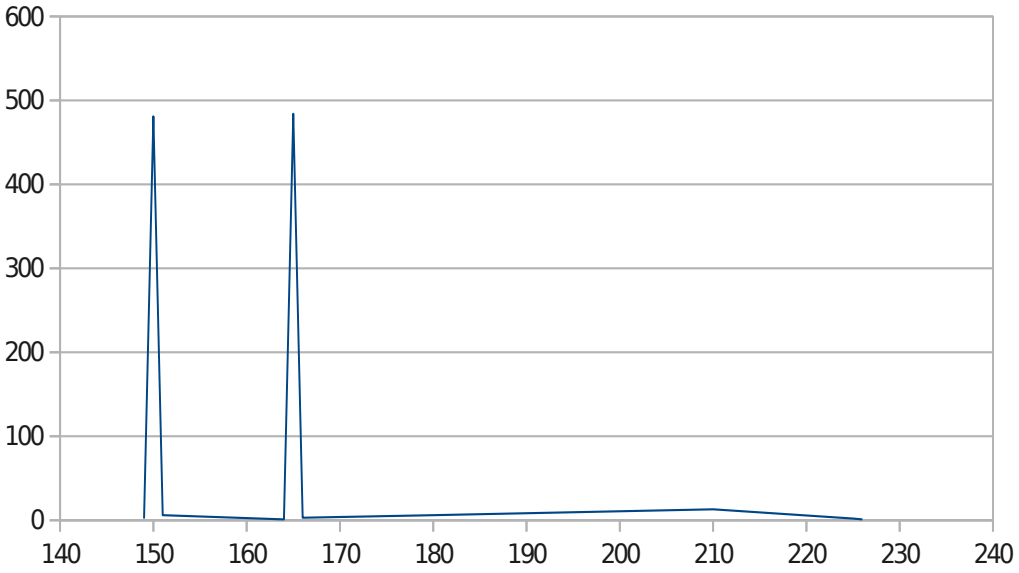


Figure 7.1: Graph on timing without operations.

7.2.1 The measurements

Table 7.1 contains the average execution times of both drivers for comparison.

Receiving

Ecrt_master_receive() is on average slower in the EtherCAT capable driver than in the generic EtherCAT driver. And if you then compare the measurements in figure 7.2 you will see that the EtherCAT capable driver has also significantly more jitter; almost no execution time has been recorded twice, and the difference between the lowest and the highest execution time is almost 20μs. It is possible that this is caused by the poor initialisation mentioned in section 6.3.3, but this will only become clear after the driver functions completely.

Sending

The measurements of ecrt_master_send() are much better in the EtherCAT capable driver than the generic EtherCAT driver with an average time which is 3 times faster.
When looking at the figure 7.3, it is also visible that with the EtherCAT capable driver there is a smaller difference between the lowest time and the largest time measured than with the generic EtherCAT driver. This means the jitter is also smaller.

Function	Generic EtherCAT	EtherCAT capable
ecrt_master_receive()	7106ns	8295ns
ecrt_master_send()	9615ns	3183ns

Table 7.1: Average execution times over 1000 samples.

7.2.2 Conclusion

There's a clearly measurable difference between the performance of the generic EtherCAT driver and the EtherCAT capable driver. Sending ethercat packets is faster with the EtherCAT capable driver, but with

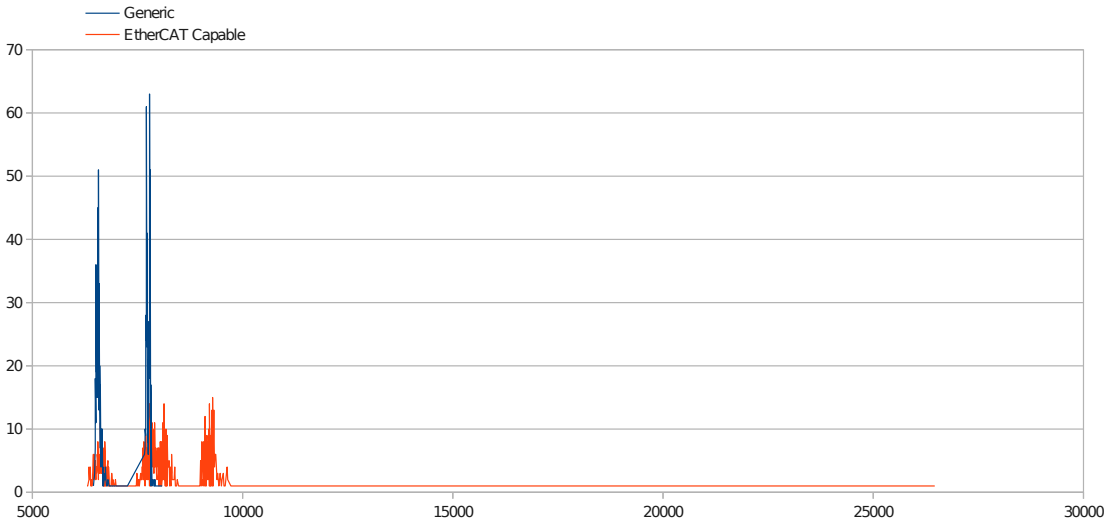


Figure 7.2: Jitter graph on ecrt_master_receive(), based on samples in appendix A.

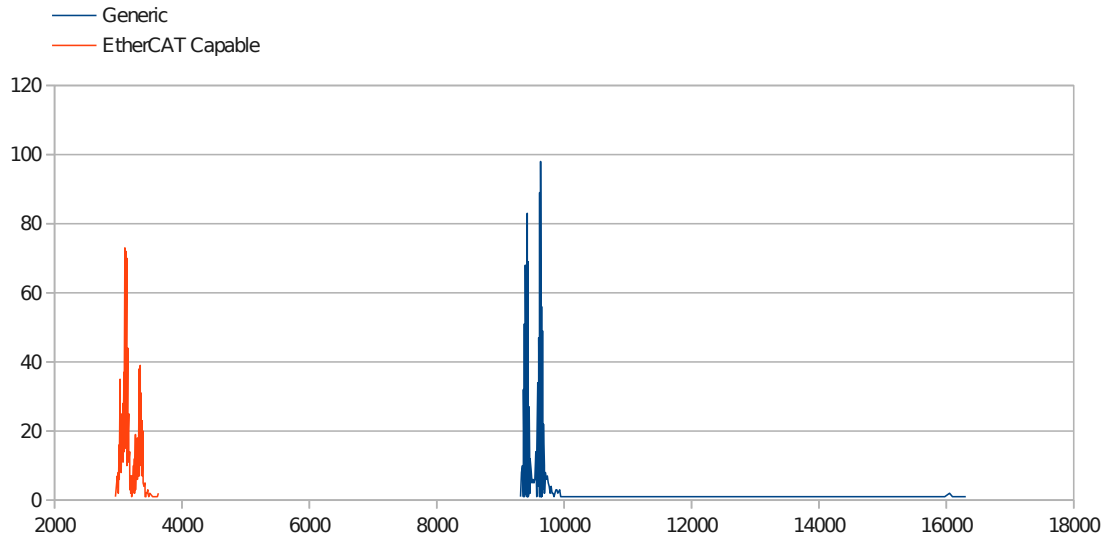


Figure 7.3: Jitter on ecrt_master_send(), based on samples in appendix. A.

receiving packets is the generic EtherCAT driver unfortunately much better.
It is possible that the lack of performance at receiving frames is because that part of the driver is not yet finished. To solve this, the driver must be finished and the performance must be tested again.

Chapter 8

Conclusion

Ethercat is very much like Ethernet because it is derived from Ethernet. Because of this it's possible to use an Ethernet device as EtherCAT master device. The IgH EtherCAT master stack already has a "generic" driver which can be used to let any Ethernet device act as EtherCAT master device, but with the PPA8548 Ethernet devices, and most other Ethernet hardware, it will lack real-time performance because there are interrupts that can interrupt the normal flow of the hardware at any moment.

This problem can be solved by writing a custom, EtherCAT capable driver. The driver is based on the original Ethernet device and makes use of polling instead of interrupts.

The polling is initiated by the EtherCAT master stack so it has control over the timing.

There are a number of parts of the Ethernet driver that need modification:

- Probing of the device
- Sending frames
- Receiving frames

At probing there is communication between the driver and the EtherCAT master stack, and the EtherCAT master stack decides whether the device should run in EtherCAT mode or not. And if so, it will initialize the ec_device_t struct for communication.

Sending frames to the sending queue is done by the EtherCAT master stack through using the existing function meant for transmitting packets.

The actual transmission and reception of the frames happens through polling. When the EtherCAT master stack tells the driver to commence polling, it clears the transmission buffer and sends the packets in the reception buffer to the EtherCAT master stack.

There are a lot of small things that need to be modified in order to let the driver work. Mainly parts that are specifically designed for the gianfar driver. The only solution for those parts is to trace back the problems very carefully, debug it and inspect for any inconsistencies.

8.1 Recommendation

There was a large performance increase for transmitting frames, but the reception is worse than the generic driver. To solve this, it's important to fix the reception problem mentioned in section 6.3.3 and then test the performance again.

For future projects on converting an Ethernet driver to an EtherCAT driver it's important not to underestimate the fact that the original author probably didn't design the driver with EtherCAT in mind.

Evaluation

This research was quite a challenge. Partially because documentation on the IgH EtherCAT master stack isn't very extensive, and partially because converting the gianfar Ethernet drivers didn't turn out to be as straightforward as the existing EtherCAT drivers. There were quite some bugs that occurred after making necessary changes because the gianfar driver wasn't as simple as the drivers Ingenieursgemeinschaft Essen has patched for their EtherCAT master stack.

This caused a large delay in the project; while I should've finished implementation in week 17, it took me until week 24(appendix C page 10).

Although not everything in this paper can be used directly for converting Ethernet drivers to EtherCAT drivers, there's a number of pitfalls documented that may be helpful.

I've learnt that I mustn't underestimate the seemingly small differences between Ethernet and EtherCAT drivers, and hopefully people will not make the same mistakes after reading this paper.

Bibliography

[1] Becker, D. Unpatched driver for the rtl-8139 ethernet adapter. http://sourceforge.net/p/etherlabmaster/code/ci/stable-1.5/tree/devices/8139too-2.6.37-orig.c, January 2012.

[2] Control, P. Ethercat for factory networking. http://www.ethercat.org/pdf/english/pcc_0409_etg_e.pdf, April 2009.

[3] Foundation, L. napi. http://www.linuxfoundation.org/collaborate/workgroups/networking/napi, November 2009.

[4] Group, E. T. Ethercat the ethernet fieldbus. http://www.ethercat.org/pdf/english/EtherCAT_Introduction_EN.pdf, October 2012.

[5] Pose, F. Ethercat patched driver for the rtl-8139 ethernet adapter. http://sourceforge.net/p/etherlabmaster/code/ci/stable-1.5/tree/devices/8139too-2.6.37-ethercat.c, January 2012.

[6] Pose, F. Igh ethercat master 1.5.2 documentation. http://www.etherlab.org/download/ethercat/ethercat-1.5.2.pdf, February 2013.

[7] Prodrive. About prodrive. http://www.prodrive.nl/en/2/about-prodrive, May 2013.

[8] Rostan, M. High speed industrial ethernet for semiconductor equipment. http://www.ethercat.org/pdf/english/Ethernet_for_Semi_0607.pdf, July 2006.

Appendix A

Performance measurement results

A.1 Generic driver

A.1.1 ecrt_master_receive()

```
Average: 7188.355957ns, modus: 6571ns, median: 6946ns, min:6510ns, max: 7876ns
6510:1, 6511:1, 6525:1, 6526:3, 6541:2, 6555:1, 6570:8, 6571:8, 6585:2, 6586:2,
    6600:3, 6601:1, 6615:2, 6616:3, 6630:1, 6631:2, 6645:4, 6646:2, 6735:1, 6751:1,
    6946:1, 7230:1, 7666:1, 7695:2, 7696:4, 7710:3, 7711:7, 7725:6, 7726:3, 7740:2,
    7741:7, 7755:1, 7756:1, 7771:1, 7785:1, 7786:4, 7801:4, 7876:2
statistics calculation time: 157681ns
Average: 7128.269043ns, modus: 7771ns, median: 6661ns, min:6495ns, max: 8221ns
6495:5, 6496:7, 6510:3, 6511:8, 6525:4, 6526:7, 6541:5, 6556:2, 6570:1, 6571:3,
    6585:1, 6586:2, 6601:1, 6661:1, 6780:1, 7755:1, 7756:1, 7770:3, 7771:14, 7785:6,
    7786:12, 7800:4, 7801:5, 7816:1, 7995:1, 8221:1
statistics calculation time: 134773ns
Average: 7205.311523ns, modus: 7786ns, median: 6825ns, min:6495ns, max: 7920ns
6495:1, 6496:2, 6510:2, 6511:4, 6525:6, 6526:4, 6540:4, 6541:4, 6555:2, 6556:4,
    6570:2, 6571:1, 6585:4, 6586:4, 6601:1, 6615:2, 6630:1, 6675:1, 6825:1, 6885:1,
    7770:1, 7771:4, 7785:4, 7786:16, 7800:4, 7801:10, 7816:6, 7831:2, 7905:1, 7920:1
statistics calculation time: 144510ns
Average: 7205.439453ns, modus: 7711ns, median: 7036ns, min:6451ns, max: 7951ns
6451:1, 6466:1, 6526:2, 6555:6, 6556:6, 6570:5, 6571:5, 6585:1, 6586:3, 6600:2,
    6601:2, 6616:3, 6631:1, 6645:1, 6646:2, 6660:1, 6661:3, 6706:1, 6721:1, 6751:1,
    6796:1, 7036:1, 7681:1, 7695:2, 7696:3, 7710:7, 7711:16, 7725:1, 7726:8, 7740:1,
    7756:3, 7786:1, 7801:1, 7816:1, 7830:1, 7831:1, 7891:1, 7906:1, 7951:1
statistics calculation time: 159616ns
Average: 7161.627441ns, modus: 7726ns, median: 6826ns, min:6526ns, max: 7920ns
6526:1, 6555:2, 6556:1, 6570:3, 6571:9, 6585:1, 6586:7, 6600:4, 6601:1, 6615:3,
    6630:1, 6631:2, 6645:4, 6661:1, 6676:3, 6690:1, 6720:1, 6721:1, 6736:1, 6750:2,
    6826:1, 7680:1, 7681:1, 7696:1, 7710:5, 7711:12, 7725:6, 7726:13, 7740:2, 7741:3,
    7755:1, 7756:1, 7770:1, 7771:1, 7846:1, 7920:1
statistics calculation time: 150855ns
Average: 7126.115234ns, modus: 7786ns, median: 6706ns, min:6466ns, max: 8086ns
6466:1, 6495:1, 6496:3, 6510:3, 6511:3, 6525:2, 6526:2, 6540:3, 6541:1, 6555:1,
    6556:4, 6570:5, 6571:4, 6586:1, 6600:2, 6601:7, 6616:2, 6645:1, 6660:1, 6661:1,
    6676:1, 6706:1, 6736:1, 6856:1, 7711:4, 7725:1, 7726:4, 7740:1, 7741:5, 7755:3,
    7756:3, 7770:5, 7771:4, 7785:1, 7786:9, 7800:2, 7801:3, 7816:1, 7890:1, 8086:1
statistics calculation time: 153541ns
Average: 7254.299316ns, modus: 7801ns, median: 6661ns, min:6481ns, max: 7861ns
6481:2, 6496:2, 6511:12, 6525:4, 6526:7, 6540:3, 6541:7, 6556:1, 6571:2, 6585:2,
    6586:2, 6600:2, 6601:1, 6631:1, 6646:1, 6661:1, 6766:1, 7771:3, 7785:2, 7786:11,
    7800:3, 7801:17, 7815:1, 7816:5, 7830:2, 7831:3, 7861:2
statistics calculation time: 136079ns
Average: 7173.921387ns, modus: 7801ns, median: 6735ns, min:6481ns, max: 8011ns
6481:1, 6496:4, 6510:2, 6511:8, 6525:1, 6526:6, 6540:1, 6541:4, 6556:5, 6570:3,
    6571:5, 6585:1, 6586:1, 6600:1, 6601:2, 6616:2, 6646:2, 6735:1, 7260:1, 7695:2,
    7710:1, 7711:2, 7725:4, 7726:1, 7741:1, 7770:2, 7785:5, 7786:9, 7800:5, 7801:10,
    7816:3, 7831:1, 7906:1, 7921:1, 8011:1
statistics calculation time: 152385ns
Average: 7133.125000ns, modus: 7711ns, median: 6720ns, min:6540ns, max: 7996ns
6540:2, 6541:3, 6555:2, 6556:8, 6570:2, 6571:10, 6585:1, 6586:6, 6601:4, 6616:5,
    6631:1, 6660:2, 6661:2, 6720:2, 7681:1, 7695:2, 7696:9, 7710:7, 7711:15, 7725:3,
    7726:9, 7741:1, 7786:1, 7830:1, 7996:1
statistics calculation time: 133919ns
Average: 7164.293945ns, modus: 6556ns, median: 7231ns, min:6525ns, max: 7801ns
```

```
6525:1, 6540:2, 6541:4, 6555:3, 6556:14, 6570:3, 6571:4, 6586:5, 6616:2, 6630:1,
    6631:3, 6646:2, 6661:1, 6736:2, 6781:1, 6796:1, 7231:1, 7666:5, 7680:6, 7681:7,
    7695:1, 7696:11, 7710:1, 7711:5, 7725:2, 7726:3, 7740:2, 7741:4, 7756:2, 7801:1
statistics calculation time: 142964ns
```

A.1.2 ecrt_master_send()

```
Average: 10387.635742ns, modus: 9421ns, median: 9601ns, min:9391ns, max: 51500ns
9391:2, 9406:3, 9421:12, 9422:1, 9436:7, 9451:3, 9465:1, 9466:3, 9481:6, 9496:4,
    9511:2, 9526:1, 9541:3, 9601:3, 9616:6, 9631:6, 9646:2, 9661:3, 9676:4, 9691:1,
    9692:1, 9706:1, 9721:2, 9736:3, 9751:1, 9766:3, 9796:1, 9826:1, 9842:1, 9871:1,
    9886:1, 9901:1, 9931:1, 10245:1, 15976:1, 16051:2, 16096:1, 16111:1, 16201:1,
    16306:1, 51500:1
statistics calculation time: 192604ns
Average: 9554.409180ns, modus: 9631ns, median: 9601ns, min:9376ns, max: 9931ns
9376:1, 9391:9, 9405:2, 9406:12, 9421:13, 9436:5, 9450:1, 9451:1, 9466:1, 9526:1,
    9556:3, 9601:2, 9616:12, 9631:15, 9646:8, 9661:4, 9676:3, 9691:2, 9751:1, 9796:1,
    9856:1, 9886:1, 9931:1
statistics calculation time: 134414ns
Average: 9529.884766ns, modus: 9421ns, median: 9556ns, min:9346ns, max: 9961ns
9346:1, 9361:1, 9376:3, 9391:8, 9406:4, 9421:12, 9436:11, 9451:4, 9466:3, 9511:1,
    9556:2, 9571:3, 9586:4, 9600:1, 9601:2, 9616:8, 9631:10, 9646:9, 9661:6, 9691:1,
    9706:1, 9721:2, 9766:1, 9781:1, 9961:1
statistics calculation time: 134878ns
Average: 9543.475586ns, modus: 9421ns, median: 9571ns, min:9346ns, max: 10096ns
9346:1, 9361:1, 9376:5, 9391:4, 9406:8, 9421:12, 9436:10, 9451:4, 9466:1, 9511:1,
    9526:1, 9571:2, 9586:4, 9601:3, 9616:8, 9631:10, 9646:4, 9661:8, 9676:6, 9691:1,
    9706:1, 9721:1, 9736:1, 9751:1, 9901:1, 10096:1
statistics calculation time: 137369ns
Average: 9485.859375ns, modus: 9361ns, median: 9571ns, min:9346ns, max: 9886ns
9346:5, 9360:1, 9361:14, 9375:1, 9376:12, 9377:1, 9391:10, 9406:2, 9407:1, 9451:1,
    9466:1, 9571:2, 9586:7, 9601:10, 9616:14, 9630:1, 9631:7, 9646:5, 9661:2, 9706:1,
    9811:1, 9886:1
statistics calculation time: 128833ns
Average: 9556.809570ns, modus: 9436ns, median: 9571ns, min:9361ns, max: 10171ns
9361:2, 9376:3, 9391:3, 9406:7, 9421:8, 9436:13, 9437:1, 9450:1, 9451:5, 9465:1,
    9466:2, 9526:1, 9541:1, 9556:1, 9571:2, 9586:3, 9601:4, 9616:5, 9631:8, 9646:8,
    9660:1, 9661:10, 9676:5, 9692:1, 9706:1, 9811:1, 9946:1, 10171:1
statistics calculation time: 142529ns
Average: 9526.508789ns, modus: 9631ns, median: 9586ns, min:9376ns, max: 10006ns
9376:1, 9391:4, 9406:12, 9421:12, 9436:14, 9451:1, 9481:1, 9496:1, 9511:1, 9526:1,
    9571:1, 9586:3, 9600:2, 9615:1, 9616:11, 9631:15, 9645:1, 9646:6, 9647:1, 9660:1,
    9661:4, 9662:2, 9691:1, 9781:1, 9826:1, 10006:1
statistics calculation time: 136018ns
Average: 9536.970703ns, modus: 9406ns, median: 9556ns, min:9361ns, max: 10021ns
9361:3, 9376:5, 9391:8, 9406:12, 9421:8, 9436:5, 9451:4, 9481:2, 9511:1, 9556:2,
    9570:1, 9571:1, 9586:4, 9600:1, 9601:6, 9616:5, 9617:2, 9631:8, 9646:5, 9661:5,
    9662:1, 9676:3, 9691:2, 9706:1, 9736:1, 9751:1, 9796:1, 9931:1, 10021:1
statistics calculation time: 144165ns
Average: 9497.200195ns, modus: 9391ns, median: 9556ns, min:9316ns, max: 9871ns
9316:1, 9331:8, 9346:1, 9361:3, 9375:2, 9376:10, 9390:1, 9391:14, 9406:3, 9421:2,
    9436:2, 9541:2, 9556:4, 9571:1, 9586:7, 9601:6, 9616:7, 9631:12, 9646:5, 9661:3,
    9706:2, 9721:1, 9751:1, 9796:1, 9871:1
statistics calculation time: 131353ns
Average: 9532.000000ns, modus: 9616ns, median: 9571ns, min:9346ns, max: 10052ns
```

```
9346:2, 9361:8, 9375:1, 9376:11, 9390:1, 9391:6, 9405:2, 9406:3, 9407:1, 9420:1,
    9421:4, 9436:2, 9451:4, 9466:1, 9556:2, 9571:2, 9586:2, 9601:11, 9616:13, 9631:7,
    9646:4, 9661:4, 9676:1, 9691:2, 9736:2, 9856:1, 9871:1, 10052:1
statistics calculation time: 143054ns
```

A.2 EtherCAT capable driver

A.2.1 ecrt_master_receive()

```
Average: 8272.509766ns, modus: 9166ns, median: 7951ns, min:6556ns, max: 21587ns
6556:1, 6570:1, 6585:2, 6601:1, 6615:1, 6616:1, 6630:2, 6645:2, 6646:1, 6660:1,
    6675:2, 6705:1, 6706:1, 6721:1, 6780:1, 6871:1, 6945:1, 7441:1, 7681:1, 7696:1,
    7726:1, 7741:1, 7755:1, 7756:1, 7770:1, 7786:1, 7800:2, 7801:1, 7816:1, 7830:1,
    7845:2, 7846:3, 7860:2, 7876:2, 7891:3, 7906:1, 7921:1, 7951:1, 7996:2, 8011:1,
    8055:1, 8101:1, 8116:1, 8131:1, 8145:1, 8146:2, 8161:1, 8176:2, 8205:1, 8206:3,
    8221:1, 8236:1, 8296:1, 8311:2, 8371:1, 8386:1, 8670:1, 9091:1, 9121:2, 9136:1,
    9151:1, 9166:3, 9195:1, 9211:1, 9226:1, 9256:2, 9271:2, 9286:1, 9301:2, 9316:2,
    9376:1, 9541:1, 9586:1, 21167:1, 21587:1
statistics calculation time: 188479ns
Average: 8269.500000ns, modus: 8116ns, median: 8011ns, min:6376ns, max: 23627ns
6376:2, 6450:1, 6465:1, 6511:2, 6556:1, 6571:1, 6585:1, 6586:1, 6691:1, 6751:1,
    6781:1, 6796:1, 6825:1, 6976:1, 7170:1, 7575:2, 7576:1, 7650:1, 7696:1, 7711:2,
    7741:1, 7755:1, 7771:1, 7786:2, 7801:3, 7816:2, 7831:1, 7845:1, 7846:2, 7861:1,
    7875:1, 7891:1, 7906:1, 7921:1, 7950:1, 7966:2, 7981:1, 7995:1, 8011:2, 8025:1,
    8040:1, 8041:1, 8056:1, 8086:1, 8101:2, 8115:1, 8116:5, 8130:1, 8131:1, 8146:1,
    8161:1, 8176:3, 8191:1, 8236:1, 8280:1, 8281:1, 8310:1, 8311:2, 8386:2, 8431:1,
    8476:1, 8836:1, 9060:1, 9091:1, 9106:1, 9181:2, 9196:1, 9241:1, 9286:3, 9301:1,
    9316:1, 9496:1, 9511:1, 9631:1, 9721:1, 21227:1, 23627:1
statistics calculation time: 185823ns
Average: 8372.583984ns, modus: 7786ns, median: 7891ns, min:6466ns, max: 25217ns
6466:1, 6480:1, 6555:1, 6585:1, 6586:1, 6661:1, 6691:1, 6735:1, 6825:1, 6826:1,
    7036:1, 7095:1, 7605:1, 7606:2, 7620:1, 7621:1, 7651:2, 7665:1, 7680:1, 7681:1,
    7696:3, 7711:2, 7726:1, 7741:1, 7756:4, 7786:5, 7801:2, 7816:1, 7831:1, 7846:1,
    7860:2, 7861:3, 7876:1, 7891:1, 7905:1, 7936:3, 7950:1, 7981:1, 7995:1, 7996:1,
    8011:1, 8041:4, 8055:1, 8056:2, 8085:1, 8086:2, 8100:1, 8101:2, 8131:4, 8160:1,
    8161:1, 8191:1, 8236:1, 8251:1, 8280:1, 8326:1, 8521:1, 8581:1, 9001:1, 9016:1,
    9120:1, 9121:1, 9181:1, 9255:1, 9271:1, 9316:1, 9361:2, 9376:1, 9736:1, 20672:1,
    21752:1, 25217:1
statistics calculation time: 181473ns
Average: 8278.861328ns, modus: 9346ns, median: 8026ns, min:6450ns, max: 24153ns
6450:1, 6480:1, 6510:1, 6525:1, 6526:1, 6540:1, 6556:1, 6571:1, 6585:1, 6615:1,
    6616:2, 6630:1, 6631:1, 6645:1, 6646:1, 6675:1, 6691:1, 6721:1, 6735:3, 6750:1,
    6751:1, 6810:1, 6811:2, 6855:1, 6886:1, 6901:1, 7515:2, 7621:1, 7636:2, 7666:2,
    7681:1, 7726:1, 7756:1, 7771:1, 7800:1, 7875:1, 7891:2, 7906:2, 7981:1, 8011:1,
    8026:1, 8056:1, 8086:1, 8101:1, 8116:1, 8175:1, 8176:2, 8190:1, 8191:3, 8206:1,
    8221:1, 8280:1, 8296:1, 8446:1, 8491:1, 8596:1, 8761:1, 9031:1, 9076:2, 9106:1,
    9121:2, 9166:1, 9211:3, 9241:1, 9286:1, 9300:1, 9301:2, 9316:3, 9331:1, 9346:4,
    9421:1, 9541:2, 9556:1, 9631:1, 10501:1, 20942:1, 24153:1
statistics calculation time: 190324ns
Average: 8348.465820ns, modus: 8131ns, median: 7890ns, min:6315ns, max: 25892ns
6315:1, 6435:1, 6436:1, 6451:1, 6465:1, 6555:2, 6556:1, 6615:1, 6660:3, 6735:1,
    6736:1, 6781:1, 6840:1, 6886:2, 6900:1, 7126:1, 7516:1, 7531:1, 7560:1, 7606:1,
    7621:3, 7650:1, 7651:1, 7665:1, 7666:1, 7696:1, 7725:1, 7726:2, 7741:1, 7755:1,
    7771:2, 7786:3, 7816:1, 7845:1, 7846:1, 7861:1, 7876:2, 7890:2, 7891:3, 7905:1,
    7965:2, 7966:1, 7981:1, 7996:2, 8011:1, 8025:1, 8026:2, 8041:1, 8055:1, 8056:1,
```

```
    8086:1, 8101:1, 8131:3, 8161:2, 8176:1, 8191:1, 8221:1, 8326:1, 8611:1, 8971:1,
    8985:1, 9001:2, 9031:2, 9106:1, 9136:2, 9165:1, 9180:1, 9256:1, 9286:1, 9331:1,
    9406:1, 9435:1, 9646:1, 9841:1, 21032:1, 23117:1, 25892:1
statistics calculation time: 187999ns
Average: 8250.683594ns, modus: 8130ns, median: 7936ns, min:6346ns, max: 21437ns
6346:1, 6465:2, 6495:1, 6675:1, 6706:1, 6781:1, 6976:1, 7486:3, 7500:1, 7530:1,
    7531:1, 7560:1, 7575:1, 7576:1, 7590:2, 7605:1, 7606:1, 7620:1, 7621:2, 7651:1,
    7666:1, 7681:2, 7695:1, 7696:1, 7710:3, 7711:1, 7725:1, 7726:1, 7740:1, 7771:1,
    7786:1, 7815:1, 7846:2, 7860:1, 7861:2, 7875:1, 7905:1, 7906:1, 7936:2, 7950:1,
    7951:1, 7966:2, 7981:1, 7995:1, 7996:1, 8011:1, 8026:2, 8055:1, 8056:2, 8070:1,
    8071:1, 8086:2, 8100:1, 8101:2, 8130:3, 8131:2, 8145:1, 8146:1, 8176:2, 8191:1,
    8206:2, 8221:1, 8251:1, 8266:1, 8295:1, 8311:1, 8446:1, 8716:1, 8746:1, 8970:1,
    9076:1, 9106:2, 9165:1, 9226:1, 9301:1, 16307:1, 20462:1, 21437:1
statistics calculation time: 192514ns
Average: 8114.049316ns, modus: 7906ns, median: 7906ns, min:6345ns, max: 21557ns
6345:1, 6346:1, 6405:1, 6406:1, 6450:3, 6451:1, 6480:1, 6481:2, 6495:1, 6525:2,
    6555:1, 6570:1, 6600:1, 6615:2, 6616:1, 6630:1, 6660:2, 6661:1, 6690:1, 6691:1,
    6706:1, 6720:1, 6870:1, 7471:1, 7516:1, 7545:1, 7546:1, 7561:2, 7620:2, 7651:1,
    7696:2, 7726:1, 7756:1, 7771:1, 7801:1, 7831:1, 7861:1, 7875:1, 7890:1, 7905:1,
    7906:4, 7921:1, 7936:1, 7951:1, 7966:1, 7981:1, 7995:1, 8010:1, 8086:1, 8101:1,
    8116:1, 8130:1, 8131:2, 8161:1, 8236:1, 8251:2, 8280:1, 9001:2, 9046:1, 9061:1,
    9076:2, 9106:1, 9135:1, 9151:3, 9166:1, 9180:1, 9196:1, 9211:2, 9226:2, 9240:1,
    9256:1, 9286:2, 9316:1, 9406:1, 9466:1, 9511:1, 10231:1, 20807:1, 21557:1
statistics calculation time: 192320ns
Average: 8328.524414ns, modus: 9106ns, median: 9046ns, min:6300ns, max: 27572ns
6300:1, 6345:1, 6346:2, 6360:2, 6361:2, 6375:2, 6376:2, 6390:1, 6391:1, 6405:1,
    6420:1, 6435:1, 6436:1, 6465:2, 6466:1, 6481:1, 6496:3, 6510:1, 6526:1, 6540:3,
    6541:3, 6555:1, 6556:1, 6585:1, 6586:1, 6600:1, 6601:1, 6615:1, 6616:2, 6631:1,
    6676:2, 6721:1, 6915:1, 7200:1, 9031:1, 9046:1, 9061:1, 9075:2, 9076:2, 9090:1,
    9091:1, 9106:4, 9121:1, 9136:2, 9151:2, 9166:2, 9195:1, 9196:2, 9210:1, 9211:3,
    9225:1, 9226:2, 9241:1, 9256:3, 9271:2, 9285:1, 9286:1, 9316:1, 9331:1, 9346:2,
    9361:2, 9376:1, 9391:1, 9436:1, 9466:1, 9631:1, 18707:1, 19578:1, 27572:1
statistics calculation time: 173072ns
Average: 8527.352539ns, modus: 9271ns, median: 8041ns, min:6511ns, max: 26463ns
6511:1, 6556:1, 6616:2, 6646:2, 6661:1, 6676:2, 6691:1, 6721:1, 6751:1, 6766:2,
    6781:1, 6796:1, 6811:2, 6841:1, 6960:1, 7111:1, 7141:1, 7246:1, 7651:3, 7665:1,
    7695:1, 7740:1, 7756:1, 7770:1, 7771:3, 7785:1, 7786:2, 7801:1, 7816:2, 7846:2,
    7860:1, 7906:2, 7921:1, 7936:1, 7981:1, 7996:1, 8041:2, 8056:1, 8070:1, 8085:1,
    8101:1, 8115:1, 8116:1, 8131:1, 8161:3, 8191:1, 8206:3, 8281:2, 8296:1, 8310:2,
    8311:1, 8371:1, 8386:1, 8401:1, 8776:1, 8881:1, 9181:1, 9211:1, 9241:3, 9271:5,
    9286:4, 9316:1, 9361:1, 9406:1, 9421:1, 9631:1, 9811:1, 9886:1, 22277:1, 22502:1,
    26463:1
statistics calculation time: 179718ns
Average: 8194.165039ns, modus: 9196ns, median: 8251ns, min:6450ns, max: 22952ns
6450:1, 6466:3, 6496:2, 6525:1, 6526:1, 6541:3, 6556:2, 6571:3, 6600:1, 6601:2,
    6616:3, 6631:1, 6645:2, 6646:3, 6691:2, 6706:3, 6721:4, 6751:1, 6766:1, 6796:3,
    6856:1, 6900:1, 6916:2, 6991:1, 7020:1, 7831:1, 8251:1, 8986:1, 9031:4, 9046:2,
    9076:1, 9106:2, 9136:1, 9151:3, 9166:1, 9195:1, 9196:6, 9211:4, 9226:3, 9241:1,
    9256:2, 9271:3, 9286:2, 9316:3, 9331:1, 9361:1, 9376:1, 9391:1, 9466:1, 9646:1,
    9826:1, 10397:1, 21588:1, 22952:1
statistics calculation time: 160772ns
```

A.2.2 ecrt_master_send()

```
Average: 3213.069336ns, modus: 3105ns, median: 3150ns, min:3046ns, max: 3495ns
```

```
3046:1, 3060:1, 3061:1, 3090:1, 3105:12, 3106:3, 3120:6, 3121:8, 3135:11, 3136:2,
    3150:7, 3151:1, 3165:5, 3166:1, 3180:2, 3181:1, 3195:1, 3255:1, 3300:1, 3315:1,
    3330:6, 3345:5, 3346:2, 3360:3, 3361:2, 3375:2, 3376:1, 3390:4, 3391:1, 3405:2,
    3420:1, 3436:1, 3465:1, 3480:1, 3495:1
statistics calculation time: 122997ns
Average: 3182.843018ns, modus: 3120ns, median: 3135ns, min:3016ns, max: 3450ns
3016:2, 3030:5, 3031:2, 3045:2, 3060:6, 3061:3, 3075:4, 3076:1, 3090:1, 3091:2,
    3105:8, 3120:10, 3121:2, 3135:9, 3150:3, 3151:1, 3166:1, 3240:2, 3255:1, 3256:1,
    3270:6, 3285:2, 3300:2, 3301:1, 3315:1, 3330:5, 3331:1, 3345:4, 3346:1, 3360:3,
    3361:2, 3375:3, 3390:2, 3450:1
statistics calculation time: 118902ns
Average: 3165.442383ns, modus: 3105ns, median: 3121ns, min:2985ns, max: 3600ns
2985:2, 2986:1, 3000:3, 3001:1, 3015:2, 3016:1, 3030:2, 3045:3, 3046:2, 3060:1,
    3061:1, 3075:4, 3076:1, 3090:6, 3091:1, 3105:9, 3106:1, 3120:8, 3121:1, 3135:8,
    3136:1, 3150:2, 3165:3, 3195:2, 3210:1, 3225:1, 3226:2, 3240:1, 3270:2, 3285:2,
    3286:3, 3300:1, 3315:3, 3330:3, 3345:5, 3360:5, 3375:4, 3600:1
statistics calculation time: 129868ns
Average: 3191.048584ns, modus: 3120ns, median: 3150ns, min:3015ns, max: 3465ns
3015:1, 3030:6, 3031:2, 3045:1, 3061:1, 3075:3, 3076:1, 3090:5, 3091:1, 3105:3,
    3106:2, 3120:9, 3121:3, 3135:8, 3136:1, 3150:7, 3151:4, 3165:2, 3166:2, 3180:2,
    3195:1, 3225:1, 3241:1, 3255:2, 3285:1, 3286:1, 3300:1, 3315:1, 3316:2, 3330:1,
    3331:1, 3345:5, 3346:1, 3360:7, 3375:3, 3390:3, 3391:1, 3406:1, 3450:1, 3465:1
statistics calculation time: 127858ns
Average: 3180.534668ns, modus: 3330ns, median: 3135ns, min:3001ns, max: 3645ns
3001:1, 3015:1, 3016:1, 3030:5, 3031:1, 3045:3, 3060:4, 3061:5, 3075:6, 3076:1,
    3090:6, 3091:2, 3105:7, 3120:6, 3135:7, 3150:2, 3151:1, 3165:1, 3166:1, 3196:2,
    3210:1, 3240:1, 3255:1, 3270:2, 3271:1, 3285:3, 3286:1, 3300:5, 3301:2, 3315:1,
    3330:7, 3331:1, 3345:1, 3346:1, 3360:2, 3375:1, 3376:1, 3390:1, 3405:1, 3465:1,
    3466:1, 3645:1
statistics calculation time: 136738ns
Average: 3181.072021ns, modus: 3120ns, median: 3121ns, min:2985ns, max: 3630ns
2985:1, 2986:1, 3000:1, 3015:2, 3030:2, 3031:1, 3045:3, 3046:1, 3060:4, 3061:2,
    3076:1, 3090:1, 3091:1, 3105:11, 3106:2, 3120:11, 3121:5, 3135:4, 3136:1, 3150:2,
    3151:1, 3165:2, 3166:2, 3180:2, 3195:2, 3211:1, 3256:1, 3285:4, 3300:1, 3315:1,
    3330:2, 3331:1, 3345:5, 3346:1, 3360:1, 3361:1, 3375:2, 3376:4, 3390:5, 3391:2,
    3435:1, 3630:1
statistics calculation time: 130678ns
Average: 3171.194580ns, modus: 3120ns, median: 3135ns, min:2985ns, max: 3630ns
2985:1, 3015:2, 3016:1, 3030:4, 3031:2, 3045:4, 3060:1, 3075:3, 3076:2, 3090:6,
    3091:3, 3105:2, 3120:13, 3121:4, 3135:5, 3136:1, 3150:5, 3151:4, 3165:1, 3180:1,
    3181:1, 3225:2, 3240:2, 3255:2, 3270:2, 3285:1, 3286:1, 3300:1, 3315:2, 3316:3,
    3330:5, 3331:1, 3345:3, 3360:2, 3361:1, 3375:2, 3390:1, 3406:1, 3495:1, 3630:1
statistics calculation time: 128863ns
Average: 3168.067871ns, modus: 3105ns, median: 3120ns, min:2986ns, max: 3615ns
2986:2, 3000:1, 3015:5, 3030:3, 3045:3, 3046:3, 3060:3, 3061:1, 3075:6, 3076:4,
    3090:5, 3091:1, 3105:7, 3106:4, 3120:3, 3121:2, 3135:4, 3136:1, 3150:3, 3151:1,
    3165:2, 3240:1, 3241:1, 3255:3, 3270:5, 3300:4, 3301:2, 3330:4, 3345:1, 3346:4,
    3360:4, 3361:2, 3375:1, 3406:1, 3421:1, 3436:1, 3615:1
statistics calculation time: 127078ns
Average: 3181.712891ns, modus: 3135ns, median: 3135ns, min:2956ns, max: 3540ns
2956:1, 2985:1, 3000:1, 3030:4, 3031:2, 3045:3, 3046:1, 3060:3, 3061:1, 3075:2,
    3090:5, 3091:3, 3105:8, 3106:3, 3120:3, 3121:1, 3135:9, 3150:8, 3165:2, 3166:1,
    3180:2, 3210:3, 3225:1, 3240:1, 3241:1, 3255:1, 3286:1, 3300:1, 3315:3, 3316:2,
    3330:3, 3331:1, 3345:5, 3360:1, 3361:1, 3375:2, 3376:1, 3390:2, 3391:1, 3405:1,
    3420:2, 3466:1, 3540:1
```

```
statistics calculation time: 139289ns
Average: 3199.409424ns, modus: 3165ns, median: 3165ns, min:2985ns, max: 3420ns
2985:2, 3000:2, 3015:3, 3016:1, 3030:4, 3060:2, 3061:2, 3090:1, 3105:6, 3120:3,
    3121:4, 3135:5, 3136:3, 3150:5, 3151:3, 3165:7, 3166:3, 3180:5, 3181:1, 3195:1,
    3210:2, 3225:1, 3240:2, 3255:1, 3270:2, 3271:2, 3286:1, 3300:1, 3301:1, 3330:2,
    3331:1, 3345:5, 3346:4, 3360:3, 3361:1, 3375:3, 3390:2, 3406:1, 3420:2
statistics calculation time: 130153ns
```

Appendix B

gianfar.c.diff

```
--- /data/projects/pmp/casroo/work/linux-ethercat/drivers/net/ethernet/freescale/gianfar.c
+++ /data/projects/pmp/casroo/work/linux-ethercat/net/ethercat/devices/gianfar.c
@@ -60,6 +60,8 @@
  *  conditions as for reception, but depending on the TXF bit).
  *  The driver then cleans up the buffer.
  */
+
+#define ecportdebug(s, ...) printk(KERN_INFO s, ## __VA_ARGS__)

 #define pr_fmt(fmt) KBUILD_MODNAME ": " fmt
 #define DEBUG
@@ -144,8 +146,10 @@
         const u8 *addr);
 static int gfar_ioctl(struct net_device *dev, struct ifreq *rq, int cmd);

-MODULE_AUTHOR("Freescale Semiconductor, Inc");
-MODULE_DESCRIPTION("Gianfar Ethernet Driver");
+void ec_poll(struct net_device *);
+
+MODULE_AUTHOR("Cas de Rooij <cas.de.rooij@prodrive.nl>");
+MODULE_DESCRIPTION("Gianfar EtherCAT Driver");
 MODULE_LICENSE("GPL");

 static void gfar_init_rxbdp(struct gfar_priv_rx_q *rx_queue, struct rxbd8 *bdp,
@@ -160,7 +164,6 @@
    lstatus |= BD_LFLAG(RXBD_WRAP);

  eieio();
-
  bdp->lstatus = lstatus;
 }

@@ -210,7 +213,7 @@
     } else {
       skb = gfar_new_skb(ndev);
       if (!skb) {
-        netdev_err(ndev, "Can't allocate RX buffers\n");
+        pr_err("%s: Can't allocate RX buffers\n", ndev->name);
```

```
         goto err_rxalloc_fail;
       }
       rx_queue->rx_skbuff[j] = skb;
@@ -472,6 +475,8 @@
 {
  int i;

+ /* Spinlocks are only necessary in Ethernet mode */
+ if (priv->ecdev == NULL)
  for (i = 0; i < priv->num_rx_queues; i++)
    spin_lock(&priv->rx_queue[i]->rxlock);
 }
@@ -480,6 +485,8 @@
 {
  int i;

+ /* Spinlocks are only necessary in Ethernet mode */
+ if (priv->ecdev == NULL)
  for (i = 0; i < priv->num_tx_queues; i++)
    spin_lock(&priv->tx_queue[i]->txlock);
 }
@@ -488,6 +495,8 @@
 {
  int i;

+ /* Spinlocks are only necessary in Ethernet mode */
+ if (priv->ecdev == NULL)
  for (i = 0; i < priv->num_rx_queues; i++)
    spin_unlock(&priv->rx_queue[i]->rxlock);
 }
@@ -496,6 +505,8 @@
 {
  int i;

+ /* Spinlocks are only necessary in Ethernet mode */
+ if (priv->ecdev == NULL)
  for (i = 0; i < priv->num_tx_queues; i++)
    spin_unlock(&priv->tx_queue[i]->txlock);
```

```
 }
@@ -617,17 +628,16 @@
  if (!np || !of_device_is_available(np))
    return -ENODEV;

+
  /* parse the num of tx and rx queues */
  tx_queues = (u32 *)of_get_property(np, "fsl,num_tx_queues", NULL);
  num_tx_qs = tx_queues ? *tx_queues : 1;
-
  if (num_tx_qs > MAX_TX_QS) {
    pr_err("num_tx_qs(=%d) greater than MAX_TX_QS(=%d)\n",
           num_tx_qs, MAX_TX_QS);
    pr_err("Cannot do alloc_etherdev, aborting\n");
    return -EINVAL;
  }
-
  rx_queues = (u32 *)of_get_property(np, "fsl,num_rx_queues", NULL);
  num_rx_qs = rx_queues ? *rx_queues : 1;

@@ -637,7 +647,6 @@
    pr_err("Cannot do alloc_etherdev, aborting\n");
    return -EINVAL;
  }
-
  *pdev = alloc_etherdev_mq(sizeof(*priv), num_tx_qs);
  dev = *pdev;
  if (NULL == dev)
@@ -646,12 +655,10 @@
  priv = netdev_priv(dev);
  priv->node = ofdev->dev.of_node;
  priv->ndev = dev;
-
  priv->num_tx_queues = num_tx_qs;
  netif_set_real_num_rx_queues(dev, num_rx_qs);
  priv->num_rx_queues = num_rx_qs;
  priv->num_grps = 0x0;
-
```

```
  /* Init Rx queue filer rule set linked list */
  INIT_LIST_HEAD(&priv->rx_list.list);
  priv->rx_list.count = 0;
@@ -681,7 +688,6 @@
        priv->tx_queue[i] = NULL;
  for (i = 0; i < priv->num_rx_queues; i++)
    priv->rx_queue[i] = NULL;
-
  for (i = 0; i < priv->num_tx_queues; i++) {
    priv->tx_queue[i] = kzalloc(sizeof(struct gfar_priv_tx_q),
            GFP_KERNEL);
@@ -835,7 +841,9 @@
 {
  struct gfar_private *priv = netdev_priv(dev);

- if (!netif_running(dev))
+ BUG_ON(priv->ecdev);
+
+ if ((priv->ecdev != NULL) || !netif_running(dev)) // TODO: nakijken
    return -EINVAL;

  if (cmd == SIOCSHWTSTAMP)
@@ -1088,6 +1096,9 @@
     priv->device_flags & FSL_GIANFAR_DEV_HAS_TIMER)
    dev->needed_headroom = GMAC_FCB_LEN;

+ // offer device to EtherCAT master module
+ priv->ecdev = ecdev_offer(dev, ec_poll, THIS_MODULE);
+
  /* Program the isrg regs only if number of grps > 1 */
  if (priv->num_grps > 1) {
    baddr = &regs->isrg0;
@@ -1163,6 +1174,10 @@
  priv->msg_enable = (NETIF_MSG_IFUP << 1 ) - 1;

  /* Carrier starts down, phylib will bring it up */
+
+ if (priv->ecdev == NULL) {
```

```
+   pr_debug("about to register device named %s (%p)...\n", dev->name, dev);
+
  netif_carrier_off(dev);

  err = register_netdev(dev);
@@ -1194,9 +1209,10 @@

  /* Create all the sysfs files */
  gfar_init_sysfs(dev);
+ }

  /* Print out the device info */
- netdev_info(dev, "mac: %pM\n", dev->dev_addr);
+ pr_info("%s: mac: %pM\n", dev->name, dev->dev_addr);

  /* Even more device info helps when determining which kernel
   * provided which set of benchmarks.
@@ -1208,6 +1224,11 @@
  for (i = 0; i < priv->num_tx_queues; i++)
    netdev_info(dev, "TX BD ring size for Q[%d]: %d\n",
          i, priv->tx_queue[i]->tx_ring_size);
+
+ if ((priv->ecdev != NULL) && ecdev_open(priv->ecdev)) {
+   ecdev_withdraw(priv->ecdev);
+   goto register_fail;
+ }

  return 0;

@@ -1257,7 +1278,7 @@

  netif_device_detach(ndev);

- if (netif_running(ndev)) {
+ if ((priv->ecdev != NULL) && netif_running(ndev)) {

    local_irq_save(flags);
    lock_tx_qs(priv);
```

```
@@ -1307,6 +1328,11 @@
  int magic_packet = priv->wol_en &&
         (priv->device_flags &
          FSL_GIANFAR_DEV_HAS_MAGIC_PACKET);
+
+ /* There Is no support for resume in EtherCAT mode */
+ if (priv->ecdev != NULL) {
+   return 0;
+ }

  if (!netif_running(ndev)) {
    netif_device_attach(ndev);
@@ -1345,7 +1371,8 @@
  struct gfar_private *priv = dev_get_drvdata(dev);
  struct net_device *ndev = priv->ndev;

- if (!netif_running(ndev))
+ /* There Is no support for restore in EtherCAT mode */
+ if ((priv->ecdev != NULL)|| !netif_running(ndev))
    return 0;

  gfar_init_bds(ndev);
@@ -1459,7 +1486,12 @@
    gfar_configure_serdes(dev);

  /* Remove any features not supported by the controller */
+ /* Only add gigabit support when the device is in Ethernet mode */
+ if (priv->ecdev == NULL)
+   priv->phydev->supported &= GFAR_SUPPORTED;
+ else
  priv->phydev->supported &= (GFAR_SUPPORTED | gigabit_support);
+
  priv->phydev->advertising = priv->phydev->supported;

  return 0;
@@ -1497,7 +1529,6 @@
   */
  if (phy_read(tbiphy, MII_BMSR) & BMSR_LSTATUS)
```

```
    return;
-
  /* Single clk mode, mii mode off(for serdes communication) */
  phy_write(tbiphy, MII_TBICON, TBICON_CLK_SELECT);

@@ -1836,11 +1867,16 @@

  /* If the device has multiple interrupts, register for
   * them. Otherwise, only register for the one
+  * The irqs should only be registered when the device doesn't
+  * run as EtherCAT master
   */
+
+ if (priv->ecdev == NULL) {
  if (priv->device_flags & FSL_GIANFAR_DEV_HAS_MULTI_INTR) {
    /* Install our interrupt handlers for Error,
     * Transmit, and Receive
     */
+
    if ((err = request_irq(grp->interruptError, gfar_error,
              0, grp->int_name_er, grp)) < 0) {
      netif_err(priv, intr, dev, "Can't get IRQ %d\n",
@@ -1869,8 +1905,8 @@
         grp->interruptTransmit);
      goto err_irq_fail;
    }
- }
-
+   }
+ }
  return 0;

 rx_irq_fail:
@@ -1901,12 +1937,14 @@

  gfar_init_mac(ndev);

+ if (priv->ecdev == NULL) {
```

```
  for (i = 0; i < priv->num_grps; i++) {
    err = register_grp_irqs(&priv->gfargrp[i]);
    if (err) {
      for (j = 0; j < i; j++)
        free_grp_irqs(&priv->gfargrp[j]);
      goto irq_fail;
+     }
    }
  }

@@ -1932,6 +1970,7 @@
  struct gfar_private *priv = netdev_priv(dev);
  int err;

+ if (priv->ecdev == NULL)
  enable_napi(priv);

  /* Initialize a bunch of registers */
@@ -2031,7 +2070,7 @@
  u32 lstatus;
  int i, rq = 0, do_tstamp = 0;
  u32 bufaddr;
- unsigned long flags;
+ unsigned long flags ;
  unsigned int nr_frags, nr_txbds, length, fcb_length = GMAC_FCB_LEN;

  /* TOE=1 frames larger than 2500 bytes may see excess delays
@@ -2046,7 +2085,6 @@
    if (ret)
      return ret;
  }
-
  rq = skb->queue_mapping;
  tx_queue = priv->tx_queue[rq];
  txq = netdev_get_tx_queue(dev, rq);
@@ -2092,6 +2130,7 @@
  /* check if there is space to queue this packet */
  if (nr_txbds > tx_queue->num_txbdfree) {
```

```
    /* no space, stop the queue */
+   if (priv->ecdev == NULL)
    netif_tx_stop_queue(txq);
    dev->stats.tx_fifo_errors++;
    return NETDEV_TX_BUSY;
@@ -2159,6 +2198,7 @@
      __skb_pull(skb, GMAC_FCB_LEN);
      skb_checksum_help(skb);
    } else {
+     if (priv->ecdev == NULL) // TCP Offload Engine
      lstatus |= BD_LFLAG(TXBD_TOE);
      gfar_tx_checksum(skb, fcb, fcb_length);
    }
@@ -2167,6 +2207,7 @@
  if (vlan_tx_tag_present(skb)) {
    if (unlikely(NULL == fcb)) {
      fcb = gfar_add_fcb(skb);
+     if (priv->ecdev == NULL) // TCP Offload Engine
      lstatus |= BD_LFLAG(TXBD_TOE);
    }

@@ -2179,6 +2220,7 @@
    if (fcb == NULL)
      fcb = gfar_add_fcb(skb);
    fcb->ptp = 1;
+   if (priv->ecdev == NULL) // TCP Offload Engine
    lstatus |= BD_LFLAG(TXBD_TOE);
  }

@@ -2212,6 +2254,7 @@
   * also must grab the lock before setting ready bit for the first
   * to be transmitted BD.
   */
+ if (priv->ecdev == NULL)
  spin_lock_irqsave(&tx_queue->txlock, flags);

  /* The powerpc-specific eieio() is used, as wmb() has too strong
@@ -2244,6 +2287,7 @@
```

```
343    * are full. We need to tell the kernel to stop sending us stuff.
344    */
345   if (!tx_queue->num_txbdfree) {
346 +   if (priv->ecdev == NULL)
347     netif_tx_stop_queue(txq);
348
349     dev->stats.tx_fifo_errors++;
350 @@ -2253,6 +2297,7 @@
351   gfar_write(&regs->tstat, TSTAT_CLEAR_THALT >> tx_queue->qindex);
352
353   /* Unlock priv */
354 + if (priv->ecdev == NULL)
355   spin_unlock_irqrestore(&tx_queue->txlock, flags);
356
357   return NETDEV_TX_OK;
358 @@ -2522,6 +2567,8 @@
359
360     bytes_sent += skb->len;
361
362 +   /* The EtherCAT Master Stack handles freeing the skb */
363 +   if (priv->ecdev == NULL)
364     dev_kfree_skb_any(skb);
365
366     tx_queue->tx_skbuff[skb_dirtytx] = NULL;
367 @@ -2530,11 +2577,16 @@
368             TX_RING_MOD_MASK(tx_ring_size);
369
370     howmany++;
371 +   if (priv->ecdev != NULL) {
372 +     tx_queue->num_txbdfree += nr_txbds;
373 +   } else {
374     spin_lock_irqsave(&tx_queue->txlock, flags);
375     tx_queue->num_txbdfree += nr_txbds;
376     spin_unlock_irqrestore(&tx_queue->txlock, flags);
377 - }
378 -
379 +   }
380 + }
```

```
+
+ if (priv->ecdev == NULL) {
  /* If we freed a buffer, we can restart transmission, if necessary */
  if (netif_tx_queue_stopped(txq) && tx_queue->num_txbdfree)
    netif_wake_subqueue(dev, tqi);
@@ -2544,6 +2596,7 @@
  tx_queue->dirty_tx = bdp;

  netdev_tx_completed_queue(txq, howmany, bytes_sent);
+ }

  return howmany;
 }
@@ -2551,7 +2604,9 @@
 static void gfar_schedule_cleanup(struct gfar_priv_grp *gfargrp)
 {
  unsigned long flags;
-
+ struct gfar_private *priv = gfargrp->priv;
+
+ if (priv->ecdev == NULL)
  spin_lock_irqsave(&gfargrp->grplock, flags);
  if (napi_schedule_prep(&gfargrp->napi)) {
    gfar_write(&gfargrp->regs->imask, IMASK_RTX_DISABLED);
@@ -2562,8 +2617,8 @@
     */
    gfar_write(&gfargrp->regs->ievent, IEVENT_RTX_MASK);
  }
+ if (priv->ecdev == NULL)
  spin_unlock_irqrestore(&gfargrp->grplock, flags);
-
 }

 /* Interrupt Handler for Transmit complete */
@@ -2756,7 +2811,6 @@
    if (unlikely(!newskb || !(bdp->status & RXBD_LAST) ||
          bdp->status & RXBD_ERR)) {
      count_errors(bdp->status, dev);
```

```
-
      if (unlikely(!newskb))
        newskb = skb;
      else if (skb)
@@ -2768,19 +2822,25 @@

      if (likely(skb)) {
        pkt_len = bdp->length - ETH_FCS_LEN;
+       if (priv->ecdev != NULL)
+       {
+         ecdev_receive(priv->ecdev,
+           (skb->data)+16,pkt_len-16);
+         dev_kfree_skb(skb);
+       } else {
        /* Remove the FCS from the packet length */
        skb_put(skb, pkt_len);
        rx_queue->stats.rx_bytes += pkt_len;
        skb_record_rx_queue(skb, rx_queue->qindex);
        gfar_process_frame(dev, skb, amount_pull,
              &rx_queue->grp->napi);
+       }

      } else {
        netif_warn(priv, rx_err, dev, "Missing skb!\n");
        rx_queue->stats.rx_dropped++;
        priv->extra_stats.rx_skbmissing++;
      }
-
    }

    rx_queue->rx_skbuff[rx_queue->skb_currx] = newskb;
@@ -2821,8 +2881,9 @@
  /* Clear IEVENT, so interrupts aren't called again
   * because of the packets that have already arrived
   */
+  /* Only do so when in Ethernet mode */
+ if (priv->ecdev == NULL)
  gfar_write(&regs->ievent, IEVENT_RTX_MASK);
```

```
-
  while (num_queues && left_over_budget) {
    budget_per_queue = left_over_budget/num_queues;
    left_over_budget = 0;
@@ -2832,10 +2893,10 @@
        continue;
      rx_queue = priv->rx_queue[i];
      tx_queue = priv->tx_queue[rx_queue->qindex];
-
      tx_cleaned += gfar_clean_tx_ring(tx_queue);
      rx_cleaned_per_queue =
        gfar_clean_rx_ring(rx_queue, budget_per_queue);
+
      rx_cleaned += rx_cleaned_per_queue;
      if (rx_cleaned_per_queue < budget_per_queue) {
        left_over_budget = left_over_budget +
@@ -2851,6 +2912,7 @@
    return budget;

  if (rx_cleaned < budget) {
+   if (priv->ecdev == NULL)
    napi_complete(napi);

    /* Clear the halt bit in RSTAT */
@@ -2901,6 +2963,17 @@
 }
 #endif

+void ec_poll(struct net_device *dev)
+{
+ int i;
+ struct gfar_private *priv = netdev_priv(dev);
+
+ for (i = 0; i < priv->num_grps; i++) {
+   gfar_poll(&priv->gfargrp[i].napi, 1);
+   gfar_receive(0, &priv->gfargrp[i]);
+ }
+}
```

```
+
 /* The interrupt handler for devices with one interrupt */
 static irqreturn_t gfar_interrupt(int irq, void *grp_id)
 {
@@ -2957,6 +3030,12 @@

      priv->oldduplex = phydev->duplex;
    }
+
+   /* Use the detected speed when the device is in Ethernet
+    * mode or 100Mbit when it's in EtherCAT mode
+
+   if (priv->ecdev == NULL)
+     phydev->speed = 100;*/

    if (phydev->speed != priv->oldspeed) {
      new_state = 1;
@@ -2988,6 +3067,7 @@
      }

      priv->oldspeed = phydev->speed;
+
    }

    gfar_write(&regs->maccfg2, tempval);
@@ -3005,7 +3085,12 @@
  }

  if (new_state && netif_msg_link(priv))
+ {
    phy_print_status(phydev);
+   /* Update the link status in the EtherCAT master stack as well */
+   if (priv->ecdev != NULL)
+     ecdev_set_link(priv->ecdev, priv->oldlink);
+ }
  unlock_tx_qs(priv);
  local_irq_restore_nort(flags);
 }
```

```
@@ -3269,7 +3354,7 @@
 /* Structure for a device driver */
 static struct platform_driver gfar_driver = {
  .driver = {
-   .name = "fsl-gianfar",
+   .name = "ec_fsl-gianfar",
    .owner = THIS_MODULE,
    .pm = GFAR_PM_OPS,
    .of_match_table = gfar_match,
```

Appendix C

Project Initiation Document

IGH ETHERCAT MASTER STACK

Project Initiatie Document

Datum voltooid: 18 Maart 2013
Auteur: Cas de Rooij

Versie: 2
Status: Concept

Bestandsnaam: Project Initiation Document IgH EtherCAT master stack.doc

Documenthistorie

Revisies

Versie	Status	Datum	Wijzigingen
1	concept	05-03-2013	Opzet document
2	concept	18-03-2013	Wijzigingen doorgevoerd naar aanleiding van feedback Cees van Tilborg

Goedkeuring

Dit document behoeft de volgende goedkeuringen:

Versie	Datum goedkeuring	Naam	Functie	Paraaf
2		Cees van Tilborg		
2		Micha Nelissen		

Distributie

Dit document is verstuurd aan:

Versie	Datum verzending	Naam	Functie
1	05-03-2013	Cees van Tilborg	Afstudeerbegeleider Fontys
1	05-03-2013	Bas van der Oest	Project Manager Prodrive
2	18-03-2013	Cees van Tilborg	Afstudeerbegeleider Fontys
2	18-03-2013	Bas van der Oest	Project Manager Prodrive
2	15-03-2013	Micha Nelissen	Afstudeerbegeleider Prodrive

Managementsamenvatting

Doel van dit document

Dit document heeft tot doel het project te definiëren, als basis te dienen voor het management ervan en de beoordeling van het succes van het project mogelijk te maken.

Dit document is bedoeld als contract tussen de afstudeerder, de school en Prodrive voor duidelijkheid over de scope van de afstudeeropdracht.

Aanleiding

EtherCAT is een veldbus, gebaseerd op Ethernet. Het gebruikt deels dezelfde hardware, en de pakketten komen ook overeen. Het grote voordeel van EtherCAT is echter dat het real-time werkt.

Voor het gebruik van de EtherCAT bus is een master nodig. Op het moment wordt daar nog een grote, dure computer voor gebruikt. Het is de bedoeling dat deze, bij bepaalde toepassingen, kan worden vervangen door een PowerPC module die goedkoper is en een kleinere voetafdruk heeft. Om dit te realiseren moeten de ethernet drivers van deze module worden omgeschreven naar EtherCAT drivers.

Globale aanpak

Ik ga eerst vooronderzoek doen op EtherCAT bus, hoe de drivers in grote lijnen werken, en wat er moet gebeuren aan de huidige ethernet drivers.
Hierna schrijf ik een Specification Document. Dit is een soort requirements document, en bevat alle eigenschappen van het eindproduct die voor de klant van belang zijn.
Vervolgens ga ik beginnen aan het Engineering Design Document(EDD). Dit document bevat alle informatie die nodig is voor de ontwikkelaars om nadat het product voltooid is, het product te kunnen onderhouden. Hierbij horen diagrammen over de opbouw van de software en uitgebreide uitleg over technische details.

Wanneer het EDD deels af is zal ik beginnen aan de implementatie van de software.
Ik zal eerst de software-interfacezijde implementeren, om vervolgens aan de hardwarezijde te gaan werken.
Na het implementeren wordt er een Qualification Plan Document geschreven. Hierin staan alle tests die moeten worden gedaan gedocumenteerd. Vervolgens zullen deze tests worden uitgevoerd en worden de resultaten in een Qualification Result Document gezet.
Hierna zal ik de afstudeerscriptie helemaal afwerken.

Doorlooptijd

De doorlooptijd is 18 weken.

Inhoudsopgave

Contents

1 Achtergrond

EtherCAT is een veldbus, gebaseerd op Ethernet. Het gebruikt deels dezelfde hardware, en de pakketten komen ook overeen. Het grote voordeel van EtherCAT is echter dat het real-time werkt.

Voor het gebruik van de EtherCAT bus is een master nodig. Op het moment wordt daar nog een grote, dure computer voor gebruikt. Het is de bedoeling dat deze, bij bepaalde toepassingen, kan worden vervangen door een PowerPC module die goedkoper is en met volume en energieverbruik een kleinere footprint heeft. Om dit te realiseren moeten de Ethernet drivers van deze module worden omgeschreven naar EtherCAT drivers.

Dit project wordt op de wijze doorlopen die bij Prodrive gebruikelijk is, zodat de documentatie en broncode uniform is met producten van andere projecten.

2 Projectdefinitie

2.1 Projectdoelstellingen

Wanneer het project voltooid is zal het mogelijk zijn een PowerPC module te gebruiken als EtherCAT Master stack. Dit is compacter en goedkoper dan de huidige oplossing.

2.2 Gekozen oplossing of aanpak

Ik zal voor het grootste deel de aanpak volgen die bij Prodrive gebruikelijk is. Op deze manier past het goed binnen het overkoepelende project, en is het gemakkelijk overdraagbaar omdat de documentatie een bekende vorm heeft.

2.3 Onderzoeksvragen

Hoofdvraag: Wat is de beste manier om Ethernet drivers voor het PowerPC platform om te schrijven naar EtherCAT drivers?

Deelvragen:

- Wat zijn de verschillen tussen de PowerPC Ethernet drivers, en bestaande EtherCAT drivers?
- Wat zijn de mogelijke manieren om Ethernet drivers voor het PowerPC platform om te schrijven naar EtherCAT drivers?
- Wat zijn de prestaties van de gekozen manier(en)?

2.4 Scope van het project

Het project omvat het omschrijven van de gianfar netwerk-driver voor het PowerPC platform, de PPA8548 zal als primair testplatform worden gebruikt. Er moet support zijn voor realtime functionaliteit van CONFIG_PREEMPT_RT gepatchte Linux kernel.

Redundante EtherCAT hoeft nog niet geimplementeerd te worden.

2.5 Producten c.q. eindresultaat

Er zullen de volgende producten worden opgeleverd:

- Specification Document
- Engineering Design Document
- Driver Source Code
- Qualification Result Document
- Afstudeerscriptie

Op de afstudeerscriptie na moeten alle producten aan de standaarden van Prodrive voldoen, en deze worden bij oplevering ook in het archiveringssysteem gezet.

Specification document

Dit document dient als "contract" tussen opdrachtgever en projectgroep waarin de requirements staan opgesteld.

Engineering Design Document

In dit document wordt omschreven hoe het product is ontworpen, en bevat alle informatie over het product die niet in de requirements waren opgenomen.

Driver Source Code

De broncode van de driver.

Qualification Result Document

Dit is het document waar de tests en testresultaten van de software in staan vermeld.

Afstudeerscriptie

De scriptie, compleet volgens richtlijnen van Fontys.

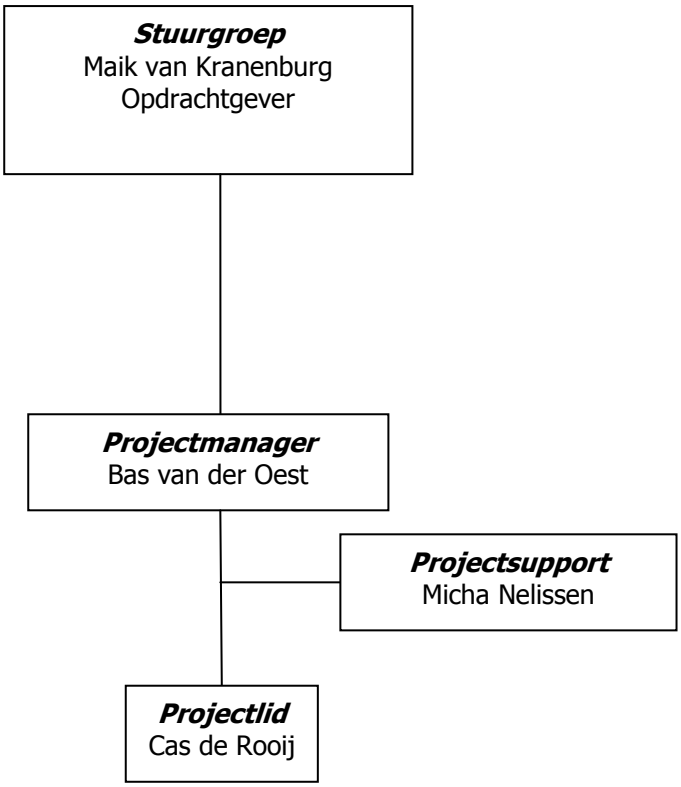
2.6 Uitsluitingen

De drivers hoeven nog geen ondersteuning te bieden voor redundante netwerken.

2.7 Benodigde Resources

Object	Kwantiteit
PPA8548 bordjes	2
Silicon Turnkey bord	1
ATX power supply	1
EtherCAT device	1

3 Projectorganisatiestructuur



3.1 Opdrachtgever

Maik van Kranenburg is de opdrachtgever. Hij heeft de eisen opgesteld, en de opgeleverde documenten en broncode moeten in ieder geval door hem gereviewd worden.

3.2 Projectmanager

Bas van der Oest is de projectmanager. Hij is verantwoordelijk voor het inplannen van de uren en het regelen van resources.

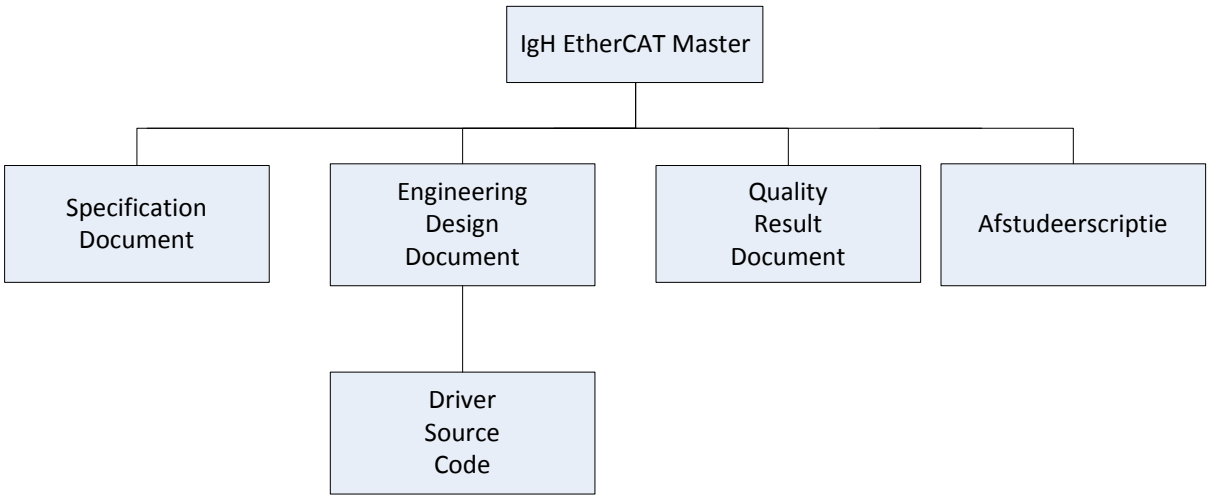
3.3 Projectsupport

Micha Nelissen is afstudeerbegeleider vanuit Prodrive, en daarnaast ook hoofdaanspreekpunt voor linux support.

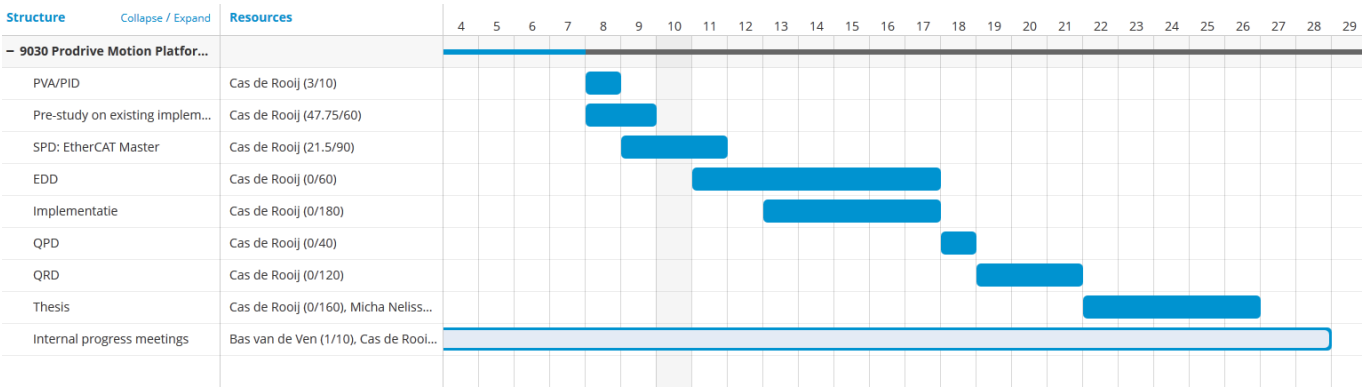
3.4 Projectlid

Cas de Rooij is als enige projectlid. Alle producten worden door hem gerealiseerd.

3. Productdecompositiestructuur



4. Planning



Dit is de basis van mijn planning. Ik ben alleen vanr plan om de implementatie in 2 stukken te verdelen; eerst de driver interface aanpassen, en vervolgens aan de hardwarezijde beginnen. Het is echter mogelijk dat ik bij het maken van het EDD er achter kom dat dat niet de meest handige planning is, dus dat gedeelte staat nog niet compleet vast.

QPD staat voor Qualification Plan Document. Dit gaat dus vooraf aan het QRD, wat voor Qualification Result Document staat.

5. Communicatieplan

Dit communicatieplan benoemt alle partijen die een (positief of negatief) belang hebben bij het project en de wijze waarop zij bij het project zullen worden betrokken en welke communicatievormen daarbij gebruikt worden.

Belanghebbenden bij het project

Wie	Namens	Belang	Communicatievorm
Maik van Kranenburg	Prodrive Holding	De drivers zijn onderdeel van een groter project.	Mail, telefoon, direct verbaal contact.

Communicatiekanalen

Van	Naar	Informatie	Medium	Frequentie of data
Cas de Rooij	Maik van Kranenburg	Opleveringen, requirements	Mail, telefoon, direct verbaal contact.	Geplande Review meetings bij release van ieder product en ongepland wanneer nodig
Cas de Rooij	Micha Nelissen	Planning, stagegerelateerde of technische vragen	Mail, telefoon, direct verbaal contact.	Geplande Review meetings bij release van ieder product, wekelijkse progress meeting, Ongepland wanneer nodig
Cas de Rooij	Bas van der Oest	Planning	Mail, telefoon, direct verbaal contact(meeting).	Wekelijkse progress meeting
Cas de Rooij	Cees van Tilborg	Voortgangsrapportage	Mail	Tweewekelijks
Cas de Rooij	Cees van Tilborg	Afwijking van het oorspronkelijke plan	Telefoon	Alleen indien nodig