MASSEY UNIVERSITY
NEW ZEALAND

DEVELOPING A BACKPLANE SOLUTION FOR THE TCS PCC

MASTERS THESIS

Thesis submitted in complete fulfilment of the requirements for the Master of Engineering

Author:

Thomas Johannes Laurentius
Leijen
tom@tjl.co.nz

Supervisor:

Dr Frans
Weehuizen
H.F.Weehuizen@massey.ac.nz

MASSEY UNIVERSITY
COLLEGE OF SCIENCES

CERTIFICATE OF REGULATORY COMPLIANCE

This is to certify that the research carried out in the Masterate Thesis entitled: Developing a backplane solution for the TCS PCC in the College of Sciences at Massey University, New Zealand:

- is the original work of the candidate, except as indicated by appropriate attribution in the text and/or in the acknowledgements;
- that the text, excluding appendices/annexes, does not exceed 40,000 words;
- all the ethical requirements applicable to this study have been complied with as required by Massey University, other organisations and/or committees, including TCS (NZ) ltd, which had a particular association with this study, and relevant legislation.

Please insert Ethical Authorisation code(s) here: (if applicable) _____

_____

Candidate's Name: Thomas Lejen
Signature:
Date: 17-8-11

Supervisor's Name: Frans Kleehuizen
Signature:
Date: 1-8-11

OGS10

MASSEY UNIVERSITY

APPLICATION FOR APPROVAL OF REQUEST TO EMBARGO A THESIS

(Pursuant to AC 98/168 (Revised 2), Approved by Academic Board 16.02.99)

Name of Candidate: Thomas Leijen ID Number: 05080657

Degree: Master of Engineering Dept/Institute/School: College of Sciences

Thesis Title: Developing a backplane solution for the TCS PCC

Name of Chief Supervisor: Dr Frans Weehuizen Telephone Extn: ~~62145~~ 04 9200466 ext 922

As author of the above named thesis, I request that my thesis be embargoed from public access until March 2013 for the following reasons:

- ✓ Thesis contains commercially sensitive information and may be applicable for a patent.
- ▪ Thesis contains information which is personal or private and/or which was given on the basis that it not be disclosed.
- ▪ Immediate disclosure of thesis contents would not allow the author a reasonable opportunity to publish all or part of the thesis.
- ▪ Other (specify): _____

Please explain here why you think this request is justified:

Massey University contract number 15226 titled: "TechNZ Supplementary Agreement" which was signed by the fellow (Tom Leijen), representative of the company (Peter Tait) and representative of Massey University (Mark Cleaver) section 5.2 states: "Approval of publications: Any disclosure of any information relating to the Project in which a party has proprietary interest must be approved by that party (Approving Party). The Approving Party will not withhold approval unless the information is commercially sensitive. Where the Approving Party withholds approval, then the parties will consult to determine whether amendments can be made to the publication so that approval to disclose the information will be granted. The Approving Party may place – and the other parties will comply with – an embargo on publication of research results from the Project for a period of up to two years from the Project's completion, if the information is the appropriate subject matter for a patent application." Since this thesis contains information that is commercially sensitive to TCS, they have requested that an embargo be placed on the publication of this thesis.

Signed (Candidate): _____ Date: 17-8-11

Endorsed (Chief Supervisor): _____ Date: 1-8-11

Approved/Not Approved (Representative of VC): _____

Date: 10/10/11

3

Acknowledgements

Thanks to my mentor and lecturer Frans Weehuizen for being supportive, inspirational and educational throughout my time at Massey University.

Thanks to my parents Herbert and Cunie Leijen for providing financial support and motivation to keep excelling throughout my education at Massey University.

Thanks to TCS and their staff for providing a great study environment and providing me the opportunity to gain valuable industry experience alongside a valuable education.

Abstract

TCS is company that specialises in systems integration and development of products for industrial application. Their flagship product for distributed control is the CPU5. This project covers the design and development of a backplane solution to solve the limitation of the CPU5 that it only has a limited number of inputs and outputs. The next iteration of the CPU5, is to be called the PCC.

Background research was carried out into the needs of TCS and their target markets. The IEC61499 standard, Windows CE platform and ISaGRAF software environment were also investigated.

The project planning was analysed to gain an understanding of R&D project planning. This project concluded that it was important to be able to make changes to the plan as the project proceeds so that it can still be completed by the deadline.

TCS had set the limitation that the backplane protocol must be a serial bus. Various different types of serial bus were investigated including CAN, USB, Light Peak (optical) and EIA485. The bus selected was a CAN bus with a custom protocol.

Research was carried out into the operation of existing bus drivers and protocols such as DeviceNet and CANOpen to determine an appropriate protocol for a serial bus based backplane. Software was developed to manage devices attached to the backplane.

Hardware and software for a digital I/O module was developed. Various ways of updating the I/O were investigated, an event-based and cyclic updating method was implemented with an adjustable debounce time to maximise performance and robustness.

Hardware and software for a serial port module was developed. A major challenge faced was to transmit a stream based protocol across a frame based CAN backplane.

Software for an Ethernet module was developed. Software design included development of an NDIS compatible miniport driver for Windows CE and software for the slave module.

To demonstrate that the PCC is an effective platform for distributed control, a test setup was created to be shown at a trade show in Nuremberg. The trade show setup was fully programmed in compliance with the IEC61499 standard and demonstrated the power of the PCC.

Contents

Introduction

TCS is a systems integrator as well as a developer of products for industrial control. TCS has over 10 years of history developing distributed control products; their current flagship product is the CPU5. The CPU5 was developed as a platform for the ISaGRAF software. ISaGRAF is a control software environment that supports the IEC61131 (ladder logic) and IEC61499 (distributed control using function blocks) standards. Although TCS has sold many CPU5 modules, they have not yet captured the market share they intended to. A few reasons for the CPU5 not being as successful as anticipated include: cost, size, lack of inputs and outputs (I/O) and component availability.

This project is part of the next major design iteration of the CPU5, the development of the CPU6. To give the CPU6 more appeal in a market where devices are named by acronyms such as PLC (Programmable Logic Controller), it has been given the name PCC which stands for “Programmable Cell Controller”. The PCC will be built with newer components so that it is smaller, more powerful and cheaper but will run a similar software platform (ISaGRAF installed on a Windows CE platform) to that installed in the CPU5. One significant difference between the CPU5 and the PCC will be that the PCC will have a system where additional I/O modules can be attached to the controller in a similar way to how traditional programmable logic controllers (PLCs) using a backplane for I/O modules. TCS is already experienced in the development of embedded controllers so they will be in charge of developing the main controller as it will only be a slight iteration from the CPU5. Because of its complexity, TCS staff will also develop the hardware for the Ethernet module. The part of the project that is completely new to TCS is the development of the backplane hardware and software to allow additional I/O modules to be attached to the PCC controller.

The parts of this project discussed in this thesis include: carrying out background research, developing a project plan, selecting a communications protocol for the backplane, developing hardware and software for the backplane, developing hardware and software for a digital I/O module, developing hardware and software for a serial port module, developing software for an Ethernet port module and demonstrating the project running an IEC61499 compliant program at a trade show.

Aim

Develop and test hardware and software for a backplane solution that allows extra I/O to be attached to TCS's latest iteration of their flagship distributed control product (the PCC). The backplane solution developed must be as flexible as (or more flexible than) current PLC rack and remote I/O solutions. Hardware and software to be developed include a digital I/O module, a serial port module and an Ethernet/IP port module. The modules developed are some of the most commonly available types and will provide platforms from which it is possible to develop additional types of I/O modules. The software developed must support Windows CE and the ISaGRAF software and be compatible with the latest in industrial control standards. The backplane and the I/O modules will be put through various performance tests to determine whether they are ready to be implemented in an industrial situation and which parts will need to be focussed on in future development. The effectiveness of the project will be tested by developing a sample application for a real-life industrial situation and demonstrating this working on the newly developed hardware at a trade show in Nuremberg.

Objective

Complete the tasks listed in the aim within 12 months using research resources sourced from Massey University and any other resources sourced from TCS that fit within the project budget. Although the final goal is to produce a product that can be released into industry, care needs to be taken that the project is also academically relevant and that emphasis is put on parts of the project that fall outside TCSs areas of expertise. Priority is to be put on the parts of the project that fall outside TCSs areas of expertise such as the development of the backplane, the I/O modules and performance testing. Other features, such as product documentation, error reporting and some hardware development, that are needed to make the project ready for release into industry will be of a lower priority.

1. Background Research

1.1 Introduction

This section includes all the background research carried out relevant to the project. It contains a description of the company sponsoring the project, a description of the IEC61499 standard which is relevant to the control methodology that the project has to be compatible with, a description of Windows CE which is the operating system that the software developed has to be compatible with and a description of ISaGRAF which is the program that will be running all the control algorithms that drive the I/O modules.

1.2 TCS ltd

About the company

TCS is a New Zealand owned and operated industrial automation and technology company that has been operating for over 25 years. Their customers are involved in a wide range of industries such as dairy, food, beverage, meat, timber, paint, quarries and fertiliser and have projects ranging from standalone equipment to complex fully integrated plants. TCS has a strong reputation for industrial communications that interface to a wide range of industrial devices and can also link with management network systems.

TCS areas of expertise are:

- Industrial automation
- Industrial communications
- Management database integration
- Production traceability
- Electronic micro-processor based technology and products
- Distributed intelligence

There are 3 interrelated engineering teams within TCS:

- The Industrial Automation team specialises in industrial automation, communications, traceability and management database integration using industrial hardened devices such PLC, SCADA / HMI and industrial communications technologies.
- The Product Development team is a highly innovative team that designs, develops and builds microprocessor based technology for a range of agricultural and industry applications, for end users or for Original Equipment Manufacturers (OEMs). The team is continually researching and developing new technologies.
- The Manufacturing team assembles, tests and quality controls all TCS products

(TCS (NZ) Ltd)

The CPU5 project

TCS has been developing distributed control products for over 10 years. At the start of this project TCS's flagship product for distributed control was the CPU5. The CPU5 is an industrial thin client controller running ISaGRAF on a Windows CE 5.0 platform. The CPU5 is equipped with the following ports: Ethernet, CAN, RS232, RS422 and RS485. There is also a version with Wi-Fi. The CPU5 however has a number of limitations that are preventing TCS from obtaining the market share they hoped to get with their distributed products. These limitations include:

- Price: The hardware and components used are too expensive compared to similar products on the market.
- Component availability: The PXA255 microcontroller used in the CPU5 is becoming obsolete and will not be available on the market for much longer.
- Cost of adding I/O modules: TCS currently does not produce I/O modules so has to purchase DeviceNet modules from other vendors to add extra I/O.

(Meek, 2010)

1.3 The IEC 61499 standard

Why it is being developed

The trend of growth in industrial control systems is a movement toward more integrated and agile systems. With concepts such as lean manufacturing becoming more popular it is important that systems are able to rapidly change from one production process to another. Industrial control systems therefore need to be easier to adapt and be changed in a quick manner. Integrating plant floor control with logistics, accounting and business management systems plays a very important role in improving the agility of a system/business. Globalisation of production is also becoming more common so there are developing needs for integrating production systems in one country with offices in other countries.

The well-established standard for industrial automation is the IEC61131-3 standard (full description of this protocol is beyond the scope of this project) which has still got several flaws, these are listed below:

- Not distributable over multiple resources.
- Execution order is not always clearly defined.
- It is not flexible enough, not all blocks can be connected directly etc.
- IEC61131 is of a scanned nature so it cannot be distributed over large networks.

The IEC61499 standard has been developed as a logical development of the IEC61131-3 standard.

One of the major benefits of the IEC61499 standard is that function blocks can be understood by mechanical, electronic and software engineers: They can be seen as components of a physical system, of an electronic circuit, or as objects of an object oriented program.

(Lewis, Introduction, 2008)

Present standard

The IEC61499 standard consists of 4 main parts, these are:

- The system model.
  The system model describes the relationships between communicating devices and applications. It describes how devices are connected together and what sort of communication links they use. The system model also shows how applications (software) are divided amongst the resources (hardware) in the system.
- The resource model.
  The resource model encompasses the physical resources in a production system, these resources range from smart sensor controllers to administration servers. The resource model describes how the controllers are connected to the machinery and provides communication interfaces between the various resources in a production system.

- The application model.
  The application model encompasses the software, in the case of the IEC61499 standard it is a network of interconnected function blocks. An application can be distributed over multiple resources. A sub application is a type of function block that is constructed from networks of basic and composite function blocks designed to provide a re-usable part of an application that can be distributed over many resources.
- The function block model.
  This is the core of the standard. A function block can represent anything from a single sensor in a plant to a complete division of a company. A function block has the following features:
  - Type name and instance name
  - Event input(s)
  - Event output(s)
  - Data input(s)
  - Data output(s)
  - Internal variables

  Behaviour is defined in terms of algorithms and state information, these algorithms can be text based, or they can be in function block diagram format.

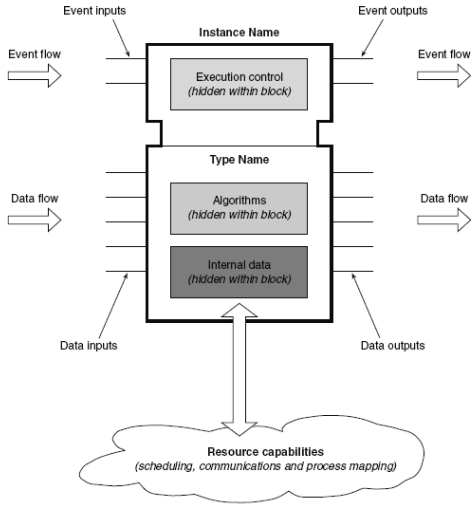


Figure 1: Function block model

In the future it is anticipated that a function block diagram will be able to be compiled into a textual format which is legible and understandable by almost anyone, even people without programming experience, this will make understanding of processes, improvement and fault finding much easier.

(Lewis, IEC 61499 models and concepts, 2008)

Current research

An article written by Jeffrey Yan and Valeriy Vyatkin discusses an implementation of IEC61499 on an airport baggage handling system (BHS). Baggage handling systems consist of many interconnected conveyors that are laid out according to airport requirements, this layout is likely to change over time as new bag routes are developed, equipment is upgraded or equipment malfunctions occur. Traditional BHS are controlled by a monolithic control program based in a centralised controller unit; this means that the program will need to be changed with every slight change to the conveyor layout. The researchers suggest a method using IEC61499 which will greatly improve the control methodology of the baggage handling system. To prove this they used a software package by ICS Triplex called ISaGRAF (this software is discussed later in this report). A number of smart conveyors (conveyor with control and sensing functionality) were set up with the ISaGRAF runtime. Using the ISaGRAF Workbench tool, function blocks were developed for each individual conveyor and these were networked together to form a model of the entire conveyor system.

In IEC61499 terms, the function block representing a conveyor in a BHS, receives an event input when a bag is passed onto the conveyor, bag specific data, such as destination, is also passed to the block through the event inputs, when the bag leaves the conveyor, the data and an event is passed to the next conveyor.

Because a BHS can be quite a complex system, it is likely that there are multiple ways to transport a bag from the source to the destination, this has the benefits of redundancy and also efficiency but can get quite complex to control. Where traditional systems would have a central unit observing the state of the system, or even just fixed routes, Yan and Vyatkin suggest implementing a Bellman Ford algorithm (similar to that used to route IP packets over the internet). Using the Bellman Ford algorithm, individual function blocks (representing conveyors with multiple outputs) can maintain an up to date routing table (by passing data to the function blocks before and after them) to automatically determine where to send the bag as to ensure it reaches its destination in the fastest possible way. (Yan & Vyatkin, 2010)

The benefits of the IEC61499 BHS control methodology proposed by Yan and Vyatkin include:

- Decentralised processing adds redundancy to a system; no longer will the whole system fail if one part of it fails.
- Increased baggage transfer speeds, with the modules re-calculating the fastest route (using the Bellman Ford algorithm) at regular intervals, bags will always reach their destination in the fastest possible way.
- Ease of re-arranging the BHS or adding extra routes. It will simply be a case of adding a few extra function blocks representing the new conveyors/apparatus using the Workbench tool. The Bellman Ford algorithm will then automatically recalculate all the routes through the system.
- Instead of viewing the control software as several pages of structured text or ladder logic, it is represented in the much more intuitive form of function blocks.
- Because bag specific data is maintained within the database of the individual conveyors data access is a lot faster and there is no need for large amounts of storage in a central database.

The concepts researched by Yan and Vyatkin can have a wide range of applications other than baggage handling in industries ranging from materials handling to food processing.

1.4 Windows CE 5

"Microsoft® Windows® CE 5.0 is an open, scalable, 32-bit operating system (OS) that integrates reliable, real time capabilities with advanced Windows technologies. Windows CE allows one to build a wide range of innovative, small footprint devices. A typical Windows CE–based device is designed for a specific use, it often runs disconnected from other computers, and requires a small OS that has a deterministic response to interrupts. Examples include enterprise tools, such as industrial controllers, communications hubs, and point-of-sale terminals, and consumer products, such as cameras, Internet appliances, and interactive televisions." (Microsoft)

The quote states that the Windows CE operating system "is an open, scalable, 32-bit operating system"; the operating system is in fact not completely open source. Only the parts that Microsoft has determined are relevant for developers of embedded devices, such as hardware interfaces are open source.

Brief history

Windows CE was first released as a scaled down version of Windows 95 in November 1995. The project from which Windows CE originated was the WinPad project. The WinPad project was developed to drastically change the way users interacted with handheld devices, unfortunately "ahead of its time" features such as handwriting recognition could not be supported due to hardware limitations at the time, the death of this project led to the introduction of the Windows CE project. Windows CE 5.0 was announced at the start of 2003 and released in Q3/4 of 2004. The current version of Windows CE is version 6.0. Version 7.0 is however on the horizon with an anticipated release date of Q3/4 2010 (Tilley, 2001)

TCS is currently using Windows CE 5.0 as it was the latest version when they started using it to develop drivers and software for the CPU5 project.

Platform Builder

"Microsoft® Platform Builder for Windows CE is an integrated development environment (IDE) for building customized embedded operating system (OS) designs based on the Microsoft Windows® CE OS. Platform Builder comes with all development tools necessary for one to design, create, build, test, and debug a Windows CE–based OS design. The IDE provides a single integrated workspace where one can work on both OS designs and projects." (Microsoft)

Platform builder is the software package used by TCS to write software and drivers for their windows CE operating system. Platform Builder contains all the source code for the windows CE operating system and allows the development of applications and board specific source code (board specific source code is known as a BSP – Board Support Package). Platform builder contains the source code for many common hardware drivers and allows these to be copied and customised to suit different types of hardware. For this project we will be creating a new BSP for the CPU6 (PCC) hardware by modifying and adding to the BSP made for the CPU5 project. Platform builder also contains many tools for debugging such as the Remote Registry Editor (allows one to observe and edit the Windows CE registry) and the Remote Kernel Tracker (allows one to see which software routines/threads are being executed in the CPU).

Current uses

Windows CE is commonly known to be used in the PDA (Personal Digital Assistant), handheld computer and mobile phone markets. Because Windows CE is a customisable RTOS (Real Time Operating System), it is however used in many more markets. A few examples of applications for Windows CE, outside of the aforementioned markets, include GPS (Global Positioning Satellite) navigation systems, automotive display units, remote storage, media players (set top and handheld), POS (point of sale) devices industrial HMI displays and industrial controllers. The advantage of using a Windows CE based device over using a PC with a traditional Windows operating system is that it can be made a lot more lightweight and the ability to create high priority system interrupts allows Windows CE devices to be a lot faster and a lot more reliable.

1.5 ISaGRAF 5.0

About ISaGRAF

"ISaGRAF is a control software environment that enables you to create local or distributed control systems. It offers a combination of a highly portable, robust control engine (Virtual Machine) and an intuitive application development environment (Workbench). ISaGRAF 5 is the world's first automation software to be compliant with both IEC 61131 and IEC 61499 industrial standards. This leading-edge software is comprised of a powerful set of new features that promise to change the way you build your control systems." (ICS Triplex)

Figure 2: ISaGRAF market coverage (ICS Triplex)

The figure above shows how ISaGRAF works in combination with various products and markets. ISaGRAF can be installed on any of the hardware platforms listed but the platform must be running one of the listed operating systems, the resultant PLC can then be used to automate systems in any of the listed markets. TCS has decided to, to best suit their target markets, install ISaGRAF on a microcontroller running the Windows CE operating system.

ISaGRAF 61499 environment

While designing the ISaGRAF IEC61499 environment, ICS Triplex research engineers kept the following factors in mind:

- The IEC61499 implementation would complement the existing IEC61131 implementation so that both IEC61131 and IEC61499 could co-exist in the system.
- The state machine used in the controller would use the IEC61131 language with a few slight changes to allow IEC61499 to function. This means that design engineers can still use ISaGRAF without needing to learn a new language.
- The basic logic engine would be cyclic, like the traditional PLC logic implementation. This was done to preserve determinism and stability and to entice other automation vendors to adopt the standard.
- The producer consumer communication model was adopted because it can be implemented on any communication stack such as Ethernet IP, DeviceNet, Profibus etc. This will make it more compatible with other systems.
- IEC61499 function blocks would live within a resource as a program organisation unit like any other in the 61131 environment.
- IEC61499 function blocks will be autonomous and only able to be manipulated by data it is consuming and producing.

(Chouinard, 2007)

1.6 Conclusion

The background research allowed a broad understanding to be gained of the concepts involved with the project and the history and future of TCS distributed control products. Having this understanding gives a clearer picture of what is required of the project and what is possible with the available tools.

2 Project planning

2.1 Introduction

This section describes the timeline developed to complete the project as described in the project description (the project description can be found earlier in this report) and discusses the process of project planning that was involved with the project. This section shows the initial timeline and then describes the change made halfway through the year and discusses why it was made.

2.2 Developing the initial timeline

The initial timeline was developed together with Nathan May and Andrew Meek, the two primary R&D staff at TCS. Nathan and Andrew both have many years of experience in project management and Andrew had previously completed a Masters Project with TCS.

The timeline was planned out as follows:

15 – 19 February: Developing a plan for the backplane driver loading process.

22 – 27 February: Finding out how a Windows CE bus driver interacts with a bus agnostic driver.

1 - 27 March: Select an electrical layer and protocol for the backplane.

29 March to 30 April: Developing a bus driver and bus agnostic driver for the 8 port I/O module.

3 May to 18 June: Developing drivers for the 8 port I/O and serial port modules and PCB development for these modules.

21 June to 2 July: Developing ISaGRAF function blocks for the I/O and serial port modules.

5 -23 July: Investigate networking architectures such as PROFI, IEP, ETCP etc...

26 July to 20 August: Develop APIs to interface other TCS software with the PCC system.

23 August to 17 September: Building a test trial setup to test the PCC and to demonstrate the system at a trade show in Nuremberg.

18 September to 29 October: Clean up software and work on thesis.

1 November to 24 December: Develop other modules such as analogue I/O and Ethernet.

Remaining project time: Clean up completed software and prepare final copy of thesis.

2.3 Timeline update in July

During the first week of July the timeline was revisited with Andrew Meek (TCS), Nathan May (TCS) and Frans Weehuizen (the University staff member in charge of the project).

At that point in time the project stages had been completed up to 18 June, making the project effectively two weeks behind schedule. The reason the project was behind schedule was the complexity of the plug and play system and the complexity of getting a stream based protocol such as EIA232 to work on a backplane with a frame based protocol (CAN data bus).

After some discussions with Frans Weehuizen, Nathan May and Andrew Meek, the decision was made to replace the stages: “Investigate networking architectures such as PROFI, IEP, ETCP etc...” and “Develop APIs to interface other TCS software with the PCC system” with “Further development and Performance testing of the 8 port I/O module and serial port module”. It was also decided to prioritise the development of a 2 port Ethernet module.

It was decided to replace the “networking architecture investigation” and “API development” stages because they were less relevant to what TCS required out of the Project. It was decided that by further developing and testing the digital I/O and serial port modules and developing an Ethernet module, it was more likely that the project would come out with a product that could be sold. TCS has previously carried out research into networking architectures and they already had a range of APIs which were developed for the CPU5 and it was decided that those could be applied to the PCC.

2.4 Recommendations

The project planning element of this project highlighted the difficulties involved with planning and carrying out such a large project. To improve the running of such a project in the future it is recommended to follow a few of the following practices:

Regular meetings

When TCS research and development (R&D) staff are working on a project they hold weekly, sometimes daily, meetings to catch up on the project progress. Regularly discussing the progress of a project would give a better overview of the project as talking about something often gives a clearer picture than just doing something blindly. Meeting with other staff will also allow staff to discuss ways to speed up the process and alternative methods of reaching the goal.

Contingency plans

When dealing with unknowns it is always good to have a contingency plan so that a product can still be developed within the set time frame. For example, if an essential part of the project such as: "Developing drivers for the serial port modules" turns out to be more work than expected, a particular nice-to-have part such as "Investigate networking architectures such as PROFI, IEP, ETCP etc..." can be left out in favour of just using the established ETCP networking architecture.

2.5 Discussion

It was interesting to see that although this project timeline was developed by engineers with many years of experience in planning R&D projects, the timeline still needed significant changes to ensure the project was completed before the deadline. This demonstrates that no matter how much experience one has in planning R&D projects, the project will never go exactly as planned. The difference between planning an R&D project and planning a project that makes use of technology which has already been developed (such as a builder building a house) is that with an R&D project one is working with completely new technology and the complexities involved with the technology and are unknown. Some parts may turn out to be easier than expected, but other parts will be harder than expected, in some cases they may even be too complicated to be completed before the deadline, even with the use of external resources. Rather than developing a project plan out of stages of known duration, gained from experience of doing the same thing over and over, the duration of the stages will have to be estimated, and contingency plans, such as time limits, external help and alternatives, will need to be put in place if certain parts of the project turn out to be too complicated to be completed within the limit of the project deadline.

2.6 Conclusion

The project planning process gone through for this project proved that when dealing with unknown technologies such as in an R&D project it is very difficult to set a project timeline and the timeline of a project with a set deadline should be reviewed regularly so that the time spent on certain parts of the project can be adjusted to produce a satisfactory result even if some features need to be left out.

3 Backplane communications protocol selection

3.1 Introduction

This section describes the process gone through to select a communications bus and protocol for the backplane. It describes the criteria for the backplane and lists a number of potential candidates for the backplane bus. The list is then narrowed down to a few options that are investigated in further depth together with some appropriate communications protocols, from that a selection of the final bus type and protocol is made.

3.2 Criteria

The following factors were important in deciding a communications protocol for the PCC backplane:

Topology

The backplane must be a serial bus with plug and play functionality. The reason for this requirement is discussed in further detail in section 3.3.

Speed

The backplane must ideally be able to transmit Ethernet frames at 100 mbps, this requirement is however of a lower priority than other requirements such as cost and availability. 100 mbps speeds would be ideal for communicating with PC networks where 100 mbps speeds are common, but many industrial devices, such as label printers, do not require speeds that high.

Frame Size

The backplane must be able to transmit frames of data of up to 1500 bytes in order to transmit Ethernet frames, if the frame size is less than 1500 bytes the protocol must have fragmentation capability.

Compatibility

TCS currently manufactures products with CAN, EIA232, EIA422 and EIA485 interfaces, the ability to interface the hardware of existing products with the backplane would be desirable as it could save some costs in terms of hardware design if such an interface is required in the future.

Cost/Availability

The PCC must be able to be made from readily available components with a low lead time to suit the Lean Manufacturing system TCS has implemented.

The price of components for the main microcontroller and communications components (excluding I/O related components that vary across the modules may not exceed $250.

3.3 Reason for using a serial bus topology

During the planning stages of the project the question came up several times as to why one of the requirements was to use a serial bus rather than a parallel bus.

Less data to be transmitted

The majority of PLCs currently on the market have a parallel backplane data bus. A parallel bus allows larger volumes of data to be transferred at a faster rate. Because the PCC is intended to be used for distributed control there will be less modules attached to each controller therefore reducing the amount of data transferred over the backplane. In distributed control a plant is divided into cells with each cell having its own controller that handles data only relevant to the cell. The control application is distributed across all the controllers in the factory rather the traditional central control where there is one large PLC that contains the whole control system and has to deal with all the data.

Different arrangement of modules

In a distributed control situation a plant is divided into production cells with one PLC per production cell. When converting a plant that was originally set up for centralised control to distributed control, there will be a lot of machinery (cells) that do not have a control cabinet big enough to fit a PLC that uses a rack (parallel backplane) to hold I/O modules. The PCC will be using a serial bus for its backplane which is less susceptible to interference allowing the modules to be spaced further apart and nearer to the hardware they are controlling.

Existing serial replacements for parallel busses

In the personal computer (PC) market there are several serial busses, such as USB, FireWire and eSATA, which have been developed to replace parallel busses. The benefits of replacing parallel busses with serial busses for connecting peripherals to PCs can be seen by the fact that peripherals that require a parallel bus are all but phased out. A few reasons why serial busses have become so popular are listed below:

- Economy: Serial cables have fewer cores so they are cheaper and require less shielding.
- Ease of interconnect: Having fewer cores means plugs can be smaller and more robust. With multiple devices on the same bus, devices on a parallel bus will need to be assigned an address before connecting, this is not necessary with a plug and play serial bus.
- Less bulky cabling: Fewer cores and smaller connectors allow the cables to be made less bulky, making the cable more user-friendly.
- Greater distances: A lot of serial buses use differential signalling over a twisted pair, this type of signalling is very robust and allows data transmissions to be reliable over a far greater distance than data transmitted over a parallel bus.

3.4 List of potential data bus options

This section contains a list of potential standards selected for the electrical layer of the backplane. Initial research was carried out only into the electrical specifications of each bus type because the development of a custom link layer protocol was an option that had to be considered.

HyperTransport

HyperTransport was developed by a group of leading computer hardware manufacturers such as Advanced Micro Devices, Alliance Semiconductor, Apple Computer, Broadcom Corporation, Cisco Systems etc... HyperTransport was developed to allow high speed communications on computer motherboards. HyperTransport allows both inter-processor communication as well as communication between processors and I/O devices. "HyperTransport technology is a powerful board level architecture that delivers high bandwidth, low latency, scalability, PCI compatibility, and extensibility. When processor-native, HyperTransport provides an integrated front-side bus that eliminates custom/proprietary buses and the need for additional glue logic. As an integrated I/O bus, HyperTransport eliminates the need for multiple local buses, ultimately simplifying overall system design and implementation."

A few of the electrical characteristics:

Maximum link transfer rate: 400Megatranfers to 2.8 Gigatransfers per second.
Data link: 2, 4, 8, 16 or 32 bits wide.
Communication can be in parallel or point to point.
Electrical protocol: 1.2 Volt differential.
Packet protocol: 4 - 64 byte data payload with 8 or 12 byte header.
Transfer medium: HyperTransport signals are transmitted over balanced/differential lines making it more immune to noise allowing higher rates of communication over longer distances, the HyperTransport specification allows communication lines to be as long as 600mm.

(HyperTransport Consortium, 2004)

I^2C

I^2C (Inter-Integrated Circuit bus) is a bus widely used in areas such as consumer electronics, telecommunications and industrial applications. I2C is designed to allow multiple ICs (Integrated Circuits) to communicate with each other over a two-wire bus. The I2C bus is a multi-master bus allowing multiple devices to talk to each other individually rather than via a single master.

A few of the electrical characteristics:

Maximum transfer rate: 34 Megabits per second.
Packet protocol: 8 bit data payload.

(NXP Semiconductors, 2007)

Intel Light Peak

Intel Light Peak is a brand new optical bus (parts are expected to ship in 2010) being developed by Intel. Light peak is designed to connect peripherals to a PC in a similar way as is currently done by USB and FireWire cables.

Light Peak uses light generating silicone to transfer data to other light sensitive silicone chips over an optical fibre. Not unlike USB, Light peak needs to be set up in a point to point topology.

A few of the characteristics:

Transfer rates: 10 to 100 Gigabits per second.
This protocol uses optical technology because existing electrical cable technology is approaching practical limits for speed and length.

(Intel, 2009)

CAN

CAN (Controller Area Network) was originally developed in the early 1980's as an automotive serial bus system. The CAN protocol is now internationally standardised in ISO 11898-1 and comprises the data link layer of the 7 layer OSI model. CAN is now widely available on most microcontrollers and there are many different types of dedicated CAN controllers. Apart from the application of CAN in the automotive field, it is also widely used in industry together with protocols such as DeviceNet and CANOpen. TCS produces various DeviceNet compatible products, so CAN is already widely used by TCS.

A few of the electrical characteristics:

Transfer rate: Up to 1 Megabit per second.
Packet protocol: 11 bit identifier and 8 byte payload.
CAN is a highly robust multi-master protocol. Signal monitoring and CRC are used to ensure all packets get transferred successfully.

(CAN in Automation e.V.)

FlexRay

"The FlexRay Communications System is a robust, scalable, deterministic, and fault-tolerant digital serial bus system designed for use in automotive applications. It was developed by the FlexRay Consortium, a cooperation of leading companies in the automotive industry, from the year 2000 to the year 2009." (Altran Technologies)

FlexRay is essentially an upgraded version of the CAN bus which was developed because new automotive electronic technologies such as ABS, ESP, Airbags etc… are requiring faster communication rates and more bandwidth than CAN which is the current standard for inter-ECU(Electronic Controller Unit) communications in the automotive industry.

A few of the electrical characteristics:

Transfer rate: Up to 10 Megabits per second.
Packet protocol: 40 bit header, 0-254 Bytes payload and 24 bit trailer.

(Millinger & Nossal, 2005)

EIA 485

EIA485 (formerly RS485) is a serial data protocol that can be set up in a bus topology. EIA485 is widely used in industry as the electrical layer for well-known interface standards such as Profibus

and Modbus. EIA485 can be implemented on most microcontrollers making it relatively easy and cheap to implement. The maximum transfer speed of EIA485 is 35Mb/s (not all controllers can run at this speed) and it can transfer data in streams, which means that a protocol can be used where large packets of data such as IP packets will not have to be fragmented.

A few of the electrical characteristics:

Transfer rate: Up to 35 Megabits per second
Maximum number of Driver/Receiver Pairs: 32 (TCS has worked with networks with more than 32 pairs, so it is possible to have more than 32 but it would not be recommended)

(Bies)

1394 FireWire and USB

FireWire and USB are high-speed serial busses that can transfer large packets of data. A lot of the peripherals planned to be made for the PCC are currently available as USB or FireWire connected peripherals for PCs. FireWire and USB are both point to point protocols which can only be arranged in a daisy chain topology if each device has a repeater inside it.

A few of the electrical characteristics:

FireWire
Transfer rate: 100 Megabits per second
Packet protocol: 20 Byte header 256 Byte payload 4 Byte trailer

(Teener)

USB 2.0
Transfer rate: 480 Megabits per second (raw data rate)**.**

(Everything USB, 2006)

SPI

SPI (Serial Peripheral Interface) is similar to I^2C and also widely used in areas such as consumer electronics, telecommunications and industrial applications. The main difference between SPI and I^2C is that SPI is better suited for applications that are naturally thought of as data streams. SPI is also much faster than I^2C transferring data in the 10s of megahertz. SPI can only be set up in a master slave arrangement where the master sends a signal to the slave to activate it.

A few of the electrical characteristics:

Transfer rate: 10s of Megabits per second.
It is a stream based protocol so there is no set packet protocol.
Slaves are addressed by a slave select line, connecting multiple slaves to a single master would require having one slave select line running from the master for each slave.

(Kalinsky & Kalinsky, 2002)

UNI/O

"As embedded systems become smaller, there exists a growing need to minimize I/O signal consumption for communication between devices. Microchip has addressed this need by developing the UNI/O bus, a low-cost, easy-to-implement solution requiring only a single I/O signal for communication."

A few of the electrical characteristics:

Transfer rate: 10 to 100 Kilobits per second.
Transfer medium: Single wire
Packet protocol: Single byte start header, 8-12 bit device address, acknowledge bits and optional: command, word address and data bytes.

(Microchip Technology Inc, 2008)

Ethernet

Modern Ethernet is usually laid out in a star topology. It is also possible to connect Ethernet cables together using a hub. The benefits of using Ethernet are the high speeds of 100s of Megabits (ranging up to Gigabits). Standard Ethernet/IP protocols on Ethernet do not support real-time data communication and don't automatically provide safety and security features. There are however various industrial Ethernet protocols (such as Ethernet PowerLink) that are known to support real time data communication and provide safety and security features.

A few of the electrical characteristics:

Transfer rate: 100 Megabits to 1 Gigabit.
Transfer medium: Differential twisted pair (making it more robust and immune to noise).

(Ethernet Powerlink Standardisation Group, 2009)

Other bus types

During this research several other bus types were found but not noted in this report for various reasons. Many of these bus types were similar to the ones mentioned in this section and either superseded by them or indistinguishable at the level at which they were being looked at. Others were obviously less suitable for this project due to factors such as cost, complexity, lack of available parts etc... After narrowing down the selection to the final few bus types, further in depth research was carried out to ensure they are the more suitable of their type.

3.5 First round of eliminations

The sections below describe why some bus types were not considered appropriate for the PCC backplane. Reasons for eliminating backplane protocols included slow transfer rate, susceptibility to interference and too complicated or expensive to fit within the project budget set by TCS.

Slow data transfer rate or susceptible to interference

UNI/O:

This was eliminated because it transfers data over a single wire, this makes it very susceptible to interference and not suitable for use in an industrial environment. It also only has a maximum speed of 100 Kilobits per second which is too slow for the PCC requirements.

I2C:

I2C is very similar to SPI; SPI however is better suited to transferring data as streams so it will allow for better and faster transmission of data. It was therefore deemed better to focus further research efforts on SPI and eliminate I2C.

Too expensive or complex

HyperTransport:

This is specifically designed for communication between processors on PC motherboards. To convert it so that it can be used to transmit data over a cable would require parts that don't exist or are very rare. There are also very few microcontrollers compatible with HyperTransport, and HyperTransport parts are very expensive.

Intel Light Peak:

Light Peak compatible products are currently still very rare because it is a very new technology. It is also a point to point protocol which falls outside of the criteria for the backplane bus.

1394 FireWire and USB:

These products are quite common on the market but they were originally designed for point to point protocols only. Suggestions were made to use the hardware but modify the protocol, which would take advantage of the high speeds and reliability. There however wasn't enough information available to make this an easy task and would likely result in the design of a bus protocol a project in itself. Using 1394 FireWire or USB therefore does not fit the project objective.

Ethernet:

Ethernet has proven to have fast data rates, high bandwidth and robustness but data speeds and throughput are not consistent. As with 1394 FireWire and USB, the hardware could be used with a custom protocol that suits the requirements, but this would also be a project in itself and is therefore not fit the project objective.

3.6 Possible communications protocols

Further investigations were carried out into the remaining bus types: CAN, FlexRay, EIA485 and SPI. Before deciding upon a final bus type, some investigation needs to be carried out into communications protocols suitable for use on the electrical hardware.

The decision to investigate a few communications protocols in detail before selecting a bus type was made because there might be a situation where a bus type may seem better in one instance but the overheads required to transmit data across it may make it actually less desirable than a solution on a different bus type.

Communications protocols investigated were mainly selected based on availability of information at TCS and from TCS staff because TCS has experience using a broad range of communications protocols in a broad range of industries, it is unlikely that there are any other protocols available that are significantly better than what TCS is currently using.

CAN bus based protocols

The most common communications protocols suitable for a CAN bus are CANOpen and DeviceNet.

DeviceNet

DeviceNet was developed by Allen-Bradley based on the Controller Area Network (CAN) which was originally developed by Bosch (GmbH). DeviceNet was designed to interconnect lower level devices (sensors and actuators) to higher level devices (controllers). DeviceNet can support up to 64 nodes which can be added or removed individually under power (plug and play) and it supports communication rates of up to 500 kbaud.

The CAN bus layer is designed to avoid collisions, this means that DeviceNet can be set up as a Master-Slave protocol and a Multi-Master protocol without the need to implement systems such as token passing to prevent collisions.

The following diagram shows the frame format of a CAN/DeviceNet frame:

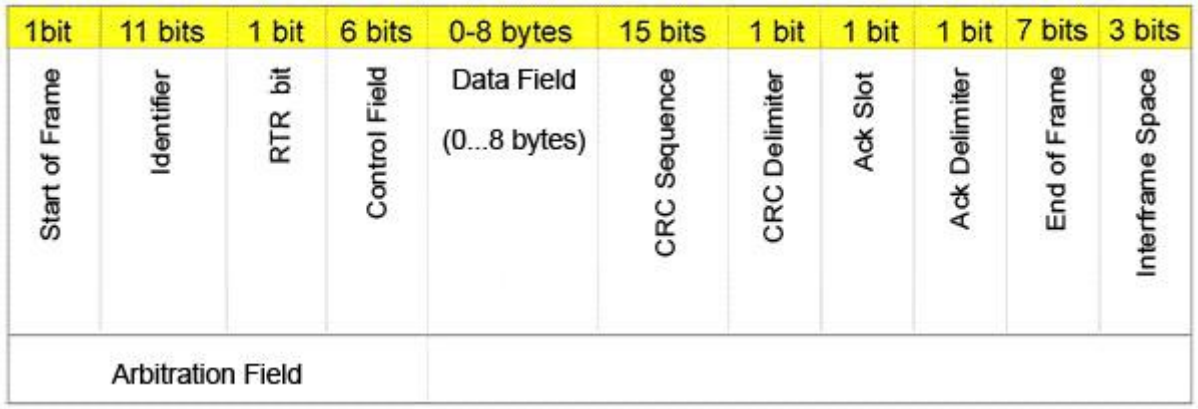


Figure 3: DeviceNet frame (SMAR Industrial Automation, 2001)

It can be seen that for up to 8 bytes of data the overhead of a DeviceNet frame is only 36 bits (4.5 bytes), this overhead can be seen as even less when counting the function codes in the 11 bit identifier.

If more than 8 bytes of data need to be transmitted over a DeviceNet network, the data will need to be fragmented; this will further increase overhead, and may reduce the number of bytes per data frame if they need to be used for data such as fragment count.

(Reynders, Mackay, & Wright, 2005)

An additional benefit of using DeviceNet is that TCS has a lot of experience using DeviceNet due to the fact that it has already developed various DeviceNet products. These DeviceNet ready products may even be able to be integrated into the PCC system.

CANOpen

In CANOpen, a frame of data is referred to as an object. The following is a list of the various types of objects and what they are used for:

- Process data object (PDO): These objects are used for real-time communications such as sensor inputs and actuator outputs.
- Service data object (SDO): These objects are used for non-real-time communications such as setting parameters in devices and carrying out diagnostics.
- Emergency object (EMCY): These objects are used to notify the control application of errors and alarm events.
- Synchronisation object (SYNC): These objects are used to achieve synchronised and coordinated operations in the system.

Even though CAN networks can support multi-master communications, CANOpen is a master-slave protocol with only one master application controlling the network.

The following diagram shows how a CANOpen network is controlled by the master to ensure a reliable network update time:

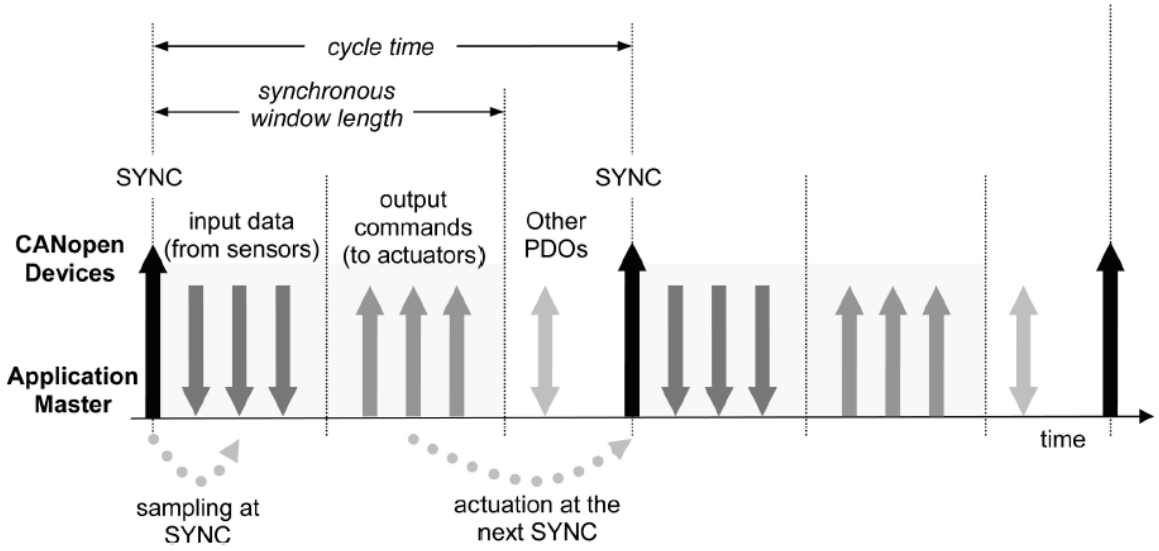


Figure 4: CANOpen synchronous operation (Cena & Valenzano, 2005)

To ensure synchronisation is carried out correctly, an appropriate cycle time needs to be set; in CANOpen this is a set value in the master controller. CANOpen therefore isn't as adaptable to plug and play as DeviceNet.

(Cena & Valenzano, 2005)

FlexRay based protocols

FlexRay is currently only being used in the automotive industry, this industry is very competitive and does not release any information on their communications protocols. FlexRay is very similar to CAN so the communications protocols that would be used on a FlexRay bus will be very similar to those used on a CAN bus (mentioned above), only with less overhead due to the larger data allocation in a FlexRay frame.

EIA485/SPI based protocols

EIA-485 is very commonly used in industrial communications applications. There are various communications protocols used in EIA-485 networks, these protocols vary from manufacturer to manufacturer and a few have been developed into industrial standards. Modbus and Profibus are two of the most common communications protocols used with EIA-485 networks. It could also be possible to develop a custom communications protocol for use on the EIA-485 data bus.

Modbus

The Modbus protocol was originally developed by American PLC manufacturers, but is now a worldwide standard. A lot of European PLCs are compatible with Modbus, but it is still most common in American PLCs.

The diagram below shows how the Modbus application layer can be applied to various physical layers:

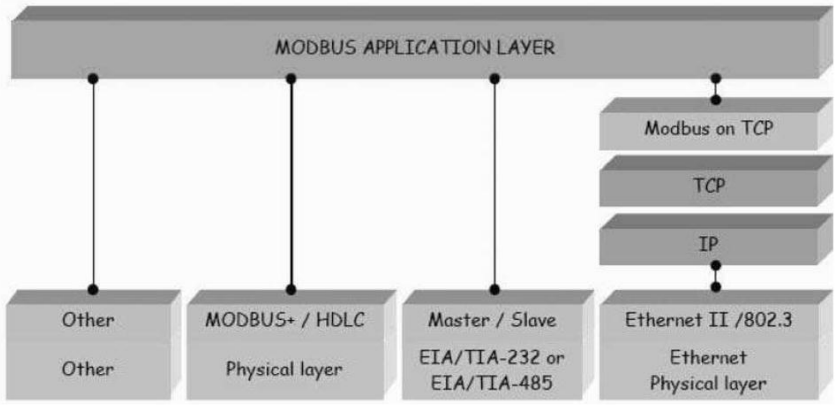


Figure 5: Modbus applications (Reynders, Mackay, & Wright, 2005)

When applied to an EIA-485 Network, Modbus operates in a Master-Slave format where the initiator (client) will send a request for data, which is responded to by the server with the data; this can be seen in the following diagram:

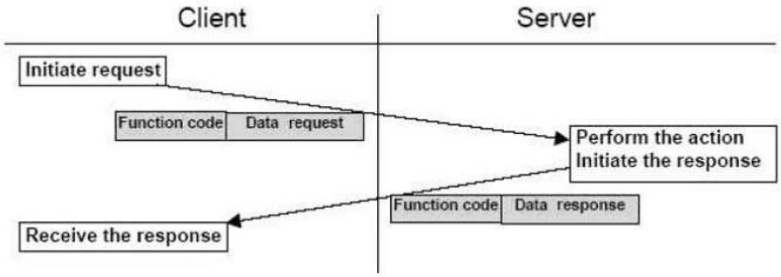


Figure 6: Modbus transaction (Reynders, Mackay, & Wright, 2005)

The following diagram shows the structure of the Modbus protocol:

Address Field	Function Field	Data Field	Error Check Field
1 byte	1 byte	Variable	2 bytes

Figure 7: Format of Modbus message frame (Reynders, Mackay, & Wright, 2005)

The first byte represents the address field, although practical limitations allow only a limited number of devices, each slave can have an address between 1 and 247. In a request message the address field is used to identify the device to which this request is being sent and in a response message this field is used to identify the device that has sent the response.

The second byte represents the function to which the message is related, this byte remains the same for a request and response message. A few examples of Modbus function codes are:

- 01: Read coil status
- 05: Force single coil
- 07: Read exception status
- Etc…

The data field is variable length. In a request frame the data field contains whatever information the PLC needs to complete the requested function. In a response frame the field contains any data requested by the host.

The last two bytes comprise the error check field. Every message is checked for errors by performing a 16bit cyclic redundancy check (CRC) on the message frame; this assures that devices do not react to corrupted messages.

(Reynders, Mackay, & Wright, 2005)

The total overhead of a Modbus message is therefore 4 bytes, this does not include the overhead caused by the fact that each message has to be preceded by a request or followed by a response.

Profibus

The Profibus protocol was originally developed by European PLC manufacturers, but is now a worldwide standard. A lot of American PLCs are compatible with Profibus, but it is still most common in European PLCs.

There are various versions of Profibus:

- Profibus DP (distributed peripheral): this protocol has only one master and multiple slaves.
- Profibus FMS (fieldbus message specification): this protocol has multiple masters that communicate peer to peer.
- Profibus DP/FMS COMBI: some Profibus devices support having multiple masters and multiple slaves on the same network.
- Profibus PA: this protocol is intrinsically safe.

A Profibus master is a controller of some form (this can range from PLC to smart sensor). Profibus master devices have the right to transfer messages without any remote request. A Profibus slave is a peripheral device such as a transmitter or a sensor, these devices may only acknowledge received messages.

When set up as a multiple master system (Profibus FMS) Profibus operates a token passing system, where the master can only transmit data when it has the token, this prevents collisions on the bus. When set up as a master-slave system (Profibus DP) slave devices are polled by the master at regular intervals. These two methods can be combined into a hybrid system (COMBI) where the master holding the token polls the slaves then passes the token on etc… the following diagram shows this setup:

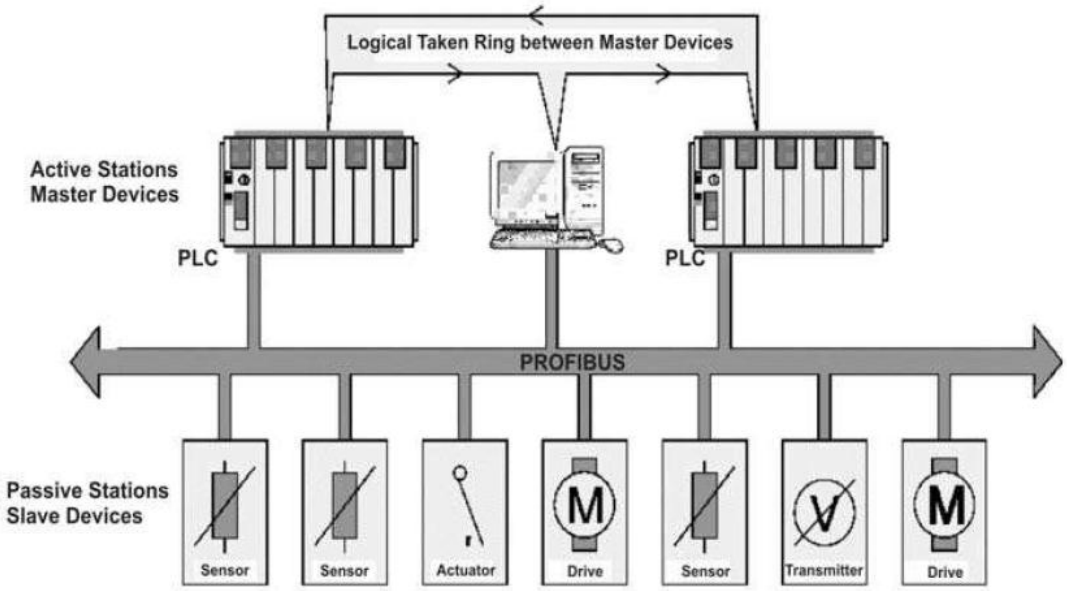


Figure 8: Profibus COMBI setup (Reynders, Mackay, & Wright, 2005)

The following diagrams show the frame formats used by Profibus:

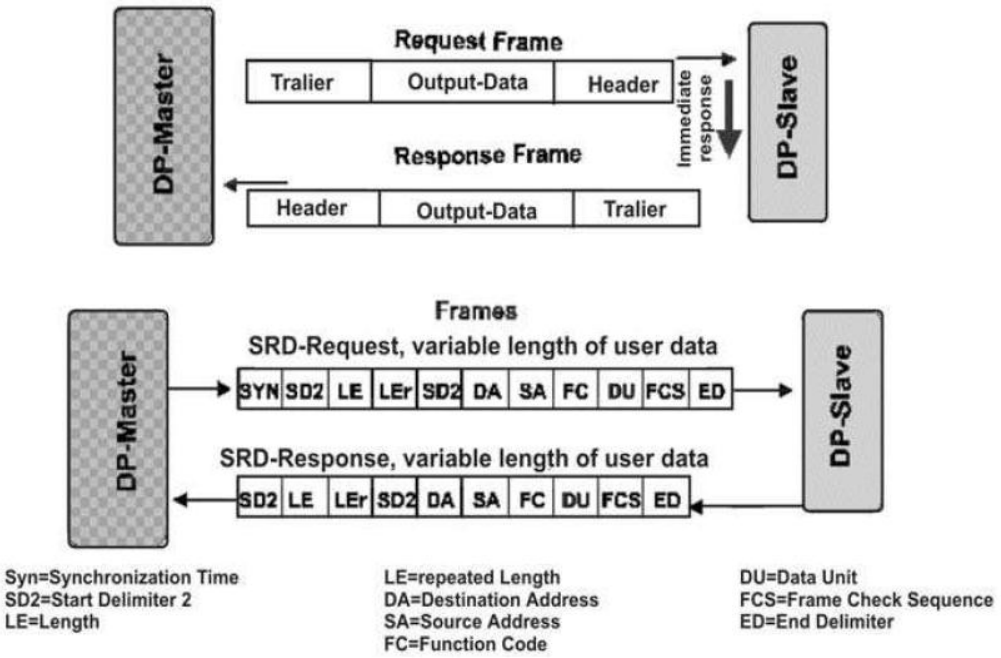


Figure 9: User data exchange for Profibus DP (Reynders, Mackay, & Wright, 2005)

It can be seen that Profibus has a similar frame structure to Modbus (see the top part of the above figure).

(Reynders, Mackay, & Wright, 2005)

SPI based protocols

During further investigation into EIA485 it was decided that SPI did not have any attributes that made it significantly better than EIA485 to warrant any further investigation.

Comparison between protocols

Due to lack of information on FlexRay and the elimination of SPI it was decided to limit further investigations to CAN and EIA485.

The following comparison was set up with information gained by research and some help from TCS staff. It compares various features relevant to the requirements set by TCS that the backplane would have if it used a CAN bus with a custom protocol based on CANOpen and DeviceNet or an EIA485 bus with a custom protocol based on Modbus and Profibus.

	CAN	**EIA485**
Recommended protocol	Custom based on: CANOpen/DeviceNet	Custom based on: Modbus/Profibus
Data Rate	1Mbps	2Mbps (depending on whether the controller can handle it)
FIFO Buffers	yes	no
Number of devices	127	32
Communication Model	Master-Slave, Client-Server or Producer-Consumer	Master-Slave only
Data	CANOpen has a 4 bit part for data length, with 8 bytes per frame => total length is 128 bytes	N*8 bytes up to 252 bytes
Overhead (Electrical)	44 bits per frame of 108 bits (always 40%)	60% for small amounts of data such as digital I/O but it can get up to as low as 20% for large amounts of data such as Ethernet
Overhead (Processor)	No need to implement software FIFO	Need to implement software FIFO
Actual data rate after overhead	600Kbps	400Kbps for small frames such as digital I/O to 800Kbps for large frames such as Ethernet. Maybe less depending on processor load due to processor overhead
Error checking	Link layer built in with CSMA/NDBA, ACK etc…	Need to set up link layer in the CPU

The values for overhead are rough estimations based on the frame structure of CAN and Modbus frames, this does not take into consideration that Modbus operates on a polled/token ring passing system because a custom version of the protocol might not use any of those systems.

Removing doubt that CAN isn't appropriate

During research there were some doubts that CAN was not an appropriate protocol for transferring Ethernet frames because of the size of Ethernet frames and the fact that most Ethernet networks run at speeds of 10 mbps to 1 gbps which is much faster than the speed at which a CAN bus is able to transmit data. This section discusses why this limitation was acceptable:

It was decided that the main controller of the PCC was to have an on-board Ethernet port capable of speeds of up to 100 mbps. The main port would be enough to connect the controller to the main factory network. Any additional ports were to be specified for use as a local low speed subnet or to attach devices such as network printers which only require transmission of low data volumes at long intervals.

A research paper was found called “Porting the Internet Protocol to the Controller Area Network” by Ditze et. al. The main object of their research carried out by Ditze et. al. was to see if they could stream video to the internet from devices on a CAN network. The following figure shows the setup they used:

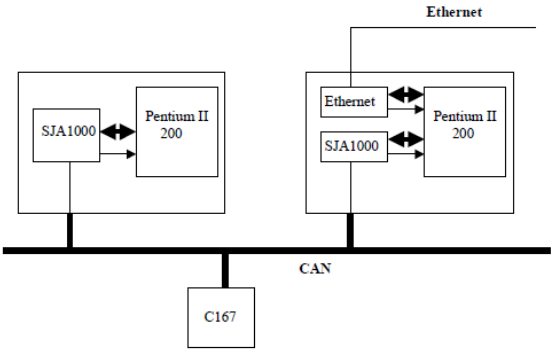


Figure 10: IP over CAN Hardware Architecture (Ditze, Bernhardi, Kamper, & Altenbend, 2003)

Because Ethernet packets can be up to 1500 bytes long, the packets needed to be fragmented into 7 byte fragments with the remaining data byte of the CAN frame used for sequencing numbers.

The test setup used consisted of two PCs with SJA1000 CAN controllers and a Microcontroller with a CAN controller (C167). One of the PCs was connected to the Internet via an Ethernet network.

The most significant test carried out was a video streaming test where video was streamed from the PC without Internet connection to the Internet via the PC with the Ethernet connection all whilst the C167 was also transmitting data. By carrying out this test Ditze et. al. proved that low quality streams of around 380 kbps could be handled over a CAN network.

(Ditze, Bernhardi, Kamper, & Altenbend, 2003)

This research gave a good indication of the speeds that could be expected from porting Ethernet over a CAN backplane, these speeds were deemed more than acceptable in terms of the requirements for speed of the Ethernet modules by TCS staff.

3.7 Recommendations

Below are a few recommendations for any further research that could be carried out into backplane options for the PCC:

- Look into the electrical layer/specifications of higher speed protocols such as USB and FireWire to see if there is potential to run a protocol allowing a bus topology on it.
- Carry out further investigations into FlexRay. At the time of this research it was still quite rare, but in the future components might become more common. Because FlexRay was only removed as an option due to lack of availability it might become a viable option when components are more readily available.

As with FlexRay, an optical bus may also become viable if components become more widely available. It is highly unlikely that an optical bus can be laid out in a bus topology, but the extremely high speeds and immunity to electrical noise could supersede the need for it to be in a bus topology. When not using a bus topology, it is important to take care that the nodes in the network still aren't reliant on each other in a way that if one module fails, the rest of the bus doesn't go down (There is no single point of failure). This is a serious factor that will need to be considered when selecting a bus option such as an optical bus.

3.8 Discussion

Overall the research went quite well. Due to loose selection criteria and time restrictions, there were an almost overwhelming amount of options. This limited the depth of investigation into the lesser known bus types. Because some backplane options couldn't be investigated too deeply, there may have been a bias towards more familiar backplane options. If more time was spent looking into certain bus types or protocols, it may have come out that there were specific features making them ideal for use as a backplane that weren't immediately apparent.

It could be seen as a good thing that a familiar bus type was picked because it allows more time for software and protocol development or less overall development time allowing TCS to determine whether the PCC is a viable product before investing large amounts into researching better backplane options.

3.9 Conclusion

The backplane bus type and protocol selected for further development was the CAN bus with a custom protocol based on features of DeviceNet and CANOpen. The comparison list (see previous section) demonstrates that CAN with a custom protocol based on DeviceNet and CANOpen fits TCS requirements better than EIA485 with a custom protocol based on Modbus.

Looking at the criteria, here is why the CAN option was selected:

Topology

CAN supports hot plug and play and DeviceNet has functionality to allow devices to be installed and uninstalled dynamically at run time. Implementing this on EIA485 would be more complicated as the existing protocols all involve the master polling the slaves which means the master must be previously aware of the existence of slaves. This means that when developing a custom protocol the CAN protocols require less complicated changes than the EIA485 based protocols.

Speed

CAN is not as fast as EIA485 at the electrical layer but CAN requires a lot less complicated processing of data in the microcontroller and all microcontrollers support 1Mbps communication on a CAN bus whereas most microcontrollers only support a maximum communication rate of 115200 bps communications on an EIA485 bus, so implementing an EIA485 based backplane that is significantly faster than 1Mbps would require special communications controllers.

Bandwidth/Frame Size

Both DeviceNet and CANOpen support data fragmentation with little more overhead than an EIA485 protocol such as Modbus would have. Because CAN frames are smaller than EIA485 packets, if the data is corrupted halfway through, a CAN protocol can stop whereas with EIA485 all the data would need to be received before error checking can occur. There is therefore too little difference in terms of bandwidth to make EIA485 better than CAN.

Compatibility

Using the CAN electrical layer with a custom protocol means that there won't be any need to redesign the circuitry to make an existing TCS product compatible with the PCC; it will only require a software change.

Cost/Availability

Most of the microcontrollers already used by TCS have built in CAN controllers. TCS also has several products that use the Phillips SJA1000 can controller. By using components that TCS is already using in other products, the costs will be significantly reduced because the reduced risk of having unused parts on the shelf means that parts can be ordered in larger quantities. Ordering parts in larger quantities means that the cost will be lower and TCS will be able to maintain a supply of parts rather than having to wait for parts to arrive every time an order comes in.

4 Windows CE backplane driver development

4.1 Introduction

This section describes development of the backplane driver. The process included carrying out background research into similar bus drivers, working out some initial ideas, looking into existing protocol device installation procedures, developing a plan for the backplane driver and then executing it and discussing various issues encountered during backplane driver development.

4.2 Background research

To develop a backplane driver for Windows CE it was important to first understand how similar backplanes and data busses worked. The two most common (most successful and effective) busses for adding hardware modules to a processor running Windows CE are the PCI bus and the USB bus. CANOpen and DeviceNet were also investigated because they are the most commonly used CAN protocols. Developing an understanding of existing busses and their strengths and weaknesses will greatly help the development of a new bus type.

General Driver loading concepts

The Windows CE operating system itself runs on a microcontroller (see figure below). The operating system is stored in the Random-Access Memory (RAM) and Read-Only Memory (ROM), calculations and changes are made in the Central Processing Unit (CPU). In short: the CPU receives input data or reads data from the RAM/ROM, processes it according to what is directed by the operating system and writes data to the outputs or the RAM.

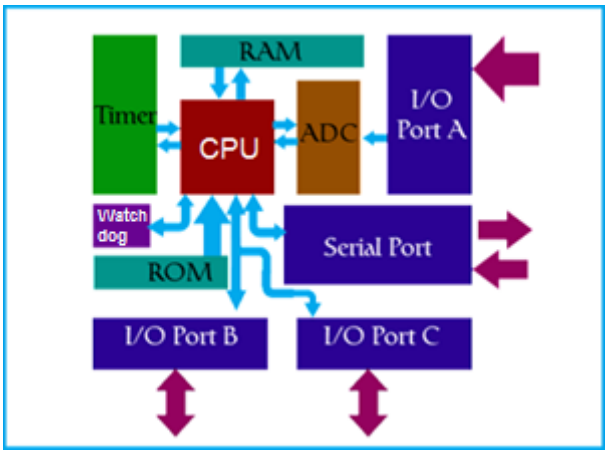


Figure 11: Basic microcontroller architecture (What is a Microcontroller?)

If a peripheral is added to the CPU, such as an I/O port, serial port or anything else, the operating system will need what is called a 'driver'. This is a piece of software which contains information telling the CPU how to process data it receives from it and how to format data it sends to it. When a manufacturer releases a microcontroller that is compatible with Windows CE, they often produce what is called a Board Support Package (BSP), this BSP contains the drivers for all the peripherals which are present in the microcontroller (in the case of the microcontroller above, that would be the I/O ports, the analogue input port and the serial port). The role of the device developer who creates the motherboard, on which the CPU is mounted, is to customise this BSP to suit the peripherals included on the motherboard.

If the motherboard is so designed that it includes a bus, to which peripherals can be dynamically added and removed, special kinds of drivers are required. The most dynamic way of adding and removing peripherals is plug and play. Plug and play allows the user to add a peripheral when required and a driver for it will be loaded automatically and upon removal, the driver will be unloaded automatically. To create a plug and play bus one first needs a driver to control the bus itself, this driver needs a layer which can sense the addition or removal of peripherals and then load or unload drivers for those peripherals accordingly.

The two most common and successful plug and play busses (which are compatible with windows CE 5.0) on the market today are the PCI bus and the USB bus. These busses are discussed in the sections below.

Different driver structures

The software which manages the drivers in terms of assigning them memory space in the RAM and processing time in the CPU is the Device Manager.

The figure below shows the way drivers interface with the windows CE device manager. The stream interface contains Init() and DeInit() functionality (amongst other functions) and these functions can be called by the bus driver to dynamically load and unload the peripherals' drivers. The three structures pictured will all be used in this project:

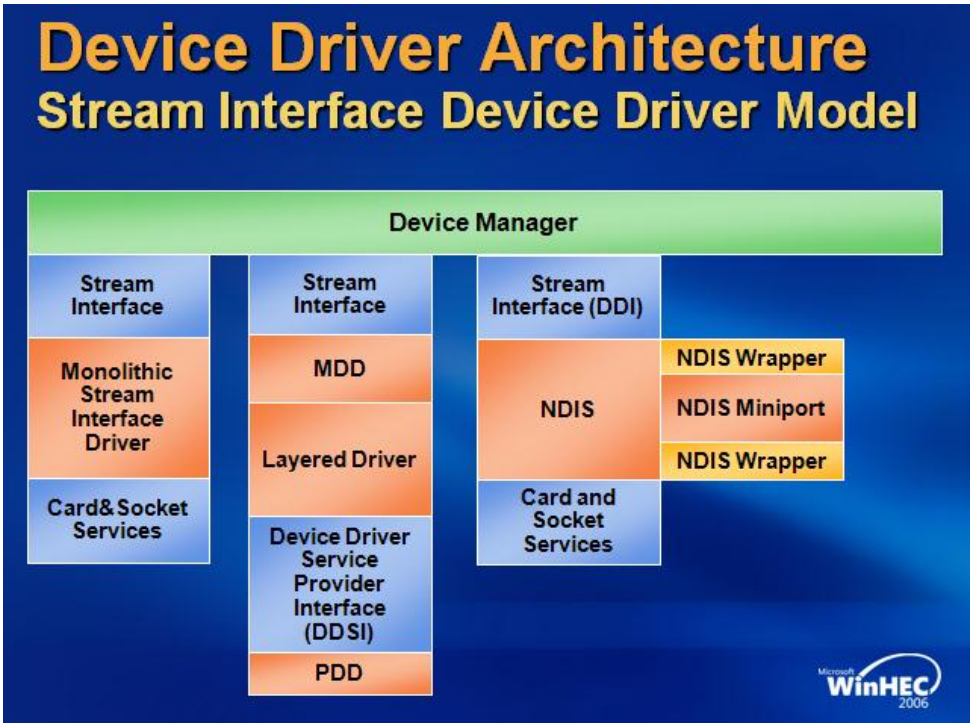


Figure 12: Stream Interface Device Driver Model (Ravalia, 2006)

The digital I/O driver uses the Monolithic Stream Interface Driver structure, the serial port driver uses a Layered Driver structure and the Ethernet port driver uses a Network Driver Interface Specification (NDIS) Miniport driver which is wrapped in the NDIS driver.

More in depth explanations of the various driver structures will be covered in the sections specific to the drivers that use those structures.

The Windows CE PCI bus

The PCI (Peripheral Component Interconnect) bus is a 64 bit parallel bus capable of transmitting data at up to 133 MHz or 1 Gigabit per second (Tyson & Grabianowski). The PCI bus is a parallel bus which means that it communicates to the PCI cards via a number of address lines, a number of data lines and some interrupt lines. The PCI bus driver is loaded at boot time, at which time it runs an enumeration routine. The PCI bus enumeration routine scans the bus by setting the address bits to access a slot and attempting to communicate with the device located in that slot; if a device responds it will be installed from a list of templates in the PCI related registry keys (details of the PCI bus installation routine are discussed later on). Once a PCI device is installed information on its location on the PCI bus is stored in the registry so that any software attempting to access the device knows how to access it. Although the PCI bus is a parallel bus and it only loads devices at boot time it is still worthwhile investigating because the way it enumerates the bus and the way it stores information about the drivers might be able to be applied to the PCC backplane driver.

The Windows CE USB bus

The USB (Universal Serial Bus) is a high speed serial bus capable of speeds of up to 480 Megabits per second. Many devices which were traditionally only able to work on a PCI bus due to speed and bandwidth limitations are now also available with a USB interface, these devices include: HDTV tuners, portable hard drives and even video cards (Everything USB, 2006). The USB bus has four lines, for power, earth and a differential pair of data lines. When a device is plugged in the current on the power lines changes, this is detected by the driver which attempts to establish a connection to the device. The USB driver maintains a number of data structures, called “pipes”, which contain information on how to address the peripheral devices. Each USB device has a number of “pipes” for different purposes, e.g. for a bulk data transfer such as a video stream, there is a “setup pipe” which sets up the communication, then the data is pushed out through the “bulk transfer pipe”, this sort of communication makes developing drivers for peripherals somewhat easier but would be a very complicated system to develop for a different bus. The way drivers are loaded and parts of the pipe system may be able to be applied to the PCC backplane driver.

4.3 Initial design ideas

Based on the research carried out into general driver loading and the PCI and USB bus standards, the following structure was proposed:

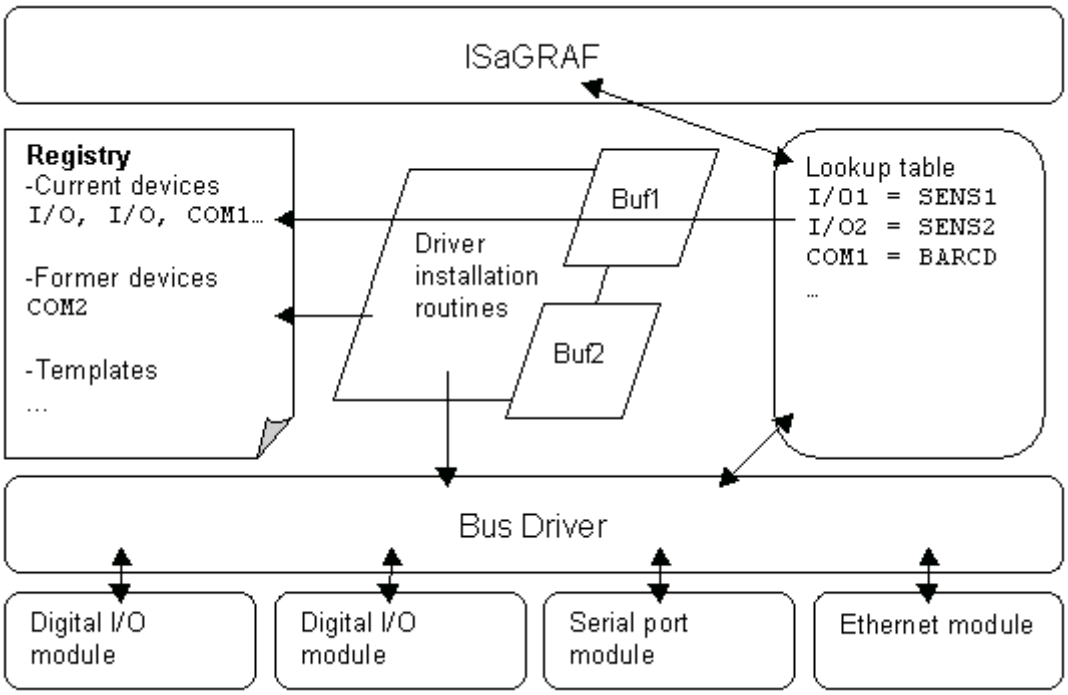


Figure 13: Proposed global overview of how the PCC modules interface with ISaGRAF

The figure above shows the proposed program structure describing how the peripherals interact with the ISaGRAF software through the bus driver which will be running on the PCC main controller. The proposed structure was designed with the knowledge of how the PCI bus and the USB bus works and some knowledge of how drivers in Windows CE work. The structure was designed as a starting point for software development and also a reference to maintain an overview of how the various objects of the software fit together. The various 'blocks' in the diagram represent parts of the software that can be individually developed and the arrows represent the interfaces between them. This allows for a more object oriented approach towards the design, making the task of designing the backplane driver easier to manage because it now consists of many small projects rather than one really large and complicated project.

Details of how the various parts of the diagram work together will be covered in the next few sections.

4.4 Examples of similar device installation procedures

Eliminating USB as a driver loading option

The USB driver loading process was deemed too complex and cumbersome to implement for the initial version of the backplane software, let alone the fact that CAN has no built in current sensing capabilities. Because investigating only the PCI bus device loading procedure was deemed not enough to gain a fair understanding, it was decided to include the driver loading process for the industrial DeviceNet and CANOpen protocols in the research.

PCI bus device loading procedure

The following steps describe how a PCI bus loads drivers for attached devices:

1. The PC or device is turned on
2. The bus driver is loaded and activated
3. The bus driver scans the PCI slots by polling them one by one. This procedure is easy on a parallel bus because it is simply a case of setting address bits. There are also only a limited number of addresses and scanning only has to happen once.
4. The bus driver requests device type information. There are two main types of devices possible on a PCI bus, these are PCI cards and PCI to PCI bridges.
5. The bus driver loads device specific data into a buffer using the HalGetBusData() function.
6. The bus driver checks the Instance registry key (see figure 14) to see if the device has been installed previously. If it has skip steps 7 and 8.

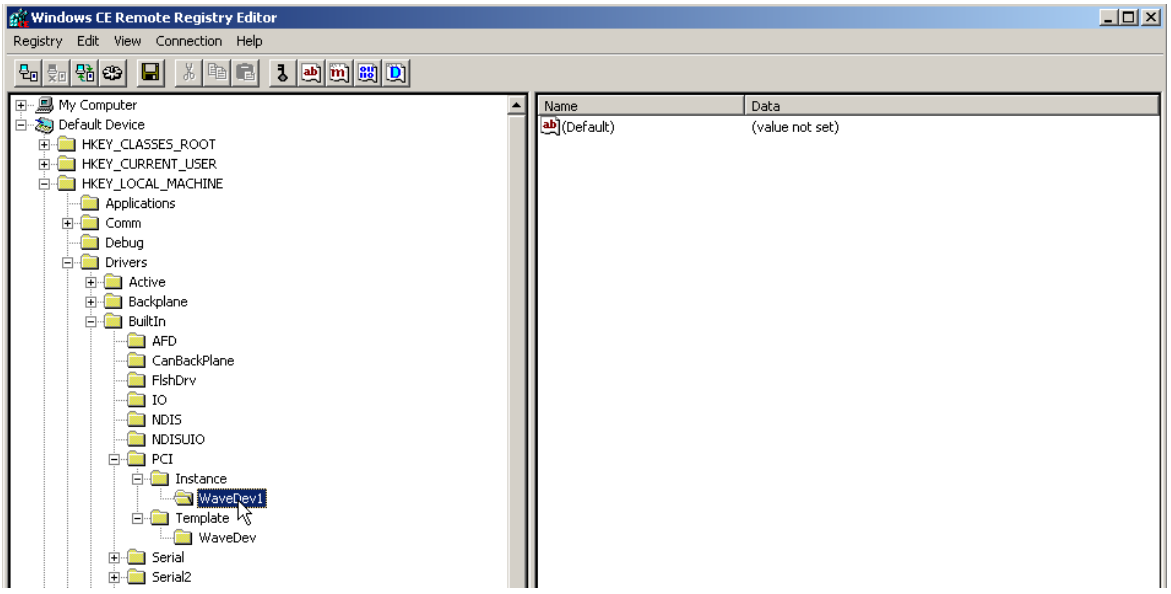


Figure 14: PCI Instance registry key

7. If no information is found in the Instance key the bus driver checks the Template registry key (see figure 15) for a matching template, if nothing is found here the device will come up as ‘unrecognised’ and step 8 will be skipped.

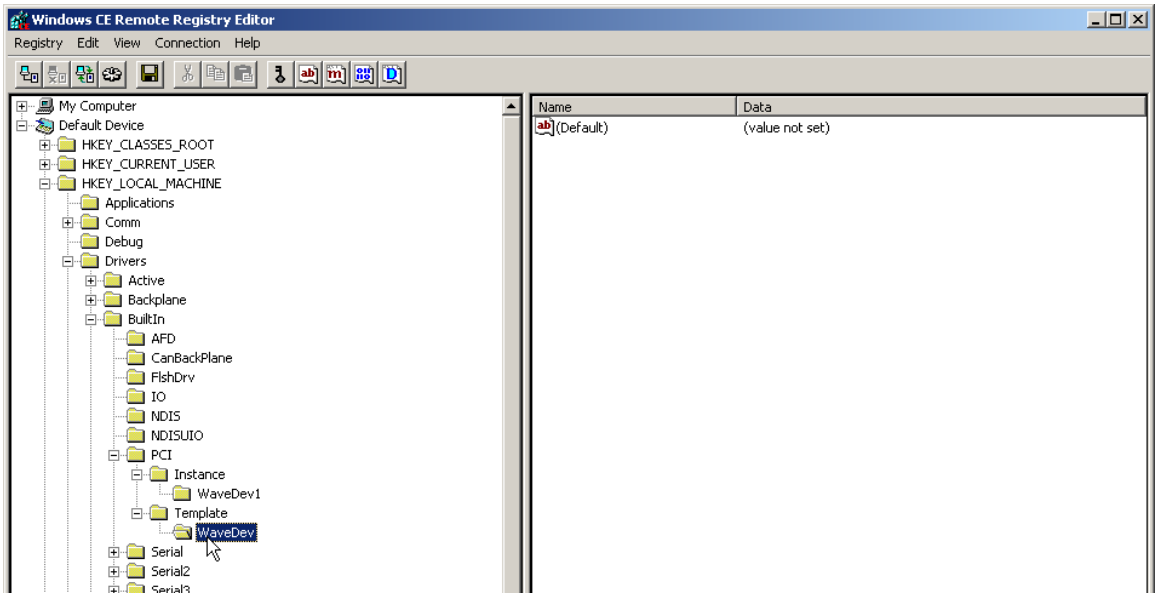


Figure 15: PCI Template registry key

8. The best matching template will be copied to the Instance key.
9. After completing the PCI bus enumeration the device manager will install all the devices listed in the Instance key.

CANOpen boot up procedure

The diagram below shows the various states of a CANOpen module, the boot up procedure below shows how these states are reached.

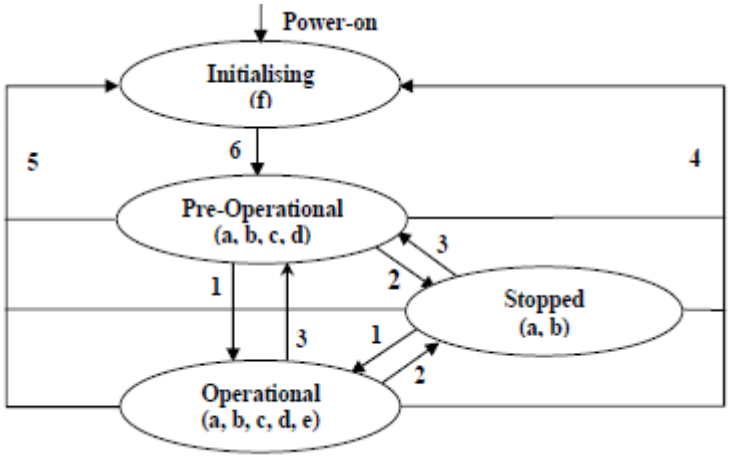


Figure 16: CANOpen state diagram

The steps of the CANOpen boot up procedure are shown below together with relevant CAN frames demonstrating how the data is transmitted across the bus.

1. Master -> All

10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0
–	–	–	1	0	0	0	0	0	0	0
–	–	–	0	0	0	0	0	0	0	0

Figure 17: Broadcast CAN frame to move slaves into pre-operational state

The master broadcasts a CAN frame to move all nodes into Pre-Operational state

2. Master -> All (unassigned)

10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	0	0	0	0	0	1
–	–	–	0	0	0	0	1	1	0	1
–	–	–	Code telling devices to send out stop command							

Frame sent to slave 1 (all un-initialised devices will be called node 1)

Figure 18: Broadcast CAN frame telling slaves to send out stop command

The master then transmits a message with a one byte code telling all devices to send out the stop command, this should result in one device (the first transmitter remaining in Pre-Operational state) sending out the stop message stopping all the other devices, preventing them from transmitting their messages.

3. All (unassigned) -> All (unassigned)

10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0
–	–	–	0	0	0	0	0	0	1	0
–	–	–	0	0	0	0	0	0	0	0

Figure 19: CAN frame sent out by slave to make other slaves move into stopped state

Stop all nodes (All nodes but the master and the first transmitter will go into the Stopped state)

4. Master -> Node 1

10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	0	0	0	0	0	1
–	–	–	0	0	0	0	1	1	0	1
–	–	–	Code assigning new macID							

Figure 20: CAN frame sent out by master to assign slave MAC ID

Since all the other devices have stopped the Master can assign a new MAC ID to the one remaining device

5. Once the new MAC ID has been assigned, the device will be moved into the Operational state.
6. All stopped devices will be moved back into the pre operational state.
7. Repeat steps 2-6 until the last device has been assigned a MAC ID.

DeviceNet boot up and address assignment procedure

The following steps show the DeviceNet boot up procedure and the frames that are transmitted across the backplane during boot up.

1. All devices start off in the communication faulted state, new devices plugged in the network also do.
2. Request vendor ID and serial number from all devices in the communication faulted state (Who Communication Faulted Request message). This is done at regular intervals set up by the systems integrator to detect newly plugged in devices and uninstall removed devices.

10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	0	1	1	0	1
–	–	–	0	0	0	0	0	0	0	1
–	–	–	0	1	0	0	1	0	1	1
–	–	–	Time delay byte offset 0-6							

Figure 21: Message group 4 communication faulted request CAN frame.

Reserved bit=0 match bit=0 value bits=000001 (this means that all devices with MAC ID = 000001 will receive the message, if bits are set to 111111 all devices in the faulted state will receive the message)

Request bit = 0 Request who (ask device to send details)

Time delay (offset in 50ms intervals)

3. Devices will return Who Response Message

10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	0	1	1	0	0
-	-	-	0	Physical port number (port 0)						
-	-	-	Vendor ID							
-	-	-								
-	-	-	Serial number							

Figure 22: Who response CAN frame

4. Assign an ID number to each received Vendor ID and serial number
5. Send out packets to devices with MAC ID 0000001 that will change the MAC ID if the vendor ID and serial number match. (Change MAC ID Communication Faulted Request Message)

10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	0	1	1	0	1
-	-	-	0	0	New MAC ID					
-	-	-	0	1	0	0	1	1	0	1
-	-	-	Vendor ID							
-	-	-	Serial Number							

Figure 23: Change MAC ID communications faulted CAN frame

6. Once a device has a MAC ID install the driver.

4.5 Device installation routine plans

Device installation/uninstallation steps

This section describes the procedures gone through to load and unload PCC backplane drivers. These procedures were developed during the initial design stages so only loosely represent the final device loading/unloading procedures.

Background:

The slave modules have been set up to respond to device management messages, these messages include request for device data and set device ID. When a request for device data message is sent out the slave module will reply with a CAN frame containing data relevant for the device management system. The responses to a request for device data are built up into a temporary list structure which can be processed by device management software.

A non-volatile data structure has been set up in the registry of the PCC to hold data relevant to the attached devices. This data structure is managed by the device management routines that process a volatile list of device data. The registry is updated every time a device is removed or added. The registry structure contains keys for active devices, former devices and it contains templates with information required for setting up new devices.

Another process which installs and removes drivers scans the registry so that the drivers installed correspond with the devices in the active devices section of the registry structure.

Driver loading procedure:

The procedure proposed for loading the device drivers in Windows CE is below:

1. Power up the PCC
2. Load the bus driver
3. Power up the bus
4. PCC will send out a request for device data
5. As device data comes in a *linked list* of device data (serial numbers etc…) will be built up (temporary data) out of the responses from the slave modules.
6. Scan the 'active devices' registry key and remove any entries from the linked list (generated in step 5) that correspond to entries in the 'active devices' key (move temporary data into permanent list).
7. Scan the 'former devices' registry key, if any devices with matching serial numbers are found in the linked list install the device, move the data to the active devices key and remove the data for that device from the linked list
8. Sort the 'former devices' registry into groups by device type (treat devices of the same type with the same mnemonic as one device)
9. Scan through the remaining entries in the linked list and find a matching device for each group of former devices with only one entry, install the device, give it their mnemonic and remove the entry from the linked list.
10. In groups with multiple entries assign the mnemonic of the most recently removed device to the new device, install it and activate a warning that the mnemonic has been automatically assigned from a potentially unrelated device.
11. If there are still unassigned devices left (remaining entries in the linked list), and no corresponding devices in the 'former devices' registry, give those devices a temporary mnemonic name according to their device type and install them from the 'template' registry.

Steps 4 to 11 of this procedure can be repeated regularly to update devices

Driver unloading procedure:

When a device is removed from the PCC, it needs to be detected to prevent data being unnecessarily sent to or read from it. Because there is no way for a module to announce its removal (especially if the device fails), the controller can only detect that it is removed by attempting to poll it. To prevent removing devices unnecessarily, the ability to implement some tolerance has been built in allowing the systems integrator to set a number of polls before the device is removed. This could be beneficial for high traffic applications where a poll response might not be fast enough due to large volumes of traffic on the bus.

The section below mentions a value represented by x, x is a number which can be set by the systems integrator to set the amount of requests for device installation a device has to miss before its driver is uninstalled.

1. Every time the bus driver sends a request for device information to the attached devices, decrement a register in all the 'attached devices' registry entries.
2. Report an error. If necessary, temporarily halt communication to this device.
3. If a poll returns the serial number of this 'active device' the register will be reset to x, if a device is installed this register will be set to x.
4. If after x polls the register has been decremented to 0 the device will be removed and its details moved to the 'former devices' registry.

If x is too high and communications are not halted the system may attempt to send data to a device which is no longer there. If x is too low the system may have to do a lot of work re-installing the driver every time.

Additional features that may be included here are:

- Making x (number of polls and the time between polls) able to be adjusted by the system integrator through the ISaGRAF workbench depending on how much traffic there is on the backplane and how time critical the application is.
- An error messaging/logging system that can be used to indicate whether the system is set up properly, this may even be able to be used to indicate whether a module is overloaded and/or in the early stages of failure.

Cleaning up the registry:

The part of the registry that could become large and cumbersome is the ‘former devices’ registry key. Having too many entries in this key will make the installation process very slow as all the entries will need to be scanned, it is therefore important to regularly clean this key. A method such as deleting history of devices which are older than a certain amount of time could be quite effective.

Further considerations:

There is a case where if two modules are replaced at the same time it could become unclear which replacement module is in which physical location, this is because the module addresses do not depend on the position, rather the serial number of the device. This situation also makes it harder knowing which device is assigned which name when installing the system, especially if the system integrator is not near the PCC. A solution to these problems could be to have a small LCD display, or a flashing LED on the slave modules to indicate the identity assigned to each device by the device installation routine.

4.6 Registry Setup

The previous section described how a non-volatile structure was set up in the registry to contain information specific to the devices attached to the backplane. This section describes this registry structure.

The figure below shows the way keys were set up in the PCC registry for the backplane driver:

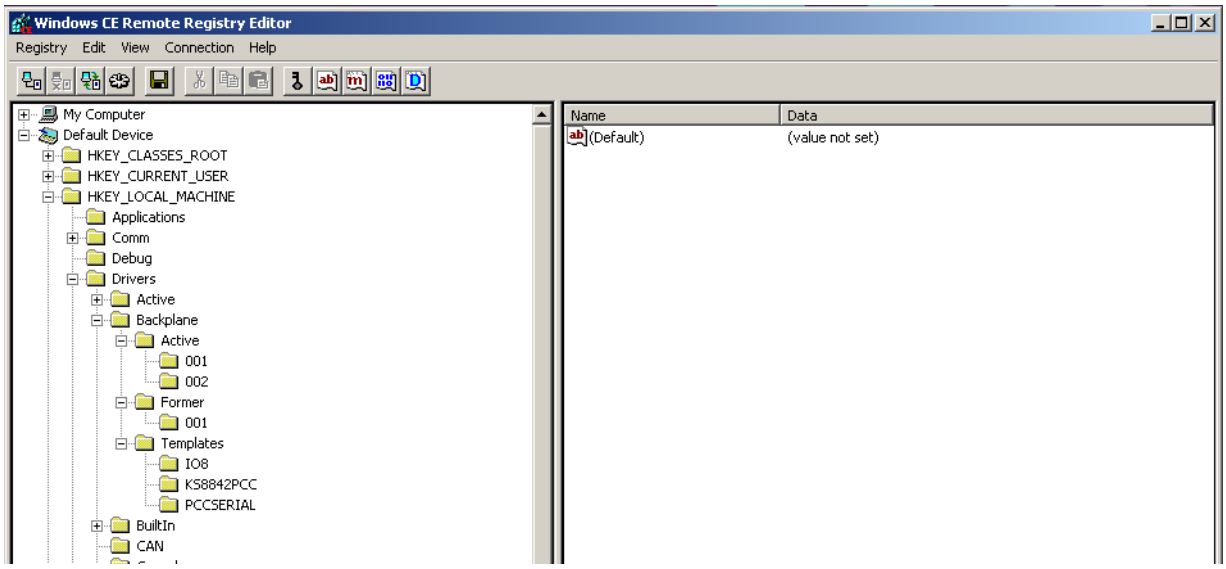


Figure 24: PCC backplane registry setup

HKEY_LOCAL_MACHINE/Drivers/Backplane/Active: This key contains information about all the currently active devices. The Active key is enumerated (scanned) by the device installation routine which installs the drivers.

HKEY_LOCAL_MACHINE/Drivers/Backplane/Former: This key contains information about devices that have previously been attached to the PCC backplane, this key is useful for storing configuration settings specific to a card so that when it is replaced or removed and then reconnected at a later stage (or in a different position in the bus), the user won't have to go through and configure the device again.

HKEY_LOCAL_MACHINE/Drivers/Backplane/Templates: This key contains information about the drivers available for PCC modules. When a new device is plugged in the backplane driver scans this key looking for a matching device ID, if this device ID is found it uses the information in the Template key to install the driver for the device.

The registry is modified by the device management routines. The device installation routine regularly scans the 'Active' key and installs/uninstalls drivers so that the drivers installed correspond to the devices in the 'Active' key.

4.7 Message prioritisation

The communications protocol that was decided to be used was the CAN protocol with a standard length header. Because the CAN protocol allows up to 127 devices on the bus, only 7 bits of the 11 bit header needed to be used for addressing devices, this left four bits over to be used for other information.

The CAN protocol has a built in message prioritisation system that allows messages with the most recessive bits (‘0’s) to pass through with a higher priority, this meant that if the four spare bits were also the first four to be transmitted through the CAN bus they directly influenced the priority of the message. A messaging scheme was set up using the four spare bits to identify and prioritise messages. After discussions with TCS staff members it was decided to prioritise messages by frame size and give explicit and device management messages the lowest priority. Messages with a smaller frame size were given priority because their data refresh rate might be faster than it takes to transfer a single larger frame, with a higher priority these devices will not be affected by the transfer of a larger frame. Having a low priority might slightly slow down the transfer of larger frames but the devices transferring these frames will be less significantly affected than a device with a small frame size would if it had a lower priority. The following chart shows some of the message types and the way they were prioritised:

Message\bit	10	9	8	7	6	5	4	3	2	1	0	
Error	0	0	0	0	MAC ID							Highest Priority
Digital In	0	0	0	1	MAC ID							
Digital Out	0	0	1	0	MAC ID							
Analogue In	0	0	1	1	MAC ID							
Analogue Out	0	1	0	0	MAC ID							
Serial In	0	1	0	1	MAC ID							
Serial Out	0	1	1	0	MAC ID							
Ethernet In	0	1	1	1	MAC ID							
Ethernet Out	1	0	0	1	MAC ID							
Explicit In	1	1	0	0	MAC ID							
Explicit Out	1	1	0	1	MAC ID							
Device Management	1	1	1	1	MAC ID							Lowest Priority

Figure 25: Message prioritisation protocol

4.8 Backplane driver software development

The backplane driver software was developed in a number of stages, they are described below in the order they were established. Some stages required further development to make the software compatible with particular features but this is the main order in which they were developed. The following diagram shows another view of how the backplane driver interfaces with the windows CE operating system. The various parts of the diagram and how they work are described in the following sections.

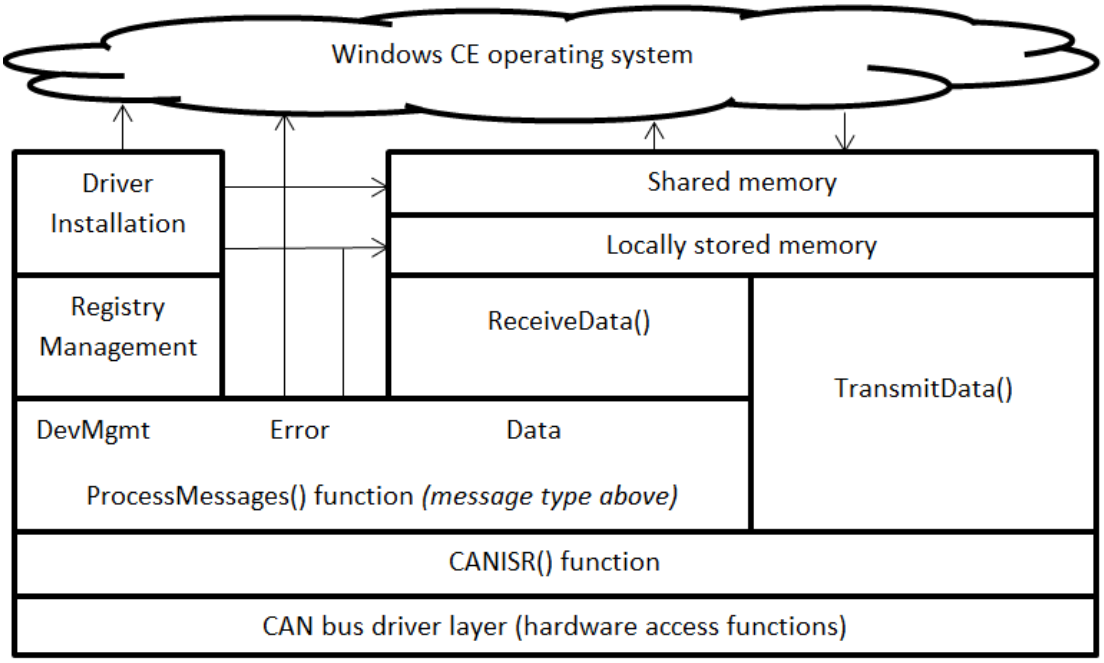


Figure 26: Backplane driver structure

Registry management

The first software that was set up was the registry management software. This part of the backplane driver was developed in the initial stages of the project when the test hardware hadn't been assembled yet because it could be developed and tested without using PCC hardware.

The registry management routine is in charge of managing the registry keys for the backplane. The idea for the device management is that at regular intervals a "heartbeat" message is sent out, this message triggers all the connected modules to respond with a short message containing information about the device. The role of the registry management code is to scan the list of responses and organise the registry appropriately, essentially carrying out steps 4-7 of the device installation pseudo code mentioned in section 4.5. The registry management software was programmed in C++ and the following three functions to be used by the rest of the backplane driver were developed:

- AddDevice(unsigned int DeviceID, unsigned long SerialNumber): This function adds device information to a linked list which is stored as a local variable which can only be otherwise accessed by the registry management functions.
- InstallDevices(): This function runs the main registry management routine which uses the linked list of device information to manage the registry keys.
- ScanReg(): This function scans the HKEY_LOCAL_MACHINE/Drivers/Backplane/Active key and calls device installation/uninstallation functions to install/uninstall the drivers listed in each sub key in the Active key.

Backplane driver template

When the hardware was set up, a driver previously used by TCS as a CAN bus driver for DeviceNet communications was used as a template. This driver communicated with the Windows layer using the stream interface*, and mapped a CAN interrupt routine (CANISR()) to the physical CAN interrupt line. The CANISR() function was already set up to read data from the CAN controller and transmit data out of the can controller. The CAN_Init() function was modified to call a function to initialise a few threads for the backplane (such as a device management thread) and the CANISR() function was modified to call a function which processes the received messages.

* Details of the stream interface are discussed in the Digital I/O software development section.

Message processing and filtering

The previous section explained that the template driver had a CAN interrupt service routine that was triggered upon receiving a CAN message and had access to the received data. The interrupt service routine was modified to call a function which processed the received data, this function is shown below:

```
int ProcessMessages(void)
{
	int recvMsgID = ((ReceiveBuffer.Identifier)>>7);
	int recvMacID = (ReceiveBuffer.Identifier&0x7F);

	//Error message group
	if((recvMsgID == MSGID_ERROR) && (BP_MaxMacID>=recvMacID)) {
		switch(ReceiveBuffer.Data[0]) {
			case 2:
				DEBUGMSG(TRUE, (TEXT("Request for retransmit
received from MacID: %i\r\n"),recvMacID));
				RetransmitData(recvMacID);
			break;
			default:
			break;
		}
	}
	//Device management message group
	if(recvMsgID==MSGID_DISCOVER) {
		DeviceManagement(recvMacID, recvMsgID);
	}
	//If MacID is set process the ordinary messages
	if((recvMacID != 0) && (BP_MaxMacID >= recvMacID)) {
		if((recvMsgID==BP_DeviceList[recvMacID].rx_MsgID)) {
			ReceiveData(recvMacID);
		}
	}
	return 1;
}
```

It can be seen that three message groups have been defined:

- Error messages: These are high priority messages. When an error message comes in the appropriate function is called to recover from this error.
- Device management messages: When a device management message is received, another function is called to determine whether it is a response to a heartbeat message and adds the device information to the linked list so that it can be processed by the registry management software. More information on this is discussed in later sections.
- Data messages: When a data message is received, a function is called which puts it in the appropriate receive buffer for the device from which the message originates. The fragmentation etc… involved with this is discussed in later sections.
- Any other messages received are dropped at this point.

Message transmission

The original template simply had a global variable in the form of a can message structure which was transmitted every time there was data and the CANISR() was triggered. The following code shows how a routine could populate a CAN structure and trigger the CAN interrupt.

```
TxBuf.Identifier = 0x781;
TxBuf.DLC = 0x02;
TxBuf.Data[0] = 0x0F;
TxBuf.Data[1] = 0xFF;
TxBuf.Data[2] = 0;
TxBuf.Data[3] = 0;
TxBuf.Data[4] = 0;
TxBuf.Data[5] = 0;
TxBuf.Data[6] = 0;
TxBuf.Data[7] = 0;

TriggerCanInt();
```

The above example shows a basic level of transmission functionality. There would need to be some form of message handling protocol to populate the structure without conflicting/interfering with the transmission of the data.

Because the backplane driver was to have several threads and several different drivers accessing the CAN, it was deemed safer to use a secure FIFO (First-In First-Out buffer) accessed by the threadsafe functions: AddToOutputFifo(CAN_IMAGE tx) and CAN_IMAGE GetFromOutputFifo(). Using a FIFO buffer would allow the higher level processes to operate as normal whilst holding back the data until the CAN controller was ready to transmit it.

Device management thread

Having developed the low level functionality for receiving and transmitting data over the CAN bus, it was possible to set up a thread to link the CAN bus communication with the registry management functions. The code executed by the thread for managing the devices can be seen below:

```
#define HEARTBEAT_INTERVAL 15000
#define DEVICE_RESPONSE_TIME 1000
DWORD WINAPI DevMgmtThread(LPVOID pParam)
{
	DEBUGMSG(TRUE, (TEXT("DevMgmtThreadStarted\r\n")));
	//Loops continuously because we never stop polling
	while(1) {
		//Broadcast a 'who request' message to all slave modules
		DEBUGMSG(TRUE, (TEXT("DevMgmtThread\r\n")));
		TxBuffer.Identifier = 0x781;
		TxBuffer.DLC = 0x00;
		TxBuffer.Data[0] = 0;
		TxBuffer.Data[1] = 0;
		TxBuffer.Data[2] = 0;
		TxBuffer.Data[3] = 0;
		TxBuffer.Data[4] = 0;
		TxBuffer.Data[5] = 0;
		TxBuffer.Data[6] = 0;
		TxBuffer.Data[7] = 0;

		AddToOutputFifo(TxBuffer);
		TriggerCanInt();

		//Wait for all the modules to finish responding
		Sleep(DEVICE_RESPONSE_TIME);

		//Process the temporary device list that was just
		// built up (update registry).
		InstallDevices();//scans active key and sets macIDs
		ScanReg();

		//Sleep until the next installation cycle
		Sleep(HEARTBEAT_INTERVAL);
	}
	return 0;
}
```

The device management thread loops continuously and runs through the following steps:

1. Broadcast a “who request” message, the slave modules are set up so that upon receiving this message they respond with information about themselves.
2. After sleeping for a set device response time (`DEVICE_RESPONSE_TIME`), it is assumed that all devices will have had enough time to respond and the thread resumes.
3. The thread calls the registry management routine to set up the registry.
4. When the registry is up to date, the thread executes the function exposed by the registry management software which installs any newly detected devices and uninstalls any removed devices.
5. The thread then sleeps again for a longer period of time. The heartbeat interval time can be reduced or extended depending on what the requirements are. A shorter heartbeat interval will make the backplane react to addition/removal of devices faster, but it will take up a lot of processor time whereas a longer heartbeat will use less processor time but makes the backplane very slow to react to any changes.

During later stages of the project the idea came up that this thread could be made to run continuously/a lot faster but at a lower priority so that it would only run when the processor is free. Running the device management thread at a lower priority would have several advantages but it could also have some serious disadvantages. If the device management thread were a lower priority it could potentially run more often, making the process of recognising devices a lot faster. If the backplane is loaded with traffic, it could also run a lot slower which could be made worse, especially if a device is physically removed and the processor is still transmitting data to it. For a final product it is likely to be more desirable to have faster installation/uninstallation, but in the prototyping/beta-testing stages of the product where reducing variables is important, it is more important that device management happens at regular intervals.

Device driver installation

Having developed a system which keeps track of the devices attached to the backplane, the next stage was to create the functions required to register the devices with the Windows CE operating system (install the drivers) and set up the memory for the volatile device information and data that could not be stored in the registry.

The specifics involved with installing each device type are discussed in their respective sections further on in this report. Setting up shared memory and shared interrupts will be discussed in the next section.

To manage the device driver installation, a program was set up that exported a set of functions and provided access to an array of structures. Details are shown below:

- A function that is called by the registry management software whenever a new device is detected: extern int Bp_InstallDevice(int bpi_MacID, unsigned int bpi_SerNum, unsigned int bpi_DeviceID, unsigned int bpi_bufLength, unsigned int driverCount)
- A function that is called by the registry management software every time a device is removed from the Active key: extern int Bp_UnInstallDevice(int bpu_MacID)
- An array that is declared globally: extern bp_Device * BP_DeviceList, the bp_Device contains pointers to memory such as the transmit buffers and sharedmemory. This list is different to the volatile list from which the registry management software updates the registry.

The bp_Device structure is shown below:

```
//Backplane device structure
typedef struct{
      //general
      unsigned int ok; //is the device there
      unsigned int installed; //device installation bit
      unsigned int bufLength; //max length of data buffer
      unsigned int driverCount;
      HANDLE driver; //handle to the driver
      HANDLE driver1; //handle to the driver
      unsigned int devtype;
      unsigned int serialNumber;    //serial number
      //related to receive
      unsigned int fragError; //has a fragmentation error occured
      unsigned int rxBufLen;
      unsigned int rxBufReady;
      unsigned int fragmentCount;
      HANDLE rxMem;
      unsigned int * rxBuf;
      unsigned int * rx_FileMemory;
      unsigned int tx_MsgID;
      //related to transmit
      unsigned int txBufLen;
      unsigned int txBufReady;
      unsigned int bytesTransmitted;
      HANDLE txMem;
      unsigned int * txBuf;
      unsigned int * tx_FileMemory;
      unsigned int rx_MsgID;
      //event handles
      HANDLE rxEvent;
      HANDLE rxEvent1;
      HANDLE txEvent;
} bp_Device;
```

Not all the information in this structure is used by every driver, the structure is populated based on what information each driver needs. The information kept in this structure is all volatile and will be lost when the PCC is powered down, when the PCC is powered up the list is populated again based on non-volatile information from the registry.

Sharing memory and interrupts

Because the device drivers will be running as separate processes to the backplane driver, there will need to be a method to transmit data frames from the backplane to the drivers and back. Windows CE has functions available for creating memory that can be shared amongst processes; these were used for creating the shared memory. For one process to notify the other process that it has finished populating the shared memory it was decided to use shared interrupts.

The Windows CE functions used for creating shared memory are CreateFileMapping() and MapViewOfFile(). Below is the section of code used in the device installation function of the backplane driver:

```
BP_DeviceList[bpi_MacID].rxMem = CreateFileMapping(INVALID_HANDLE_VALUE,
NULL, PAGE_READWRITE,0,(bpi_bufLength+3)*sizeof( unsigned int ), bpi_name);
if(BP_DeviceList[bpi_MacID].rxMem) {
     BP_DeviceList[bpi_MacID].rx_FileMemory = (unsigned int*)
     MapViewOfFile(BP_DeviceList[bpi_MacID].rxMem, FILE_MAP_ALL_ACCESS,
     0,0,0 );
     if(BP_DeviceList[bpi_MacID].rx_FileMemory ) {
         memset( BP_DeviceList[bpi_MacID].rx_FileMemory, 0x0,
         (bpi_bufLength+3) * sizeof( unsigned int ));
     }
}
```

After creating the shared memory on the backplane driver side, the handle to the shared memory (`BP_DeviceList[bpi_MacID].rx_FileMemory`) was put in the registry of the active device.

When installed, the initialisation routine of the device driver then reads the handle to the shared memory from its registry key, saves it in the variable: `pds->rx_FileMap` and executes the following code:

```
RegQueryValueEx(hKey, TEXT("rxMem"), NULL, NULL, (LPBYTE)&pds->rx_FileMap,
&dwLen);
      if(pds->rx_FileMap) {
            pds->rx_FileMemory = (unsigned int*)MapViewOfFile(pds-
            >rx_FileMap, FILE_MAP_ALL_ACCESS, 0,0,0 );
            if(pds->rx_FileMemory ) {
                  memset( pds->rx_FileMemory, 0x0, (pds->BufLength+3)
                  *sizeof(unsigned int));
            }
      }
```

The Windows CE functionality for sharing interrupts involves giving the interrupt a name upon creation in the device installation routine, and then creating another interrupt in the driver initialisation routine with the same name. The following function creates a named interrupt:

```
txEvent = CreateEvent(NULL, FALSE, FALSE, L"txEvent");
```

Data reception and transmission

It was decided to set up the data buffers and shared memory to the expected size of a frame of data. The size of the shared memory for each device would therefore depend on device type, for example: a digital I/O module would only require one byte and an Ethernet module would require 1500 bytes. Having a shared buffer the size of a complete data frame allows the drivers to process an entire frame at once. For frames larger than 8 bytes, a fragmentation/assembly layer needed to be implemented to fragment the data buffers into CAN-frame sized fragments and to assemble the incoming CAN frames into a buffer of the appropriate frame size for each device. The two functions that handle fragmentation and assembly are:

- TransmitData(): This function is called whenever the CAN controller is free to transmit data and there is data ready to be transmitted. This function will continue to be called until the entire frame of data is transmitted, at which time it will release the transmit buffer for the driver to write another frame of data to it.
- ReceiveData(): This function is called whenever a data frame is received by the CAN controller. The function checks the receive buffer associated with the device from which the message originated and determines whether it is a valid message then writes the data to the buffer. When the ReceiveData() function processes the final fragment of data it triggers an event called “DataEvent”, that event causes the following thread to run:

```
DWORD WINAPI RXThread(LPVOID pParam)
{
	int i;
	while(1) {
		//Wait for the backplane to finish processing a message
		WaitForSingleObject(DataEvent, INFINITE);
		for(i=1; i<=BP_MaxMacID; i++) {
			copyToSharedMem(i)
		}
		Sleep(0);
	}
	return 0;
}
```

The copyToSharedMem() function scans through all the receive buffers to see if any of them contain complete frames and then copies the receive buffer to the shared memory and releases the receive buffer so that the ReceiveData() function can start populating it with the next frame of data.

The fragmentation protocol of the ReceiveData() and TransmitData() functions is discussed in the “Fragmentation” section.

Look up table

Each time a new instance of a device is installed Windows will give it an Index number, for example, the fifth instance of a serial port to be installed will get index number 5 (COM5). To make programming in the ISaGRAF environment more intuitive for the end user, there will be functionality to give the peripheral modules mnemonics, for example: if COM5 is controlling a barcode scanner, it could be named BARCODE1. Due to time limitations this part of the backplane driver was put aside because it is a “nice to have” feature which is not essential for the operation of the backplane.

4.9 Error logging

Because the PCC is a headless platform (system where screen, keyboard and mouse are missing) and there is not necessarily an operator continuously monitoring the system, errors can occur and go unnoticed for long periods of time. Since it is not industrially sound to have a system that stops and requires an operator to clear any error that may have occurred before continuing, the PCC was designed to recover itself (from minor errors) and simply log the error to a text file. Error logs can be regularly accessed to check what sort of minor errors (un-noticeable errors) have been occurring to prevent major errors (errors which cause a halt in operation) and they can be useful in diagnosing causes of major errors.

The error logging code was written to log errors to a text file. It can be called from anywhere and will log an error message with time stamp. The code block on the following page shows this.

```
int LogError (char* message, int length)
{
  HANDLE hLogFile;
  DWORD dwBytesWritten, dwPos;
  int i, n;
  char buff[4096];
  SYSTEMTIME time;

  // Open the existing file.
  hLogFile = CreateFile (TEXT("\\errorLog.txt"),// Open errorLog.txt
                      GENERIC_WRITE,        // Open for writing
                      0,                    // Do not share
                      NULL,                 // No security
                      OPEN_ALWAYS,          // Existing file only
                      FILE_ATTRIBUTE_NORMAL,// Normal file
                      NULL);                // No template file

  //check if file opened correctly
  if (hLogFile == INVALID_HANDLE_VALUE)
  {
    return 0;
  }

  //create timestamp
  TimeStamp timestamp = GetTimeStamp();
  buff = timestamp.text;
  n = timestamp.length;
  //space
  buff[n] = " ";
  n++;
  //copy error string to buffer
  for(i = 0; i < length; i++) {
        buff[n] = message[i];
        n++;
  }
  //add newline and carriage return to end of string
  buff[n] = "/n";
  n++;
  buff[n] = "/r";

  //add error message to end of log file
  dwPos = SetFilePointer (hLogFile, 0, NULL, FILE_END);
  WriteFile (hAppend, buff, n,
             &dwBytesWritten, NULL);

  //close handle to log file
  CloseHandle (hLogFile);

  return 1;
}
```

4.10 Fragmentation

Because different modules transmit data frames of different sizes, a layer needed to be established in the software to fragment buffers of different sizes into frames suitable to be transmitted over the CAN bus and assemble can frames into the original frame size.

The figure below shows the structure of the code in the TransmitData() function, this function is called whenever the CAN bus is free and there is data ready to be transmitted. It is so designed that it is non-blocking. Whenever it is called it transmits one frame. After transmitting a frame, the processor can spend time doing other operations and will get back to transmitting the rest of the data when the CAN bus is free again.

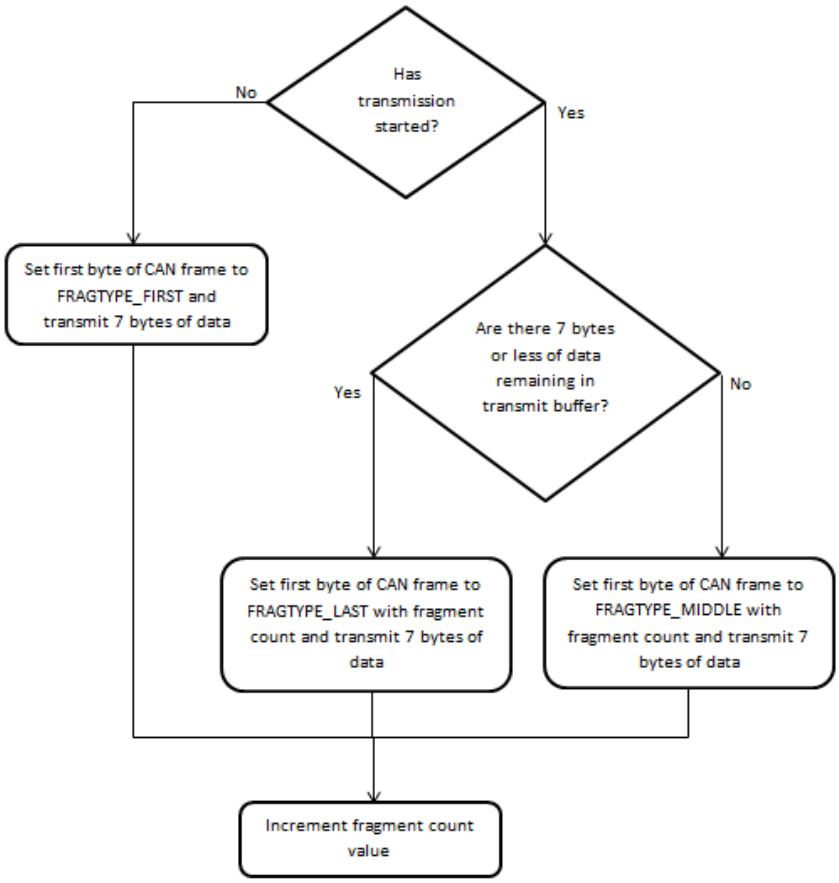


Figure 27: Backplane frame fragmentation flow chart

The first byte of data in the can frame is structured as follows:

7	6	5	4	3	2	1	0
Fragment type		Fragment count					

Figure 28: Structure of first byte of CAN frame

The receive routine, described below, uses the fragment type and fragment count values to check for errors in transmission. Since the maximum value of Fragment count is only 63 (6 bit), functionality was implemented to wrap around the count for frames that need to be split into more than 63 fragments.

The figure below shows the structure of the code in the ReceiveData() function. This function is called every time a message identified as a data message is received. Every time a data buffer is filled up, an event is set which triggers the driver for the applicable device to process the received frame.

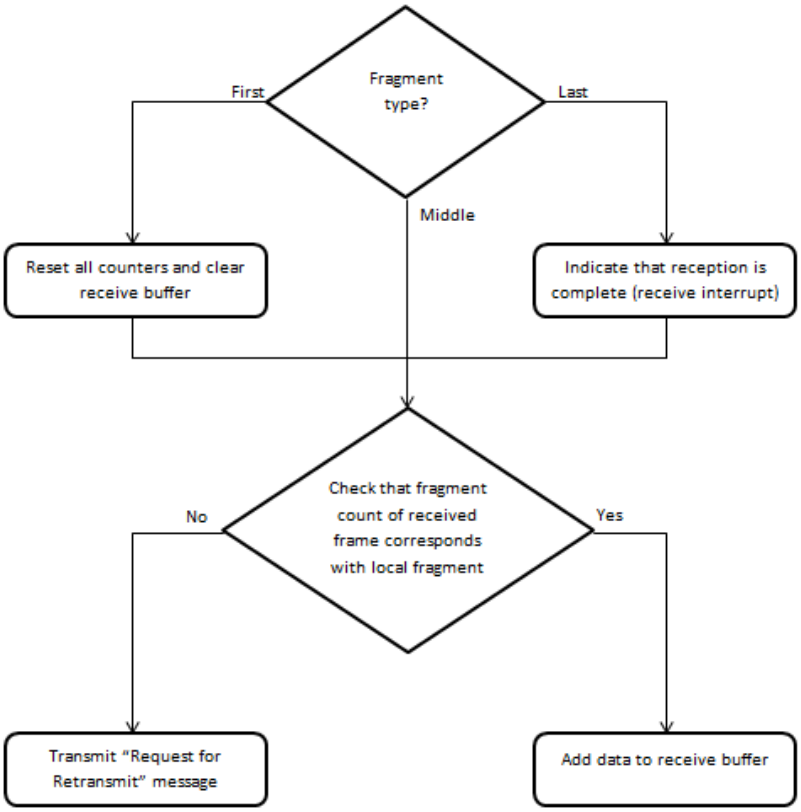


Figure 29: CAN fragment assembly flow chart

4.11 Multithreading

What is it?

The backplane driver software uses several threads such as `DevMgmtThread()`, `RXThread()` and `CANISR()` each driver and the Windows CE operating system also all run at least one or more threads. Because the PCC controller only has one processor core, it can only carry out one instruction at a time so to give the appearance that multiple threads are running simultaneously it divides its processing resource up in time slices, giving each thread a few slices of its processing time to carry out a few instructions.

"The Windows Embedded CE base execution unit is a thread. Each thread has its own context (stack, priority, access rights, and so on) and is executed in the process container. Each process contains at least one thread that is the primary thread. Windows Embedded CE has a theoretical limitation of 32,000 processes that the system can simultaneously load. The number of threads is not theoretically limited, but that number is limited by the number of available descriptors." (Pavlov & Belevsky)

The following figure shows how the processor provides time slices for threads of various priorities:

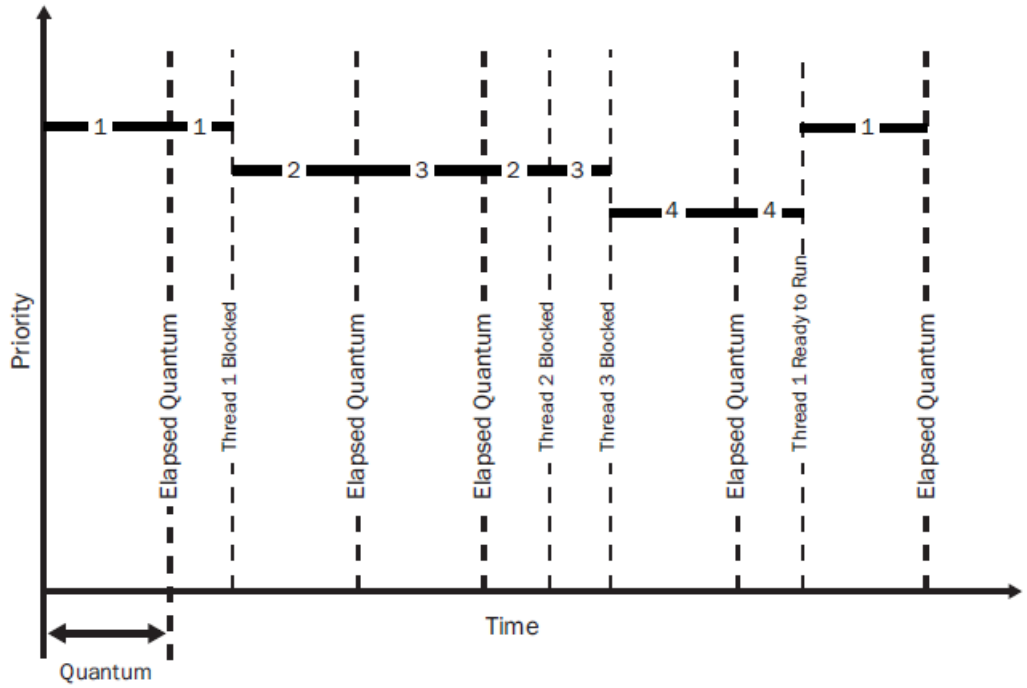


Figure 30: Windows Embedded CE scheduler thread execution (Pavlov & Belevsky)

It can be seen that the thread with the highest priority is allowed to run until it is blocked, at that time the thread with the next lowest priority is released to run. If there are multiple threads with the same priority, a different thread is run when the time quantum/slice ends. After the second lowest priority threads (threads 2 and 3) are blocked, only then can the third lowest priority thread run (thread 4), this thread will run until a higher thread is released.

Precautions to take

Because the processor is a limited resource it must be used as efficiently as possible. Efficient software is therefore defined by how few threads it uses and how limited the time of execution of those threads is. The aim for driver development is therefore to use as few threads as possible and to limit their execution times to the shortest periods possible.

For a thread that is running continuously the period of execution can be limited by making it sleep for periods of time or by making it wait for an interrupt before it executes its next cycle. Examples of this in the backplane driver are shown below:

- The Sleep() function: The device management thread makes use of the sleep function by calling it to wait for response messages to arrive and calling it after each cycle to sleep for a certain amount of time before sending out the next heartbeat message.
- The WaitForSingleObjecct() function: The receive thread uses this function to wait for an event triggered by the CANISR() thread before it copies the complete receive frame into the shared memory.
- System Interrupts: The CANISR() function is run by a system interrupt mapped to a particular pin on the hardware which is triggered when an CAN frame is received.

4.12 Slave microcontroller software

The slave microcontroller software has very similar features to that of the backplane driver only without the functionality for routing the data frames to various drivers because the slave module only receives data frames from one driver.

Operating system

The slave microcontrollers are from the Renesas M16C family. The functionality of the slave modules is limited to only carrying out the I/O task and communicating over the backplane, so they do not require a full operating system such as Windows CE. Instead of having an operating system, the slave microcontrollers run a C program with only two threads:

- The CANISR() thread: This thread fragments data frames and transmits the fragments and assembles incoming fragments into data frames. Fragmentation and assembly are carried out in the same manner as in the backplane driver, see “Fragmentation” section (previous) for more details.
- The main process thread: this thread runs the main application and handles functions such as error checking as well as processing any messages received from the main processor.

Developing the CAN software

The current range of CAN/DeviceNet products produced by TCS use a lower specification microcontroller in the M16C family than the microcontroller used for the PCC slave modules. Previously for all TCS’ CAN-based (DeviceNet) products using the M16C TCS has used the Philips SJA1000 as an external CAN controller. The M16C microcontroller used for the PCC slave modules however has a built in CAN controller. Because this CAN controller is new to TCS products, a software library needed to be developed. To ensure compatibility with existing and future TCS products, the library for the built in CAN controller was made to contain the same functions and functionality as the existing library for the SJA1000. The SJA1000 library contains the following functions:

```
void DisableCAN( void );
void EnableCAN( void );
void CANReset( void );
BOOL CANInit( unsigned char baudRate );
BOOL CheckCanBusOff( void );
BOOL CANIdle(void);
BOOL CANTxing(void);
BOOL CanStatTxIdle(void);
unsigned char GetCanInt(void);
BOOL CanIntRx(unsigned char canIR);
BOOL CanIntTx(unsigned char canIR);
BOOL CanIntError(unsigned char canIR);
unsigned int GetId(void);
unsigned int GetDlc(void);
unsigned char GetDataByte(unsigned char byteNum);
void CanClrRxBuff(void);
void CanTx(void);
void CanAbortTx(void);
void CanResetTxErrorCounter(void);
```

Also the interrupts needed to be set up to call the following function:

```
void far CANISR( void );
```

Making identical libraries for the two types of CAN controller would mean that all the existing software developed by TCS (e.g. DeviceNet stack) could be run on the higher specification microcontroller without needing to change the software interface. Another benefit of using existing library functions is that TCS developers are familiar with the CAN library functions for the SJA1000 so when they are working with the new library they will be working with functions they already understand. This will reduce development time of new software and lower the barrier to move to the higher specification microcontroller.

Setting up the CAN controller

Before creating any of the functions for reception and transmission it needed to be established how to set up the controller and what modes of operation were available for reception and transmission.

The following diagram shows the various modes in which the CAN controller can be set:

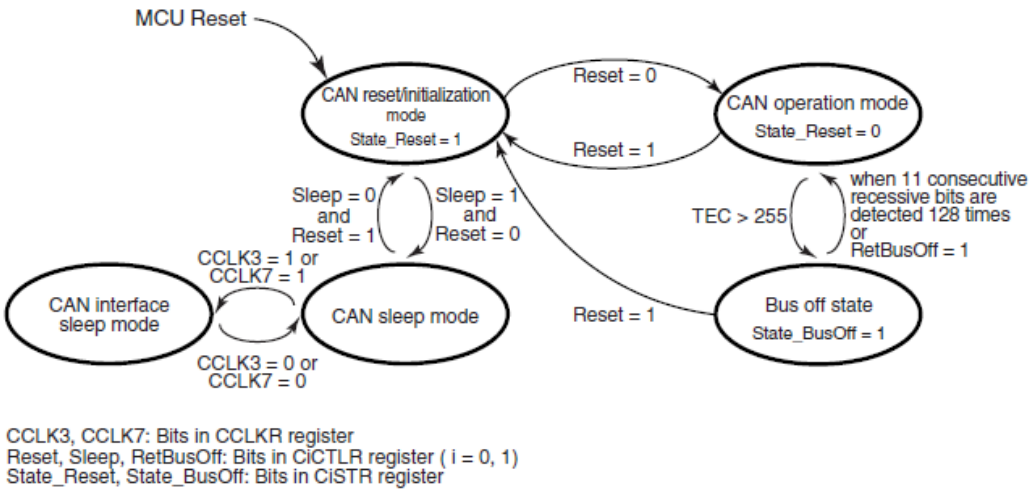


Figure 31: Transition between operational modes (Renesas, 2005)

It can be seen that when the microcontroller is powered up, the CAN controller is in CAN reset/initialisation mode. Setting the reset bit to 0 moves the CAN controller into CAN operation mode. When data reception/transmission needs to be stopped, or when an error has occurred (Bus off state), the reset bit needs to be set to 1 to shift the controller back into initialisation mode. For the initial version of the software it was deemed unnecessary to include any power saving features, the sleep modes were therefore not used, but may be used in future versions.

By observing and controlling the states in the “Transition between operational modes” diagram the following functions could be set up:

```
void DisableCAN( void );
void EnableCAN( void );
void CANReset( void );
BOOL CANInit( unsigned char baudRate );
BOOL CheckCanBusOff( void );
```

The following diagram shows the various modes of operation of the CAN controller when it is in CAN operation mode:

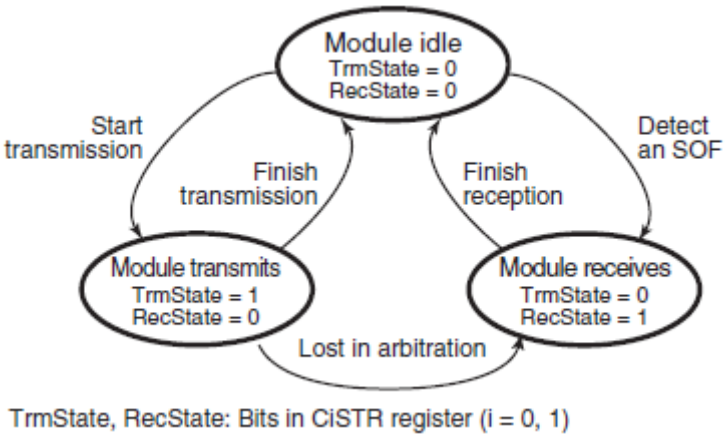


Figure 32: Sub Modes of CAN Operation Mode (Renesas, 2005)

By observing and controlling the states in the “Transition between operational modes” diagram the following library functions could be set up:

```
BOOL CANIdle(void);
BOOL CANTxing(void);
BOOL CanStatTxIdle(void);
void CanTx(void);
void CanAbortTx(void);
void far CANISR(void);
```

Message buffering:

To build up an understanding of the existing message buffering functionality the SJA1000 message buffering features were first investigated. This would make it easier to adapt it to suit the built-in controller in order to keep both CAN libraries as similar as possible.

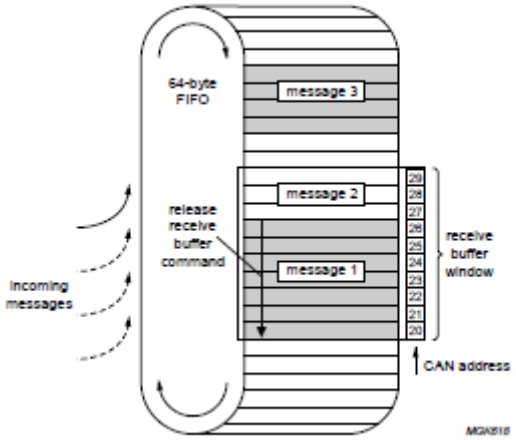
The SJA1000 transmit buffer is described as follows:

“The transmit buffer is an interface between the CPU and the Bit Stream Processor (BSP) that is able to store a complete message for transmission over the CAN network. The buffer is 13 bytes long, written to by the CPU and read out by the BSP.” (Philips Semiconductors, 2000)

The SJA1000 receive buffer is described as follows:

“The receive buffer is an interface between the acceptance filter and the CPU that stores the received and accepted messages from the CAN-bus line. The Receive Buffer (RXB) represents a CPU-accessible 13-byte window of the Receive FIFO (RXFIFO), which has a total length of 64 bytes. With the help of this FIFO the CPU is able to process one message while other messages are being received.” (Philips Semiconductors, 2000)

The following figure shows the SJA1000 receive buffer structure:



Example of the message storage within the RXFIFO (Philips Semiconductors, 2000)

The M16C has a completely different message buffering structure which has a fixed area of memory that can be set up as transmit or receive slots. The figure below shows the memory mapping of the M16C CAN message box:

Address		Message Content (Memory mapping)	
CAN0	CAN1	Byte access (8 bits)	Word access (16 bits)
0060h + n • 16 + 0	0260h + n • 16 + 0	SID10 to SID6	SID5 to SID0
0060h + n • 16 + 1	0260h + n • 16 + 1	SID5 to SID0	SID10 to SID6
0060h + n • 16 + 2	0260h + n • 16 + 2	EID17 to EID14	EID13 to EID6
0060h + n • 16 + 3	0260h + n • 16 + 3	EID13 to EID6	EID17 to EID14
0060h + n • 16 + 4	0260h + n • 16 + 4	EID5 to EID0	Data Length Code (DLC)
0060h + n • 16 + 5	0260h + n • 16 + 5	Data Length Code (DLC)	EID5 to EID0
0060h + n • 16 + 6	0260h + n • 16 + 6	Data byte 0	Data byte 1
0060h + n • 16 + 7 ⋮ 0060h + n • 16 + 13	0260h + n • 16 + 7 ⋮ 0260h + n • 16 + 13	Data byte 1 ⋮ Data byte 7	Data byte 0 ⋮ Data byte 6
0060h + n • 16 + 14	0260h + n • 16 + 14	Time stamp high-order byte	Time stamp low-order byte
0060h + n • 16 + 15	0260h + n • 16 + 15	Time stamp low-order byte	Time stamp high-order byte

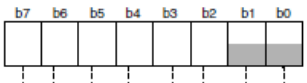
i = 0, 1

n = 0 to 15: the number of the slot

Figure 33: Memory mapping of CANi Message box (Renesas, 2005)

It can be seen that the M16C/6N microcontroller has 2 CAN ports with 16 message slots. These message slots can be configured in various ways by various registers such as the Message Control Register shown below:

CANi Message Control Register j (i = 0, 1) (j = 0 to 15) (4)



Symbol	Address	After Reset
C0MCTL0 to C0MCTL15	0200h to 020Fh	00h
C1MCTL0 to C1MCTL15	0220h to 022Fh	00h

Bit Symbol	Bit Name	Function	RW
NewData	Successful Reception Flag	When set to reception slot 0: The content of the slot is read or still under processing by the CPU. 1 The CAN module has stored new data in the slot.	RO (1)
SentData	Successful Transmission Flag	When set to transmission slot 0: Transmission is not started or completed yet. 1: Transmission is successfully completed.	RO (1)
InvalData	"Under Reception" Flag	When set to reception slot 0: The message is valid. 1: The message is invalid. (The message is being updated.)	RO
TrmActive	"Under Transmission" Flag	When set to transmission slot 0: Waiting for bus idle or completion of arbitration. 1: Transmitting	RO
MsgLost	Overwrite Flag	When set to reception slot 0: No message has been overwritten in this slot. 1: This slot already contained a message, but it has been overwritten by a new one.	RO (1)
RemActive	Remote Frame Transmission/ Reception Status Flag (2)	0: Data frame transmission/reception status 1: Remote frame transmission/reception status	RW
RspLock	Auto Response Lock Mode Select Bit	When set to reception remote frame slot 0: After a remote frame is received, it will be answered automatically. 1: After a remote frame is received, no transmission will be started as long as this bit is set to "1". (Not responding)	RW
Remote	Remote Frame Corresponding Slot Select Bit	0: Slot not corresponding to remote frame 1: Slot corresponding to remote frame	RW
RecReq	Reception Slot Request Bit (3)	0: Not reception slot 1: Reception slot	RW
TrmReq	Transmission Slot Request Bit (3)	0: Not transmission slot 1: Transmission slot	RW

NOTES:
1. As for write, only writing "0" is possible. The value of each bit is written when the CAN module enters the respective state.
2. In Basic CAN mode, slots 14 and 15 serve as data format identification flag.
   The RemActive bit is set to "0" if the data frame is received and it is set to "1" if the remote frame is received.
3. One slot cannot be defined as reception slot and transmission slot at the same time.
4. This register cannot be set in CAN reset/initialization mode of the CAN module.

Figure 34: C0MCTLj and C1MCTLj registers (Renesas, 2005)

In the above figure it can be seen that setting bits b7 and b6 of the Message Control Register determine whether the slot is a receive slot or a transmit slot.

Having 16 message slots that can be configured as both receive and transmit slots is quite flexible in that it allows queuing of messages in both transmit and receive buffers. It is also more complicated in that a software algorithm needs to be implemented to manage the buffers. To keep things simple, the initial version of the CAN library was set up to use one receive slot and one transmit slot. Future versions of the CAN library may include buffering depending on the effectiveness of using just one slot.

With the message reception and transmission set up it was possible to complete the library. Only due to the different interrupt system the library wasn't exactly identical to the existing TCS library. With more time it could be possible to make it identical but this was not a requirement set by TCS so it will need to be addressed when there is a need to use the new library with existing TCS software.

4.13 Recommendations

Device management

As the device management method used is completely new there remain extensive possibilities for improving its operation. Due to time limitations, only a basic version could be set up. The following improvements are only a few of the many possibilities but are deemed the most important at the current stage:

Adjustable poll rate

The time period between polling the devices to update the device list is currently fixed. To improve the device management system it is recommended that the poll time be adjustable so that the system integrator can set it through the ISaGRAF workbench software. For more time-critical applications the poll rate would need to be reduced to allow faster response to addition and removal of devices but for applications where there is a lot of traffic on the backplane it would be more efficient to slow the poll rate to prevent adding additional unnecessary traffic on the backplane.

Error reporting

Code has been developed to log errors to a text file, this has been used in some parts of the backplane driver but an investigation will need to be carried out to determine where else logging is needed. Apart from some basic error logging, there is no other way of reporting errors. It is recommended that a few other forms of error reporting are implemented. One form of error reporting that could be implemented is the capability to send an error message over the industrial network so that other controllers in the distributed control system can be informed that the PCC is faulty. Another form of error reporting that could be implemented is to light up the red module status LED of the PCC and slave modules and to flash error codes so that engineers can diagnose modules by looking at them.

Reacting to failed device management responses

The device management software currently allows a device to miss up to three polls before the driver is uninstalled, this means that if a device is removed or fails, it will take up to four times the time between polls before the driver for that device is uninstalled. The previously described method of uninstalling the driver for a device may be robust and in some cases save processor time for some applications, but may also be too long and inefficient for other applications. To make the device management software as flexible as possible it must also be possible to adjust the number of polls missed before uninstalling a driver. Since it might start getting complicated for a systems integrator to have to adjust both the time and the number of polls before uninstallation, it might even be possible to implement some sort of algorithm to determine which values are suitable based on a single selection made by the systems integrator, or maybe even implementing a self-optimising system.

Reduce memory usage

One significant inefficiency in the backplane software is that there is quite a bit of memory double up, especially with the receive and transmit buffers. The memory use could be made a lot more efficient by sharing the receive and transmit buffers directly rather than transmitting the contents of the receive buffer to the shared memory when a complete frame is received and transmitting the contents of the shared memory to the transmit buffer when a frame is ready to be transmitted.

The decision to have the shared memory separate was initially made to allow the CANISR() routine to already start filling up the next frame while the driver (in some cases slowly) processes the data in the shared memory. Because of this it may be safer to have two separate buffers. The Windows CE operating system however has several features to make memory blocks threadsafe so it may still be possible to make the shared memory threadsafe without having two separate buffers. This would require quite significant software changes and extensive testing and would therefore be recommended to be carried out only if memory use becomes critical.

Retransmission of faulty messages

The current method of retransmitting faulty messages is that when a corrupt fragment comes in, message transmission is stopped and the entire frame is retransmitted. Retransmission of faulty messages could be improved by only retransmitting the corrupted fragment (a single CAN frame); this solution was looked at in initial development, but deemed too complicated to implement at that stage. To reduce traffic on the backplane future revisions of the backplane driver might benefit from implementing this.

CAN buffering

The CAN software in the slave modules was developed to replace software developed for the SJA1000 CAN controller. The SJA1000 CAN software makes use of a receive FIFO buffer built into the SJA1000. The built in CAN controller used in the slave modules has a number of memory “slots” that need to be activated to receive data. The current version of the software only activates one slot so there is no buffering in the electrical layer. Some performance testing will need to be carried out to see if the CAN bus still performs adequately without electrical layer buffering.

4.14 Discussion

There are a few remaining issues to be looked at to improve the operation of the backplane driver, these include: implementing additional error reporting in the device management software, reducing memory usage, improving the retransmission of faulted messages process and buffering CAN frames. Thread prioritisation is also an issue that was looked at but may need further investigation. These issues will need to be thoroughly investigated and resolved before releasing the product.

There are many different kinds of backplanes and even more serial busses on the market, each of these have different features that could be applied to the PCC backplane. A lot of care has had to be taken, and will need to be taken, to ensure that the PCC backplane takes from the existing technology what it needs but not too much that it becomes overly complicated and expensive.

4.15 Conclusion

Various plug and play communications protocols were investigated to develop a device installation process. A method of transmitting frames of variable size over a CAN bus was developed. These features were implemented in the form of a Windows CE driver and software for a slave module with a Renesas microcontroller. The software developed has the following features:

- A universal platform for transmitting frames (of a theoretically infinitely variable length) of data from the bus interface layer of a Windows CE driver (the shared memory) to the bus interface layer of a peripheral device (the main function of the slave modules).
- A bus management system for maintaining a record of devices attached to the CAN bus and installing/uninstalling the Windows CE drivers for the attached devices.
- A platform (the slave module) which can interface a peripheral function to the CAN backplane.

5 Digital I/O module development

5.1 Introduction

This section describes the process gone through to develop the digital I/O software. It describes the hardware used for initial development, the development of the Windows CE software, the development of the Renesas (slave module) software, the development of the ISaGRAF software and the development of the prototype hardware. Having developed the hardware and software, it was put through various performance tests to see if it met the performance standard required by TCS.

5.2 Hardware used for initial development

Because the hardware development for the PCC had not yet been completed, initial software testing was done using products based on similar hardware as what was proposed for use as the hardware of the PCC.

Main Controller

The product used to test the software for the main controller was the CPU5 controller. The CPU5 had already been used for distributed control with windows CE and ISaGRAF. It also had a DeviceNet interface which meant that it already had all the hardware and a driver for the CAN bus. Using a processor which already had Windows CE, ISaGRAF and CAN on it meant that the software and hardware for the new module could be designed in parallel. This allows the designers to work together and customise hardware and software to work together rather than just making the software design constrained by hardware limitations or the other way. Another benefit of being able to develop hardware and software in parallel is that there is a faster time to market, which has many cost and competitive advantages for TCS.

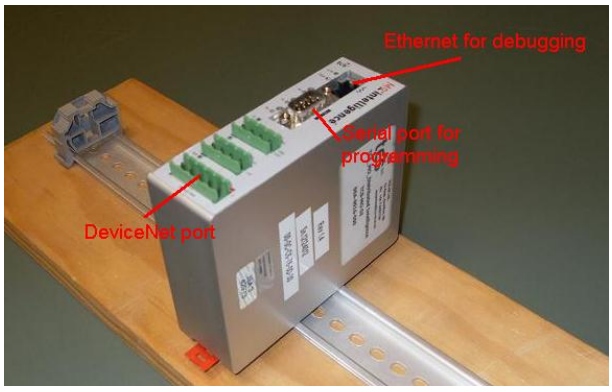


Figure 35: CPU5 master controller module

Slave modules

The microcontroller chosen to control the slave modules was to be a microcontroller from the Renesas M16C family. TCS has various products using M16C controllers; one of these products has the in-house name of 'CPU4'. Although it is the predecessor of the CPU5, the CPU4 is quite a different product and does not have Windows CE or ISaGRAF. The CPU4 was designed as a DeviceNet interface with a serial port for EIA232, EIA422 and EIA485 communications and it can also support a daughterboard for 8 bit digital I/O. Because the CPU4 hardware has an M16C processor, supports DeviceNet (CAN), has a serial port and can be used for digital I/O it was the perfect hardware platform to use to test the slave module software.

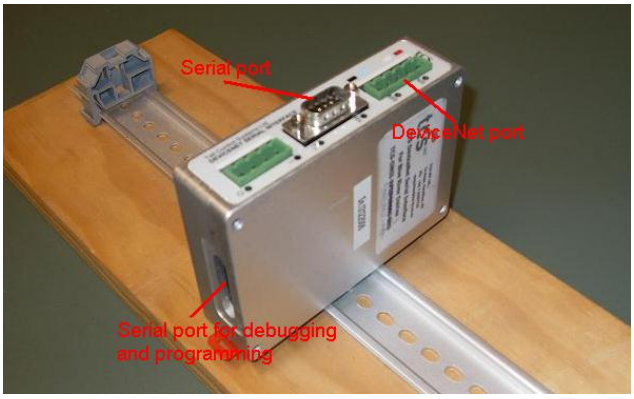


Figure 36: CPU4 slave module controller with serial port

Figure 37: Bare CPU4 board with digital I/O daughterboard

Test setup

The figure below shows the way the CPU4 and CPU5 modules were set up to carry out initial software tests.

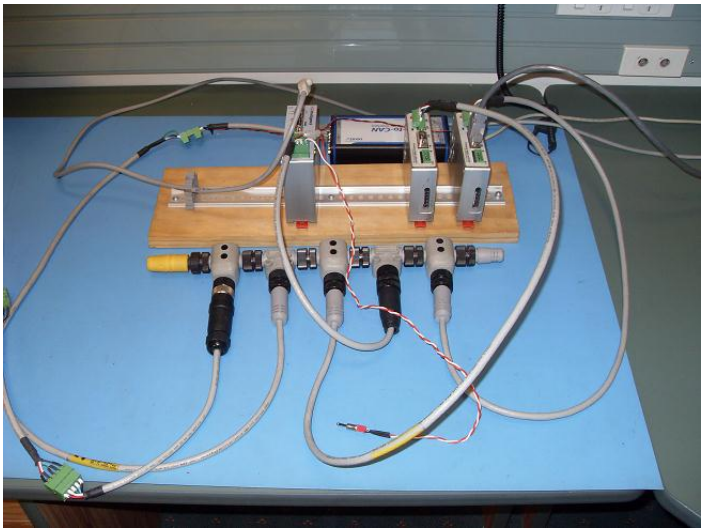


Figure 38: Initial test setup with CPU5 and CPU4 hardware

The module mounted on the DIN rail on the left hand side is the main controller (CPU5) module. The two modules on the right hand side, also mounded on DIN rail are slave modules (CPU4). The module in the background, behind the wooden plank, is a CAN analyser. All four of the aforementioned modules are interconnected with DeviceNet standard cable. The CAN analyser is connected to a PC via a USB cable to analyse the data on the CAN bus. The main controller is attached to the PC via an Ethernet cable for downloading and debugging. The slave modules are attached to the PC via a serial cable for downloading and debugging.

5.3 Windows CE stream interface driver

Development of the digital I/O driver was started with a template for a stream interface driver supplied by Andrew Meek. The template was similar to this one:

```
#include <windows.h>
#include <Devload.h>

BOOL XXX_Deinit( DWORD hDeviceContext )
{
      return TRUE;
}

DWORD XXX_Init(ULONG   RegistryPath)
{
      HKEY hKey;
      RETAILMSG( 1, (TEXT("XXX_Init\n")));
      hKey = OpenDeviceKey((LPCTSTR)RegistryPath);
      if ( !hKey ) {
           RETAILMSG(1, (TEXT("Failed to open devkeypath,\r\n")));
      }
      else
      {
            // Read values from registry if needed
            RegCloseKey (hKey);
      }
      return TRUE;
}

BOOL WINAPI DllEntry(HINSTANCE DllInstance, ULONG Reason, LPVOID Reserved)
{
    RETAILMSG( 1, (TEXT("DriverShell: DllEntry\n")));
    return TRUE;
}

VOID XXX_PowerUp(DWORD hDeviceContext )
{
}

VOID XXX_PowerDown(DWORD hDeviceContext)
{
}

DWORD XXX_Open(DWORD hDeviceContext, DWORD AccessCode, DWORD ShareMode)
{
    return hDeviceContext;
}

BOOL XXX_Close(DWORD hOpenContext)
{
    return TRUE;
}

DWORD XXX_Read(DWORD hOpenContext, LPVOID pBuffer, DWORD Count)
{
    return 0;
}

DWORD XXX_Write(DWORD hOpenContext, LPCVOID pSourceBytes, DWORD
NumberOfBytes)
{
```

```
    return 0;
}

DWORD XXX_Seek(DWORD hOpenContext, long Amount, DWORD Type)
{
    return 0;
}

BOOL XXX_IOControl(DWORD hOpenContext, DWORD dwCode, PBYTE pBufIn, DWORD
dwLenIn, PBYTE pBufOut, DWORD dwLenOut, PDWORD pdwActualOut)
{
      BOOL RetVal = TRUE;
      switch(dwCode)
      {
            default:
            RetVal = FALSE;
            break;
      }
      return RetVal;
}
```

(Eitman, 2008)

The functions listed in this shell are all compatible with the stream interface which means that they can be called with standard windows functions such as ActivateDeviceEx() which activates the device by calling the XXX_DllEntry() and XXX_Init() stream interface functions and CreateFile() which calls the XXX_Open() stream interface function and returns a handle to the device (The 'handle' is a pointer to the device object in the memory, this is passed to the programs that use the device). The counterparts to ActivateDeviceEx() and CreateFile() are: DeactivateDevice(), which calls XXX_Deinit(), and CloseHandle() which releases the object handle returned by CreateFile() .

XXX corresponds with the device prefix. For example a serial port with prefix COM would have functions such as COM_Init(), COM_DllEntry() etc…

5.4 Software Development

This section discusses the software development carried out to complete the digital I/O driver. The only stream interface functions used in the digital I/O driver were the XXX_Init, XXX_Deinit and XXX_IOControl functions.

Installing a stream interface driver

The function used to install a driver with a stream interface is ActivateDeviceEX(). One of the parameters needed by ActivateDeviceEX() is the registry key containing information needed to install the driver. The registry key must contain a value such as "Dll = IO8.dll" for ActivateDeviceEX() to load the driver, a number of other parameters are also required, these parameters are all managed by the registry management software discussed in the backplane driver development section. The code below is used to install the digital I/O driver:

```
REGINI bpi_data[8] = {
            {L"MacID", (LPBYTE)(&bpi_MacID), sizeof(DWORD), REG_DWORD },
            {L"BufLength", (LPBYTE)(&bpi_bufLength), sizeof(DWORD),
            REG_DWORD },
            {L"rxMem", (LPBYTE)(&BP_DeviceList[bpi_MacID].rxMem),
            sizeof(DWORD), REG_DWORD },
            {L"txMem", (LPBYTE)(&BP_DeviceList[bpi_MacID].txMem),
            sizeof(DWORD), REG_DWORD },
            {L"BusName", (LPBYTE) TEXT("CAN_0_0_0"), 20*sizeof(TCHAR),
            REG_SZ },
            {L"BusParent", (LPBYTE)(&BP_Hnd), sizeof(DWORD), REG_DWORD },
            {L"InterfaceType", (LPBYTE)(&InterfaceType), sizeof(DWORD),
            REG_DWORD },
            {L"Instance", (LPBYTE)(&BP_Instance), sizeof(DWORD), REG_DWORD
            }
      };

BP_DeviceList[bpi_MacID].driver =
ActivateDeviceEx((LPCWSTR)bpi_fullPath,bpi_data,8,NULL);
```

It can be seen that another fairly important parameter passed to the ActivateDeviceEX function is bpi_data. This parameter is an array of REGINI structures, ActivateDeviceEX allows one to pass it an array of REGINI structures to add extra parameters to the HKEY_LOCAL_MACHINE/Drivers/Active key, this key is accessed by the XXX_Init routine when the driver is initialised, it allows the backplane driver to store important non-volatile configuration information at the same time as passing important parameters such as the handle to the shared memory to the driver.

XXX_IOControl function

The following block of code shows the IOControl function:

```
BOOL IO8_IOControl(DWORD hOpenContext,
                           DWORD dwCode,
                           PBYTE pBufIn,
                           DWORD dwLenIn,
                           PBYTE pBufOut,
                           DWORD dwLenOut,
                           PDWORD pdwActualOut)
{
      PDEVICESTATE pds = (PDEVICESTATE) hOpenContext;
      unsigned int i;
      DWORD ActualOut = 0;
      //copy data from IOControl buffer to shared memory
      if(dwLenIn) {
            if(pds->tx_FileMemory[0] == 0) {
                  for(i=0; i<dwLenIn; i++) {
                        pds->tx_FileMemory[i+1] = (unsigned int) pBufIn[i];
                  }
                  pds->tx_FileMemory[i+2] = 2;
                  pds->tx_FileMemory[0] = 1;
                  ActualOut = 1*sizeof(BYTE);
            }
            SetEvent(pds->TXEventHandle);
      }

      if(dwLenOut) {
            if(pds->rx_FileMemory[0] == 1) {
                  for(i=0; i<dwLenOut; i++) {
                        pBufOut[i] = (BYTE) pds->rx_FileMemory[i+1];
                  }
                  ActualOut = 1*sizeof(BYTE);
                  pds->rx_FileMemory[0] = 0;
            }
      }
      *pdwActualOut = ActualOut;

      return TRUE;
}
```

This IOControl function is called by an application such as ISAGRAF to read the inputs and/or set the outputs depending on how the function arguments are set.

The decision was made to use the IOControl stream interface function for the digital I/O because it allows the inputs to be read and the outputs to be set in one go rather than making the program controlling the I/O (e.g. ISaGRAF) to have to call a Read() and then a Write(). If the software wants to do just a read, all it needs to do is set the dwLenIn argument to 0.

XXX_Init/XXX_Deinit functions

Other than the XXX_IOControl function, which carries out the main functionality of the digital I/O driver, there are also the XXX_Init and XXX_Deinit functions. The XXX_Init and XXX_Deinit functions are called at initialisation and de-initialisation respectively. These functions are used to carry out the operations that only need to be done once such as initialising and freeing the shared memory and setting up the shared interrupts.

Multiple instances of the same driver

While developing the IO8 driver it was found that any variables declared globally in the stream interface driver code would be global for all instances of that driver, meaning that it was not possible to store information specific to an instance of a driver as a global variable. This problem was worked around by storing information specific to each instance in a globally declared array of structures as shown below:

```
typedef struct _DeviceState_tag {
	DWORD OK;
	DWORD MacID;
	DWORD BufLength;
	HANDLE rx_FileMap;
	HANDLE tx_FileMap;
	unsigned int*  rx_FileMemory;
	unsigned int*  tx_FileMemory;
} DEVICESTATE, *PDEVICESTATE;

#define	MAXDEVICES		127

DEVICESTATE gDevices[MAXDEVICES];
```

Some code was then written in the Init() routine to initialise the array and add entries:

```
DWORD IO8_Init(LPCTSTR RegistryPath, DWORD dwBusContext)
{
	DWORD dwHandle = 0;
	DWORD dwLen = sizeof(DWORD);
	HKEY hKey;

	// look for a free device instance
	PDEVICESTATE pds;
	if(devCount==0) {
		pds = &gDevices[devCount];
		pds->OK = 1;
		devCount++;
	}
	else{
		unsigned int i;
		pds = &gDevices[0];
		for(i = 1; pds->OK!=0; i++) {
			pds = &gDevices[i];
		}
		if(i>devCount)
			devCount++;
		pds->OK = 1;
	}
```

The stream interface allows a pointer to be passed from the Init() and Open() functions to the IOControl(), Read(), Write(), Close() and DeInit() functions. This feature was used to pass device specific information such as the shared memory and MAC ID to the functions using it.

The code below shows a pointer being returned by a function, and the pointer being used in another function.

```
dwHandle = (DWORD) pds;

      return dwHandle;
}


BOOL IO8_IOControl(DWORD hOpenContext,
                           DWORD dwCode,
                           PBYTE pBufIn,
                           DWORD dwLenIn,
                           PBYTE pBufOut,
                           DWORD dwLenOut,
                           PDWORD pdwActualOut)
{
      PDEVICESTATE pds = (PDEVICESTATE) hOpenContext;
```

5.5 Slave microcontroller software

Operating system

Unlike the main controller, the slave microcontroller only has limited functionality so it does not need to run an operating system such as Windows CE. The slave microcontroller is programmed using software developed by TCS for other products using the Renesas M16C microcontroller. This section describes the modifications and additions made to the existing software for the digital I/O slave module.

Initialisation

Port 10 and 0 of the Renesas microcontroller were used for the digital I/O pins.

To initialise port 10 and port 0 the port directions needed to be set. This was done using the following code:

```
void IO8_Init() {
      pd10_1 = INPUT;
      pd10_3 = INPUT;
      pd10_6 = INPUT;
      pd10_4 = INPUT;
      pd10_0 = INPUT;
      pd10_2 = INPUT;
      pd10_7 = INPUT;
      pd10_5 = INPUT;
      pd0_6 = OUTPUT;
      pd0_4 = OUTPUT;
      pd0_2 = OUTPUT;
      pd0_0 = OUTPUT;
      pd0_7 = OUTPUT;
      pd0_5 = OUTPUT;
      pd0_3 = OUTPUT;
      pd0_1 = OUTPUT;
```

Main function

The main function of the slave microcontroller was written as follows:

```
void ProtocolMain(void)
{
	unsigned char *newdata;
	unsigned char txdata[BufLength];
	int i;
	int testin[8];
	int testout[8];

	//This part will be inactive until the device has been installed, prevents
	// unecessary traffic on the backplane
	if( MacID > 0 ) {
		//If data has been received
		if( ProtocolReceiveBufferReady ) {
			ProtocolReceiveBufferReady = 0;
			//Send the data to the digital outputs
			setOutputs( ProtocolReceiveBuffer );
		}

		//if the transmit buffer is ready
		if( !ProtocolTransmitBufferReady ) {
			//get data from inputs
			newdata = getInputs();
			//scan the data received from the io
			for ( i = 0; i<BufLength; i++)
			{
				//if the data has changed, change the transmit buffer
				ProtocolTransmitBuffer[i] = newdata[i];
				if( ProtocolTransmitBuffer[i] != txdata[i] ) {
					txdata[i] = ProtocolTransmitBuffer[i];
					change = 1;
				}
			}
			//if any of the data has changed transmit the transmit buffer
			if( change ) {
				change = 0;
				ProtocolTransmitBufferLength = BufLength;
				ProtocolTransmitBufferReady = 1;
				asm("\t int #002h"); //trigger the can
			}
		}
	}
}
```

Upon successful reception of a frame from the backplane, the outputs are set. Every time an input changes, or every time the cyclic update timer times out, the input data is sent across the backplane.

SetOutputs()

The function to set the outputs was fairly simple:

```
void setOutputs( unsigned char* setval ) {
	//set outputs
	char temp = setval[0];
	p0_0 = ( temp & 0x01 );
	p0_1 = ( ( temp & 0x02 ) >> 1 );
	p0_2 = ( ( temp & 0x04 ) >> 2 );
	p0_3 = ( ( temp & 0x08 ) >> 3 );
	p0_4 = ( ( temp & 0x10 ) >> 4 );
	p0_5 = ( ( temp & 0x20 ) >> 5 );
	p0_6 = ( ( temp & 0x40 ) >> 6 );
	p0_7 = ( ( temp & 0x80 ) >> 7 );
}
```

Bitwise shifting was used in case the pinout of the hardware connected to the port of the processor was not in the same order as the pinout of the port. This will allow the hardware developer to have some more freedom and will save the end user from having to rearrange the bits at the application level.

GetInputs()

Some of the code for the getInputs() function is shown below:

```
//defines
#define INPUT_DEBOUNCE_TIME 1 //10ms

//global variables
BOOL InputHold[8];
BOOL DebouncedInputs[8];

unsigned char* getInputs(void) {
	static unsigned char retval[1];
	unsigned char temp = 0x00;
	//get inputs
	if( InputDebounce[0] == 0 ) {
		InputDebounce[0] = INPUT_DEBOUNCE_TIME;
		if(p10_4 == InputHold[0])
			DebouncedInputs[0] = InputHold[0];
		InputHold[0] = p10_4;
	}
```

It can be seen that every time an input changes, the debounce timer is started and data is not sent across the backplane until the debounce has completed.

Input Debounce

To make the digital inputs more robust it was necessary to implement a debounce feature. The debounce function made use of a hardware timer interrupt service routine available in the TCS library for the Renesas microcontroller. The interrupt was set up to trigger every 10 milliseconds; a global variable was put in the interrupt service routine to be decremented every time it was called. Part of the code for the interrupt service routine is shown below:

```
void ProtocolTimerFunc(void) {
      int i;
      //Count down debounce timer - used to debounce trigger signal,
      // which can be I/O
      for(i=0; i<8; i++) {
            if (InputDebounce[i] > 0)
                  InputDebounce[i]--;
      }
```

Each input was given their own debounce timing variable to allow them to be sampled individually.

The debounce time of 10ms was recommended by staff at TCS as an adequate debounce time, an article on the EE Times Design website written by Jack Ganssle also recommends using a 10ms debounce time (Ganssle, 2004). A feature was also added to allow the debounce time to be adjusted for applications that require a faster input response time.

The figure below shows a timing diagram of a Maxim MAX6861 debounce chip, the debounce routine used was based on this timing diagram:

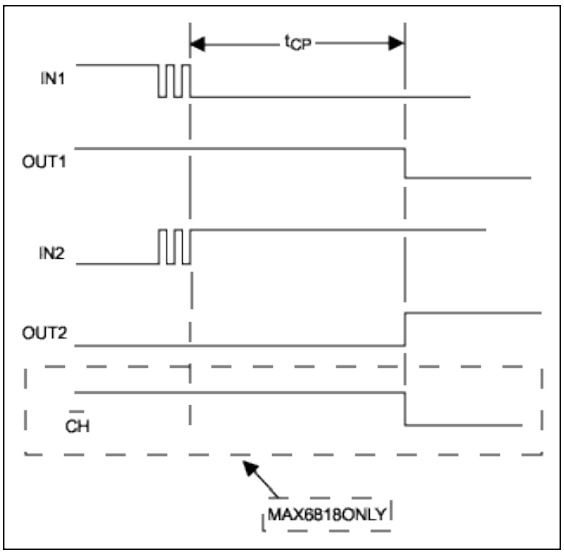


Figure 39: MAX6861 timing diagram (Maxim Integrated Products, 2000)

Only in the software implementation, rather than requiring the input to be stable over the continuous period of t_{CP} before changing the output, the input is sampled when it changes state, and if the input is still in the same state when it is sampled 10ms later, the output will change to that state.

Cyclic vs. Change-Of-State

The initial software design only sent the input status to the main controller (digital I/O driver) when an input value had changed. This method of updating inputs is called Change-Of-State (COS). COS was used because it used the least amount of traffic on the backplane and the least amount of processing power in the main controller. During the development of the serial port driver, it was found that occasionally, due to a heavy load on the CAN bus, the digital I/O driver would miss a packet and therefore not react to the change of an input. This usually happened when the processor was under a high load and unable to process incoming CAN messages fast enough. To reduce the effects of missing a packet, it was decided to send the input status to the main controller at regular intervals (Cyclically) as well as at change of state. The slave module was set up with functionality for the systems integrator to set the cycle time to give them the freedom to decide between having a low load on the CAN bus or fast and reliable reactions to a change of input state.

5.6 ISaGRAF software development

The final part of the software development for the digital I/O driver was the development of the ISaGRAF software to read/drive the digital I/O. The ISaGRAF software was developed based on a template for digital I/O provided by ISaGRAF.

ISaGRAF I/O structure

The figure below shows the digital I/O structure as shown in the ISaGRAF workbench software:

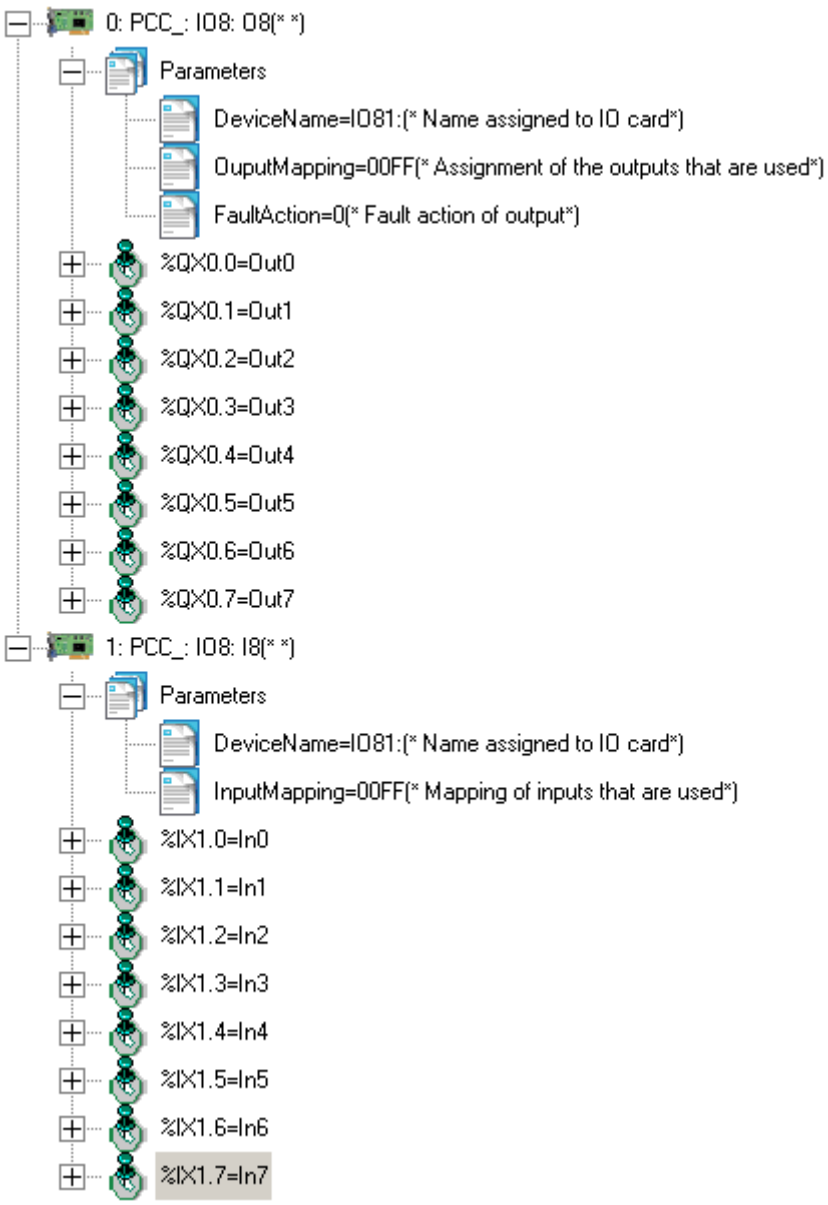


Figure 40: ISaGRAF digital I/O structure

Looking at the digital I/O structure, it is important to note the DeviceName parameter. The DeviceName parameter is used by the underlying software to connect to the driver for the device labelled IO81.

Opening, Reading and Writing

The following code shows some of the code used by ISaGRAF to open the digital I/O driver:

```
typSTATUS pcc_io8i8IosOpen(
   strRtIoSplDvc* pvRtIoDvc /* Run time io struct of the device to open */
   )
{
      strI8 *params = (strI8*)pvRtIoDvc->pvOemParam;
      sprintf(io8Devices[firstfree].DeviceName, "%s",params->DeviceName);
      io8Devices[firstfree].deviceHandle =
      CreateFile((LPCWSTR)io8Devices[firstfree].DeviceName, GENERIC_WRITE,
0, NULL, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, 0);
```

It can be seen that ISaGRAF uses standard Windows CE system calls to create a handle to the device giving the software read/write access to it.

The software below shows some of the code used by the ISaGRAF software to read data from the digital I/O:

```
void pcc_io8i8IosRead (
   strRtIoSplDvc* pRtIoSplDvc /* Run time io struct of the device to read
*/
   )
{
      strI8 *params = (strI8*)pRtIoSplDvc->pvOemParam;
      if(DeviceIoControl(
            io8Devices[count].deviceHandle,     //Handle
            0,                        //IO control code
            NULL,                     //input buffer
            0,                        //input buffer size
            &inputState,              //output buffer
            1,                        //output buffer size
            &lpBytesReturned,NULL     //actual number of bytes returned
            )) {
            if(lpBytesReturned) {
                  if(io8Devices[count].lastInputState != inputState)
                  {
                        io8Devices[count].lastInputState=inputState;
                        io8Devices[count].changed = 1;
                  }
            }
            else {
                  io8Devices[count].changed = 0;
                  inputState = io8Devices[count].lastInputState;
            }
```

It can be seen that ISaGRAF uses a standard Windows DeviceIoControl function to read data from the (stream interface) driver. The write function works in much the same way as the read function, it also uses the DeviceIoControl function to write data to the digital I/O.

Preventing errors

The software not shown in the previous sections of code is the code that checks whether the device is still active. Because the digital I/O modules are plug-and-play, there is a possibility that a device is removed at run time. If ISaGRAF attempts to access that I/O module after it has been removed, there will be a serious error because the memory that used to be taken up by the device will be unallocated or allocated to something else. To prevent the software from crashing when an error occurs, there needs to be a system in place to check that the device being accessed is still there. The following list shows the steps taken by the ISaGRAF software when it finds out that the handle to a device is no longer active:

1. Check if the device handle is inactive or if the DeviceIoControl fails more than a certain amount of times (IO8MAXIOCTLFAILS).
2. Set a variable in the device structure that it needs re-opening
3. Close the current handle to the device (set it to inactive)
4. The next time ISaGRAF attempts to access the device and the re-open variable is set
5. Wait a for a certain number of scans (SCANS_BEFORE_REOPEN)
6. Search for a device in the registry with the same device name* and open a handle to that device if it is found. When a device is removed and replaced with another device, or just put back in, it will have a different handle but the registry management software (mentioned previously) will ensure that the device has the same name.

This allows the software to continue running when a module fails, the backplane driver logs an error in a text file but there is no error reporting in ISaGRAF. This feature was built in to allow the system to keep running as best it could until an operator or engineer becomes available to resolve the error.

*Devices are assigned a device name by the system integrator; this name represents the functionality of the device. The registry management system ensures this name is retained and any replacement devices also get this name. The device name is therefore the most consistent value by which to recognise a hardware connection.

5.7 Hardware development

Schematic design

It was decided to use a microcontroller from Renesas M16C family for the slave modules because TCS uses the M16C microcontrollers in quite a range of their products. To reduce design time of the backplane communications software, the same microcontroller was used for all modules. Because TCS uses the M16C microcontroller in other products, many of the features common to their products had already been designed and tested, so features such as module status lights, programming interfaces and EEPROM memory could be copied directly from the schematics of other products. The only part of the Digital I/O hardware that had not been used in another TCS product was the combination of digital input and output pins, this had to be carefully designed to make the hardware able to withstand industrial conditions and handle a range of voltages around 24 volts without damaging the microcontroller which is running at a 3.3 volt voltage level.

The following image shows the schematic design of one of the eight I/O lines:

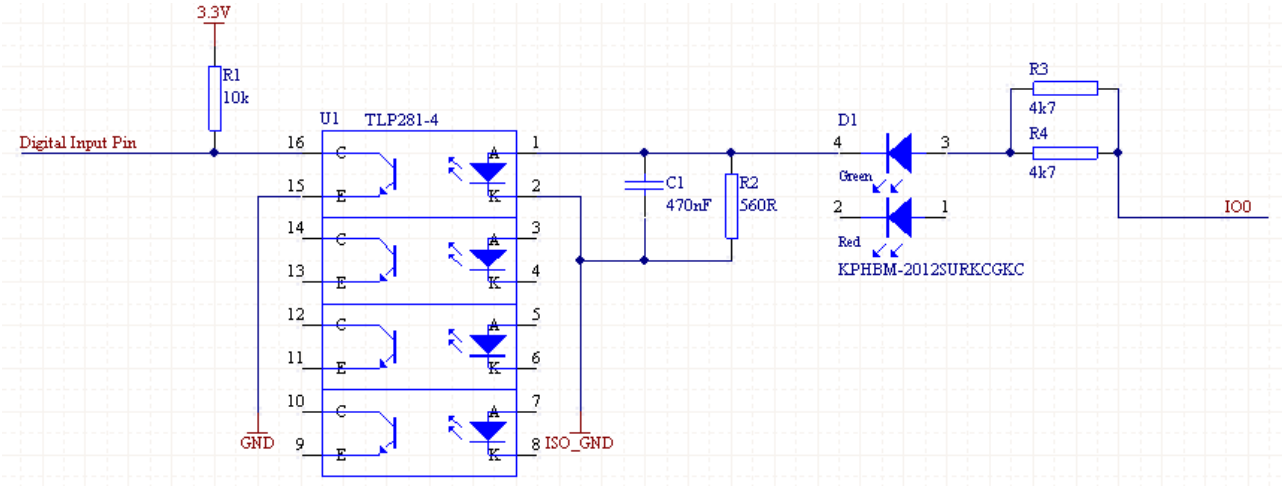


Figure 41: Digital input part of I/O schematic

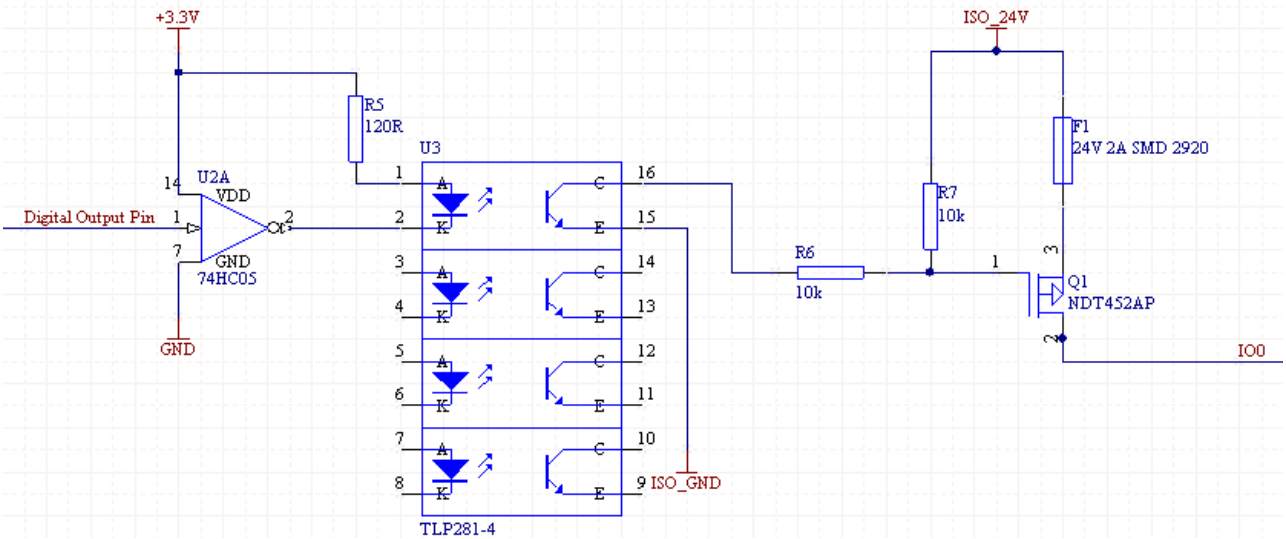


Figure 42: Digital output part of I/O schematic

The schematics above show that two separate pins of the microcontroller are used for input and output respectively but they come together at the same point (IO0) making the module have a single combination input/output point. It can be seen that the 24 volt side is optically isolated and fused, together with the selection of industrial temperature rated components this makes it an industrial grade module with adequate protection to prevent damages to the microcontroller (and PCC) if the module is attached to an unreliable voltage source to the I/O connector.

Hardware prototype

The figure below shows the prototype design of the digital I/O hardware without the cover showing how tightly the components are placed together to fit all the components required to have eight combination input and output lines going into the microcontroller.

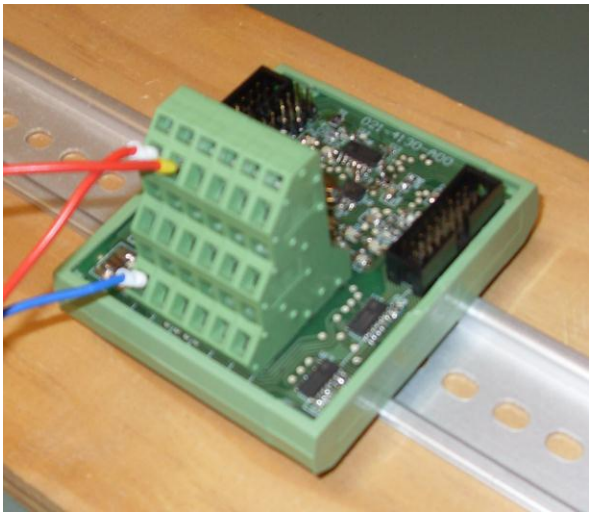


Figure 43: Digital I/O hardware prototype

The figure below shows the digital I/O module with cover as it was taken to the SPS tradeshow in Nuremberg:

Figure 44: digital I/O module with cover

5.8 Performance testing

The following figure shows the ISaGRAF program used to performance test a single pin of the digital I/O module:

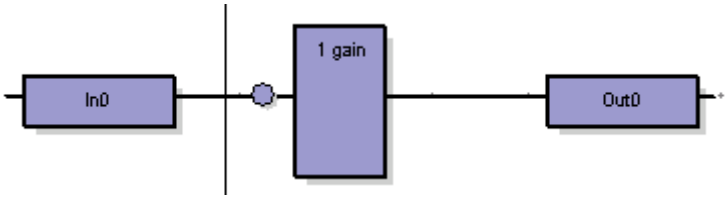


Figure 45: ISaGRAF program for testing the digital I/O

The leftmost block represents a digital input. The block in the centre is a 1 gain block which inverts the value received at its input and sends it to its output in the next scan cycle. The block on the right represents a digital output. It can be seen that the same graphical block is re-used for input and output blocks, this could be confusing for a systems integrator because the difference between blocks is not instantly apparent; this issue will need to be raised with ISaGRAF software developers.

The program functionality was to blink an output on and off at the I/O scan rate set by ISaGRAF. When the software was tested with a scan rate of 100ms the digital I/O module worked really well producing a 100ms output pulse. When the scan rate was raised however performance issues started becoming visible.

Debounce time

When the scan rate was set faster than 10ms the output pulse would still only be 10ms, this was due to the debounce time of 10ms preventing the inputs from responding immediately. Because the debounce time was seen as a performance limitation, it was made adjustable so that systems integrators could decide for themselves whether they preferred having a longer debounce time, or a very fast system that was susceptible to switch bounce.

Thread priority

Observing the output pulse over longer periods of time it could be seen that at regular intervals the output pulse was slightly longer than expected. After some investigation it was found that the longer output pulses occurred at the same time that the device management thread was active. To increase the performance of ISaGRAF in this situation, the thread priority of the CAN, transmit and ISaGRAF threads were increased to a level higher than that of the device management thread. Later on it was also found that there were a few other processes running in the Windows CE operating system that had a higher priority than the digital I/O threads, so these processes also had their thread priorities reduced. Reducing the priorities of the less important threads made the Digital I/O run a lot smoother and also reduced the occurrence of missed packets for which a cyclic updating system had to be implemented.

5.9 Recommendations

The initial prototype was completed to a point where it worked reasonably reliably so that development could move on to the rest of the modules. This section describes a few features that are recommended to be included in future revisions:

Fault and error handling in the code

This was addressed quite extensively for the digital I/O module as the rest of the functionality was quite basic and it was to be an essential part of the Nuremberg show display. However it is quite likely that there are still small chances of errors that could pop up due to unexpected conditions occurring. The main focus in this would be to safeguard the outputs from randomly switching, which could have catastrophic results in an industrial application.

Fault and error actions

Although a few fault actions were hard-coded into the slave module, no method of passing fault information back to the PCC controller had been implemented. Most industrial PLCs also have a method for the systems integrator to set the fault state to which the digital outputs of a module default when something like a communications fault occurs. These issues would need to be addressed before releasing the digital I/O module as a reliable industrial I/O module.

5.10 Discussion

The digital I/O driver uses a monolithic stream interface driver, which is the most basic driver type available in Windows CE, so it was a good starting point for the first driver developed for the PCC backplane. A number of issues, such as error reporting and fault actions, remain that need to be addressed before releasing the digital I/O driver in industry. The issues that remain were excluded due to time limitations and do not relate to the specific operation of a stream interface driver loaded by a bus driver so they are not important in terms of the project definition. There are many more features which aren't essential for an industrial digital I/O module to have which could be included; it will be up to TCS or their customers to decide whether they need to be included.

5.11 Conclusion

Reliable and robust software was successfully developed to expand the functionality of the PCC to include digital inputs and outputs. The software included a Windows CE driver and code to handle the digital I/O in the slave module. The hardware developed was an 8 bit digital I/O module based on the CPU4 hardware to connect to the PCC via the CAN backplane. This hardware and software not only expanded the functionality of the PCC but it also demonstrated that the backplane driver worked. The digital I/O module was so developed that it could also be used as a platform/sample for the development of future modules based on direct I/O such as an Analogue I/O module and a DOL type motor control station. The digital I/O module was put through extensive performance tests and after some adjustments proved to work according to the requirements set by TCS.

6 Serial port module development

6.1 Introduction

This section discusses the development of the hardware and software for the PCC serial port module. The various parts of the development include: developing the serial port driver for windows CE, finding a way to reliably transmit a stream of data over a CAN bus, developing software for the slave module, developing ISaGRAF function blocks and developing a serial port module. The completed hardware and software was put through various tests to prove that it performed as expected by TCS.

6.2 Hardware used for initial testing

A similar test setup was used for the serial port software development as for the digital I/O software. The only difference between the hardware used for the testing and the final design of the PCC serial port module was that the test hardware only had one serial port. The expected final design was to be a module with two serial ports, both providing functionality for EIA232, EIA422 and EIA485. Tests could only be carried out with one EIA232 serial port; the design had to allow for the extra port, and extra functionality. Some more time needed to be allocated for further software development after the arrival of the new hardware modules.

The figure below shows the test setup used for the serial port software development:

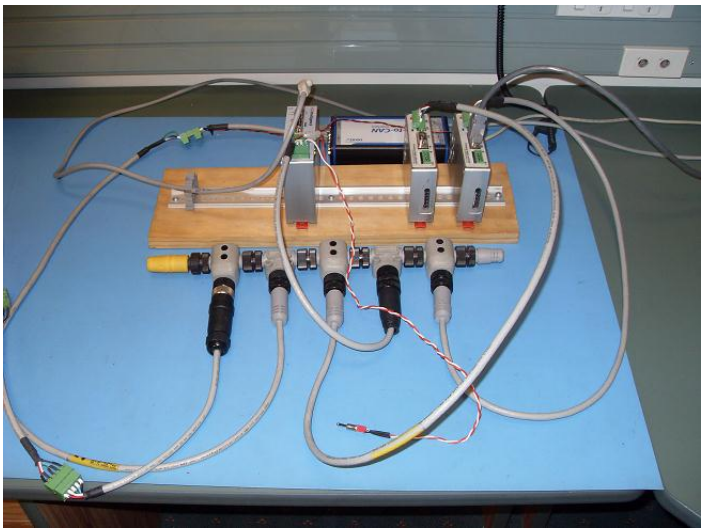


Figure 46: serial port software test setup

The module mounted on the DIN rail on the left is the CPU5 module which carries out the role of the main controller. The two modules on the DIN rail on the right side are the CPU4 modules; they carry out the role of the slave modules. The module behind the board is the CAN analyser used to monitor the messages sent across the backplane on a PC. All the aforementioned modules are connected together by the DeviceNet cabling also seen in the photo.

Other hardware used was a laptop with check weigher simulation software on it. This was used to test the performance of the serial port. A barcode scanner was also used for some of the performance tests.

6.3 Layered drivers

Windows CE uses a concept called layered drivers to make it easier to transport drivers across different platforms. Layered drivers usually have two main layers, in some cases these layers are split into further layers. The two main layers are called MDD and PDD which are explained below:

MDD = Model Device Driver: This driver is platform independent and provides a generic interface (such as the stream interface) to the OS and driver clients. The MDD calls PDD functions to access the hardware.

PDD = Platform Dependent Driver: This driver contains the functions specific to the hardware for which it is designed (dependent). The PDD provides functions that can be called by the MDD.

(Ng, 2007)

Using this structure, it is only necessary for the embedded device developer to understand and modify the hardware access functions in the PDD layer rather than having to completely understand the operation of a complicated driver such as a the serial port driver.

6.4 COM MDD and PDD interaction

The image below shows the structure of the serial port driver used for the PCC serial port modules. The reason this structure was used instead of a fully monolithic stream interface driver as used for the digital I/O driver was because the operation of a serial port is quite a bit more complicated and Windows CE provides the upper layers of the driver so it was only necessary to design software for the PDD (platform dependent) layer. Most of the sample drivers for Windows CE were also designed for the layered structure.

Figure 47: Serial port layer structure (Ravalia, 2006)

It was decided to develop a layered driver so that development focus could be put more on the transmission of the serial data across the backplane rather than the interaction of the serial port driver with the Windows CE kernel. Because there are a number of layers between the stream interface and the PDD layer, the software had to be designed with a completely different interface to that of the digital I/O driver. To obtain details of this interface, the driver for the built in serial port of the PCC was used as a template.

6.5 Software development

The following sections describe the various parts of the software, roughly in the order they were developed. A serial port driver supplied with the board was used both as a template for the new driver and to work out the functionality of the various features and how they interfaced with the upper layers.

Connecting the PDD to the upper layers

The template driver was programmed in the C++ style with each layer encapsulated by the layer above. The entry point of the PDD driver therefore calls the CreateSerialObject() function to create an instance of a serial port and then binds the PDD functions to this object. Upon successful binding of the PDD functions to the serial object the Init() function was called to initialise the serial port. The code which connects the PDD to the upper layers is shown below:

```
CSerialPDD * CreateSerialObject(LPTSTR lpActivePath, PVOID pMdd,PHWOBJ
pHwObj, DWORD DeviceArrayIndex)
{
    CSerialPDD * pSerialPDD = new PCCUart(lpActivePath,pMdd, pHwObj);
    if (pSerialPDD && !pSerialPDD->Init()) {
        delete pSerialPDD;
        pSerialPDD = NULL;
    }
    return pSerialPDD;
}
```

The important functions in the PDD object that interface with the upper layers were established to be the Constructor, Destructor, Init, Deinit, XmitInterruptHandler, ReceiveInterruptHandler and the serial port setup functions. There was also a ThreadRun() function which ran as a thread for processing incoming messages.

Constructor and Destructor

Because the serial PDD was set up as a C++ class it had a constructor and a destructor. The advantage of using a class, meant that the constructor and destructor were called separately for each instance and the variables constructed were accessible by all the functions in the instance of that particular class. This avoided the need to use an array of global variables and pass handles to these variables between functions as had been done with the digital I/O driver. The code below shows the constructor initialising a few variables:

```
//Constructor of the (serial port) class
PCCUart::PCCUart (LPTSTR lpActivePath, PVOID pMdd, PHWOBJ pHwObj )
:   CSerialPDD(lpActivePath,pMdd, pHwObj)
,   m_ActiveReg(HKEY_LOCAL_MACHINE,lpActivePath)
,   CMiniThread (0, TRUE)
{
      //initialise global variables
      DWORD MacID = 0;
      DWORD Instance = 0;
      DWORD BufLength = 0;
      DWORD rx_handle = 0;
      DWORD tx_handle = 0;
      BOOL PCC_Xmit_DataReady = FALSE;
      HANDLE rx_FileMap = NULL;
      HANDLE tx_FileMap = NULL;
      HANDLE EventHandle = NULL;
```

Initialisation/De-initialisation

Apart from the functions looking slightly different due to the fact that a different interface was being used, the functionality of the initialisation and de-initialisation functions of the serial port was exactly the same as that of the digital I/O because the shared memory and shared interrupts were the only thing that needed to be set up.

Serial port setup functions

The serial port PDD object contained four functions for setting up the various serial port functions, these functions are listed below:

```
BOOL    PCCUart::SetBaudRate(ULONG BaudRate,BOOL bIrModule)
BOOL    PCCUart::SetByteSize(ULONG ByteSize)
BOOL    PCCUart::SetParity(ULONG Parity)
BOOL    PCCUart::SetStopBits(ULONG StopBits)
```

A list of command codes were set up to represent the various configurations of the serial port features. The above functions were modified to transfer these commands to the slave module to set up the serial port. Some of these codes can be seen below:

```
typedef enum {
      PCCSERIAL_Reserved = 0x00,
      PCCSERIAL_Mode = 0x01, //mode as in 232,422 or 485
      PCCSERIAL_Setup = 0x02, //setup the baud rate, databits, parity and
stopbits
      PCCSERIAL_Command = 0x03, //commands such as clear txfree etc
      PCCSERIAL_Data = 0x04, //transmit/receive data
      PCCSERIAL_Ack = 0x0A, //request for acknowledge/acknowledge received
      PCCSERIAL_Ack2 = 0x0B //acknowledge a received message
} PCCSERIAL;

typedef enum {
      PCCSERIAL_MODE_232 = 0x00,
      PCCSERIAL_MODE_422 = 0x01,
      PCCSERIAL_MODE_485 = 0x02
} PCCSERIAL_MODE;

typedef enum {
      PCCSERIAL_SETUP_CODE_Baud = 0x00,
      PCCSERIAL_SETUP_CODE_ByteSize = 0x01,
      PCCSERIAL_SETUP_CODE_Parity = 0x02,
      PCCSERIAL_SETUP_CODE_StopBits = 0x03
} PCCSERIAL_SETUP_CODE;
```

ThreadRun()

The ThreadRun() function was developed alongside all the other functions as their need to transmit data over the backplane developed. The ThreadRun() thread in the template was only being used to buffer received data until a receive interrupt occurred. Because there were multiple functions (set up and transmit interrupt) writing to the transmit buffer and there was a chance of having to buffer data due to the difference in speed of the PCC processor and the serial port, it was decided to let the ThreadRun() thread also handle transmission of the frames from a threadsafe buffer. More specific details of the data transmission/reception are covered in the Data transmission section.

XmitInterruptHandler()

To see which functions were being called during data transmission, a program was written that created a serial object and attempted to transmit some data. By writing a program that attempted to transmit data, it was found that the upper layers called the XmitInterruptHandler() function. It was also found that this function was passed a pointer to a buffer of data and the length of the buffer. With the knowledge that the function was passed a pointer to a buffer of data and the length of that buffer, the code was written to transfer the data from the buffer fragmented in frames of a predetermined size across to the slave module until the buffer length passed to the function was equal to 0. Details of how the transmission was handled are described in the Data transmission section.

ReceiveInterruptHandler()

The ReceiveInterruptHandler() function was also found to be passed a pointer to a buffer and a buffer length value. In the case of the ReceiveInterruptHandler() function the buffer length value represented the amount of bytes that could be written into the buffer pointed to by the pointer to the buffer. Code was written in the ReceiveInterruptHandler() function to continue taking bytes of data from the receive buffer until either there was no data left, or until there was no room left in the buffer pointed to by the pointer passed to the ReceiveInterruptHandler() function. More details on reception of data are described in the Data transmission section.

6.6 Data transmission

Problems

Two major problems were encountered while developing a protocol to transmit data across the backplane to the serial port modules.

Stream vs. Frame based transmission

One of the big difficulties with creating the serial port software was that serial communication is a stream based protocol and CAN is a frame based protocol. When a serial port is transmitting data, it will keep transmitting until it is done, a frame based protocol sends data out in fragments; these fragments may also have to share the bus with fragments from other data sources so there is a chance of there being small delays between fragments. Many devices with a serial port require the specification of a size for the receive buffer and any streams that are longer will be cut off once the buffer is full. To make the PCC as flexible as possible it was decided to leave it up to the systems integrator to specify the receive buffer length at the ISaGRAF level. Performance tests will indicate the maximum length of streams that the system can handle before the backplane gets overloaded or the slave module runs out of memory. These results will be provided to the systems integrator as recommendations/guidelines. It will be at their discretion to set the maximum stream length as to the requirements of reliability, speed and compatibility.

Data buffering

Another problem which may be encountered with the PCC is that at low baud rates data will be being transmitted across the backplane to the slave module at a much faster rate than the slave module can transmit data out of the serial port, with large streams of data, the slave modules could run out of memory. Some data also needs to be retained in the memory of the CPU in case a request for retransmission is triggered due to a corrupted data frame.

Solutions

Stream vs. Frame based transmission

The serial PDD layer consists of a number of functions that can be called from higher layers and a thread designated for data reception.

When a message is transmitted the higher layers call the `XmitInterruptHandler(PUCHAR pTxBuffer, ULONG *pBuffLen)` function, this function has as arguments: a pointer to a transmit buffer containing the data to be transmitted, and a pointer to a variable containing the length of data to be transmitted. Normally this interrupt will write directly to the serial port by writing data to a register until the register is full, the interrupt will continue to be called until the value pointed to by *pBuffLen is equal to 0.

Instead of using the interrupt function to write data to the serial port, the data was written to a threadsafe linked list called TX FIFO in the following manner:

```
DWORD tx = 0;
DWORD dwDataAvaiable = *pBuffLen;
DWORD xmitStop = dwDataAvaiable;
*pBuffLen = 0;
PCC_Xmit_DataReady = TRUE;

//Pause data reception
Rx_Pause(TRUE);

while(tx < dwDataAvaiable) {
      SerialData *TxFrame = new SerialData(BufLength+1);
      if((dwDataAvaiable - tx) >= (BufLength-2)) {
            xmitStop = (BufLength-2);
            TxFrame->DataLength = xmitStop;
      }
      else {
            xmitStop = (dwDataAvaiable - tx);
            TxFrame->DataLength = xmitStop;
      }
      TxFrame->Data[0] = PCCSERIAL_Data;
      for(i = 0; i < xmitStop; i++) {
            TxFrame->Data[i+1] = *pTxBuffer;
            pTxBuffer ++;
            tx++;
      }
      SerialTxFifo.addElement(TxFrame);
}
*pBuffLen = tx;
EnableXmitInterrupt(TRUE);
```

The ReceiveInterruptHandler(PUCHAR pRxBuffer,ULONG *pBufflen) function was set up to re-assemble fragments of data from a buffer (RX FIFO) into the buffer pointed to by the pRxBuffer argument of the ReceiveInterruptHandler function.

```
DWORD dwBytesStored = 0 ;
DWORD dwRoomLeft = *pBufflen;
m_bReceivedCanceled = FALSE;

//will keep receiving until the buflen limit has been reached
while (dwRoomLeft && !m_bReceivedCanceled) {
      DWORD dwNumRxInFifo = SerialRxFifo.getNumberOfElements();
       if (dwNumRxInFifo) {
      while (dwNumRxInFifo && dwRoomLeft) {
              SerialData *RxFrame;
              RxFrame = (SerialData*)SerialRxFifo.getFirstElement();
              for(i=0; i < RxFrame->DataLength; i++) {
                     UCHAR uData = RxFrame->Data[i];
                     *pRxBuffer++ = uData;
                     dwRoomLeft--;
                     dwBytesStored++;
              }
              SerialRxFifo.removeCurrentElement();
              delete RxFrame;
              dwNumRxInFifo --;
      }
}
else
      break;
}
```

The above modifications mean that data streams will be split into fragments that can be easily handled by the backplane and incoming data fragments will be assembled into a buffer, the size of which is specified by the upper layer serial port functions. The buffering will allow the data fragments to be transmitted whenever the backplane is free.

The serial port message handling thread, ThreadRun(), was modified to handle transmission of the data in the TX FIFO buffer as well as reception of data and fragmenting the received data appropriately to fit into the RX FIFO. The flowchart below shows the basic operation of the message handling thread:

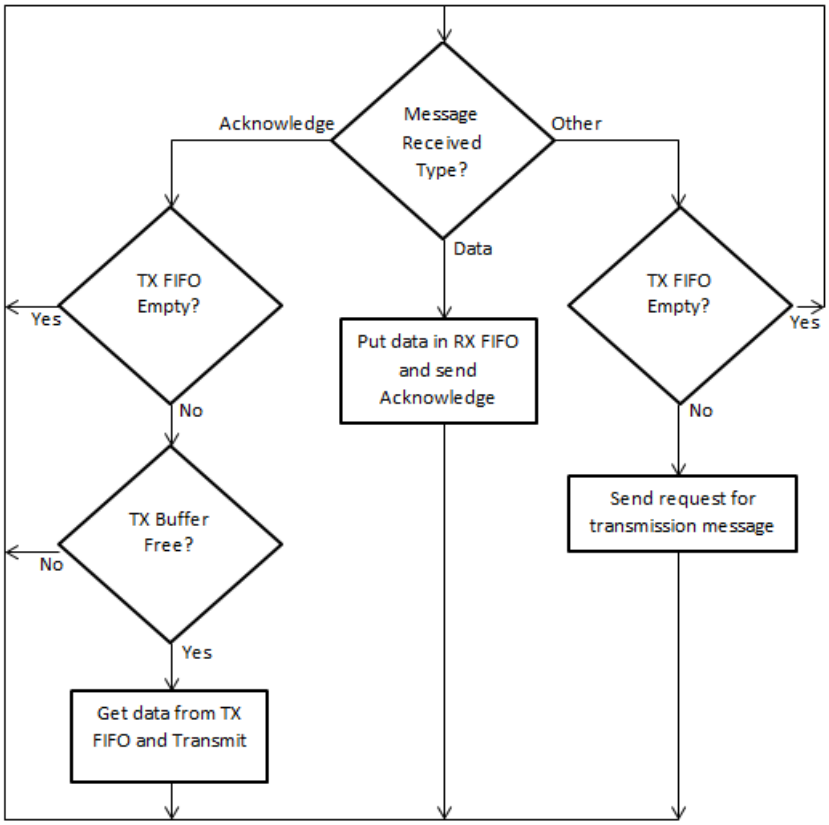


Figure 48: Serial communications flow chart

This flowchart was designed so that it could be applied to both the master and slave modules. The process was so designed to ensure received data was processed as fast as possible preventing the receiver from acting as a bottle neck but that data to be transmitted was held in the module until an acknowledgement message was received to ensure the receiving module is ready to process more data.

If there is data ready to be transmitted, but no acknowledge has been received (either the first message to be transmitted or if the receiver fails to send an acknowledge) a request for transmission message is sent, this will cause the receiver to either send an Acknowledge if it is ready to receive data or send nothing if it is not yet ready, request for transmission messages will continue to be sent out periodically until the TX FIFO is empty.

This method of flow control is based on the “Stop-and-Wait” protocol discussed in Industrial Control Technology by Peng Zhang (Zhang, 2008) according to Zhang; this method of flow control is slow but simple. Because each frame needs to be acknowledged, data transmission speed is essentially half of what it could be. This method was selected because the limited memory of the slave module could cause it to be a bottle neck, it was therefore important to have functionality to stop transmission for short periods while the slave modules process the data.

Data buffering

The serial port module uses a circular array based queue to buffer the data to be transferred out of the serial port. A circular array based queue is a vector of a certain size with pointers pointing to the beginning and end of the queue, the difference in the position of the pointers in the vector is used to indicate how full the queue is. When the pointers reach the end of the vector, they wrap around to the start. The following images demonstrate the operating principle of a circular queue:

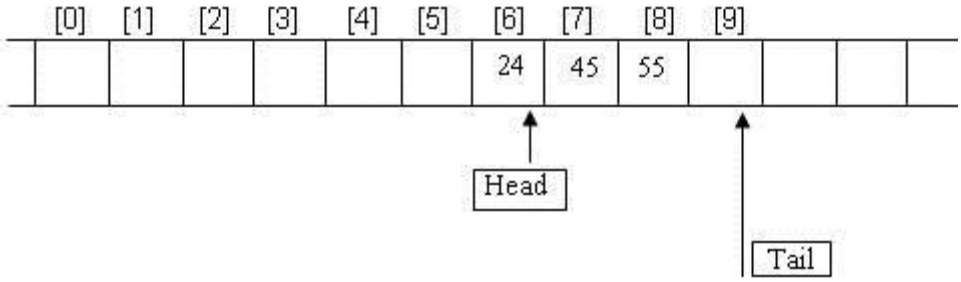


Figure 49: circular queue demonstration before wrap (HN Computing, 2008)

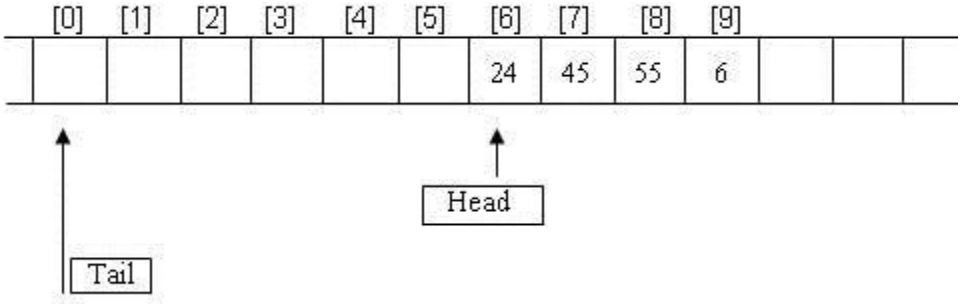


Figure 50: circular queue demonstration wrap around (HN Computing, 2008)

This means that when there is too much data being stored in the module, the data won't overflow into other parts of the memory which could contain vital data for the running of the module, but instead it will overflow back into itself, overwriting the oldest data in the buffer. To stop data from being lost, the buffer will regularly need to be checked to ensure the buffer isn't about to overflow, this sort of checking is called "checking the watermark".

The following code was written to check how close the RX and TX buffers in the slave modules are to overflowing.

```
BOOL U1TransmitFree (void)
{
	static BOOL returnValue = TRUE;
	if(U1EnQueuePtr>=U1DeQueuePtr) {
		if((U1EnQueuePtr-U1DeQueuePtr) > (U1_TX_BUFFER_SIZE*0.8) )
			returnValue = FALSE;
		if((U1EnQueuePtr-U1DeQueuePtr) < (U1_TX_BUFFER_SIZE*0.6) )
			returnValue = TRUE;
	}
	else {
		if((U1DeQueuePtr-U1EnQueuePtr) < (U1_TX_BUFFER_SIZE*0.8) )
			returnValue = FALSE;
		if((U1DeQueuePtr-U1EnQueuePtr) > (U1_TX_BUFFER_SIZE*0.6) )
			returnValue = TRUE;
	}

	return returnValue;
}
```

It can be seen that once the buffer is 80% full the high watermark is tripped and the function returns false, it will keep returning false until the low watermark is tripped when the buffer is less than 60% full. When the data rate is high enough for the watermarks to be tripped, a big difference between the watermarks such as 1% to 99% means that data can be transferred over the backplane in longer bursts but there will also be longer pauses, setting the watermarks closer together like 45% to 55% means bursts will be shorter, but pauses will also be shorter. A good feature to include in the future would be to make the watermarks adjustable so that burst times could be set to the appropriate rate to keep traffic on the backplane running smoothly, ensuring all devices perform at their best.

6.7 Slave microcontroller software development

Operating system

Unlike the main controller, the slave microcontroller only has limited functionality so it does not need to run an operating system such as Windows CE. The slave microcontroller is programmed using custom software developed by TCS for other products using the Renesas M16C microcontroller. This section describes the modifications and additions made to the existing software for the serial port module. TCS has already developed products using the M16C microcontrollers that made use of the serial ports so there was already a library of functions available for serial communications and serial port setup. The software development for the slave microcontroller therefore only needed to process the data received through the backplane and either transmit it or use it for setting up the serial port.

Transmitting and receiving data

As discussed in the “Data transmission” section, there was quite a bit more to the data transmission/reception than just transmitting frames of a fixed size across the backplane. The solution that was developed was to fragment the streams of serial data into frames of a fixed size, allowing the backplane driver to process the frames of fixed size but requiring additional software to assemble/fragment the data streams. A layer of software was developed for the slave modules that followed the same fragmentation/assembly process as in the PCC controller. This simplified software development and maximised compatibility as TCS would be able to use the product for any other serial over CAN application.

Data buffering

This part is also discussed in more detail in the “Data transmission” section. Before transmitting the data out of the serial port, the fragments of data were assembled into a circular buffer. The circular buffer method was used to prevent the slave module from running out of memory as it will never overflow, only replace existing data in the buffer.

6.8 ISaGRAF software development

Function blocks

TCS has already used the CPU5 with ISaGRAF to control devices through the serial port in industry so they already had function blocks for the serial port. Because the PCC serial port software only made changes to the PDD layer, there was no need to write any new serial port software in ISaGRAF. The figures below show the function blocks for the serial port:

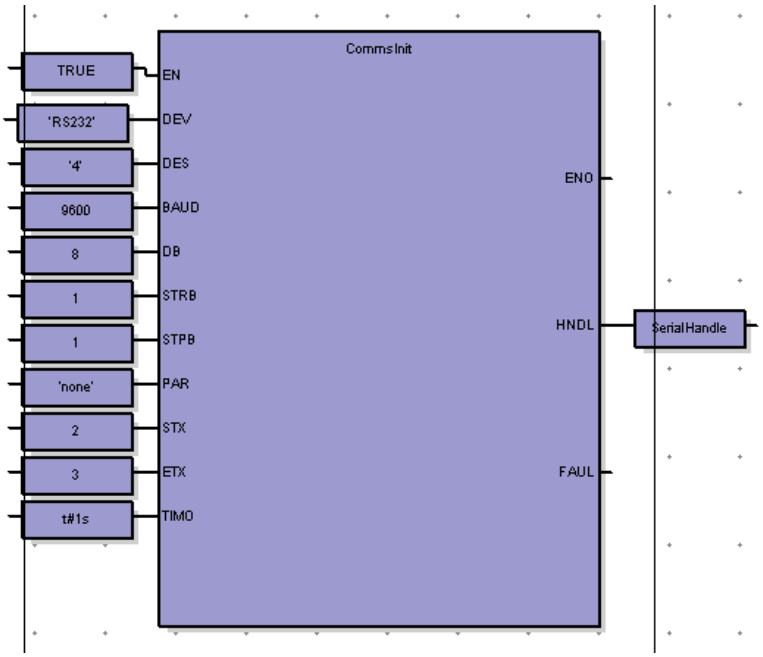


Figure 51: Serial port initialisation function block (Meek, IO_Test, 2010)

It can be seen that the serial port initialisation function block accepts a whole range of parameters for initialising the serial port, when the enable value (EN) is set to TRUE the serial port is initialised and the function block returns a handle to the serial port for use by the receive and transmit blocks.

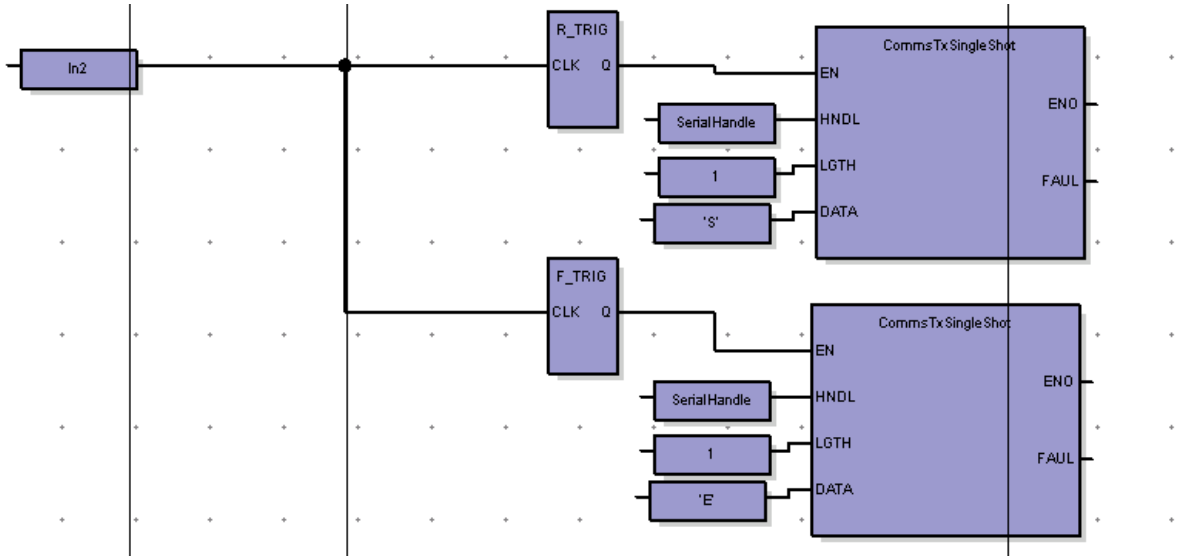


Figure 52: Serial port transmission function blocks (Meek, IO_Test, 2010)

When the EN input of the transmit function block goes high, the data at the inputs are used to transmit the parameter passed to the DATA input through the serial port. The particular program shown above will transmit the character “S” out of the serial port when In2 goes high (rising edge) and transmits the character “E” when In2 goes low (falling edge).

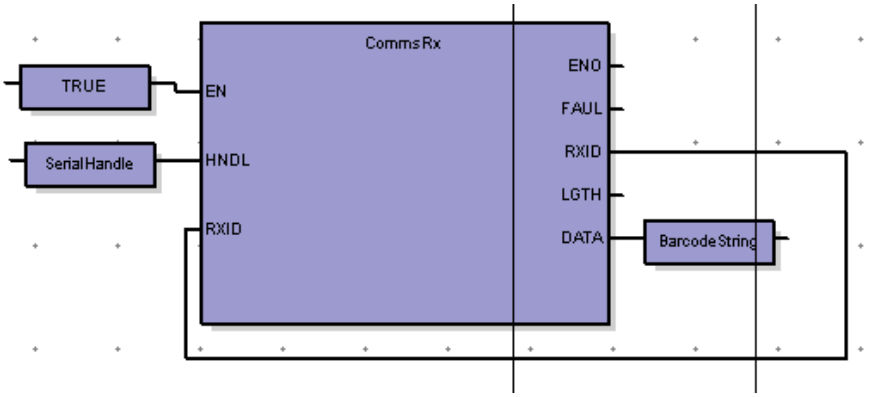


Figure 53: Serial port receive function block (Meek, IO_Test, 2010)

When the EN input is set to TRUE the receive function block will wait until the serial port pointed to by the SerialHandle (passed to HNDL input) handle receives data, this data is then written to the Barcode String parameter.

6.9 Hardware development

This section contains details of the hardware development for the PCC serial port module

Isolation

To protect the M16C microcontroller from noise and voltage spikes, the serial hardware needed to be isolated from the microcontroller. The digital I/O module used optical isolators to isolate the hardware from the microcontroller. The TLP281 optical isolators used for the digital I/O module were selected because they easily fitted the requirements for cost and speed. Their 2 micro second rise-time and 3 micro second fall-time (Toshiba, 2002) is however too slow for a protocol which supports speeds of up to 115200kbps. Another form of isolation used in TCS products is galvanic isolation. The component used by TCS is the Silicon Labs SI8441; this component has an output rise time of 3.8 nanoseconds and an output fall time of 2.8 nanoseconds (Silicon Labs, 2009) which is much more suitable for high speed data transmissions.

The difference between the galvanic isolator and the optical isolator that makes it faster more robust and more expensive is that the galvanic isolator uses a modulated RF carrier instead of light to transmit data between the isolated sides. The following is stated about the RF carrier in the SI8441 datasheet:

“This RF on/off keying scheme is superior to pulse code schemes as it provides best-in-class noise immunity, low power consumption, and better immunity to magnetic fields.” (Silicon Labs, 2009)

The following figure shows the modulation scheme of the si8441:

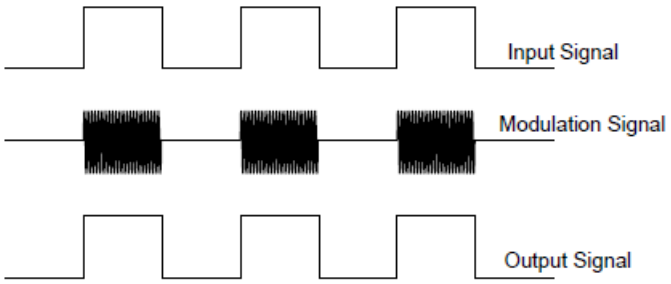


Figure 54: Modulation scheme (Silicon Labs, 2009)

Another problem faced was that the serial port hardware needed to be powered and the serial cables don’t contain power supply lines. The hardware was therefore powered by an isolated power supply.

The following schematic shows the galvanic isolators and the power supply isolator used in the serial port module hardware:

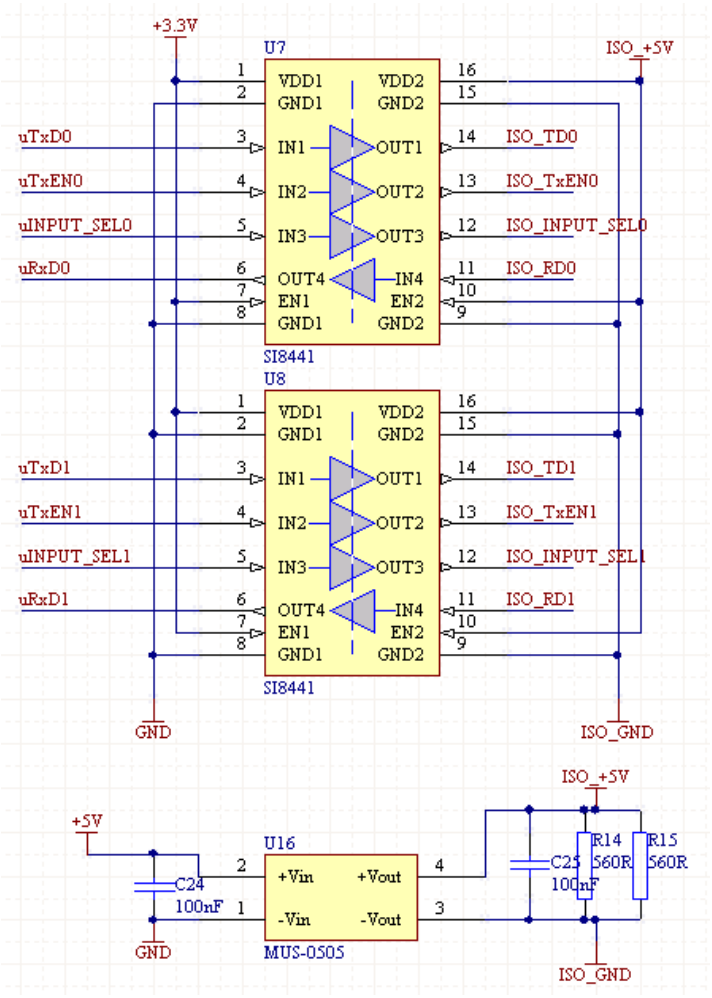


Figure 55: Galvanic isolators (top) and power supply isolator (bottom) to protect microcontroller

Transceivers

The serial port outputs on the microcontroller are only capable of switching between 0 volt and 3.3 volt; with the isolators attached they can still only switch between ISO_GND and ISO_+5V (TTL levels). EIA232/EIA422/EIA485 standard devices often use different voltage levels and EIA422/EIA485 standard devices use differential signalling. To be able to communicate with EIA232, EIA422 and EIA485 devices it is therefore necessary to use a transceiver to convert the TTL signals to the appropriate standards.

The EIA232 standard is designed to switch from voltages lower than -3V to voltages higher than +3V but no higher or lower than 12V and -12V respectively, this has been done to filter out voltage ripples between -3V and 3V making the line more robust. (Bies) The following schematic shows the MAX202 which is used to convert TTL level signals to EIA232 level signals:

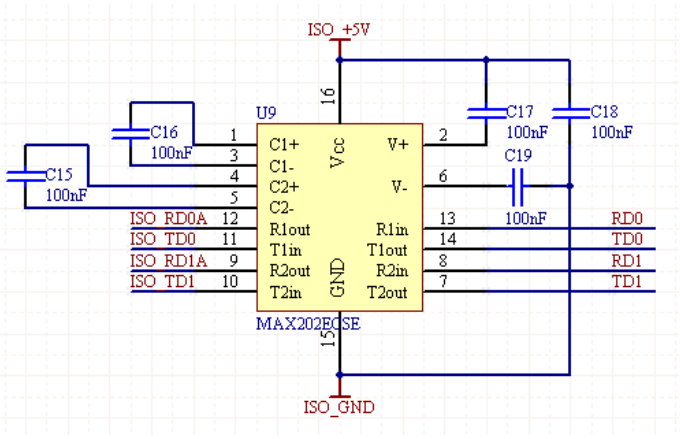


Figure 56: MAX202 used to convert TTL to EIA232

The EIA422 and EIA485 standard uses a differential pair of signals, where on a transition from 0 to 1 (in terms of TTL), the signal on one wire switches from 0V to 5V (or higher) and the other wire switches from 5V (or higher) to 0V (Bies). Because the signals are differential, it means that the transceivers measure the difference between the signals and because both signals are affected equally by interference, they are very robust against interference. The following figure shows the schematic of the EIA422/ EIA485 transceivers:

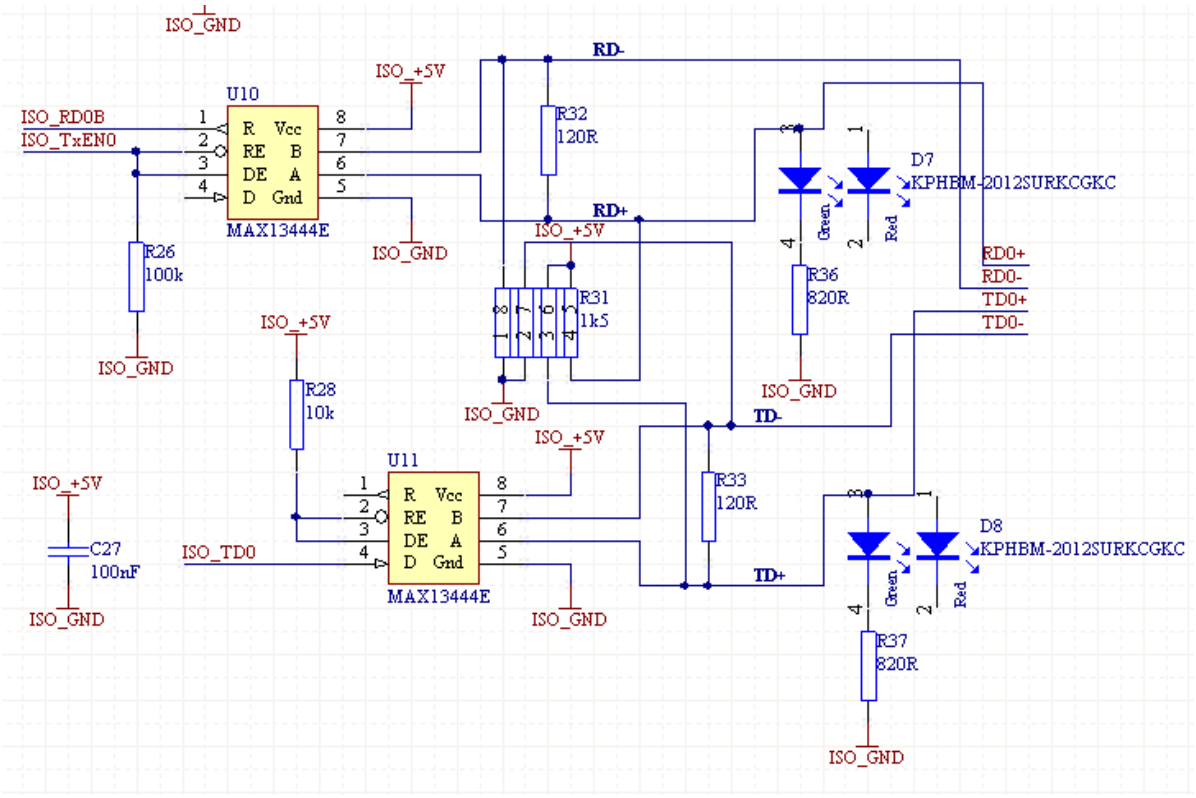


Figure 57: EIA422/ EIA485 transceiver schematic

Serial port input switching

Because the microcontroller only had three built in serial ports and the serial port module used one of those as a programming port and the serial port module was required to have two EIA232/EIA422/EIA485 ports used only in one mode at a time, it was necessary to design a way of switching the input to the serial ports on the microcontroller between the TTL side of the MAX202 transceiver and the MAX13444 transceiver. It was decided to use digital logic gates driven by a digital output on the microcontroller to select the serial protocol used. A design needed to be made with logic gates that switched between two inputs like a toggle switch.

A basic design representing a toggle switch in digital logic was designed:

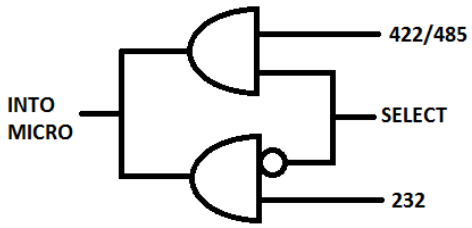


Figure 58: Basic digital logic toggle design

Because no such component as shown in the previous figure exists (or is widely available) a design needed to be made that used widely available components. To minimise the footprint and cost of the switching logic the design was simplified to use only one type of logic gate. The following figure shows the final schematic for the switching logic:

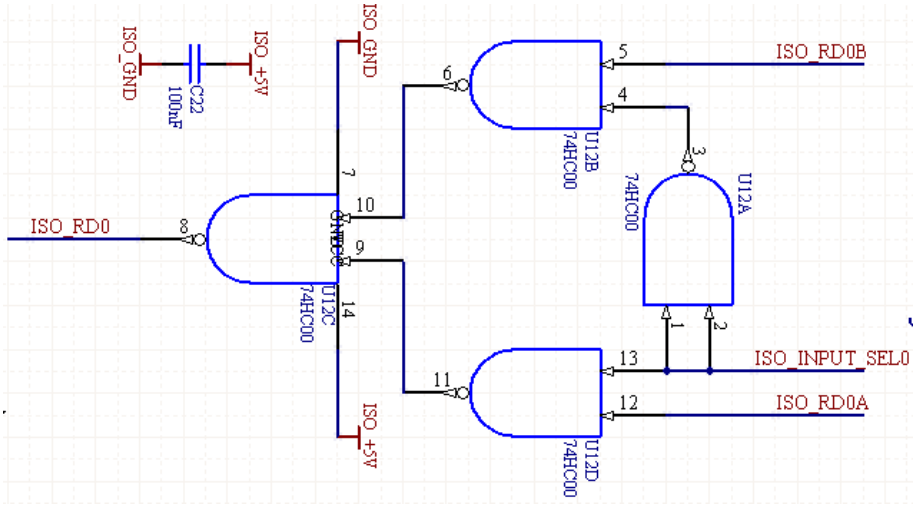


Figure 59: Switching logic for PCC serial port module

It can be seen that the design only uses four NAND gates; this means that the only component necessary is a single quad in line package, minimizing the footprint to the smallest and lowest space and monetary cost possible.

Hardware prototype

The following figure shows the prototype of the serial port hardware as it was taken to the Nuremberg trade show:

Figure 60: Serial port hardware prototype

It can be seen that the hardware uses a cage clamp connector instead of DB-9 standard connectors, this was done to save space on the module and to allow devices that have non-standard connectors to be connected to the module. The right hand side of the connector is for the 232 standard and the left hand side is for 422/485 communications. There can only be two devices attached to the serial port module at a time since the four interfaces are only wired up to two serial ports on the microcontroller.

6.10 Performance testing

Because the performance testing was carried out to test the Windows CE and slave module software, it was decided to do the bulk of the testing using the EIA-232 interface. It was decided to use the EIA-232 interface so that performance testing could be carried out using a PC without needing to rely on a 232 to 485 converter which may in itself limit performance.

Loopback test

Initial testing was carried out to test the features, such as the maximum and minimum baud rates, of the serial port. The initial test program was a loopback program which simply repeated whatever was sent into the serial port back out of the serial port. A serial terminal was used to send a file containing a list of random ASCII characters out of the PC serial port to the PCC serial port module.

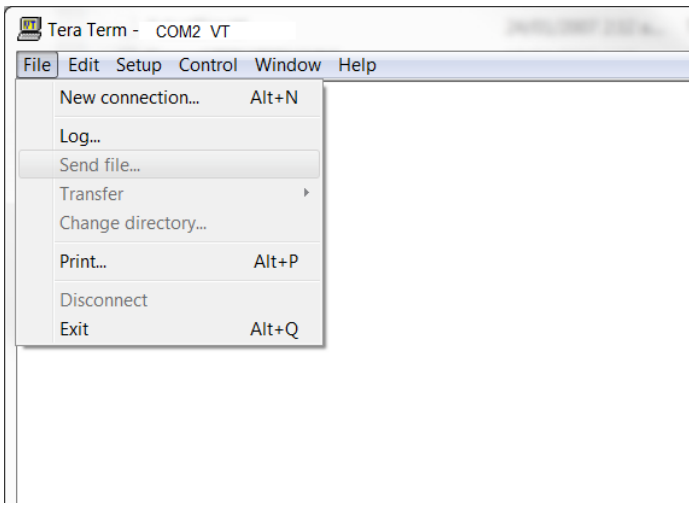


Figure 61: Serial terminal used to test the performance of the serial port module

The list of ASCII characters was sent out at a wide range of baud rates to test the performance of the serial port at each baud rate. The returned characters were then compared to the original characters sent out to give an indication of the amount of bytes lost during transmission due to factors such as buffer overflow.

During initial testing using this method, it was found that a lot of bytes were being dropped during the transmission from the PCC back to the serial port module, especially at lower baud rates. This testing resulted in the development of the watermark system discussed in the previous sections.

After applying the watermark system there were no more bytes lost during initial tests of transmission from the PCC to the serial port module. There was however still some data loss at higher baud rates. It was found that at a baud rate of 9600bps, data loss occurred after 4000 bytes of successful data transfer and at higher baud rates data loss occurred after 2000 bytes of successful data transfer. It was found that the remaining data loss was caused by the stop and wait (waiting for an acknowledgement before transmitting the next frame of data) buffering method, which was holding data back in the serial port module which was causing the transmit buffer to overflow causing data loss. The amount of data loss could possibly be reduced by using a different buffering method. After some discussion with TCS staff it was deemed unlikely that there would be a lot of devices transferring data in **bursts** of more than 2000 bytes. It may however be something that would need to be addressed in future revisions of the serial port software.

Finding it strange that at baud rates ranging from 14400 bps to 115200 bps data loss occurred after 2000 bytes for each rate, some more investigations were carried out to find out why the data loss didn't decrease as the baud rate increased and it was found that although the characters were being transmitted at the correct baud rate, there was actually an approximate 1.4ms pause between transmission of each byte. The 1.4ms delay was put down to the time taken for the serial terminal to read a character from the file and send it to the serial port. This meant that, depending on the device, the actual performance of the serial port module may be even less than what was found during these tests. As mentioned previously, this performance issue would have to be addressed in future software revisions.

Reliability over longer periods of time

This test was carried out using a program that simulated a more realistic industrial situation for the PCC serial port module to be used. Having gained an idea of the performance in terms of speed and reliability to handle longer streams of data, it was necessary to test the endurance of the software to run for longer periods of time without locking up due to a memory leak, buffer overflow or thread deadlock situation.

The software used for this test was a program designed by TCS to interface a check weigher with a serial interface to an Ethernet network so that it could transmit data to a remote PLC. The software was modified so that it no longer contained the Ethernet communication elements and interfaced with the serial port number given to the serial port module by the PCC and installed on the PCC controller.

The reason the check weigher software was used was that TCS had designed a simulator so that a PC could be used to simulate the behaviour of a check weigher. The principle of operation of the check weigher software was that it would continue sending the weight reading to the serial port at regular intervals until it receives an acknowledgement. The software would keep track of the number of messages sent and the number of messages that weren't acknowledged, making it ideal for performance testing the serial port.

The following figure shows the check weigher simulation software:

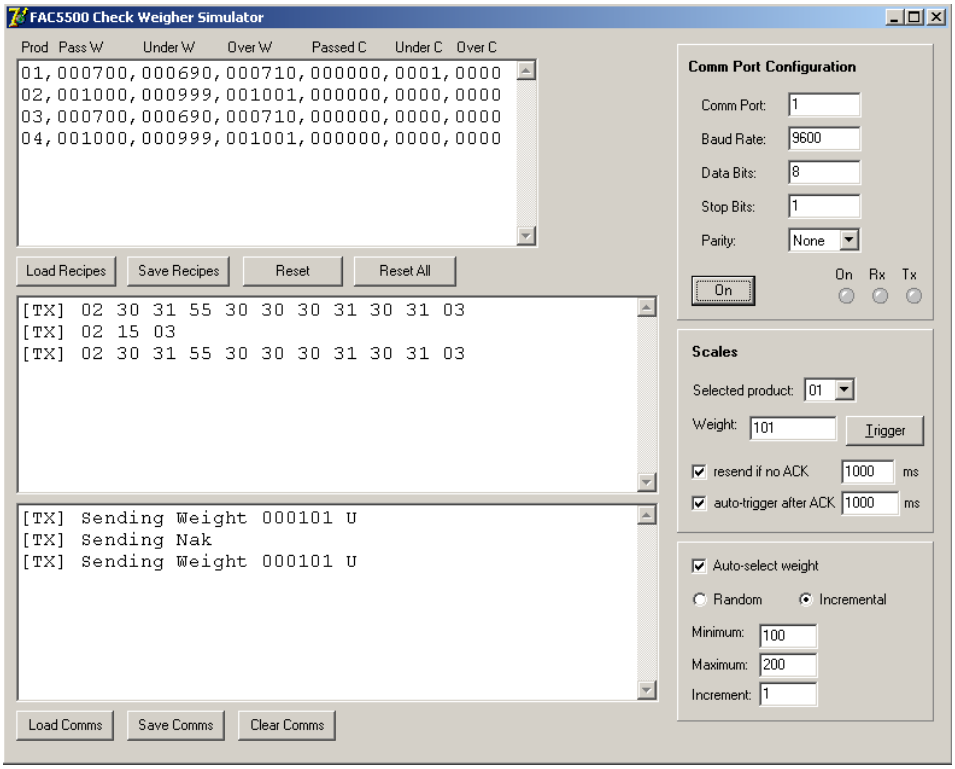


Figure 62: TCS check weigher simulation software

Some of the initial tests carried out showed that occasionally messages weren't acknowledged. This problem was fixed by adjusting the thread priorities, giving the serial port thread a higher priority and reducing less important threads such as the device management thread.

Later tests were carried out with the serial port running overnight without missing any messages.

Due to limitations of time and equipment such as a spare PC and a spare PCC with serial port module, any tests running for longer than overnight weren't carried out. It is recommended that TCS carry out longer tests before releasing the serial port module in industry.

6.11 Recommendations

The recommendations for further development include improving buffering and transmission methods, adding fault handling procedures and testing the serial port module over a longer time period.

Improving the buffering and transmission methods

Initial testing showed that there were still some issues with data being lost during large transmissions, especially at higher serial baud rates. To improve the serial port software, a review of the method with which the stream of data is being fragmented into frames would need to be done. One relatively simple solution to improve the reliability of the serial port could be to increase the buffer size in the slave module because this is where the most data is being lost when loss occurs.

Fault handling

The only fault handling currently in the module is a basic attempt to reconnect. A recommendation for future software developments would be to improve the way faults are handled with features such as messaging to indicate module status back to the main controller, serial port settings copied to the EEPROM so that if the module is powered down it doesn't revert back to its default settings.

Longer term testing

It is highly recommended that in any future developments before it is released into industry the module is put through its paces by running a high traffic serial communications application over a longer period of time. An ideal situation would be a low risk industrial application (beta release), maybe even an application inside TCS where the module can be monitored during a real-life application.

Determining maximum receive buffer size

The system will need to be tested with a device that transmits streams of data with the minimum possible pause between bytes (less than 1.4ms delay) to determine the maximum amount of bytes that can be received without losing data. The values obtained from these tests will allow TCS to set up specifications for the maximum receive buffer size at each baud rate.

6.12 Discussion

There are still quite a few issues remaining, these include: buffering, fault handling and testing, that all still need to be addressed before releasing the module. There are also several features that could be added to make the serial port more user-friendly and feature-rich. The main aim for this project however was to get a functional serial port that is compatible with the PCC and its backplane and this goal was reached.

6.13 Conclusion

A Windows CE driver, compatible with the function blocks developed by TCS was developed to control a serial port module attached to the PCC backplane. A dual serial port module that adhered to EIA232, EIA422 and EIA485 standards was developed to interface through the PCC backplane with the serial port driver. The serial port software and hardware was put through the initial testing phases with good results.

7 Ethernet port module development

7.1 Introduction

This section discusses the development of the Ethernet port module. This includes the hardware used, the concepts involved with designing a Windows CE network driver, the software development of the Windows CE driver and the software development for the slave module. Having developed the hardware and software, the module was put through various performance tests to see how well it performed compared to previously implemented Ethernet over CAN solutions.

7.2 Hardware used for testing

Because TCS had no existing hardware with similar features to the Ethernet module, testing of the Ethernet hardware could not be done until the PCC module was developed. The only testing that could be done was for the loading and unloading of a stub driver with no Ethernet functionality.

The figure below shows the module used for testing the Ethernet software:

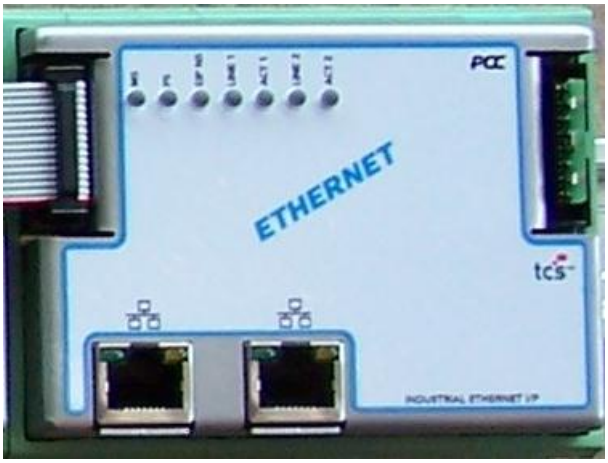


Figure 63: PCC Ethernet module

The PCC Ethernet module was developed by TCS staff because it was quite a complicated module and there was no time for developing the module in the project timeline. The module is similar in design to the modules for the digital IO and serial port module apart from the fact that there is an Ethernet controller with two industrial rated Ethernet ports connected to the Renesas microcontroller instead of just the discrete components required for the digital I/O and serial modules.

7.3 Network driver architectures

Windows CE has two different main groups of Ethernet drivers, with many sample drivers for different types of hardware in each group. The two groups are RNDIS and NDIS. NDIS stands for Network Driver Interface Specification and RNDIS stands for Remote Network Driver Interface Specification. The original specification was the NDIS specification. It was designed to increase compatibility across various platforms of Windows such as Windows CE and pc based platforms such as Windows NT.

The following three diagrams show examples of the NDIS architecture:

Figure 64 shows the most common layout for an NDIS driver, it can be seen that the NDIS layer fully encapsulates the Miniport layer (the customisable layer) because it uses standard hardware access functions.

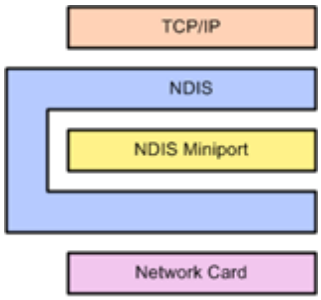


Figure 64: Most common NDIS architecture (Microsoft, 2010)

Figure 65 illustrates an NDIS structure for a USB network device. The NDIS layer is accessed by a custom Miniport driver which transfers the data over a USB bus to a remote device with custom firmware.

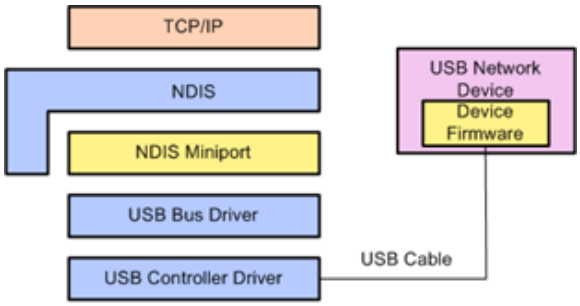


Figure 65: Serial NDIS driver structure (Microsoft, 2010)

Figure 66 shows the structure used by RNDIS. The only custom software used is that in the remote device. This structure seemed ideal for use in the PCC but with further investigation, it was found that it required the software in the module to be quite complicated. Using RNDIS would still require customisation of the Miniport driver because it would have to be interfaced with the backplane driver instead of a USB driver. Another limitation was that functionality not applicable to the PCC would have to be supported in the device firmware to make it compatible with the RNDIS driver, making the overall design of this software no easier than the using a serial NDIS interface as shown in Figure 65.

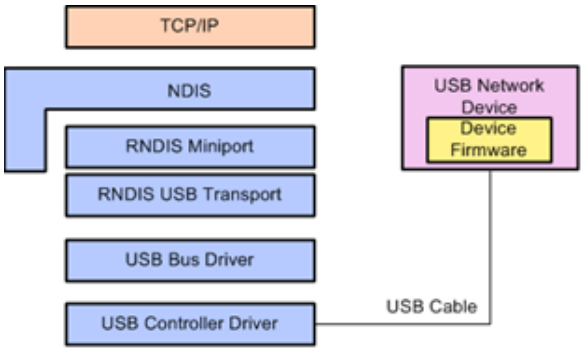


Figure 66: RNDIS Architecture (Microsoft, 2010)

The main benefit of making a device compatible with the RNDIS Architecture is that the device is compatible with all sorts of other devices supporting RNDIS. Because the PCC Ethernet module will only* be used with the PCC controller, it is however not necessary to spend the time creating a fully RNDIS compatible device.

**In the future the Ethernet module may have to be compatible with other TCS products but that is not currently a requirement for the project.*

The final network driver architecture selected for the PCC was based on the NDIS structure for a USB network device. Figure 67 shows this structure:

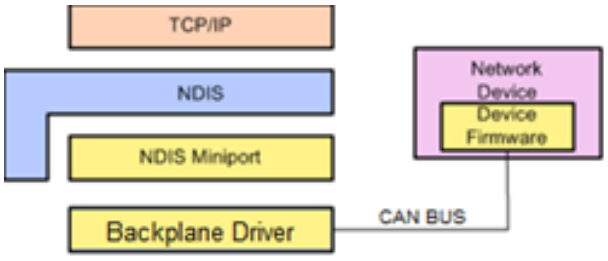


Figure 67: Network driver architecture for Ethernet driver

7.4 Ethernet driver software development

Loading and unloading the Ethernet driver

To find out how Ethernet drivers are loaded and unloaded, a stub driver (limited-functionality driver with no hardware interaction) needed to be created. This stub driver was created from a sample Ethernet driver supplied with the Windows CE Platform Builder software package for a KS8851 Ethernet controller.

Research found that the HKEY_LOCAL_MACHINE/Comm registry key contained all the Ethernet driver specific information. It was also found that the NDIS driver was loaded at boot time and enumerated the HKEY_LOCAL_MACHINE/Comm key to install all the drivers for the attached network cards.

This theory was tested by adding some entries to the registry.

The figure below shows the keys and sub-keys created in the HKEY_LOCAL_MACHINE/Comm key, these keys are HKEY_LOCAL_MACHINE/Comm/KS8842, HKEY_LOCAL_MACHINE/Comm/KS88421 and HKEY_LOCAL_MACHINE/Comm/KS8842Switch and their respective subkeys.

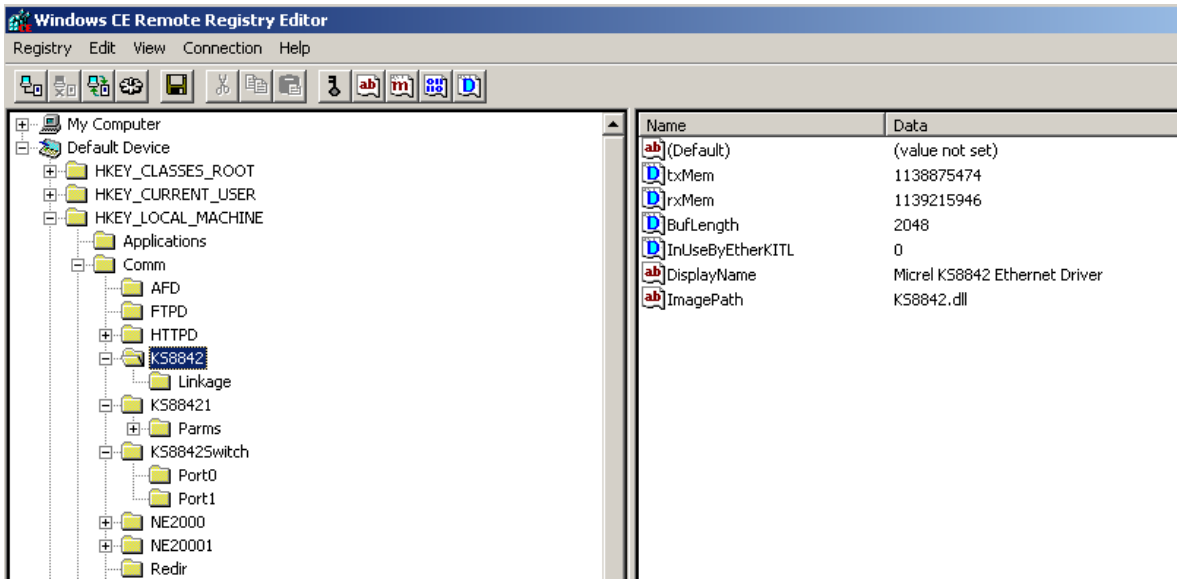


Figure 68: Ethernet related registry keys

It could be seen that the stub driver was loaded at boot time. This however was not what was required for the PCC, some more research needed to be carried out into the dynamic loading and unloading of the NDIS Miniport driver.

One solution that was found for adding/removing NDIS Miniport drivers at run time was the method used by PCI bus drivers. This involved creating a registry key similar to that below:

```
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\PCI\Template\MYNIC]
"Class"=dword:02
"SubClass"=dword:00
"ProgIF"=dword:0
"VendorID"=multi_sz:"1234","5678"
"DeviceID"=multi_sz:"1111","2222"
"Dll"="NDIS.dll"
"Entry"="NdisPCIBusDeviceInit"
```

(Microsoft, 2003)

Attempting to install the DLL NDIS.dll causes the NDIS driver to re-enumerate the HKEY_LOCAL_MACHINE/Comm key and load any newly added Miniport drivers. This was tested in a debug build.

When a Windows CE debug build is loaded onto a platform, it is possible to read debug messages sent to the debugger application (platform builder) through the Ethernet connection between the target device and PC. The debug messages produced by the application that loads the Miniport driver (NDIS.dll) are shown below:

```
4294868576 PID:3f74dba TID:c3e79642 0x83e79400: NDIS:ndisAddPCMCIADevice INdisOpenDeviceKey failed

4294868577 PID:3f74dba TID:c3e79642 0x83e79400: NDIS:: CE_AddBusFriendlyName() [successfully created]
[HKLM\Comm\BusFriendlyNames\CAN_0_1_0\001]

4294868580 PID:3f74dba TID:c3e79642 0x83e79400: NDIS:: Failed GetBusNamePrefix() on [Drivers\Active\02].
Set it to "???"

4294868581 PID:3f74dba TID:c3e79642 0x83e79400: NDIS:: ndisAddBusAgnosticDeviceFromNdisInit(): Error
no "InterfaceType" in ActivePath[Drivers\Active\02]

4294868581 PID:3f74dba TID:c3e79642 0x83e79400: DEVICE!LaunchDevice: Init() failed for device
0x0005ac50

4294868582 PID:3f74dba TID:c3e79642 0x83e79400: DEVICE!I_ActivateDeviceEx: couldn't activate: prefix
NDS, index 1, dll NDIS.dll, context 0x630edf4

4294868583 PID:3f74dba TID:c3e79642 0x83e79400: Driver installation failed for dev1!
```

Looking at the debug output, it can be seen that there are quite a few errors occurring in functions related to bus drivers such as: “ndisAddBusAgnosticDeviceFromNdisInit()”. Quite a bit of time was spent attempting to work around these errors but eventually it was decided to temporarily put this method of loading NDIS miniport drivers aside and search for alternative methods.

Further research found that, as with any stream interface driver*, the NDIS driver can also be accessed and manipulated using IOControl commands. Some of these IOControl commands included:

- IOCTL_NDIS_BIND_ADAPTER
- IOCTL_NDIS_REGISTER_ADAPTER
- IOCTL_NDIS_DEREGISTER_ADAPTER
- IOCTL_NDIS_LOAD_MINIPORT
- Etc…

* Details of the stream interface driver functions are discussed in the Digital I/O software development section.

This suggested that there was a possibility for the backplane driver to tell the NDIS driver to load and unload Miniport drivers at runtime. Because the NDIS driver has stream interface functions, all that was needed to send it IOControl commands was to call a CreateFile to obtain a handle to the NDIS driver and then to do the IOControls to load and unload the Miniport driver. Some of the code used to load and unload the NDIS driver can be seen below:

The following code was put in the device management software to initialise the interrupt and call the code that loads the Ethernet driver:

```
if( ( BP_DeviceList[bpi_MacID].devtype == BP_DEVTYPE_ETH ) && bpi_new ) {
      DWORD RetVal;
      int count = 2;
      AdapterRegisterCommand( count, &arguments );
      bpi_new = 0;
}
```

The following code block shows the code for the AdapterRegisterCommand function:

```
void
AdapterRegisterCommand(
      int argc,
      TCHAR *argv[])
{
      argv[1] = *argv+0x07;
      if (g_bVerbose)
            Print(TEXT("Registering miniport %s, adapter %s"), argv[0],
argv[1]);

      DoNdisMultiSzIOControl(IOCTL_NDIS_REGISTER_ADAPTER, argv[0], argv[1],
NULL);
}
```

The AdapterRegisterCommand function calls the DoNdisMultiSzIOControl function; this function can be seen in the following code block:

```
BOOL
DoNdisIOControl(
      DWORD dwCommand,
      LPVOID      pInBuffer,
      DWORD cbInBuffer,
      LPVOID      pOutBuffer,
      DWORD *pcbOutBuffer     OPTIONAL)
//
//    Execute an NDIS IO control operation.
//
{
      HANDLE      hNdis;
      BOOL  bResult = FALSE;
      DWORD cbOutBuffer;

      hNdis = CreateFile(DD_NDIS_DEVICE_NAME, GENERIC_READ | GENERIC_WRITE,
                        FILE_SHARE_READ | FILE_SHARE_WRITE,
                        NULL, OPEN_ALWAYS, 0, NULL);

      if (hNdis != INVALID_HANDLE_VALUE)
      {
            cbOutBuffer = 0;
            if (pcbOutBuffer)
                  cbOutBuffer = *pcbOutBuffer;

            bResult = DeviceIoControl(hNdis,
                                                dwCommand,
                                                pInBuffer,
                                                cbInBuffer,
                                                pOutBuffer,
                                                cbOutBuffer,
                                                &cbOutBuffer,
                                                NULL);

            if (g_bVerbose || bResult == FALSE)
                  Print(TEXT("IoControl result=%d"), bResult);

            if (pcbOutBuffer)
                  *pcbOutBuffer = cbOutBuffer;

            CloseHandle(hNdis);
      }
      else
      {
            Print(TEXT("CreateFile of '%s' failed, error=%d"),
DD_NDIS_DEVICE_NAME, GetLastError());
      }


      return bResult;
}
```

The DoNdisMultiSzIOControl function first creates a handle to the NDIS driver using the standard CreateFile function (which is used to create a handle to a stream interface driver). Once a handle to the NDIS driver is created, the function uses the DeviceIoControl function (also a standard stream interface function) to register the adapter with the NDIS driver. After this the NDIS driver takes over and attaches the miniport driver to its interface.

Initialising the shared memory

Once the miniport driver is loaded, the first thing that needs to be done is to establish a link with the backplane driver, this is done by setting up the shared memory and the shared interrupts. After some investigation it was found that the MiniportInitialise function was the first function to be called by the NDIS driver upon initialisation. The following code shows how the shared memory is initialised:

```
NDIS_STATUS MiniportInitialize (
    OUT PNDIS_STATUS pnsOpenErrorStatus,
    OUT PUINT        puiSelectedMediumIndex,
    IN  PNDIS_MEDIUM pnmMediumArray,
    IN  UINT         uiMediumArraySize,
    IN  NDIS_HANDLE  hAdapter,
    IN  NDIS_HANDLE  hConfiguration )
{
    PNDIS_ADAPTER pAdapter;
    NDIS_STATUS   nsStatus;
    UINT          i;
    UCHAR         bInterrupt = FALSE;

    RETAILMSG(TRUE, (TEXT("Micrel : KS884X ISA MiniportInitialize \r\n")));

    // Allocate memory for the adapter block now.
    nsStatus = NdisAllocateMemory(( PVOID* ) &pAdapter, sizeof(
NDIS_ADAPTER ),
        0, HighestAcceptableMax );
    if ( nsStatus != NDIS_STATUS_SUCCESS )
    {
        return( nsStatus );
    }

    // Clear out the adapter block, which sets all default values to FALSE
    // or NULL.
    NdisZeroMemory( pAdapter, sizeof( NDIS_ADAPTER ));

    //Set PCC variables
    backplane_interface_init(&pAdapter->m_Hardware);
```

The NdisAllocateMemory and NdisZeroMemory functions initialise the memory for the miniport driver in the NDIS (upper) layer. The backplane_interface_init function initialises the memory and interrupts shared with the backplane driver and is shown in the code block below:

```
//Called by MiniportInitialize () function in NdisDriver.c
void backplane_interface_init(PHARDWARE phwi) {
      TCHAR name[6];
      TCHAR event_name[8];

      phwi->ethMacID = ethMacID;
      phwi->eth_BufLength = eth_BufLength;
      phwi->eth_rx_FileMap = eth_rx_FileMap;
      phwi->eth_tx_FileMap = eth_tx_FileMap;
      phwi->eth_TXcomplete = 1;

      //set up link to shared memory
      //------------------------
      if(phwi->eth_rx_FileMap) {
            phwi->eth_rx_FileMemory = (unsigned int*)MapViewOfFile(phwi-
>eth_rx_FileMap,
                  FILE_MAP_ALL_ACCESS, 0,0,0 );
            if(phwi->eth_rx_FileMemory ) {
                  memset( phwi->eth_rx_FileMemory, 0x0, (phwi-
>eth_BufLength+3)*sizeof(unsigned int));
            }
      }

      if(phwi->eth_tx_FileMap) {
            phwi->eth_tx_FileMemory = (unsigned int*)MapViewOfFile(phwi-
>eth_tx_FileMap,
                  FILE_MAP_ALL_ACCESS, 0,0,0 );
            if( phwi->eth_tx_FileMemory ) {
                  memset( phwi->eth_tx_FileMemory, 0x0, (phwi-
>eth_BufLength+3)*sizeof(unsigned int));
            }
      }
      //------------------------

      phwi->eth_TXEventHandle = CreateEvent(NULL, FALSE, FALSE,
L"txEvent");

      phwi->eth_DataEvent = CreateEvent(NULL, FALSE, FALSE, L"dataEvent");

      wcscpy(event_name,(LPTSTR)TEXT("RX"));
      _itow(ethMacID,name,10);
      wcsncat(event_name,(LPTSTR)name,sizeof(name));
      phwi->eth_RXEventHandle = CreateEvent(NULL, FALSE, FALSE,
event_name);
      pccSetRxInt( phwi );
      active = 0x01;
}
```

Establishing communication with the slave modules

After developing the code to unload and load a stub driver it was time to create a driver which could communicate with the actual Ethernet hardware. Further development was carried out using a driver supplied by the manufacturer of the Micrel KSZ8842 Ethernet controller.

Writing to CAN instead of writing to registers

Looking into the driver supplied by the hardware manufacturer, it was found that the driver had functions dedicated to interfacing the driver software with the physical registers. These functions were modified to send data over the Backplane instead of writing data to a register.

It is worth noting that the code written to do this contained little to no error checking so there were a lot of crashes, these errors were not dealt to because it was decided to use an alternative method of communicating with the Ethernet port.

The code block below shows some of the originally defined functions used to read and write data to/from the Ethernet controller registers:

```
#define HW_WRITE_WORD( phwi, addr, data )\
{\
MIO_WORD(( phwi )->m_ulVIoAddr + (addr<<ADDR_SHIFT))=(USHORT)data;\
}

#define HW_READ_WORD( phwi, addr, data ) \
{\
      *( data ) = MIO_WORD(( phwi )->m_ulVIoAddr + (addr<<ADDR_SHIFT));\
}
```

These function definitions were modified to contain functions that accessed the Ethernet controller through the backplane:

```
#define HW_WRITE_WORD( phwi, addr, data )\
{\
      writeWord(phwi, addr, data);                    \
}

#define HW_READ_WORD( phwi, addr, data ) \
{                                                     \
      *data = (USHORT)readWord(phwi, addr);                 \
}
```

To interface the above functions with the backplane, the functions below were written:

```
unsigned int readWord(PHARDWARE phwi, unsigned int addr) {
      int counter;
      unsigned int retWord = 0x00;
      unsigned int retry_counter = 0x00;
      counter = 0;
      if( ( phwi->eth_tx_FileMemory ) && ( phwi->eth_rx_FileMemory ) &&
active ) {
            phwi->eth_tx_FileMemory[1] = ETHMSG_READ_WORD;
            phwi->eth_tx_FileMemory[2] = addr>>8;
            phwi->eth_tx_FileMemory[3] = addr;
            phwi->eth_tx_FileMemory[phwi->eth_BufLength+2] = 4;
            phwi->eth_tx_FileMemory[0] = 1;
            SetEvent(phwi->eth_TXEventHandle);
            Sleep(25);//wait for a response
            retWord = phwi->eth_rx_FileMemory[2] << 8;
            retWord |= phwi->eth_rx_FileMemory[3];
            phwi->eth_rx_FileMemory[0] = 0;
      }
      return retWord;
}

void writeWord(PHARDWARE phwi, unsigned int addr, unsigned int data) {
      int counter;
      unsigned int retry_counter = 0x00;
      counter = 0;
      if( ( phwi->eth_tx_FileMemory ) && ( phwi->eth_rx_FileMemory ) &&
active ) {
            phwi->eth_tx_FileMemory[1] = ETHMSG_WRITE_WORD;
            phwi->eth_tx_FileMemory[2] = addr>>8;
            phwi->eth_tx_FileMemory[3] = addr;
            phwi->eth_tx_FileMemory[4] = data>>8;
            phwi->eth_tx_FileMemory[5] = data;
            phwi->eth_tx_FileMemory[phwi->eth_BufLength+2] = 6;
            phwi->eth_tx_FileMemory[0] = 1;
            SetEvent(phwi->eth_TXEventHandle);
            phwi->eth_rx_FileMemory[0] = 0;
      }
}
```

Because the existing function definitions were changed, all the code referring to those functions for hardware access now called the backplane interface functions. Using this method of communicating with the Ethernet controller over CAN worked but was too slow and cumbersome to be used to control the Ethernet port.

A few of the messages sent between the PCC and Ethernet module were recorded using the CAN analyser, these are shown with explanation of what they mean below:

> 03 000e 0033
Select Bank 51
< 03
Success
> 02 0004
Offset 4 : Port 2 Status Register
< 02 0000
Byte read “0x00”:
Auto MDI-X mode
Polarity not reversed
Receive flow control not enabled
Transmit flow control inactive
10mbps link speed
Half duplex
MDI-X state
AN not done
Link not good
Link partner not flow control capable
Link partner not 100bt full duplex capable
Link partner not 100bt half-duplex capable
Link partner not 10bt full duplex capable
Link partner not 10bt half-duplex capable

> 03 000e 002a
Select Bank 42
< 03
Success
> 04 0000
Offset 0: Indirect access control register
< 04 0000
Word read “0x00”:
Write cycle
Static MAC address table selected
Indirect Access = 0x000

> 03 000e 0012
Select Bank 18
< 03
Success
> 04 0002
Offset 2: Interrupt status register
< 04 ab00
Word read “0xAB00”:
Link change interrupt state
Receive interrupt state
Receive overrun interrupt state
Transmit process stopped
Receive process stopped

> 03 000e 0031
Select Bank 49
< 03
Success
> 02 0002
Offset 2: Port1 control register
< 02 009f
Byte read “0x9F”:
Advertise 10bt half-duplex capability
Advertise 10bt full-duplex capability
Advertise 100bt half-duplex capability
Advertise 100bt full-duplex capability
Advertise flow control capability
Enable auto negotiation

> 03 000e 0031
Select Bank 49
< 03
Success
> 02 0005
Offset 5: Port1 status register
< 02 009e
Byte read “0x9E”:
HP Auto MDI-X mode
Receive flow control enable
Transmit flow control enable
100mbps link speed
Full duplex operation

> 03 000e 0031
Select Bank 49
< 03
Success
> 02 0004
Offset 4: Port1 status register
< 02 007f
Byte read “0x7F”:
Link partner 10bt half-duplex capable
Link partner 10bt full-duplex capable
Link partner 100bt half-duplex capable
Link partner 100bt full-duplex capable
Link partner flow control capable
Link good
AN done
MDI-X

> 03 000e 0033
Select Bank 51
< 03
Success

> 02 0002
Offset 2: Port 2 Control Register 4
< 02 009f
Byte read “0x9F”:
Advertise 10bt half-duplex capability
Advertise 10bt full-duplex capability
Advertise 100bt half-duplex capability
Advertise 100bt full-duplex capability
Advertise flow control capability
Enable auto negotiation

> 03 000e 0033
Select Bank 51
< 03
Success
> 02 0005
Offset 5: Port2 Status register
< 02 00A0
Byte read “0xA0”:
HP Auto MDI-X mode
Polarity is reversed
Flow control is inactive
10mbps link speed
half duplex operation

> 03 000e 0033

A better method of communicating with the Ethernet controller would involve carrying out the following steps:

1. Find out what initialisation can be done by the slave module without requiring dynamic information from the main controller.
2. Find out which miniport functions carry out dynamic initialisation and settings changes (such as setting the multicast table) and develop a code for that so that data could be sent over in a block rather than sending a can message for each registry access.
3. Write code to receive a full Ethernet frame in the slave module and then transmit it in fragments to the PCC
4. Write code to transmit a full Ethernet frame to the slave module and write code in the slave module to populate the frame into the appropriate registers to transmit it over Ethernet.

The following sections discuss how these steps were carried out to create the Ethernet driver.

Triggering the ISR

From what was found during initial tests with writing to the Ethernet controller registers over the backplane, it was established that the Ethernet driver polled the receive, transmit and link state interrupt statuses regularly. This however was not fast enough to create a reliable Ethernet driver. Looking into the sample driver provided by the Ethernet controller manufacturer it was found that the driver used the MiniportISR function to process interrupts. It was quite difficult to find out how to trigger the MiniportISR because it was called by a function in a higher NDIS layer which could not be accessed. Instead the following block of code was found in the MiniportInitialize function:

```
nsStatus = NdisMRegisterInterrupt( &pAdapter->m_Interrupt,
        pAdapter->m_hAdapter, pAdapter->m_ulInterruptNumber,
        pAdapter->m_ulInterruptNumber, FALSE, FALSE,
        NdisInterruptLatched );
```

Because the MinportISR function was found to be the only interrupt processing function, it was assumed that the NdisMRegisterInterrupt function was used to register the interrupt that triggered the MiniportISR function via the upper layer functions. It was found that the third parameter (pAdapter->m_ulInterruptNumber) referred to the ‘bus relative interrupt vector’ (Microsoft), in other words, the memory address linked to the interrupt line of the bus to which the device was attached. After looking into how bus drivers such as PCI bus register interrupts for Ethernet cards it was found that the bus driver creates a value called SysIntr in the HKEY_LOCAL_MACHINE/Comm/KS88421/Parms key to represent the system interrupt vector of the Ethernet card. Upon installation of the Ethernet driver this value is registered with the Ethernet driver. The following figure shows the HKEY_LOCAL_MACHINE/Comm/KS88421/Parms key:

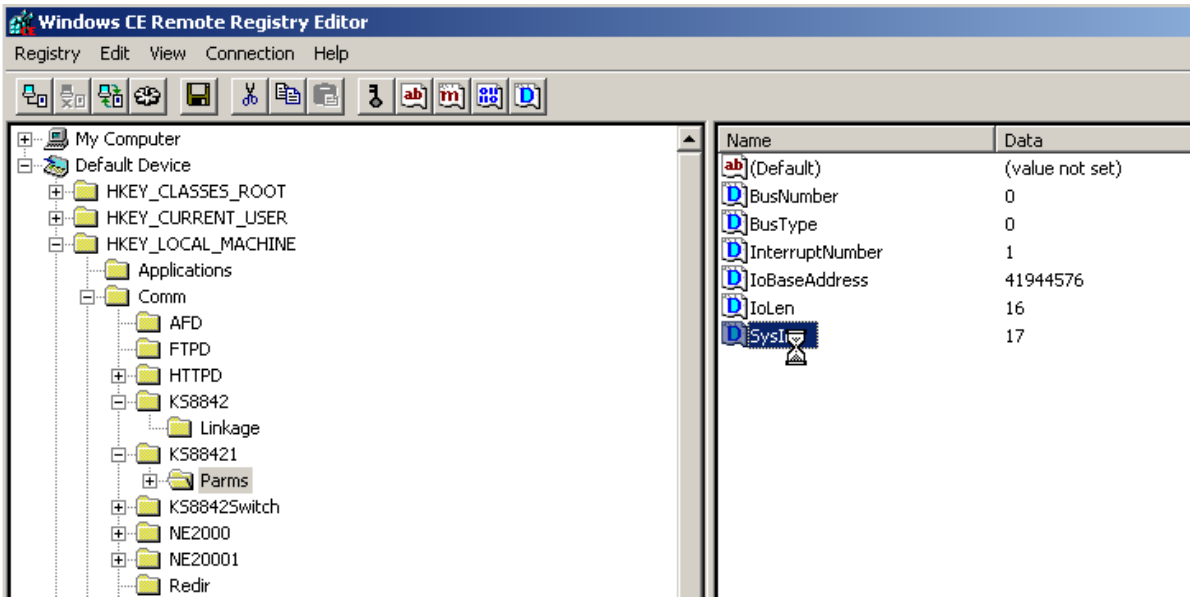


Figure 69: HKEY_LOCAL_MACHINE/Comm/KS88421/Parms registry key

The only problem with using system interrupt vectors is that according to the Microsoft documentation these can only be triggered by physical interrupt lines in the hardware. Because the backplane driver is supposed to be able to trigger the interrupt it was necessary to find a way to trigger a system interrupt from an application.

Further research in the Windows CE documentation found a function called SetInterruptEvent which allows a device driver to cause an artificial interrupt event (Microsoft). This function could be used to artificially trigger a system interrupt. The description of the function in the MSDN library contained a warning that the interrupt should only be used by the PowerOn and PowerOff related functions, this warning was however deemed to be aimed more towards developers of software for third party devices where it was not known which system interrupts were assigned to which device and devices where the system interrupts were actually able to be triggered by hardware.

Looking into system interrupts; it was found that Windows CE had defined a range of system interrupt values for peripheral devices, the unused value SYSINTR_FIRMWARE + 1 was picked as the system interrupt to be used for the Ethernet driver. When the driver is modified to be able to support multiple instances, a function will have to be developed to search for the first free system interrupt, but at the moment TCS only requires one Ethernet card to be attached to each PCC.

The code:

```
SetInterruptEvent(SYSINTR_FIRMWARE + 1);
```

Was put in the backplane driver to trigger the system interrupt upon completion of receiving an Ethernet frame, this proved to successfully trigger the MiniportISR function.

Process received frames

Every time the slave module sends a message across the backplane, this message is transferred from the backplane to the shared memory, since the shared memory is only big enough for one frame, this frame will be overwritten every time the next frame comes in. It is therefore necessary that the driver has a way of recognising the arrival of a frame and then instantly processing it. The serial port driver uses the ThreadRun function to do this (see serial port driver development section), this function is running continuously in a thread separate to the serial port driver. The ThreadRun thread is blocked waiting for an interrupt that is triggered when a frame arrives in the shared memory. When a frame comes into the shared memory the thread is released to process the frame. It was decided to use the MiniportISR function in a similar way to process received frames.

The following code block shows the MiniportISR function:

```
VOID MiniportISR (
    OUT PBOOLEAN pbInterruptRecognized,
    OUT PBOOLEAN pbQueueDpc,
    IN  PVOID    pContext )
{
    PNDIS_ADAPTER pAdapter = ( PNDIS_ADAPTER ) pContext;
    PHARDWARE     pHardware = &pAdapter->m_Hardware;
    UINT    wIntStatus;

    printf("ethernet interrupt");

    //future versions of this driver might have functionality to
    // determine whether the interrupt needs to be processed further
    wIntStatus = 0x01;

    if ( wIntStatus )
    {
        *pbInterruptRecognized = TRUE;
        *pbQueueDpc = TRUE;
    }
    else
    {
        // not our interrupt!
        *pbInterruptRecognized = FALSE;
        *pbQueueDpc = FALSE;
    }
}
```

According to MSDN documentation the pbInterruptRecognised parameter needs to be set to TRUE if the interrupt is recognised as a valid interrupt related to this driver and the pbQueueDpc needs to be set to TRUE if the MiniportHandleInterrupt function needs to be called to complete the interrupt-driven operation (Microsoft).

The following block of code shows the MiniportHandleInterrupt function:

```
VOID MiniportHandleInterrupt (
    IN  NDIS_HANDLE hAdapterContext )
{
    PNDIS_ADAPTER pAdapter = ( PNDIS_ADAPTER ) hAdapterContext;
    PHARDWARE     pHardware = &pAdapter->m_Hardware;
    BOOLEAN       bReceiveDone = FALSE;
    int           port = MAIN_PORT;

#ifdef DEBUG_DRIVER_INTERRUPT
DbgPrint( "MiniportHandleInterrupt"NEWLINE );
#endif

    if ( ( msgType( pHardware ) == ETH_TXDONE ) )
    {
      pHardware->eth_TXcomplete = 1;

      // Acknowledge the interrupt
      freeSharedMem(pHardware);

#ifdef SEND_QUEUE
      SendNextPacket( pAdapter );
#endif
    }

    if ( ( msgType( pHardware ) == ETH_DATA1 ) || ( msgType( pHardware ) ==
ETH_DATA2 ) )
    {
pHardware->eth_TXcomplete = 1;

      NdisAcquireSpinLock( &pAdapter->m_lockAdapter );

      bReceiveDone |= ProcessReceive( pAdapter );

      NdisReleaseSpinLock( &pAdapter->m_lockAdapter );

#ifdef DEBUG_COUNTER
      pHardware->m_nGood[ COUNT_GOOD_INT_RX ]++;
#endif
    }

    if ( msgType( pHardware ) == ETH_LSC )
    {
      pHardware->eth_TXcomplete = 1;

      pHardware->m_bLinkIntWorking = TRUE;
      populateLinkStatus( pHardware );
      SwitchGetLinkStatus( pHardware );
      if ( pAdapter->m_ulNdisMediaState !=
            pHardware->m_ulHardwareState )
      {
           NDIS_STATUS nsStatus;
           nsStatus = ( NdisMediaStateConnected ==
            pHardware->m_ulHardwareState ) ?
            NDIS_STATUS_MEDIA_CONNECT : NDIS_STATUS_MEDIA_DISCONNECT;

           NdisMIndicateStatus( pAdapter->m_hAdapter, nsStatus, NULL, 0 );
           NdisMIndicateStatusComplete( pAdapter->m_hAdapter );
           pAdapter->m_ulNdisMediaState = pHardware->m_ulHardwareState;
      }
```

```
      if ( ( pAdapter->m_ulDriverState & DRIVER_STATE_RESET )  &&
          NdisMediaStateConnected == pAdapter->m_ulNdisMediaState )
      {
          pAdapter->m_ulDriverState &= ~DRIVER_STATE_RESET;
          NdisMResetComplete( pAdapter->m_hAdapter, NDIS_STATUS_SUCCESS,
          FALSE );
      }
      // Acknowledge the interrupt
      freeSharedMem(pHardware);
   }

   // Finally, indicate ReceiveComplete to all protocols which received
   // packets.
   if ( bReceiveDone )
   {
      // only allow received packets after Filter is defined.
      if ( pAdapter->m_bPacketFilterSet )
      {
          NdisMEthIndicateReceiveComplete( pAdapter->m_hAdapter );
      }
   }
}  // MiniportHandleInterrupt
```

This function was modified to call the msgType function which checks what kind of message it is, examples of message types are listed below:

- Data frame
- Link status change message
- Transmission complete message

The block below shows a list of message types defined for communication between the Miniport driver and the Ethernet module:

```
typedef enum {
      ETH_RESERVED            = 0x00,
      ETH_DATA1               = 0x01,
      ETH_DATA2               = 0x02,
      ETH_LSC                 = 0x03,
      ETH_TX                  = 0x04,
      ETH_TXDONE              = 0x05,
      ETH_SETMULTI            = 0x06,
      ETH_RX_INT              = 0x07,
      ETH_TX_INT              = 0x08,
      ETH_DIS_INT             = 0x09
} ETH;
```

The message types are put in the first byte of the frame sent across the backplane, for data messages these frames can be up to 2048 bytes long.

When a data frame comes in, the ProcessReceive function is called to process the received frame of data and pass it to the upper layer. The original ProcessReceive function supplied by the driver manufacturer contained quite complicated code to read the received data from the registers in the Ethernet controller and pass it to the upper layers, this was replaced with a function to read the data from the shared memory. The block of code shows the ProcessReceive function.

```
BOOLEAN ProcessReceive (
    IN  PNDIS_ADAPTER pAdapter )
{
    PHARDWARE pHardware = &pAdapter->m_Hardware;
    BOOLEAN   bReceiveDone = FALSE;
    UINT      uiIndicateLength;
    UINT      uiLength;
    int       port = MAIN_PORT;
    BOOLEAN   bUseEmergencyPacket=FALSE;
#ifdef DEBUG_INTERRUPT
    DbgPrint("rev-process\n");
#endif
    // DAVIDCAI first find a available packet
    if(pHardware->m_nReadyBuf[pHardware->m_nCurBufIndex]==PKT_NOT_READY) //
not available
    {
            UINT n;
            for(n=0;n<MAX_NDIS_PKT;n++)
            {
                   if(pHardware->m_nReadyBuf[n]==PKT_READY)
                        {
                          pHardware->m_bLookahead=pHardware->m_pBuf[n];
                          pHardware->m_nCurBufIndex=n;
                          RETAILMSG(TRUE, (TEXT("current
pkt=%d\r\n"),pHardware->m_nCurBufIndex));
                          break;
                        }
            }
            if(n>=MAX_NDIS_PKT) //all packets is not available
               {
                      RETAILMSG(TRUE, (TEXT("no packets, using emergency
pkt\r\n")));
                      bUseEmergencyPacket=TRUE;
               }
      }

      //Get buffer of data from shared memory
      pccHwRecData( pHardware );

        if ( !pHardware->m_nPacketLen )
            goto next_packet;

        uiLength = pHardware->m_nPacketLen - ETHERNET_HEADER_SIZE;
        uiIndicateLength = uiLength;

        // Only allow received packets after Filter is defined.
        if ( pAdapter->m_bPacketFilterSet )
        {
        NDIS_PACKET *PacketArray[MAX_PACKETS_PER_INDICATE];
    if(bUseEmergencyPacket==FALSE)
      {
            PacketArray[0] = CURRENT_PACKET;
            NdisAdjustBufferLength(CURRENT_BUFFER,pHardware->m_nPacketLen);
```

```
            NDIS_SET_PACKET_STATUS(CURRENT_PACKET, NDIS_STATUS_SUCCESS);
            pHardware->m_nReadyBuf[pHardware->m_nCurBufIndex]=PKT_NOT_READY;
            NdisMIndicateReceivePacket(pAdapter->m_hAdapter, PacketArray,
1);
           {
            NDIS_STATUS ReturnStatus;
            ReturnStatus = NDIS_GET_PACKET_STATUS(PacketArray[0]);
            if(ReturnStatus!= NDIS_STATUS_PENDING)
              pHardware->m_nReadyBuf[pHardware-
>m_nCurBufIndex]=PKT_READY//unmark the packet
           }
      }
      else
      {
      //all pass to up-level packets have been used. we need to use this
        //Emergency Packet and ask NDIS do a immediately copy
               PacketArray[0] = pHardware->m_pEmergencyPacket;
            NdisAdjustBufferLength(pHardware->m_pEmergencyNdisBuf,pHardware-
>m_nPacketLen);
            //indicate NDIS to do immediately copy
            NDIS_SET_PACKET_STATUS(pHardware->m_pEmergencyPacket,
NDIS_STATUS_RESOURCES);
            NdisMIndicateReceivePacket(pAdapter->m_hAdapter, PacketArray, 1)
      }
             bReceiveDone |= TRUE;
             pHardware->m_cnCounter[ port ][ OID_COUNTER_RCV_OK ]++;
        }
next_packet:
    // Acknowledge the interrupt
    freeSharedMem(pHardware);
    return( bReceiveDone );
}  // ProcessReceive
```

It can be seen that the ProcessReceive function contains various calls to NDIS layer functions to prepare the memory for the received frame, to process the frame and to indicate that processing is complete.

The function pccHwRecData(pHardware) copies the received frame from the shared memory to the pHardware->m_bLookahead buffer. After populating the m_bLookahead buffer, the NdisMIndicateReceivePacket function informs the NDIS upper layers that the frame is ready to be processed (Microsoft).

Sending Frames

After having established how frames are received and implemented the code to process a received frame it was possible to send frames (Ping frames) to the Ethernet driver to cause a response. By sending a Ping frame to instigate a response it was possible to see which function was called first to send a frame of data through the Ethernet port. The first function to be called turned out to be the MiniportSend function, some of the code is shown below:

```
NDIS_STATUS MiniportSend (
    IN  NDIS_HANDLE  hAdapterContext,
    IN  PNDIS_PACKET pPacket,
    IN  UINT         uiFlags )
{
    PNDIS_ADAPTER pAdapter = ( PNDIS_ADAPTER ) hAdapterContext;
    PHARDWARE     pHardware = &pAdapter->m_Hardware;
    NDIS_STATUS   nsStatus = NDIS_STATUS_FAILURE;
    PNDIS_BUFFER  pndisBuffer;
    UINT          uiPacketLength;

#ifdef DEBUG_COUNTER
    pHardware->m_nGood[ COUNT_GOOD_SEND_PACKET ]++;
#endif
    NdisQueryPacket( pPacket, NULL, NULL, &pndisBuffer, &uiPacketLength );

    // If the returned packet length is zero, there is nothing to transmit.
    if ( !uiPacketLength )
    {
        nsStatus = NDIS_STATUS_SUCCESS;
        goto SendLockDone;
    }

if ( !pHardware->eth_TXcomplete )
    {
        nsStatus = NDIS_STATUS_RESOURCES;
        goto SendBlockDone;
    }
    nsStatus = SendPacket( pAdapter, pPacket,pndisBuffer, uiPacketLength );
```

It can be seen that the MinportSend calls the NdisQueryPacket function to extract a buffer containing a packet of data from the upper layers, and then it calls the SendPacket function to transmit the frame.

Part of the SendPacket function is shown below:

```
NDIS_STATUS SendPacket (
    PNDIS_ADAPTER pAdapter,
    PNDIS_PACKET  pPacket,
    PNDIS_BUFFER  pBuffer,
    UINT          uiPacketLength )
{
    PHARDWARE   pHardware = &pAdapter->m_Hardware;
    NDIS_STATUS nsStatus = NDIS_STATUS_FAILURE;
    UINT        uiLength;
    BOOLEAN     bBroadcast;
    BOOLEAN     bResult;
    int         port = MAIN_PORT;

    NdisAcquireSpinLock( &pAdapter->m_lockAdapter );

    bResult = AdapterCopyDownPacket( pAdapter, pPacket,pBuffer,
uiPacketLength,
        &uiLength, &bBroadcast );

   NdisReleaseSpinLock( &pAdapter->m_lockAdapter );
```

The AdapterCopyDownPacket function is called to extract the packet from the buffer (returned by NdisQueryPacket) and transmit the packet. The AdapterCopyDownPacket contained a whole lot of code to write a packet of up to 2048 bytes long into the Ethernet controller registers and transmit it. Instead a function was used to write the data to the shared memory.

Some of the new AdapterCopyDownPacket function is shown below:

```
BOOLEAN AdapterCopyDownPacket (
    IN  PNDIS_ADAPTER pAdapter,
    PNDIS_PACKET      pPacket,
    IN  PNDIS_BUFFER  pndisBuffer,
    IN  UINT          uiPacketLength,
    OUT PUINT         puiLength,
    OUT BOOLEAN*      pbBroadcast )
{
    PHARDWARE pHardware = &pAdapter->m_Hardware;
    // Length of current source buffer
    UINT      uiBufferLength, nSecondLen;
    static    UCHAR pFirstBuf[ MAX_BUF_SIZE ];
    // Address of current source buffer
    PUCHAR    pBuffer;
    PETHERHDR pEhterHdr;
    ASEND_ACTION sendAction=DIRECT_SEND;
    BOOLEAN fHdr=TRUE;
    DWORD dwCase=0;

    *puiLength = 0; // initialize;

    pBuffer = GetFirstBuffer( &pndisBuffer, &uiBufferLength );

  // may include less than 16 byte data
    if(pBuffer==NULL)
      return FALSE;

    MOVE_MEM(pFirstBuf, pBuffer,uiBufferLength);

      while(uiBufferLength <16) // first buffer must include more than 16
byte
    {
       pBuffer = GetNextBuffer( &pndisBuffer, &nSecondLen);
       if(pBuffer==NULL)
             return FALSE;
      MOVE_MEM(pFirstBuf+uiBufferLength,pBuffer,nSecondLen);
       uiBufferLength+=nSecondLen;
    }

      pBuffer=pFirstBuf;
…

    pEhterHdr=(PETHERHDR)pBuffer;
  if(sendAction!=DISCARD_PACKET)
  {
      pccHwTxData( pHardware, pHardware->m_bLookahead, *puiLength );
  }
…
}
```

The pccHwTxData function copies the data from the m_bLookahead buffer to the shared memory and transmits it.

The following figure is a screenshot taken from a program called WireShark which is used to monitor activity on the Ethernet port of a PC:

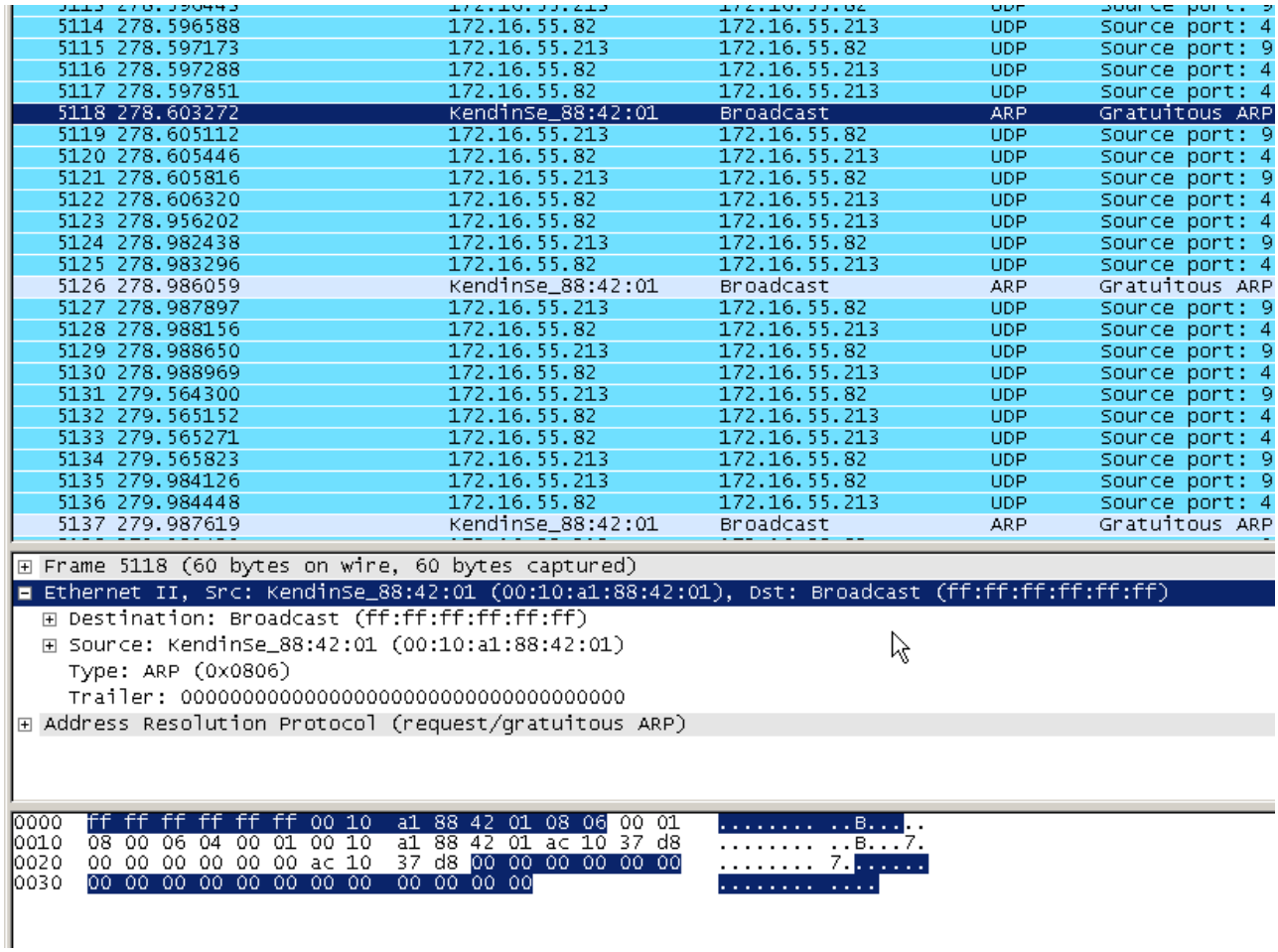


Figure 70: WireShark screenshot showing first Ethernet transmission

The highlighted line at the top of the screenshot shows the frame broadcast by the Ethernet module announcing its presence on the network. In the middle part it can be seen that the length of the frame is 60 bytes and at the bottom the actual data contained in the frame is seen.

Configuration – Getting the Link Status

During early software development it was already established that the Ethernet driver regularly polled the link status of the Ethernet controller, the MiniportISR also contained functionality to process a link status changed interrupt.

The function that polled the link status called the SwitchGetLinkStatus function and the following code was called if the message received from the backplane was a link status changed message:

```
if ( msgType( pHardware ) == ETH_LSC )
    {
      pHardware->eth_TXcomplete = 1;

      pHardware->m_bLinkIntWorking = TRUE;
populateLinkStatus( pHardware );
      SwitchGetLinkStatus( pHardware );
if ( pAdapter->m_ulNdisMediaState !=
             pHardware->m_ulHardwareState )
      {
           NDIS_STATUS nsStatus;
           nsStatus = ( NdisMediaStateConnected ==
            pHardware->m_ulHardwareState ) ?
            NDIS_STATUS_MEDIA_CONNECT : NDIS_STATUS_MEDIA_DISCONNECT;
           NdisMIndicateStatus( pAdapter->m_hAdapter, nsStatus, NULL, 0 );
           NdisMIndicateStatusComplete( pAdapter->m_hAdapter );
           pAdapter->m_ulNdisMediaState = pHardware->m_ulHardwareState;
```

The populateLinkStatus function copied the link status values and some other hardware related values into the appropriate variables of the hardware structure pointed to by the driver. This code is shown below:

```
void populateLinkStatus( PHARDWARE phwi ) {
      phwi->bData[0] = phwi->eth_rx_FileMemory[2];
      phwi->bStatus[0] = phwi->eth_rx_FileMemory[3];
      phwi->bLinkStatus[0] = phwi->eth_rx_FileMemory[4];
      phwi->bData[1] = phwi->eth_rx_FileMemory[5];
      phwi->bStatus[1] = phwi->eth_rx_FileMemory[6];
      phwi->bLinkStatus[1] = phwi->eth_rx_FileMemory[7];
      phwi->bData[2] = phwi->eth_rx_FileMemory[8];
      phwi->bStatus[2] = phwi->eth_rx_FileMemory[9];
      phwi->bLinkStatus[2] = phwi->eth_rx_FileMemory[10];
      phwi->m_bOverrideAddress[0] = phwi->m_bPermanentAddress[0] = phwi-
>eth_rx_FileMemory[11];
      phwi->m_bOverrideAddress[1] = phwi->m_bPermanentAddress[1] = phwi-
>eth_rx_FileMemory[12];
      phwi->m_bOverrideAddress[2] = phwi->m_bPermanentAddress[2] = phwi-
>eth_rx_FileMemory[13];
      phwi->m_bOverrideAddress[3] = phwi->m_bPermanentAddress[3] = phwi-
>eth_rx_FileMemory[14];
      phwi->m_bOverrideAddress[4] = phwi->m_bPermanentAddress[4] = phwi-
>eth_rx_FileMemory[15];
      phwi->m_bOverrideAddress[5] = phwi->m_bPermanentAddress[5] = phwi-
>eth_rx_FileMemory[16];
}
```

After getting the link status from the hardware, the link status information is passed to the upper layers.

Configuration – Setting the multicast table

The MiniportSetInformation and MiniPortGetInformation functions were also found to be regularly polled. One of the functions that seemed important was HardwareSetGroupAddress. The HardwareSetGroupAddress function copied the multicast table from the upper layers into the Ethernet controller's multicast list. The HardwareSetGroupAddress function was replaced by the following function:

```
void pccSetMulticast( PHARDWARE phwi ) {
      int i = 0;
      if(phwi->eth_tx_FileMemory[0] == 0) {
            phwi->eth_tx_FileMemory[1] = ETH_SETMULTI;
            for( i = 0; i < 8; i++ ) {
                  phwi->eth_tx_FileMemory[i+2] = phwi->m_bMulticastBits[ i
];
            }
            phwi->eth_tx_FileMemory[phwi->eth_BufLength+2] = 0x09;
            //indicate that the transmit buffer is ready to transmit
            phwi->eth_tx_FileMemory[0] = 1;
            //trigger transmit event
            SetEvent(phwi->eth_TXEventHandle);
      }
}
```

Configuration – Other settings

While modifying the above configuration functions, a few others were found including a function that read the VLAN table from the Ethernet controller. These other functions were all deemed unnecessary for the basic operation of the Ethernet controller so were left out of the initial design. A future revision of the driver may include getting these functions working to allow features such as VLAN to be carried out using the Ethernet module.

7.5 Slave microcontroller software development

This section describes the software development carried out for the slave module.

Operating system

Unlike the main controller, the slave microcontroller only has limited functionality so it does not need to run an operating system such as Windows CE. The slave microcontroller is programmed using software developed by TCS for other products. This section describes the modifications and additions made to the existing software specifically for the Ethernet module. Because of limited development time, the initial version of the slave microcontroller software was kept simple, using just two threads (main process thread and CANISR()) that were also used for the digital I/O and serial port modules. Future versions of the software might include additional interrupt driven threads to speed up reception and transmission of data.

Adding the Ethernet controller

The digital I/O and the serial port modules both only needed discrete components connected to the microcontroller for input and output protection etc… the Ethernet module however required an integrated circuit to be attached to the address and data bus of the microcontroller to interface it to the Ethernet.

The Address and Data Bus

The address and data bus is a parallel bus to attach peripheral devices to the microcontroller. When a device is attached to the address and data bus, the microcontroller can write to the registers in the device the same way it writes to its own registers. The microcontroller is able to write to the registers of other devices using a concept called memory mapping. The following figure shows the memory structure of the Renesas M16C when the address data bus is set up, the grey areas show which parts of memory can be mapped to peripheral devices:

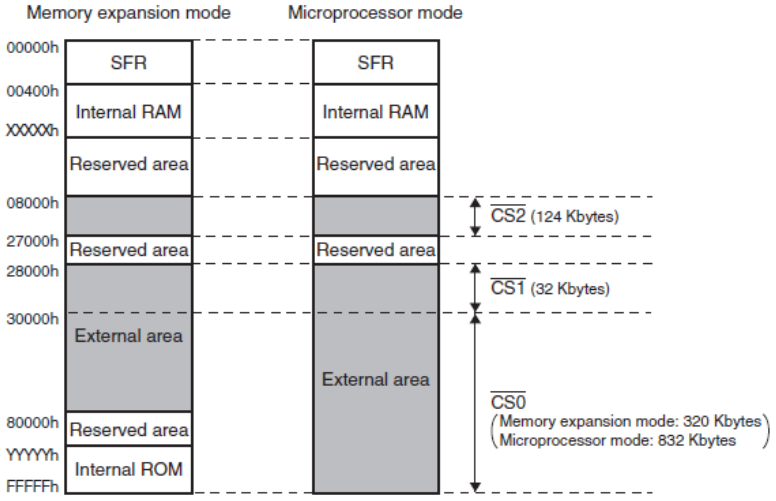


Figure 71: Memory Map and CS area in memory expansion and microprocessor mode (Renesas, 2005)

The enable line of the Ethernet controller was connected to the CS2 pin of the microcontroller, so if the microcontroller is set up to use the above memory map, the processor can write to the Ethernet controller by writing to the memory space between 8000h and 27000h.

Challenges faced

The initial version of the software didn't work, so the following was done to attempt to get it to work:

- Checking the software: going through the M16C datasheet to see if there were any more registers that needed to be set up to initialise the address and data bus. This resulted in a clearer code layout but did not fix the problem.
- Checking the physical connections of the hardware for dry joints and other PCB faults. A few dry joints were resolved but this did not fix the problem.
- Measuring the address and data bus lines (and chip select lines) with an oscilloscope. It was found that the pins were being driven with the appropriate timing, so no problems were found there.

None of the above mentioned diagnostics worked so the datasheet of the Ethernet controller was checked:

- There were various steps for initialising the Ethernet controller, but the address and data bus connection was just standard so that could not have caused any problems.
- Just to be sure there was nothing hidden in the text, the whole datasheet was read in detail: There was a small section about the base address register which stated: "When the EEEN pin is tied to Low, the default base address is 0x0300" (Micrel Inc., 2007). If the base address of a device mapped to 0x8000 is 0x0300, the base address in terms of the microcontroller memory is 0x8300. When the code was changed to use 0x8300 as a base address it worked perfectly.

Writing to and reading from registers

The first version of the Ethernet driver was a modified version of a driver for a device attached to an ISA or PCI bus. This driver was designed to directly populate the registers for the Ethernet module and was modified to write these in the form of commands over the CAN bus so it was necessary to write software in the slave module that converted the read/write commands from CAN frames to read and write to its registers. The following block of code shows part of the main (continuously looping) function which contains the Ethernet writeWord and readWord functions:

```
void Ethernet_Main( void ) {
	//main function
	unsigned int addr = 0x0000;
	unsigned int data = 0x0000;
	unsigned int tempdata = 0x0000;
	unsigned int errStatus = 0x0000;

	//read/write registers
	if(ProtocolReceiveBufferReady) {
		switch( ProtocolReceiveBuffer[ 0 ] ) {
			case ETHMSG_WRITE_WORD :
				addr = ProtocolReceiveBuffer[ 1 ] << 8;
				addr |= ProtocolReceiveBuffer[ 2 ];
				data = ProtocolReceiveBuffer[ 3 ] << 8;
				data |= ProtocolReceiveBuffer[ 4 ];
				writeWord(addr, data);
				//Send back confirmation write was completed
				ProtocolTransmitBuffer[ 0 ] = ETHMSG_WRITE_WORD;
				ProtocolTransmitBufferLength = 1;
				ProtocolTransmitBufferReady = 1;
				asm( "\t int #003h" );
			break;
			case ETHMSG_READ_WORD :
				addr = ProtocolReceiveBuffer[ 1 ] << 8;
				addr |= ProtocolReceiveBuffer[ 2 ];
				data = readWord(addr);
				//Send back data
				ProtocolTransmitBuffer[ 0 ] = ETHMSG_READ_WORD;
				ProtocolTransmitBuffer[ 1 ] = data >> 8;
				ProtocolTransmitBuffer[ 2 ] = data;
				ProtocolTransmitBufferLength = 3;
				ProtocolTransmitBufferReady = 1;
				asm( "\t int #003h" );
			break;
			default:
				//Do nothing
			break;
		}
		ProtocolReceiveBuffer[ 0 ] = 0x00;
	}
}
```

The writeWord and readWord functions were written to accept a register address and data parameter to reduce code repetition for the read and write operations.

As an example, part of the writeWord function is shown below:

```
void writeWord(unsigned int addr, unsigned int value) {
      switch(addr) {
            case 0x00 :
                  ksz8842reg0 = value;
            break;
            case 0x02 :
                  ksz8842reg2 = value;
            break;
            case 0x04 :
                  ksz8842reg4 = value;
            break;
            case 0x06 :
                  ksz8842reg6 = value;
            break;
            case 0x08 :
                  ksz8842reg8 = value;
            break;
            case 0x0A :
                  ksz8842regA = value;
            break;
            case 0x0C :
                  ksz8842regC = value;
            break;
            case 0x0E :
                  ksz8842regE = value;
            break;
            default:
                  //do nothing
            break;
      }
}
```

Initialisation

During the initial stages of software development it was noted that a lot of the initialisation carried out was simply writing static data from the main controller to the slave module. To reduce load on the backplane, this code was removed from the main controller so that most of the initialisation could be carried out inside the slave module without input from the main controller. A helpful resource for writing the initialisation code was an application note provided by the manufacturer of the Ethernet controller. The application note showed the steps required to initialise the Ethernet controller for basic operation. The figure below shows a screenshot taken of the application note showing a few of the initialisation steps:

Steps Sequence	Read\write	Register Name[bit]	Value	Description
0		BAR [15-0] Bank 0, 0ffset 0x00	0x0300	Just to make sure that your host bus controller has assigned the low 16bit base address to the KSZ8841/2M is same as the device default base address 0x0300.
1	Read	SIDER [15-4] Bank 32, 0ffset 0x00	0x880	Read the device chip ID, make sure it is correct ID (0x880 for KSZ8842M), otherwise there are some errors on the host bus interface.
2	Write	MARL[15-0] Bank 2, 0ffset 0x00	0x89AB	Write QMU MAC address (low). MAC address are generally expressed in the form of 01:23:45:67:89:AB. (we use this MAC as a example).
3	Write	MARM[15-0] Bank 2, 0ffset 0x02	0x4567	Write QMU MAC address (Medium). MAC address are generally expressed in the form of 01:23:45:67:89:AB. (we use this MAC as a example).
4	Write	MARH[15-0] Bank 2, 0ffset 0x04	0x0123	Write QMU MAC address (High). MAC address are generally expressed in the form of 01:23:45:67:89:AB. (we use this MAC as a example).
5	Write	MACAR1[15-0] Bank 39, 0ffset 0x00	0x0123	Write Switch MAC address 1. MAC address are generally expressed in the form of 01:23:45:67:89:AB. (we use this MAC as a example).

Figure 72: Micrel application note showing initialisation steps (Micrel Semiconductor, 2007)

The initialisation steps were compared to the initialisation steps of the Ethernet driver and it was found that they were very similar. With a few adjustments the code for the initialisation steps was written into the slave module.

Part of the initialisation code can be seen below:

```
void Ethernet_Init( void ) {
      unsigned int temp;
      //Initialise ethernet controller
      NetworkLEDOff();

      //Write QMU MAC address
      writeWordDir(0x02, HostMacAddrLow, 0x4301);
      writeWordDir(0x02, HostMacAddrMid, 0xA188);
      writeWordDir(0x02, HostMacAddrHigh, 0x0010);
      //Write switch MAC address
      writeWordDir(0x27, MacAddr1, 0x4301);
      writeWordDir(0x27, MacAddr2, 0xA188);
      writeWordDir(0x27, MacAddr3, 0x0010);

      //Enable QMU Transmit flow control / Transmit padding / Transmit CRC
      writeWordDir(0x10, TransmitCtl, 0x000E);
      //Enable QMU Receive flow control / Receive all broadcast frames /
      // Receive all multicast frames / Receive unicast frames / Receive
      // strip the CRC
      writeWordDir(0x10, RecCtl, 0x000E);
      //Enable QMU Transmit Frame Data Pointer Auto Increment
      writeWordDir(0x11, TxFrameDataPtr, 0x4000);
      //Enable QMU Receive Frame Data Pointer Auto Increment
      writeWordDir(0x11, RxFrameDataPtr, 0x4000);
```

Receive ISR

The step-by-step programmers guide used to obtain the initialisation procedure also included a step by step guide for receiving data. Before being able to apply these steps it was necessary to set up an ISR (Interrupt Service Routine) and to actually understand how the data reception process of the Ethernet controller works.

Setting up the ISR

The following figure shows the Ethernet interrupt line (ENETINT) is connected to pin 20 on the microcontroller. Pin 20 can be set up as an external interrupt (INT0), the ISR therefore needed to be set up to be triggered by INT0.

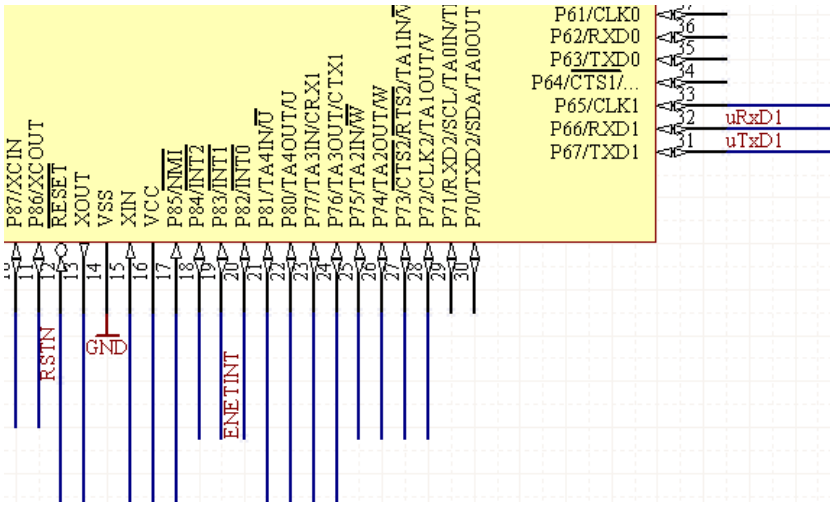


Figure 73: Ethernet interrupt wired to INT0 (Meek, V4-128-B PCC Ethernet.SchDoc, 2010)

TCS has already developed products using an M16 microcontroller with an external SJA1000 CAN controller that was wired up to an external interrupt. The code for these products was used as an example to base the Ethernet ISR on.

The first step for setting up the Ethernet ISR was to define pin 20 as an input and set its interrupt priority, this was done by the following function:

```
void EthEnable(void) {
      pd8_2 = INPUT;
      int0ic = 0x02; //Interrupt priority 2
}
```

It is likely that the interrupt priority may need to be adjusted to make the slave module run more efficiently. This is something that may need to be addressed in future software revisions.

The next step was to set up a function to be called by the Interrupt and define the function as an Interrupt; this was done by the following code:

```
#pragma INTERRUPT EthRxISR
void far EthRxISR( void );

//Ethernet ISR
void far EthRxISR( void ) {
        //Ethernet interrupt code goes here
}
```

Having set up the interrupt pin and the function to service the interrupt it was necessary to map the function to the interrupt vector; this was done by modifying the interrupt vector table in the microcontroller code:

```
;-----------------------------------------------------------------
; variable vector section
;-----------------------------------------------------------------
      .section    vector      ; variable vector table
      .org  VECTOR_ADR

      .lword      dummy_int   ; BRK (vector 0)
      .lword      dummy_int   ;     (vector 1)
      .glb        _CANISR
      .lword      _CANISR     ; c0rec (vector 2)
      .glb        _CANTXISR
      .lword      _CANTXISR   ; c0trm (vector 3)
      .lword      dummy_int   ; int3(for user)(vector 4)
      .lword      dummy_int   ; timerB5(for user)(vector 5)
      .lword      dummy_int   ; timerB4(for user)(vector 6)
      .lword      dummy_int   ; timerB3(for user)(vector 7)
      .lword      dummy_int   ; si/o4 /int5(for user)(vector 8)
      .lword      dummy_int   ; si/o3 /int4(for user)(vector 9)
      .lword      dummy_int   ; Bus collision detect (for user)(vector 10)
      .lword      dummy_int   ; DMA0(for user)(vector 11)
      .lword      dummy_int   ; DMA1(for user)(vector 12)
      .lword      dummy_int   ; Key input interrupt(for user)(vector 13)
      .lword      dummy_int   ; A-D(for user)(vector 14)
      .lword      dummy_int   ; uart2 transmit(vector 15)
      .lword      dummy_int   ; uart2 receive(vector 16)
      .glb        _Serial0TxISR
      .lword      _Serial0TxISR   ; uart0 transmit(for user)(vector 17)
      .glb        _Serial0RxISR
      .lword      _Serial0RxISR   ; uart0 receive(for user)(vector 18)
      .lword      0FF900H     ; uart1 transmit (vector 19) KD30 Monitor V2
      .lword      0FF900H     ; uart1 receive  (vector 20) KD30 Monitor V2
      .glb        _HWTIMERISR
      .lword      _HWTIMERISR ; timer A0(for user)(vector 21)
      .glb        _TimerA1ISR
      .lword      _TimerA1ISR ; timer A1(for user)(vector 22)
      .lword      dummy_int   ; timer A2(for user)(vector 23)
      .lword      dummy_int   ; timer A3(for user)(vector 24)
      .lword      dummy_int   ; timer A3(for user)(vector 24)
      .lword      dummy_int   ; timer B0(for user)(vector 26)
      .lword      dummy_int   ; timer B1(for user)(vector 27)
      .lword      dummy_int   ; timer B2(for user)(vector 28)
      .glb        _EthRxISR
      .lword      _EthRxISR   ; int0 (vector 29)
      .lword      dummy_int   ; int1 (for user)(vector 30)
      .lword      dummy_int   ; int2 (for user)(vector 31)
```

It can be seen that EthRxISR is mapped to vector 29 which is representative of int0.

The step-by-step guide supplied by the Ethernet controller manufacturer instructed the programmer to set up only the receive interrupt. With the receive interrupt set up, the ISR would be triggered upon receiving an Ethernet frame.

After some further investigation, looking into the full data sheet of the Ethernet controller, it was found that the interrupt service routine could also be triggered by types of interrupts other than the receive interrupt. The following figure shows the structure of the Interrupt Enable Register that allows the programmer to enable various types of interrupts.

Bank 18 Interrupt Enable Register (0x00): IER

This register enables the interrupts from the QMU and other sources.

Bit	Default Value	R/W	Description
15	0x0	RW	**LCIE Link Change Interrupt Enable** When this bit is set, the link change interrupt is enabled. When this bit is reset, the link change interrupt is disabled.
14	0x0	RW	**TXIE Transmit Interrupt Enable** When this bit is set, the transmit interrupt is enabled. When this bit is reset, the transmit interrupt is disabled.
13	0x0	RW	**RXIE Receive Interrupt Enable** When this bit is set, the receive interrupt is enabled. When this bit is reset, the receive interrupt is disabled.
12	0x0	RW	Reserved
11	0x0	RW	**RXOIE Receive Overrun Interrupt Enable** When this bit is set, the Receive Overrun interrupt is enabled. When this bit is reset, the Receive Overrun interrupt is disabled.
10	0x0	RW	Reserved
9	0x0	RW	**TXPSIE Transmit Process Stopped Interrupt Enable** When this bit is set, the Transmit Process Stopped interrupt is enabled. When this bit is reset, the Transmit Process Stopped interrupt is disabled.
8	0x0	RW	**RXPSIE Receive Process Stopped Interrupt Enable** When this bit is set, the Receive Process Stopped interrupt is enabled. When this bit is reset, the Receive Process Stopped interrupt is disabled.
7	0x0	RW	**RXEFIE Receive Error Frame Interrupt Enable** When this bit is set, the Receive error frame interrupt is enabled. When this bit is reset, the Receive error frame interrupt is disabled.
6-0	-	RO	Reserved

Figure 74: Interrupt Enable Register (Micrel Inc., 2007)

The interrupts that were decided to be enabled for the initial version of the Ethernet module software were:

- Receive Interrupt: Triggered upon receiving a frame. This triggers the microcontroller to read the frame and transmit it to the PCC main controller.
- Transmit Interrupt: Triggered upon completing transmission. When this is triggered, a message is sent to the PCC main controller that it can transmit the next message.
- Link Change Interrupt: Triggered upon completion of auto-negotiation after a cable is plugged in or when a cable is removed. When this is triggered a message is sent to the PCC to indicate the link status.

Because the different types of interrupts all trigger the same interrupt pin it was necessary to check what kind of interrupt had triggered the interrupt pin before carrying out the related task. The following figure shows the Interrupt Status register, this register was looked at to decide what kind of interrupt had occurred:

Bank 18 Interrupt Status Register (0x02): ISR

This register contains the status bits for all QMU and other interrupt sources.

When the corresponding enable bit is set, it causes the interrupt pin to be asserted.

This register is usually read by the host CPU and device drivers during interrupt service routine or polling. The register bits are not cleared when read. The user has to write "1" to clear

Bit	Default Value	R/W	Description
15	0x0	RO (W1C)	**LCIS Link Change Interrupt Status** When this bit is set, it indicates that the link status has changed from link up to link down, or link down to link up. This edge-triggered interrupt status is cleared by writing 1 to this bit.
14	0x0	RO (W1C)	**TXIS Transmit Status** When this bit is set, it indicates that the TXQ MAC has transmitted at least a frame on the MAC interface and the QMU TXQ is ready for new frames from the host. This edge-triggered interrupt status is cleared by writing 1 to this bit.
13	0x0	RO (W1C)	**RXIS Receive Interrupt Status** When this bit is set, it indicates that the QMU RXQ has received a frame from the MAC interface and the frame is ready for the host CPU to process. This edge-triggered interrupt status is cleared by writing 1 to this bit.
12	0x0	RO	Reserved
11	0x0	RO (W1C)	**RXOIS Receive Overrun Interrupt Status** When this bit is set, it indicates that the Receive Overrun status has occurred. This edge-triggered interrupt status is cleared by writing 1 to this bit.
10	0x0	RO	Reserved
9	0x1	RO (W1C)	**TXPSIE Transmit Process Stopped Status** When this bit is set, it indicates that the Transmit Process has stopped. This edge-triggered interrupt status is cleared by writing 1 to this bit.
8	0x1	RO (W1C)	**RXPSIE Receive Process Stopped Status** When this bit is set, it indicates that the Receive Process has stopped. This edge-triggered interrupt status is cleared by writing 1 to this bit.
7	0x0	RO (W1C)	**RXEFIE Receive Error Frame Interrupt Status** When this bit is set, it indicates that the Receive error frame status has occurred. This edge-triggered interrupt status is cleared by writing 1 to this bit.
6-0	-	RO	Reserved

Figure 75: Interrupt Status Register (Micrel Inc., 2007)

The following code was used to determine the interrupt type:

```
#pragma INTERRUPT EthRxISR
void far EthRxISR( void );

//Ethernet ISR
void far EthRxISR( void ) {
	unsigned int temp;
	unsigned int intr_type;

	//Read value from ISR to check interrupt type
	temp = 0x00;
	temp = readWordDir(0x12, 0x02);
	intr_type = temp & 0x2000;
	if(intr_type == 0x2000) {
		//Receive
	}
	intr_type = 0x0000;
	intr_type = temp & 0x8000;
	if(intr_type == 0x8000) {
		//Link Change
	}
	intr_type = 0x0000;
	intr_type = temp & 0x4000;
	if(intr_type == 0x4000) {
		//Transmit
	}
}
```

The data reception process

The main part of the receive process is getting data from the receive Queue in the Ethernet controller. Figure 76 shows the structure of the Queue Management Unit in the microcontroller. The Queue Management Unit is a structure of Ethernet controller registers. There are two registers called “QMU Data Low” and “QMU Data High”, these registers are used to read data from the queue management unit. The following figure shows the structure of the queue management unit:

Packet Memory Address Offset	Bit 15 2nd Byte	Bit 0 1st Byte
0	Status Word	
2	Byte Count	
4 - up	Packet Data (maximum size is 1916)	

Figure 76: Receive queue management unit structure (Micrel Inc., 2007)

Every time the registers are read, the offset is incremented allowing the entire queue to be read through the two registers. To read a received message the following steps need to be followed:

1. Read Status Word (Packet memory address offset 0) to check if the message is valid etc… from QMU Data Low.
2. Read and store Byte Count (Packet memory address offset 2) from QMU Data High.
3. While the byte count isn’t exceeded read Packet Data (Packet memory address offset 4 up) from QMU Data Low and QMU Data High.

As with the initialisation code the manufacturer had supplied a step-by-step guide for processing a received message. The step-by-step guide included a few other steps to be carried out around the receive interrupt that were also included in the receive code.

Transmitting data

The Ethernet controller also used a queue management unit to transmit frames of data. The structure of the queue management unit is shown below:

Packet Memory Address Offset	Bit 15 2^{nd} Byte	Bit 0 1^{st} Byte
0	Control Word	
2	Byte Count	
4 - up	Packet Data (maximum size is 1916)	

Figure 77: Transmit queue management unit structure (Micrel Inc., 2007)

The transmit process worked similarly to the receive process:

1. Write the control word to QMU Data Low to set various control settings such as “interrupt on complete”
2. Write the byte count to QMU Data High
3. While the byte count isn’t exceeded write the packet data to QMU Data High and QMU Data low.

For this there were also a few more steps involved with transmitting a frame of data that were described in the step-by-step programmers guide provided by the manufacturer.

7.6 Data transmission

Main controller

Because most of the NDIS upper layer functions for message queuing were used, it was not necessary to develop a data transmission protocol for the miniport driver loaded in the main controller. For details of the functions that were used, see the section “Ethernet driver software development”.

Slave module

Because the Ethernet port was able to operate at much higher speeds than the backplane, there needed to be a method to buffer the incoming frames in the slave module. The following figure shows the flow chart implemented for buffering data in the slave module:

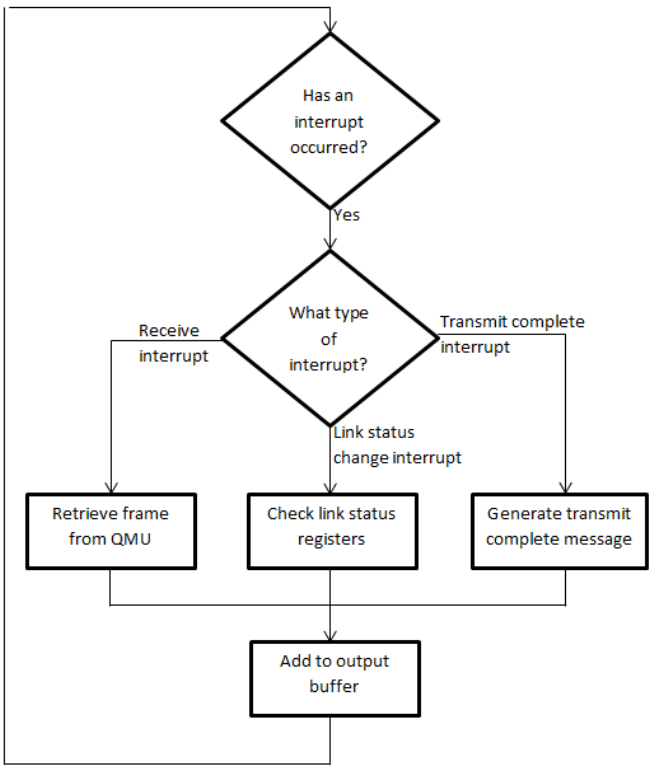


Figure 78: Slave module data buffering

The interrupts mentioned in the flow chart are interrupts generated by the Ethernet controller (e.g. the Receive interrupt is generated when an Ethernet frame comes in). The output buffer mentioned is the queue of messages waiting to be transmitted across the CAN backplane.

This flow chart was implemented in the slave module but due to lack of memory the output buffer was only big enough for one frame. To prevent conflicts occurring, software was written to block any writes to the output buffer until the data in it had been transmitted. To be able to buffer more data, memory would have to be freed up in the microcontroller and it may even be necessary to add additional external memory to the microcontroller. A future revision of the software may need this extra memory to prevent frames being lost during high traffic situations.

The following flowchart shows the basic operation of the main function of the slave module:

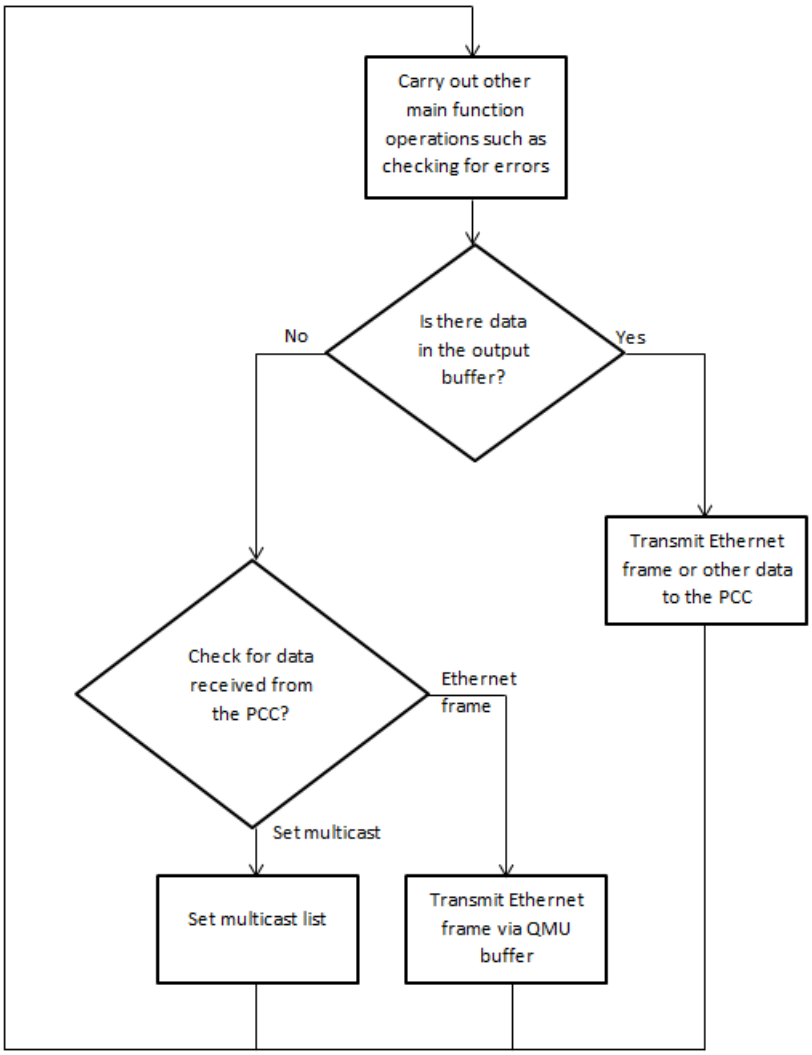


Figure 79: Ethernet module main loop

It can be seen that processing of received messages (the output buffer gathers messages received from the Ethernet before transmitting them to the main controller) takes priority over processing messages from the PCC to be sent out via the Ethernet since the buffer is limited in the slave module compared to the buffering available in the PCC main controller.

7.7 Performance testing

Performance tests carried out included a ping test and a data transfer test. Initial tests carried out were adequate to prove that the Ethernet port worked and was adequate for the basic industrial Ethernet requirements TCS had set for the Ethernet module.

Registering the IP address in the ARP table

When an Ethernet device attempts to communicate with another Ethernet device that it has not communicated with before it first broadcasts a message asking for the device that has a particular IP address to reply with routing information (its MAC address) so that the device can update its routing table, this shows the basic functionality of an Ethernet device.

The following figure is a screenshot taken from WireShark showing the PCC's Ethernet module replying to an ARP request:

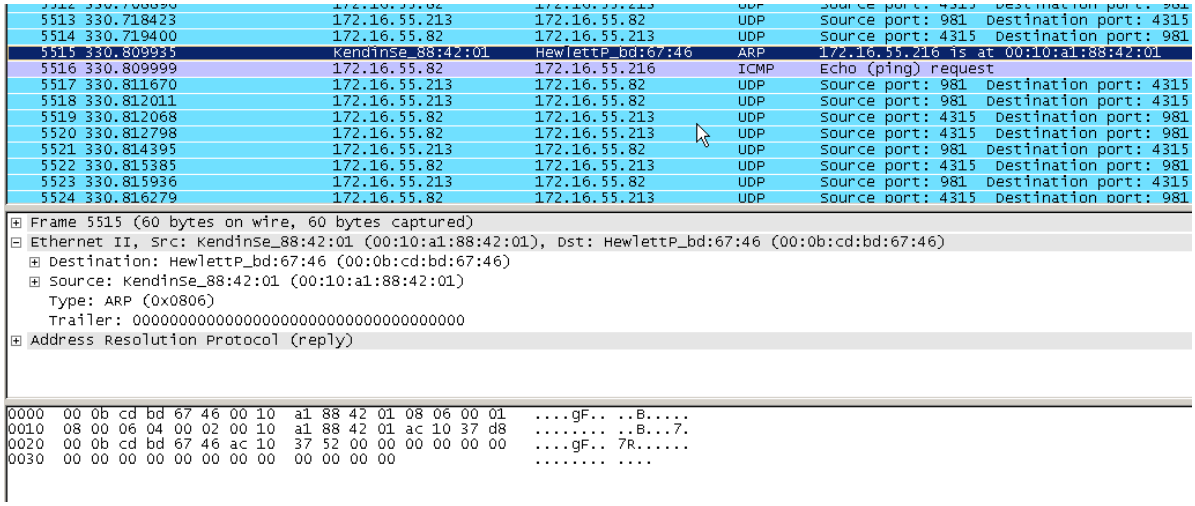


Figure 80: ARP reply message

It can be seen that the message is transmitted from KendinSe_88:42:01 (the Ethernet module) to HewlettP_bd:67:46 (the PC) and it is of type ARP. The message states that the IP 172.16.55.216 is at the MAC address 00:10:a1:88:42:01.

Looking at the ARP routing table in the PC, it can be seen that the IP (172.16.55.216) and MAC (01-10-a1-88-42-01) of the Ethernet module are now listed in the routing table.

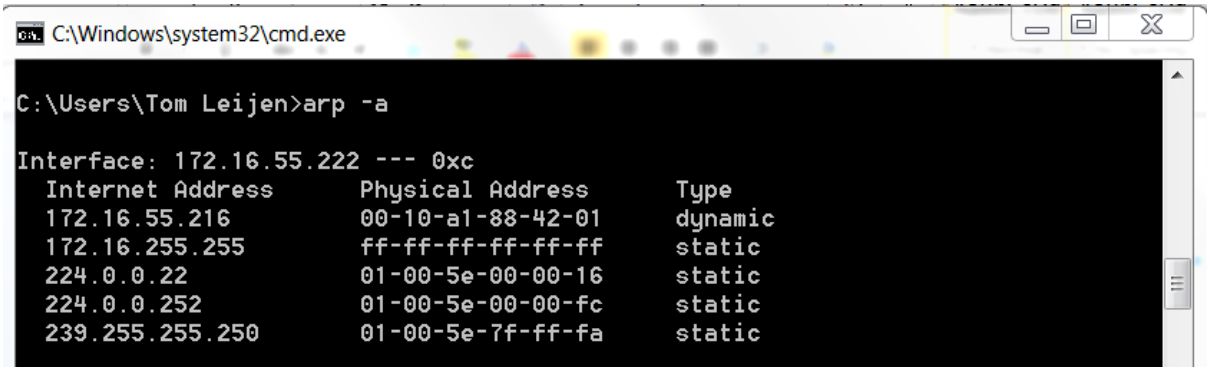


Figure 81: ARP routing table

Ping testing

The following figure shows one of the early Ping tests carried out using the built in Microsoft Windows Ping program:

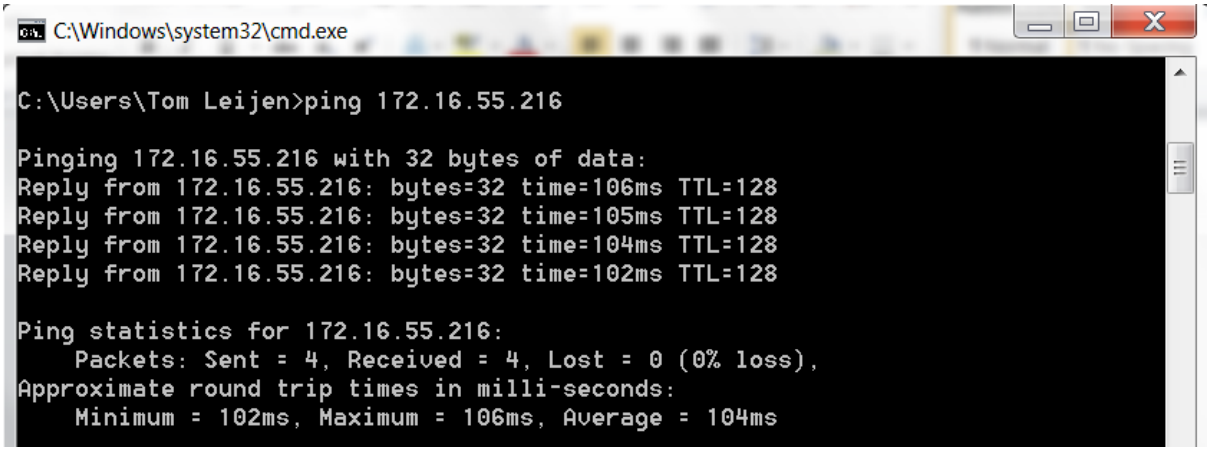


Figure 82: Early Ping test

A Ping reply time of 104ms was deemed quite slow, so a few of the frame acknowledgement systems put in place to prevent collisions on the backplane were streamlined to operate faster, that resulted in a drastically reduced Ping reply time:

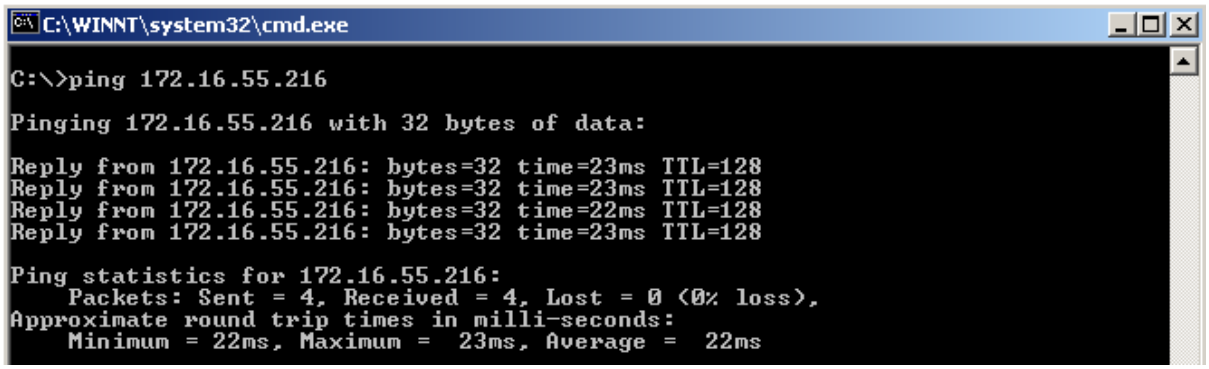


Figure 83: Shorter Ping reply times

The following figure is a screenshot taken from WireShark showing the Ping request and reply messages being transmitted over the Ethernet:

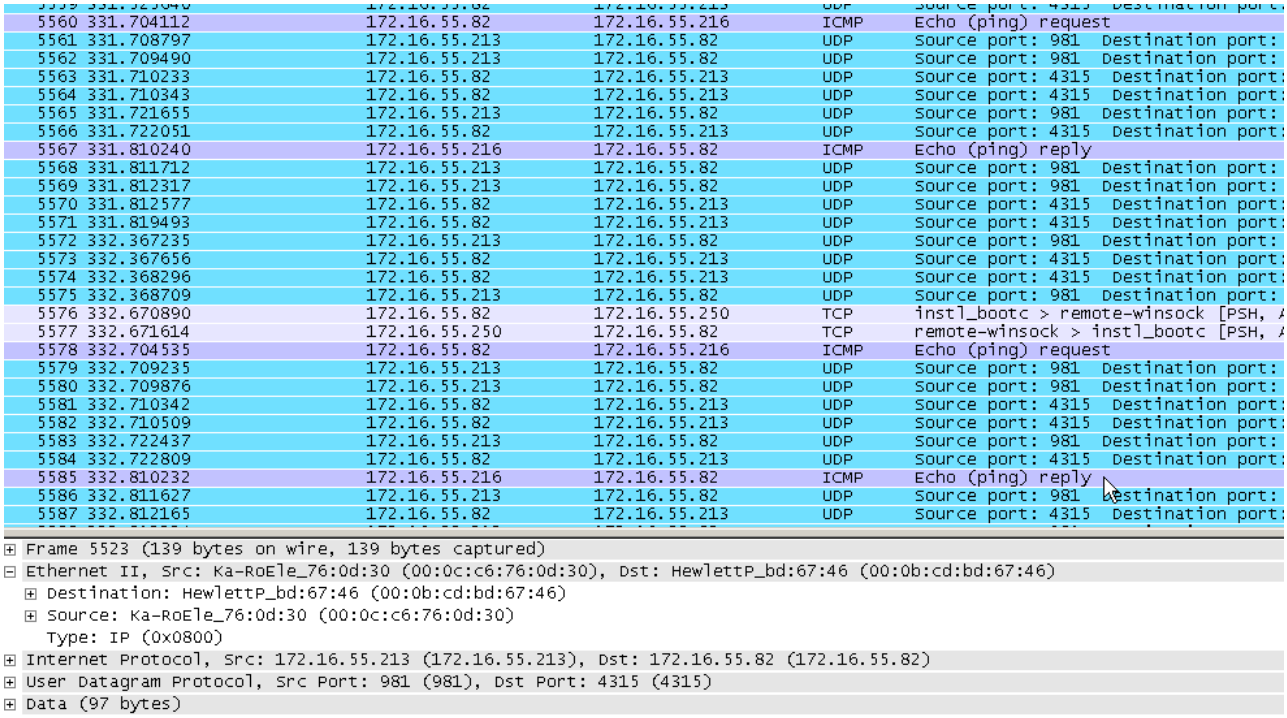


Figure 84: Ping Request and Reply messages

The version of the driver used when the above Ping test was carried out used the Sleep() function to wait a few milliseconds before transmitting more data over the backplane to prevent conflicts on the backplane. The sleep period was 20ms and then a further 10ms every time the backplane was found to be occupied. Instead of using Sleep() it was decided to use a Mutex to see if the backplane was free.

“A mutex object is a synchronization object whose state is set to signalled when it is not owned by any thread and non-signalled when it is owned. Its name comes from its usefulness in coordinating mutually exclusive access to a shared resource. Only one thread at a time can own a mutex object.” (Microsoft)

By using the WaitForSingleOjbect(phwi->mutex, MAX_MUTEX_WAIT) function the time waited by the transmitting thread was reduced to the minimum amount of time needed to wait for the backplane to become free. The following screenshot shows the results of using a mutex instead of Sleep:

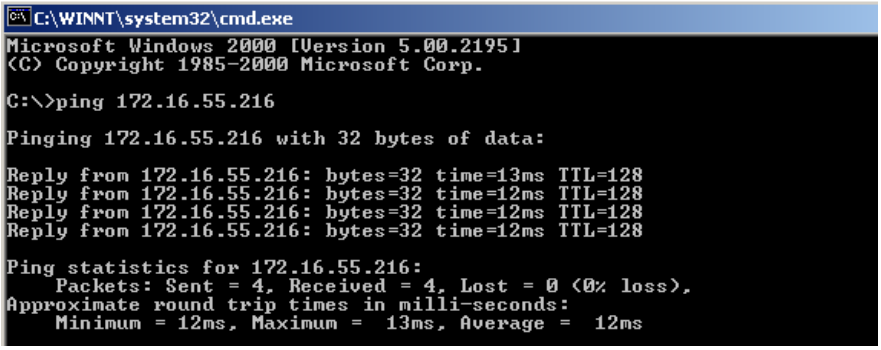


Figure 85: Faster Ping times

File transfer

The command line FTP functions are useful to test the performance of an Ethernet port because they show detailed results, such as speed and number of bytes transferred during the transfer. The following image file was uploaded to the PCC main controller via FTP, and then downloaded back to the PC:

Figure 86: Windows CE 5.0 logo (Microsoft)

The following figure shows the FTP results for loading it via the built in port of the main controller:

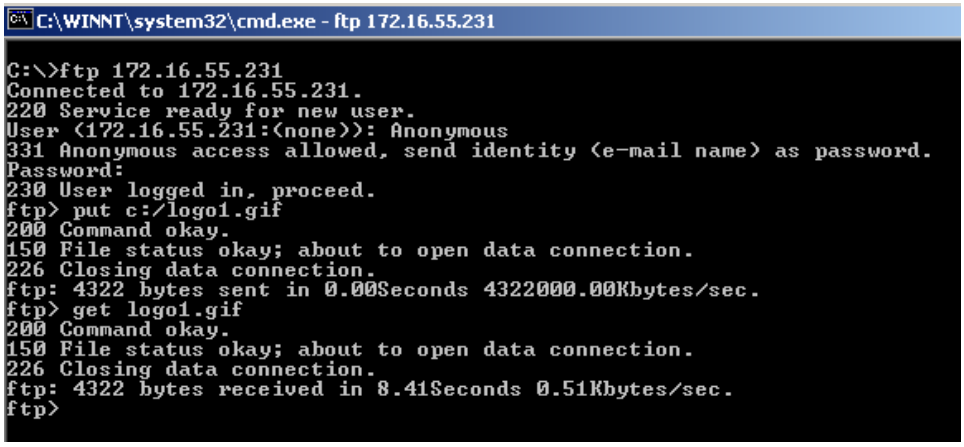


Figure 87: Command line FTP showing speed to transfer over built in Ethernet port

It can be seen that the upload speed is very high 4322000Kbytes/sec* but the download speed is quite slow at 0.15Kbytes/sec, this is suspected to have been due to the fact that the main controller was continuously sending data out over the Ethernet for the debug messages.

*This value is unrealistic because the maximum speed of the CAN bus is only 1Mbps, it should be seen as the PC reporting that the transfer was too fast for it to calculate the speed.

The following figure shows the FTP results for loading it via the Ethernet port module:

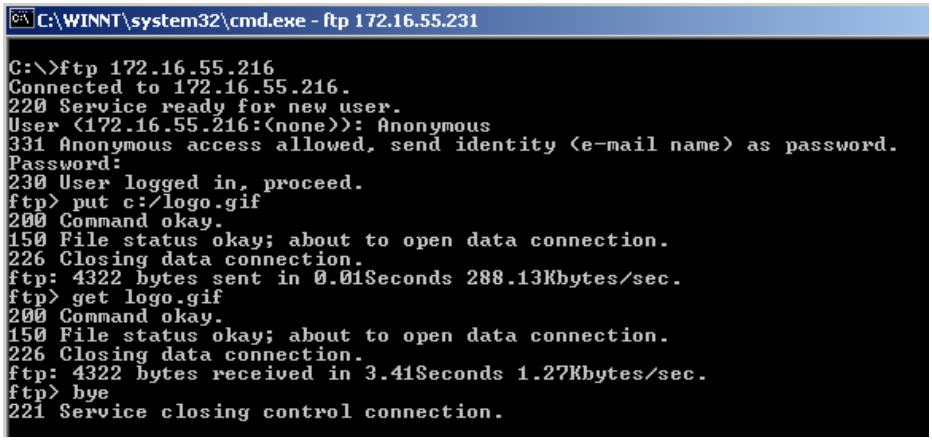


Figure 88: Command line FTP via Ethernet port module

It can be seen that the upload speed has changed significantly to 288.13Kbytes/sec and the download speed is somewhat faster than that of the built in port at 2.7Kbytes/sec.

It is suspected that the download speed is quite slow because the debugger is busy sending data out the built in Ethernet port, this is taking up most of the bandwidth allowing little more for downloading, this however is an issue that needs further investigation.

The upload speed of 288.13Kbytes/sec over the CAN bus is quite a realistic speed because it is similar to what was found by Ditze et. al. who managed to stream video at a rate of 380Kbytes/sec (Ditze, Bernhardi, Kamper, & Altenbend, 2003). With some buffering in the slave module and some more adjustments in the performance of the backplane a transfer rate of about 380Kbytes/sec would be obtainable.

7.8 Recommendations

Implementing additional features

During prototype development a number of features were disabled because they weren't necessary for the basic operation that needed to be available for proof of concept testing. In the future these features may need to be added to make the Ethernet port module compatible with a wider range of devices.

Buffering

Due to a lack of memory, the current slave module can only buffer one incoming Ethernet frame; this could cause trouble if the Ethernet port is subjected to a heavy load of incoming data. The performance and reliability of the Ethernet module could be increased significantly in situations of high load by increasing the memory so that it can be used to buffer Ethernet frames while they wait to be transmitted across the backplane.

Interrupts

The slave microcontroller software may be able to be sped up by putting the code that handles transmission of data in an interrupt service routine rather than in the main routine. This is something that will have to be investigated when the current software is deemed too slow for a particular application.

Two separate ports instead of one switch

Initial development was carried out with the default settings of the Ethernet controller enabled; this meant that it was set up as a switch connected to the PCC through a port with a single IP address. During research it was found in the datasheet that it was possible to disable the switching engine so that messages could be directed to a particular port and that each message coming in was labelled with the port number from where it came, these features all indicate that it is possible to use the Ethernet module as two separate Ethernet ports with two separate IP addresses. It would be more complicated to set up the Ethernet module as two separate ports and traffic would be slower since now two signals are being transmitted across a single 1mbps line but it may also have its benefits in an industrial application.

Fault and error handling

The prototype Ethernet port did not include any fault or error handling other than a basic attempt to reset the fault if the Ethernet controller produced a fault message. Future revisions of the software would need to include more fault handling systems to make it reliable enough for release. There also needs to be a way to inform ISaGRAF that a fault has occurred so that it can react appropriately and, for example, display the fault on an HMI screen.

7.9 Discussion

Several types of Ethernet driver were investigated and several ways to load them were attempted, eventually a miniport driver that interfaced the NDIS upper layer with the PCC backplane deriver that was loaded via an NDIS IOControl was developed. The miniport driver and slave module software developed provided basic Ethernet functionality and a platform which can be modified to suit the specific requirements of TCS and their customers. Before releasing the Ethernet module in industry, features such as fault and error handling will need to be implemented. Features such as two separate ports can be added as TCS and their customers require it.

7.10 Conclusion

A basic industrial-rated Ethernet module was added to the range of available slave modules for the PCC. This module is not fully featured and the performance remains relatively low. The module may be adequate for basic industrial Ethernet applications but some of the issues will need to be addressed in the next revision of the Ethernet module software.

8 Nuremberg show setup (PCC Test and Demonstration)

8.1 Introduction

To promote the PCC and demonstrate its IEC61499 capabilities TCS had decided to take a fully functioning PCC with a few modules attached to a trade show in Nuremberg. This section shows the PCC being used in a theoretical baggage handling application to demonstrate a typical application of the PCC in an IEC61499 environment. As part of the Nuremberg show setup, TCS had also developed a motor control station module. It is shown how TCS used hardware and software developed for the digital I/O module to develop the hardware and software for the motor control station module.

8.2 Hardware setup

Show display

The following figure shows the setup of the hardware displayed at the Nuremberg show in front of a poster demonstrating several possible applications for the PCC:

Figure 89: Nuremberg show setup

The show setup shows the PCC as if it were controlling a “cell” of three conveyor belts making up part of a larger baggage handling system. In terms of IEC61499, this “cell” can be seen as a function block, or a number of function blocks, provided by the resource which is the PCC setup shown.

Motor control station

An interesting aspect of this setup is that it includes a module not mentioned in this report. The motor control station module hardware was developed by a contractor employed by TCS and the software was developed by Andrew Meek. The motor control station module is a direct on line (DOL) motor controller, this means that it can switch a three phase motor on or off but it cannot vary the motor speed. The motor control module also has some digital I/O connections, increasing its functionality as a PCC module and making the hardware also able to operate as a standalone module with different firmware. The following figure shows the motor control station module:

Figure 90: PCC Motor control station module

The interesting thing about this module is that the hardware is for the largest part based on the digital I/O module developed for this project, because of this, the software is based on the digital I/O software. The fact that contractor Aaron Wilson only had to carry out design calculations for the three phase part of the hardware and that Andrew Meek only needed to spend a day to modify the digital I/O software demonstrates the versatility of the platform developed for the PCC in this project.

8.3 SCADA screen

SCADA stands for **S**upervisory **C**ontrol **A**nd **D**ata **A**cquisition. The following graphic was displayed on a screen next to the cell display to show the context of which the cell was a part. The graphic was designed by Sarah Clark of TCS.

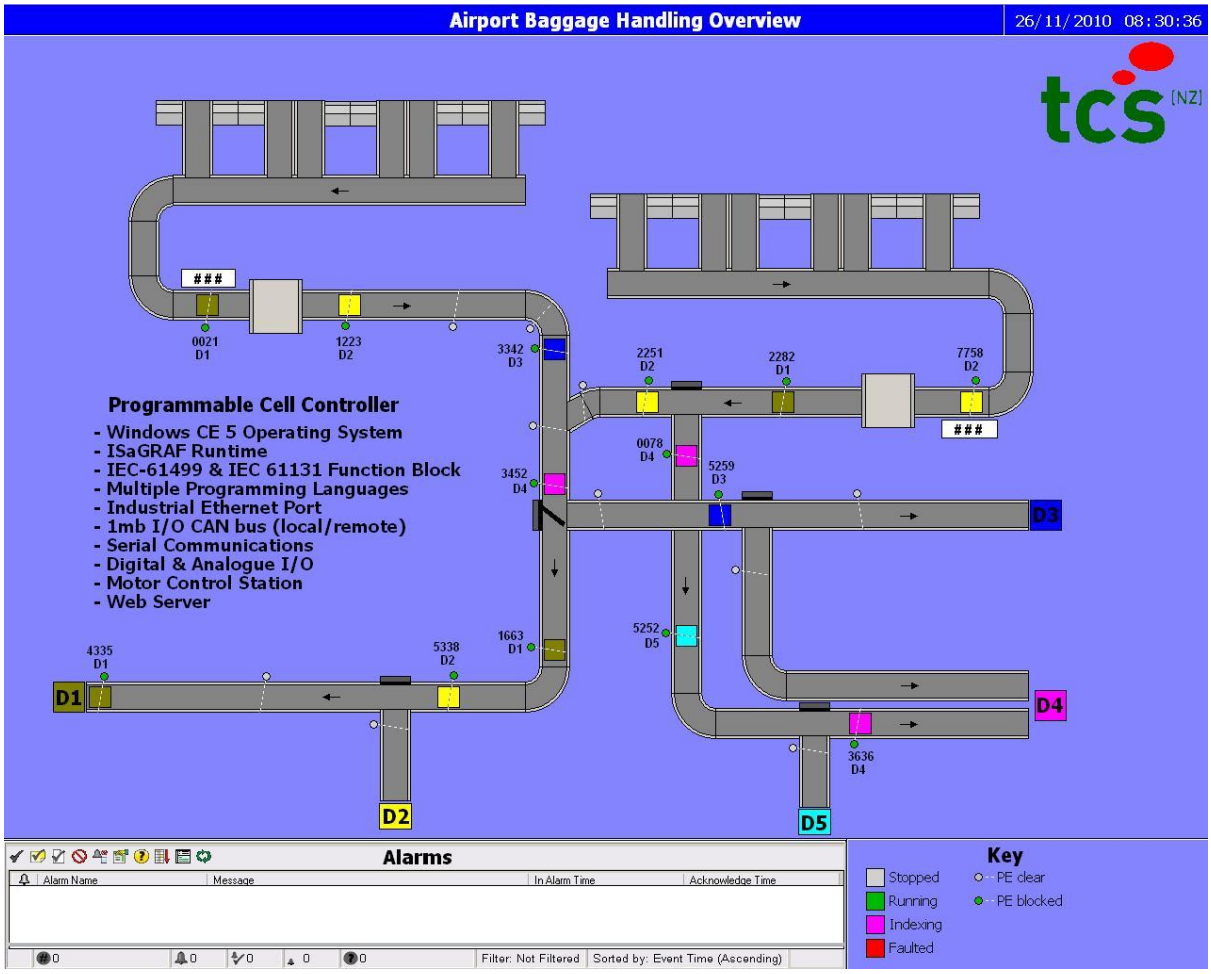


Figure 91: SCADA demonstration (Clark, 2010)

The SCADA screen shows an overview of a basic baggage handling setup as the operator of a baggage handling system would use to monitor the system. Using a SCADA screen allows operators of large material handling systems that may span different rooms and even buildings to be able to monitor the status of the system and sometimes even control various elements of the system from a single location. These systems are beneficial as they reduce the amount of staff required in a plant and in some situations increase plant safety as staff can control a system from inside a safe control room/area.

The small blocks on the conveyor belts represent the position of baggage. The demonstration panel used in the show has LED lights that light up when a bag is on one of the conveyors in the PCC controlled “cell” (one of the three conveyors).

8.4 IEC61499

For the show application, TCS used one of their IEC61499 capable CPU5 modules with no physical I/O attached to simulate the operation of the rest of the baggage handling system that was not controlled by the PCC setup. Most of the ISaGRAF software was developed by Andrew Meek of TCS. It is however included in this report because it demonstrates the functionality of the PCC when running the latest in industrial control standards and it shows that an understanding of the latest standards in industrial control standards was maintained during the development of the PCC.

IEC61499 Resource model

The following figure is a screenshot taken from the ISaGRAF workbench displaying the resources used in the baggage handling application for the Nuremberg show:

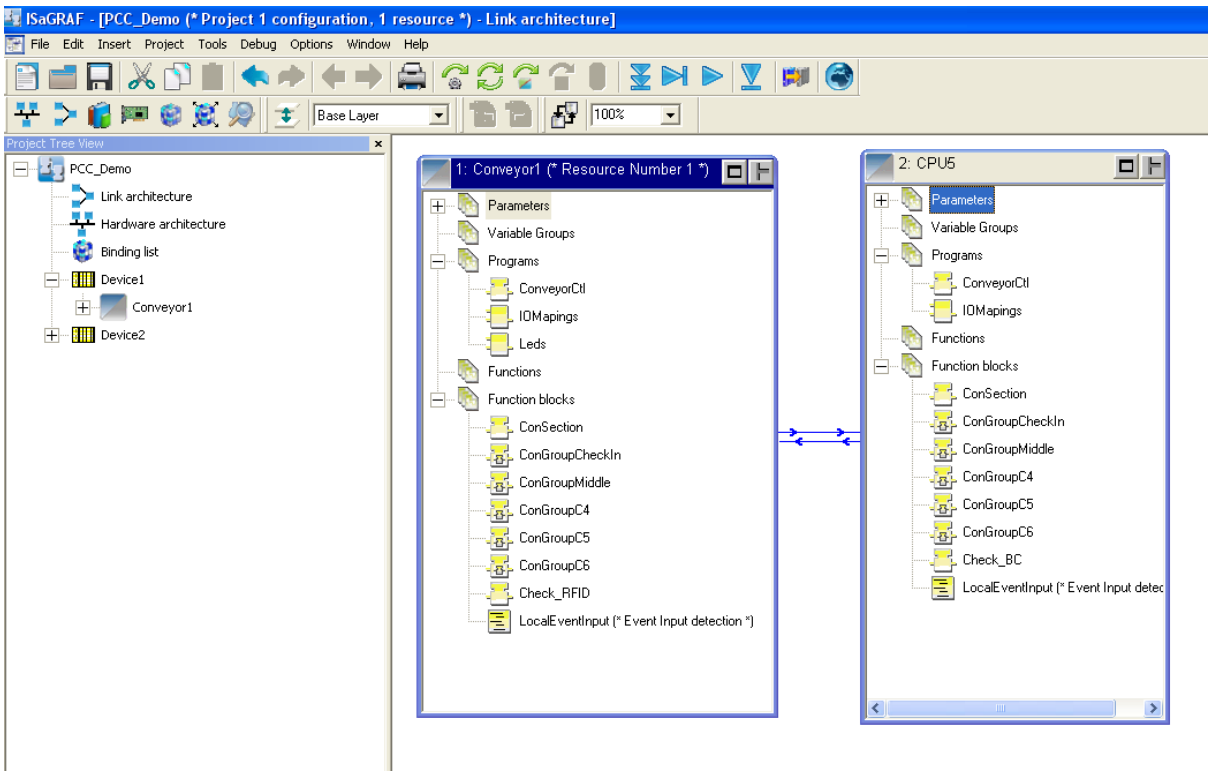


Figure 92: IEC61499 resource model (Meek, PCC_Demo, 2010)

The figure shows the two resources used in the baggage handling application. Device1 (Conveyor1) is the PCC setup containing the software to control the cell and Device2 (CPU5) is the CPU5 containing the simulation software for the rest of the baggage handling system. It can be seen that there is bi-directional communication between the two resources. Communication is done over a switched Ethernet I/P connection between the two modules.

It can be seen that some of the resources contain double ups of function blocks and programs (such as the ConveyorCtl program), this is done to allow the system to continue running even if one of the resources fails. The power of IEC61499 is that the control system is fully distributed, a production cell in a system may have programs specific to controlling the various elements inside that cell, but the control system that manages the cells at a global level is contained inside all the cells so that any single cell or PLC could fail without affecting the control of the rest of the material handling plant or production environment.

IEC61499 System model

The system model describes the controllers in the system and how they are interconnected. The system model of the Nuremberg show setup isn't really representative of the system model of a distributed baggage handling system but it can still be described in IEC61499 terms. The system is a PCC connected to a CPU5 and HMI PC over Ethernet.

IEC61499 Application model

The entire baggage handling system is represented by an application consisting of a series of interconnected function blocks. The previous section mentioned the ConveyorCtl program as being a globally defined program; this program represents the application model of the entire baggage handling system. Figure 93 shows the application model of the baggage handling system demonstrated at the Nuremberg show. The figure shows a series of interconnected function blocks demonstrating the interactions between the various conveyor belts and other elements of the baggage handling system.

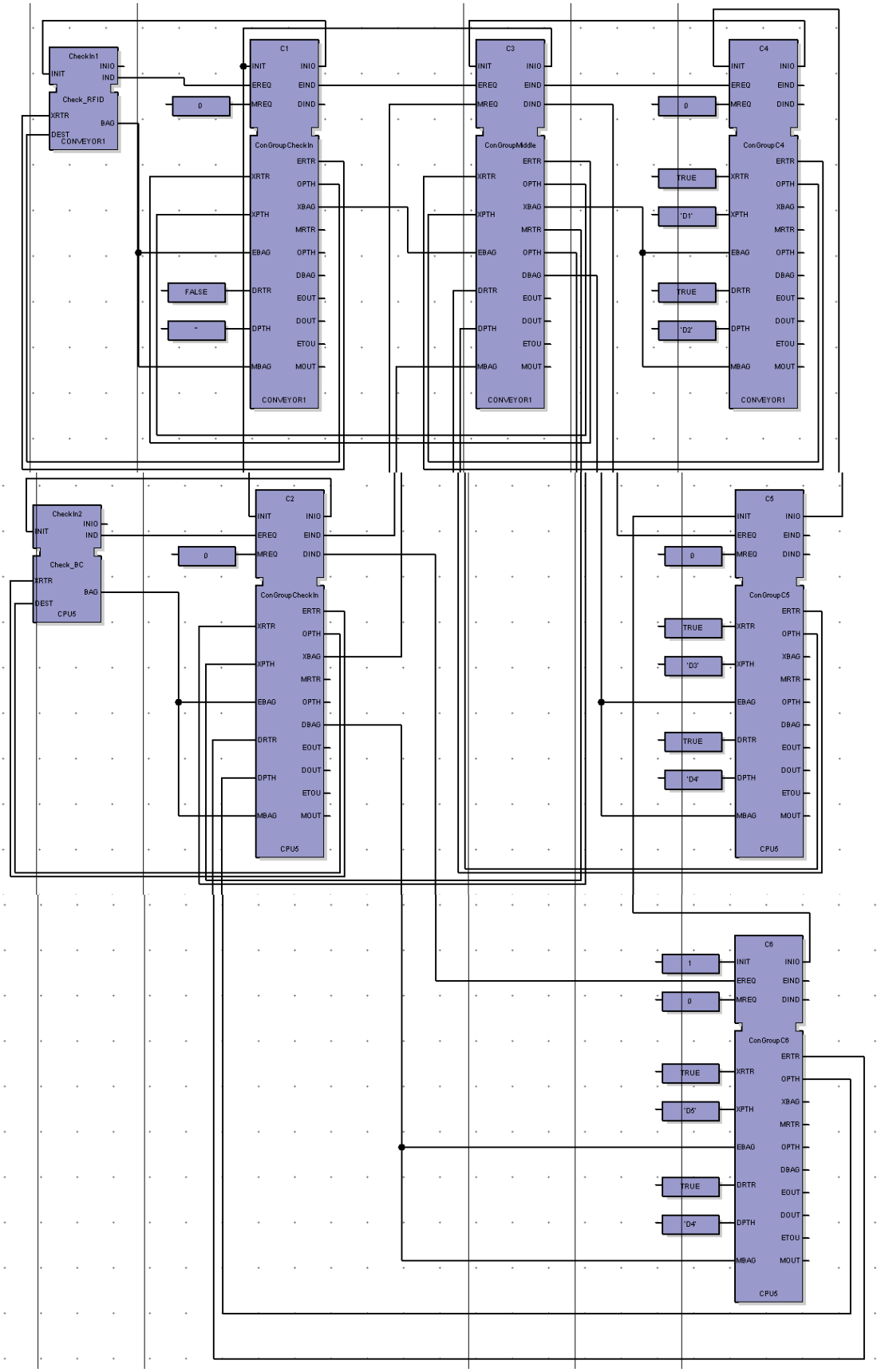


Figure 93: IEC61499 Application model (Meek, PCC_Demo, 2010)

The top three conveyor function blocks (figure 93) represent the three conveyor sections controlled by the PCC setup and the bottom three function blocks represent the rest of the baggage handling system which is simulated by the CPU5 module.

One notable feature is that the INIO outputs of the function blocks are fed back into the INIT inputs of the previous function blocks, this prevents the preceding function block from processing any data (and sending it to the outputs) until the following function block has finished processing the current data.

IEC61499 Function block model

The main (larger) function blocks in the baggage handling system represent conveyor belt sections. The following figure shows one of the conveyor belt section function blocks:

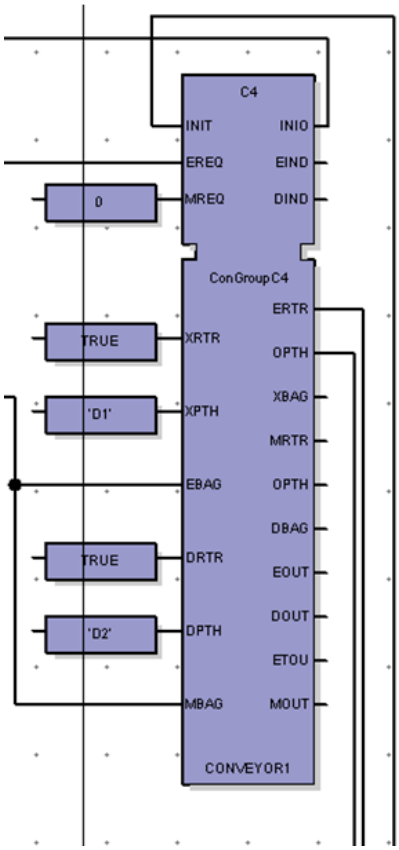


Figure 94: Conveyor belt function block (Meek, PCC_Demo, 2010)

Looking into the software from which the function block is built up, it can be seen that it contains decision logic; this is shown in the following figure:

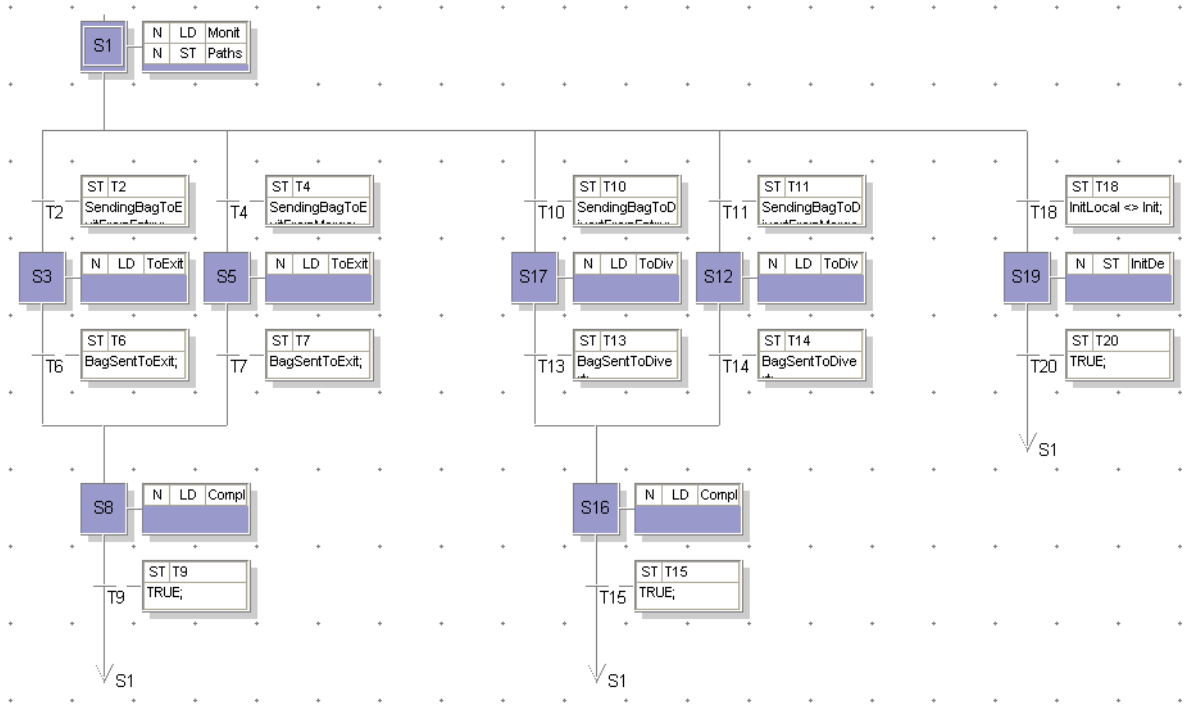


Figure 95: Decision logic inside a function block (Meek, PCC_Demo, 2010)

The decision logic is designed to carry out various tasks and populate the data outputs of the function block. The following figure shows the S8 task which has been programmed in IEC61131 ladder logic:

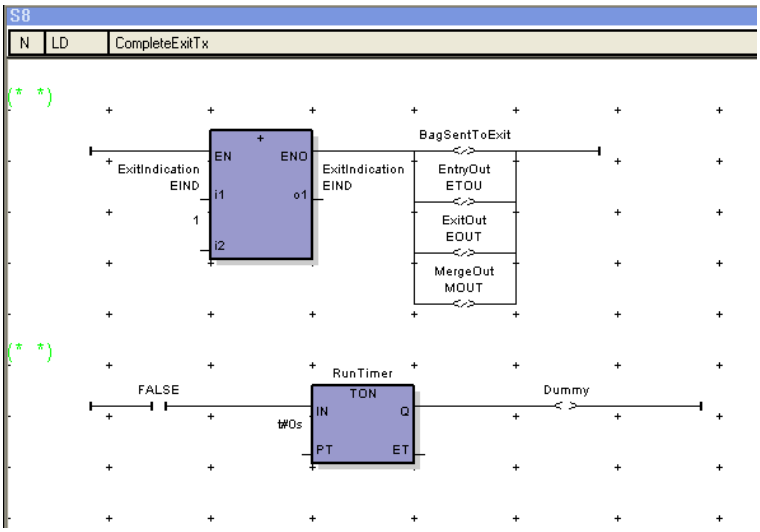


Figure 96: IEC61131 ladder logic inside a function block (Meek, PCC_Demo, 2010)

Tasks can also be programmed in structured text. The following figure shows an example of task S19 which is written in structured text:

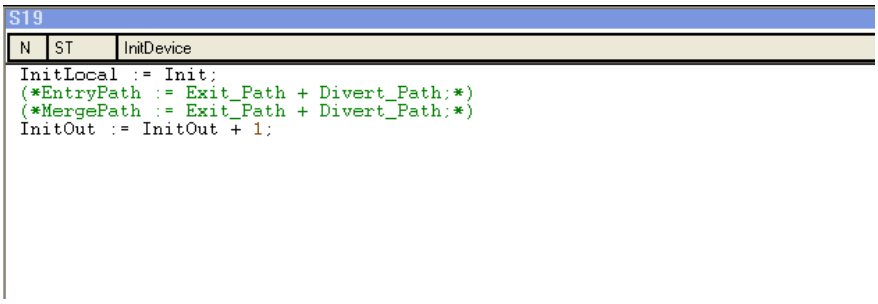


Figure 97: Structured text inside a function block (Meek, PCC_Demo, 2010)

Although the function blocks contain ladder logic and structured text, they still comply with the IEC61499 standard because the standard states that:

"A function block can represent anything from a single sensor in a plant to a complete division of a company. A function block has the following features:

- Type name and instance name
- Event input(s)
- Event output(s)
- Data input(s)
- Data output(s)
- Internal variables

Behaviour is defined in terms of algorithms and state information, these algorithms can be text based, or they can also be in function block diagram format" (Lewis, IEC 61499 models and concepts, 2008).

8.5 Public reaction

The PCC setup was shown at the SPS/IPC/DRIVES exhibition for automation technology in Nuremberg, Germany where TCS had a stand located centrally in the ISaGRAF area. Upon return from the trip to Germany Peter Tait (Director of TCS) described how ISaGRAF staff had used the PCC to demonstrate how their product was used in an OEM automation platform. Another interesting thing discovered at the trade show was that the PCC was one of the few products purpose-built for IEC61499 compliant distributed control. Peter also stated that he had received quite a bit of interest in the PCC and other TCS products from companies located in countries such as South Africa, Italy and India (Tait, 2010).

At the time of completion of this project TCS had already received two substantial orders for PCC sets. 8 PCC sets consisting of the PCC controller, a digital I/O module and serial port module were ordered by a company in South Africa for a citrus fruit handling and packing application. 4 PCC sets consisting of the PCC controller, a digital I/O module and serial port module were ordered by a New Zealand company for testing and development.

8.6 Recommendations

Creating additional modules

To complete the show setup an extra module, the motor control station, was developed. During the development of the motor control station software there were a few complications with values such as bufLength (the size of the frames transmitted across the backplane) being located in different places in the software. It would make it even easier for the developer of software for additional modules to have important values that need to be different for each module located in a single file and potentially have a function that generates these values that can be called from a separate object of software with functions specific to the new device.

A more advanced distributed control application

The demonstration setup used only a very basic setup consisting of only two controllers controlling a really basic system. The performance of the PCC could be better tested with a much larger system with more complicated control. With a larger more complex setup tests could also be carried out to see how the system reacts if a module fails or if a cell is added. The advantages of IEC61499 could have also been better demonstrated with a larger more complex system.

8.7 Discussion

The Nuremberg show setup very effectively demonstrated the application of IEC61499 in a basic setup. It also proved that the PCC platform was able to support the latest ISaGRAF software and run IEC61499. The show setup however did not demonstrate the performance of the PCC as it was a very basic application and, being a demo, it was not running very fast. Another thing the show setup did not fully demonstrate was the ease of reconfiguration of the PCC using its backplane management software, this would require quite a bit of interaction (swapping modules etc…) which would be quite difficult to demonstrate on a trade show.

8.8 Conclusion

The Nuremberg show setup successfully demonstrated the operation of the PCC as a powerful platform for an industrial control system complying with the latest industrial control standards. It demonstrated the versatility of the IEC61499 standard for use in material handling processes as well as the versatility of using the PCC as a platform to control material handling systems, by displaying the power of distributed control together with the flexibility of the PCC backplane architecture.

Project Discussion

The background research was broad and covered all the required information for the project. In an industrial situation, this research would not have been as extensive but as an academic project it was important to make it well rounded and include concepts such as IEC61499.

The project plan set out at the start of the project needed to be altered halfway through the year in order to suit a change in requirements of the parties involved and to ensure the project was finished within the set timeframe. This also demonstrated the realities of project planning where it is highly important to be able to make compromises to ensure a project is completed within the set time.

Although a highly robust backplane protocol was selected that met most requirements of TCS, it could still be improved. Further research into protocols such as the USB protocol will be needed if future design iterations require a faster backplane.

A working backplane driver has been developed to load the various I/O modules and manage their information in the registry. This driver is adequate to prove the concept but will require some adjustments, such as error reporting, before making it available to industry.

A range of I/O modules including digital I/O, serial and Ethernet have been developed. The development of the motor control station by TCS staff proved that the I/O modules can easily be adapted to create additional modules. The performance of the I/O modules has been tested and limitations such as transmit speed have been obtained. Some of these modules may need changes made to their software to match speed requirements for particular applications. All of the I/O modules still need features such as error reporting added to them to make them ready for use in industry.

The Nuremberg show setup was very effective in proving that the PCC was capable of running ISaGRAF and compliant with the IEC61499 and IEC61131 standards. Aspects that were lacking in the show setup included: demonstrating the ease of reconfiguration of the PCC, testing the performance of the I/O modules and demonstrating the flexibility of an IEC61499 distributed control system.

Project Conclusion

The following figure shows the final product obtained at the end of this project:

Figure 98: Complete PCC set with I/O modules

From left to right in the above figure:

- Motor control station: this module was developed by TCS staff and is largely based on hardware and software developed for the digital I/O module.
- PCC main controller: a design iteration of the CPU5 developed by TCS as a platform to run ISaGRAF and the drivers for the I/O modules on the Windows CE operating system.
- 8 way bi-directional digital I/O module: The first module developed for this project. This module uses a basic Windows CE stream interface driver and can be driven by the ISaGRAF software.
- Dual EIA232/EIA422/EIA485 serial port module: The second module developed for this project. This module uses a Windows CE driver that interfaces with existing serial port MDD layers. The module is compatible with serial port function blocks developed by TCS for the on-board serial port of the CPU5. A major design challenge faced was to transmit the stream of data over the frame based CAN bus.
- Dual Ethernet port module: The third module developed for the project. This module uses a miniport driver that needs to be registered with an already running NDIS driver. Challenges faced include: selecting a driver structure, loading the driver, generating interrupts and transmitting data frames over the CAN bus.

The result of the research and development carried out during this project is that TCS now has a distributed control product with a fully functioning backplane and several I/O modules that are almost ready to be used in industry. The I/O modules developed also provide a platform for additional types of I/O modules with similar features to be developed. It is anticipated that the PCC, with its more advanced features, will soon take over from the CPU5 as TCS flagship distributed control product. The amount of features made available due to the introduction of the backplane will allow TCS to customise the PCC to suit their target market and hopefully gain the market share they missed with the CPU5.

Bibliography

Altran Technologies. (n.d.). *About FlexRay and this website*. Retrieved August 2010, from FlexRay: http://www.flexray.com/

Bies, L. (n.d.). *Introduction to RS485*. Retrieved July 2010, from Lammert Bies: http://www.lammertbies.nl/comm/info/RS-485.html

CAN in Automation e.V. (n.d.). *Controller Area Network (CAN)*. Retrieved August 2010, from CAN in Automation: http://www.can-cia.org/index.php?id=16

Cena, G., & Valenzano, A. (2005). Controller Area Network: A Survey. In *The Industrial Communication Technology Handbook.* Boca Raton: CRC Press.

Chouinard, J. (2007, October). An IEC61499 configuration with 70 controllers; challenges, benefits and a discussion on technical decisions. Brossard, Quebec, Canada: ICS Triplex ISaGRAF Inc.

Clark, S. (2010, November). PCC Show Screenshot.JPG. Hamilton, New Zealand.

Ditze, M., Bernhardi, R., Kamper, G., & Altenbend, P. (2003). Porting the Internet Protocol to the Controller Area Network. *RTLIA*.

Eitman, B. (2008, June 8). *Windows CE: A Stream Interface Driver Shell*. Retrieved August 2010, from Windows CE Musings: http://geekswithblogs.net/BruceEitman/archive/2008/06/09/windows-ce-a-stream-interface-shell.aspx

Ethernet Powerlink Standardisation Group. (2009). *Technology*. Retrieved June 2010, from Ethernet Powerlink: http://www.ethernet-powerlink.org/index.php?id=4

Everything USB. (2006, August 31). *USB 2.0, Hi Speed USB FAQ*. Retrieved June 2010, from Everything USB: http://www.everythingusb.com/usb2/faq.htm

Ganssle, J. (2004, April 21). *Solving Switch Bounce Problems*. Retrieved September 2010, from EE Times Design: http://www.eetimes.com/discussion/break-point/4024956/Solving-Switch-Bounce-Problems

HN Computing. (2008). *Outcome 2: Operations on Array Data Structures*. Retrieved 2010, from HN Computing: http://www.sqa.org.uk/e-learning/ArrayDS02CD/page_20.htm

HyperTransport Consortium. (2004, June). HyperTransport I/O Technology Overview.

ICS Triplex. (n.d.). *ISaGRAF Overview*. Retrieved August 2010, from ISaGRAF: http://www.isagraf.com/index.htm

Intel. (2009). Light Peak: Overview.

Kalinsky, D., & Kalinsky, R. (2002, February 2). *Introduction to Serial Peripheral Interface*. Retrieved June 2010, from EE Times Design: http://www.eetimes.com/discussion/other/4023908/Introduction-to-Serial-Peripheral-Interface

Lewis, R. (2008). IEC 61499 models and concepts. In *Modelling Control Systems Using IEC 61499* (pp. 21-41). London: The Institute of Engineering and Technology.

Lewis, R. (2008). Introduction. In *Modelling Control Systems Using IEC 61499* (pp. 1-20). London: The Institution of Engineering and Technology.

Maxim Integrated Products. (2000, September 1). *Application Note 287 - Switch bounce and other dirty little secrets*. Retrieved September 2010, from Maxim IC: http://www.maxim-ic.com/app-notes/index.mvp/id/287

Meek, A. (2010, February). (T. Leijen, Interviewer)

Meek, A. (2010, September). IO_Test. Hamilton, New Zealand.

Meek, A. (2010, November). PCC_Demo. Hamilton, New Zealand.

Meek, A. (2010, August). V4-128-B PCC Ethernet.SchDoc. Hamilton, New Zealand.

Micrel Inc. (2007, October). M9999-102207-1.9. San Jose, California, United States of America.

Micrel Semiconductor. (2007, April 5). Micrel KSZ8842M-16 Step-by-Step Programmer's Guide. San Jose, California, United States of America: Micrel Semiconductor.

Microchip Technology Inc. (2008). UNI/O Bus Specification. Microchip Technology Inc.

Microsoft. (2003, December). *HOW TO: Use the PCI Bus Enumerator to Configure an NDIS Miniport Driver*. Retrieved November 2010, from Microsoft Support: http://support.microsoft.com/kb/320912

Microsoft. (2010, January). *RNDIS*. Retrieved November 2010, from MSDN: http://msdn.microsoft.com/en-us/library/ee484414.aspx

Microsoft. (n.d.). *Platform Builder User's Guide*. Retrieved August 2010, from MSDN: http://msdn.microsoft.com/en-us/library/aa448756.aspx

Microsoft. (n.d.). *Welcome to Windows CE 5.0*. Retrieved August 2010, from MSDN: http://msdn.microsoft.com/en-us/library/ms905511.aspx

Microsoft. (n.d.). *Windows Embedded Developer Centre*. Retrieved November 2010, from MSDN: http://msdn.microsoft.com/en-us/library/gg144992(WinEmbedded.0).aspx

Millinger, D., & Nossal, R. (2005). FlexRay Communication Technology. In R. Zurawski, *The Industrial Communication Technology Handbook* (pp. 30-1 to 30-14). CRC Press.

Ng, S. (2007, April 11). *Sakito's Blog.* Retrieved November 2010, from MSDN Blogs: http://blogs.msdn.com/b/saki/archive/2007/04/11/windows-ce-drivers-101-layered-drivers-vs-monolithic-drivers.aspx

NXP Semiconductors. (2007, June 19). I2C bus specification and user manual.

Pavlov, S., & Belevsky, P. (n.d.). *Windows Embedded CE 6.0 Fundamentals.* Microsoft Press.

Philips Semiconductors. (2000, Jan 4). Data Sheet. *SJA1000 Stand-alone CAN controller*.

Ravalia, J. (2006). Future Directions For The Windows CE Device Driver Architecture.

Renesas. (2005, November). M16C/6N Group Hardware Manual. *User's Manual*. Renesas.

Reynders, D., Mackay, S., & Wright, E. (2005). *Practical Industrial Data Communications.* Oxford: Elsevier.

Silicon Labs. (2009, December). ISOpro Low-Power Quad-Channel Digital Isolator. *Data Sheet*. Silicon Labs.

SMAR Industrial Automation. (2001). *Tutorial DeviceNet*. Retrieved 2010, from SMAR: http://www.smar.com/devicenet.asp

Tait, P. (2010, November). SPS trade show recap.

TCS (NZ) Ltd. (n.d.). *About Us*. Retrieved August 2010, from TCS (NZ): http://www.tcs-nz.co.nz/173/pages/102-overview

Teener, M. J. (n.d.). Technical Introduction to IEEE 1394.

Tilley, C. (2001, Februrary 18). *The History of Windows CE*. Retrieved August 2010, from HPC: Factor: http://www.hpcfactor.com/support/windowsce/

Toshiba. (2002, June 27). TLP281, TLP281-4. *Data Sheet*. Toshiba.

Tyson, J., & Grabianowski, E. (n.d.). *How PCI Works*. Retrieved August 2010, from How stuff works: http://computer.howstuffworks.com/pci.htm

What is a Microcontroller? (n.d.). Retrieved August 2010, from Pic Tutorials: http://www.pictutorials.com/what_is_microcontroller.htm

Yan, J., & Vyatkin, V. V. (2010). Distributed Intelligent Automation of Baggage Handling Systems with Commercial Implementation of IEC61499. *IEEE Transactions on Automation Science and Engineering*.

Zhang, P. (2008). *Industrial Control Technology.* Norwich: William Andrew.

Glossary

API	Application Programming Interface: a software library that exposes functions that can be used by other software.
BHS	Baggage Handling System.
BSP	Board Support Package: a collection of drivers put together to run Windows CE on a specific hardware platform.
CAN	Controller Area Network: a serial communications bus, see section 3.4 for further details.
CANOpen	An industrial communications protocol using the CAN bus.
CPU	Central Processing Unit used by computers to carry out calculations and process data.
CPU5	The predecessor of the PCC was called this.
CPU6	The original name of the PCC.
DeviceNet	An industrial communications protocol using the CAN bus.
DOL	Direct On Line: a term used in motor control, it is a type of control where the motor is either on at full speed (frequency of power source) or off.
Drivers	Software used to allow a microcontroller/CPU to interpret data sent to and received from external devices.
EIA232	A serial communications bus type, see section 3.4 for further details.
EIA422	A serial communications bus type, see section 3.4 for further details.
EIA485	A serial communications bus type, see section 3.4 for further details.
FIFO	First In First Out: a technique often used for buffers or queues.
FTP	File Transfer Protocol: a protocol used by computers to transmit files over a network.
HMI	Human Machine Interface: a device such as a screen to allow a human to view information about the status of the machine.
I/O	Input and Output.
IEC61131	Industrial control standard see section 1.3 for further details.
IEC61499	Industrial control standard see section 1.3 for further details.
ISaGRAF	A software package used to run control algorithms on the PCC to control inputs and outputs see section 1.5 for further detail.
M16C	The type of microcontroller produced by Renesas used in the PCC slave modules.

MAC	Media Access Control: this is often used together with the words ID and Address, this is a hardware-specific address.
MDD	Model device driver: platform independent device driver that provides a generic interface to the operating system and calls PDD functions to access the hardware.
Modbus	An industrial communications protocol using the EIA485 bus.
NDIS	Network Driver Interface Specification.
PC	Personal Computer.
PCC	Programmable Cell Controller: the product around which the project is based.
PCI	Peripheral Component Interface: a parallel data bus commonly used in computers.
PDD	Platform dependent driver: this driver contains functions specific to the hardware for which it is designed, the PDD provides functions that can be called by the MDD.
PLC	Programmable Logic Controller: the most common type of controller used in industrial control
Profibus	An industrial communications protocol using the EIA485 bus.
R&D	Research and Development.
RNDIS	Remote Network Driver Interface Specification.
SCADA	Supervisory Control and Data Acquisition: a software system that allows factory staff to monitor the plant from a centralised location.
SJA1000	A CAN controller chip produced by Phillips used in the PCC controller and several other TCS products.
Slave modules	Peripheral devices to add extra functionality such as digital I/O and serial to the PCC controller.
TCS	TCS is the name of the company sponsoring the project.
USB	A serial communications bus type, see section 3.4 for further details.
Windows CE	A version of the Microsoft Windows operating system used in the project, se section 1.4 for further details.