Ontwerp van intelligente aanstuurcircuits voor het automatiseren van de configuratie bij modulaire beeldschermsystemen

Design of Intelligent Drivers for Automatic Configuration in Modular Display Systems

Ir. Pieter Bauwens

Promotor: Prof. dr. ir. J. Doutreloigne

Proefschrift ingediend tot het behalen van de graad van
Doctor in de Ingenieurswetenschappen
Elektrotechniek

Vakgroep Electronica en Informatiesystemen
Voorzitter: Prof. dr. ir. J. Van Campenhout
Faculteit Ingenieurswetenschappen
Academiejaar 2010–2011

UNIVERSITEIT GENT

Ontwerp van intelligente aanstuurcircuits voor het automatiseren van de configuratie bij modulaire beeldschermsystemen

Design of Intelligent Drivers for Automatic Configuration in Modular Display Systems

Ir. Pieter Bauwens

Promotor: Prof. dr. ir. J. Doutreloigne

Proefschrift ingediend tot het behalen van de graad van
Doctor in de Ingenieurswetenschappen
Elektrotechniek

Vakgroep Electronica en Informatiesystemen
Voorzitter: Prof. dr. ir. J. Van Campenhout
Faculteit Ingenieurswetenschappen
Academiejaar 2010–2011

UNIVERSITEIT GENT

ISBN nummer:

NUR code:

Depot nummer:

Promotor:

Prof. dr. ir. J. Doutreloigne

Onderzoeksgroep CMST
Vakgroep Electronica en Informatiesystemen
Technologiepark 914A
B-9052 Zwijnaarde

Dankwoord

Het is een apart gevoel. Vier jaar lang lijkt het einde nog ergens in een verre, onbekende toekomst te liggen, en dan opeens is het er. De laatste tests zijn afgelopen, de laatste hoofdstukken geschreven. Ietwat onwennig zet je de eerste stapjes uit het mentaal isolement waar je de voorbije maanden vertoeft hebt. Je kijkt nog even achterom en slaakt een tevreden zucht: "Het is af.".

Dit gevoel gaat gepaard met een sterke dankbaarheid voor iedereen die op één of andere manier een steentje heeft bijgedragen. De grootste stenen komen van André Van Calster en Jan Doutreloigne. Zij zijn degenen die mij de kans hebben aangeboden om dit doctoraat te beginnen. Met de verscheidene etentjes en andere evenementen zorgde André voor een sterke sociale band onder de collega's, één van de pijlers waarop de aangename CMST-sfeer steunt.
Met Jan als promotor weet je dat je het goed getroffen hebt. Je wordt ten volle gesteund in de keuzes die je maakt en je kan ervan op aan dat, als je even geen uitweg ziet, Jan met veel plezier (en het nodige geduld) je de juiste richting zal wijzen. Aan beiden, bedankt!

Hoe eenzaam een doctoraat ook mag lijken, met de juiste collega's merk je daar helemaal niets van. Een speciale bedanking gaat dan ook naar mijn bureaugenoten. Ann, je hebt me als groentje onder je vleugels genomen en ook erna kon ik altijd bij je terecht. Bedankt en veel plezier met je kleine Silke! Stefaan en Benoit, bedankt voor de vele interessante discussies, gaande van electronica en fysica tot religie en ethiek. Vincent wil ik in eerste instantie bedanken als onze persoonlijke Linux-helpdesk, maar verder ook nog voor het jaarlijkse optreden met het GUSO-orkest. Jodie, je passie voor muziek viel in goede smaak, en gaf zo nu en dan de motivatie zelf nog eens aan de piano te gaan zitten. Dominique en Liang, misschien niet de grootste babbelaars in de hoop, maar twee sympathieke jongemannen die zeker hebben bijgedragen aan de sfeer in de bureau. Bedankt!

Het grootste deel van voorgenoemden hoort bij een groep die ook nog een speciale vermelding verdient: De Designers. We vormen misschien een kleinere groep binnen CMST, maar samen staan we sterk. Een band die nog versterkt wordt door het sporadisch "*ribbekes fretten*".
Ook de andere collega's (binnen en buiten CMST) krijgen een hoop bedankjes, voor hun al dan niet werkgerelateerde bijdragen. Wim Meeuws voor de hulp met

de Cadence en Synopsys programma's, Lieven voor het layouten van de bordjes, Bjorn voor zijn ultieme handigheid bij het solderen, Nadine voor haar spontaniteit, Erwin voor zijn leuke verhalen, Wim voor zijn schaterlach, ...en iedereen anders, bedankt!

Ook buiten de werkomgeving zijn er heel wat mensen om te bedanken. Zij zorgen dan misschien niet voor het bijdragen van de professionele stenen, maar vormden eerder de mortel die de stenen bijeen hield. Bart, Jasper en Leen, Bas en Gudrun, Steven,..., bedankt om goede vrienden te zijn! Ook aan alle anderen die ik misschien net iets te weinig kan zien.
Aan de mensen van de Chinese les wil ik dit nog kwijt:
同学们和万老师，我们一起学汉语学了已经四年了。你们是好朋友！每星期一晚上，我很高兴去上课。谢谢你们！

En laatst maar niet minst, mijn ouders en zusje. Dankzij hen sta ik waar ik nu sta, en geen woorden kunnen uitdrukken hoe dankbaar ik daarvoor ben. Bedankt voor het steunen, het vertrouwen, voor alle evidente en minder evidente zaken. Zusje, lieve zus, hoe oud je ook mag worden, je zal altijd mijn klein zusje blijven, bedankt voor de levendigheid die je brengt. Je weet het misschien niet, maar je hebt me veel geleerd!

Kortom, bedankt!

Contents

List of Tables

List of Figures

Samenvatting

We zijn er ons allen van bewust dat we in een wereld leven waar displays niet uit weg te denken zijn. Ze behoren tot elk aspect van ons leven. Tegelijk worden de eisen die we stellen aan deze displays steeds strenger. We willen een hogere resolutie, een beter contrast, een grotere flexibiliteit (zowel fysische flexibiliteit als aanpasbaarheid). Modulaire display systemen kunnen een handje helpen de displays van enkele van deze vereisten te voorzien. Een modulair display systeem is een soort display waarbij het display zelf is onderverdeeld in (al dan niet volledig) onafhankelijke modules. Elke module heeft zijn eigen (klein) display en alle modules werken tesamen om de illusie van één groot display te creëren. Intelligente modulaire display systemen geven nog wat extra functionaliteit aan deze displays. Dit doctoraat stelt vier van zo'n intelligente systemen voor.

Hoofdstuk 2 geeft een overzicht van de bestaande beeldschermtechnologieën en hun eigenschappen. Het verschil tussen actief en passief matrix aansturing wordt er ook aangehaald. Er kan aangetoond worden dat, wanneer passieve aansturing gebruikt wordt bij een aantal beeldschermtechnologieën, er altijd een afweging zal zijn tussen de bereikbare resolutie en contrast (of helderheid). Als je het aantal rijen in een display wilt verhogen, zal dit ten koste gaan van het contrast. Een modulair display kan dit verhelpen. Elke module heeft zijn eigen onafhankelijk display. Het verhogen van het aantal rijen kan eenvoudig gebeuren door het toevoegen van enkele modules. Dit zal uiteraard geen invloed hebben op het contrast van de individuele moduledisplays.

Een andere handige eigenschap van modulaire displays is dat ze kunnen gebruikt worden voor de zogenaamde free-form displays. Dit zijn displays waarvan de gebruiker de vorm grofweg zelf kan bepalen. Hun vorm is niet vast en kan worden veranderd. In zijn meest eenvoudige vorm zijn dit displays die schaalbaar zijn. Modules kunnen hierbij gebruikt worden om een zeer groot display te bouwen met een zelf te kiezen beeldverhouding. In een meer algemene vorm kan dit betekenen dat de precieze vorm van het display kan aangepast worden. Hoofdstuk 3 geeft enkele voorbeelden van systemen die dit trachten te bereiken. Een modulair display systeem kan hier ook voor geschikt zijn. De modules worden aan elkaar gekoppeld om zo een willekeurige vorm te creëren, maar dat toch werkt als één geheel. Hiertoe zullen de modules met elkaar moeten communiceren. Alle modules zijn identiek en zullen via een zelfde datalijn met de microcontrol-

ler verbonden zijn. Om de verschillende modules uit elkaar te houden, moet er met adressen gewerkt worden. Maar net omdat alle modules identiek zijn (geen hardgecodeerde adressen) zal deze adrestoewijzing via software moeten gebeuren. Dit vereist enige intelligentie van het systeem. Maar dat is dan ook het onderwerp van dit doctoraat.

Aangezien de modules verbonden zijn in een netwerk, lijkt het logisch een kijkje te nemen naar de beschikbare netwerkprotocollen. Dit gebeurt in Hoofdstuk 4. Het zal blijken dat geen van die protocollen onze noden volledig vervult en dat het efficiënter is onze eigen specifieke protocollen te definiëren. In het algemeen zal zo'n protocol bestaan uit twee fasen. De eerste fase is de initialisatiefase, waar de modules hun adres verkrijgen. De tweede fase is de normale werkingsfase. Elke module heeft een bypass tussen de ingang en uitgang. Deze bypass zal inactief zijn tijdens de initialisatiefase, waardoor een module alleen maar zal kunnen communiceren met zijn directe buren. Tijdens de normale werkingsfase wordt de bypass actief, zodat alle modules op dezelfde datalijn verbonden worden. In deze normale werkingsfase zullen alle drivers reageren op parameterdata en beelddata. De parameterdata bevat informatie over de te gebruiken refreshrate, het aantal rijen en kolommen die de module moet aansturen en specifieke beeldschermeigenschappen (bv. de instelstroom voor een LED display).

We starten in Hoofdstuk 5 met het beschrijven van de eerste driver, de 'modular display driver'. Deze driver is bedoeld voor het aansturen van modules die verbonden zijn in een bus netwerk. De modules zijn verbonden als een daisy chain. Dit wil zeggen dat elke module één ingang en één uitgang heeft. De uitgang van de ene module is de ingang van de volgende. Dit is een zeer eenvoudig netwerk, en heeft bijgevolg een zeer eenvoudig protocol. Elke module krijgt een adres van de vorige module in de rij en stuurt het volgende adres door naar de volgende module. Vanwege deze eenvoud, kunnen we hier nog niet echt spreken van een free-form display driver. Deze driver is vooral bedoeld voor het verhogen van de multiplexeerbaarheid in passive matrix beeldschermen.

De driver uit Hoofdstuk 6, de 'improved modular display driver', is een uitbreiding op de vorige driver. Hij is geschikt voor modules in een mesh netwerk. In zo'n netwerk heeft elke module vier in- en uitgangen: aan elke zijde één. Met deze driver is het mogelijk een eenvoudig free-form display te maken. De manier waarop de modules hun adres verkijgen lijkt sterk op de manier die hierboven beschreven is. Een module krijgt een adres toegestuurd op één van zijn ingangen en zal aan zijn andere uitgangen een berekend adres uitsturen. We moeten echter wel opletten voor datalussen. Deze treden op wanneer er zich meer dan twee paden tussen twee modules bevinden. Data, zoals adressen, kunnen vast komen te zitten in zo'n lus. In dit hoofdstuk wordt dan ook een primitieve manier van data routing aangehaald.

De eerste echte free-form modular display driver krijgen we te zien in Hoofdstuk 7. Deze wordt ook gebruikt in mesh netwerken, dus er kunnen dezelfde

displays gemaakt worden als met de vorige driver. Deze driver zal echter het systeem van wat meer functionaliteit voorzien. Hij zal het systeem in staat stellen om de gecreëerde vorm van het display te detecteren, hoe de verschillende modules geconnecteerd zijn. Dit wordt dan weergegeven in een Grafische User Interface (GUI) op een computer. Aangezien het netwerk niet veranderd is ten opzichte van het vorige, kan een adres van een module nog steeds berekend worden door zijn buren. Om de GUI te laten weten welke modules er aanwezig zijn (i.e. hoe het display eruit ziet), volstaat het dat alle modules hun adres naar de microcontroller sturen. Uiteraard zal dit op een gecontroleerde manier moeten gebeuren. De displayvoorstelling in de GUI is real-time. Wanneer er na de initialisatiefase modules worden toegevoegd of verwijderd, zal de GUI geupdated worden, terwijl het display zelf zonder problemen verder werkt. Dit vereist een grotere complexiteit in het routeringsalgoritme. Om de problemen met datalussen tegen te gaan, zal elke module slechts naar één input kijken (gekozen tijdens de initialisatiefase). Dit zal een databoom creëren waarbij elke module data ontvangt van één parent node en die zal doorsturen naar nul tot drie child nodes. Maar wanneer een parent node wordt verwijderd, betekent dit nog niet dat de child nodes geen data via een andere weg zouden kunnen krijgen. Het datarouteringsprotocol moet er dus voor zorgen dat de databoom wordt aangepast als een module verwijderd wordt. Wanneer er een nieuwe module toegevoegd wordt, moet deze uiteraard ook aan de databoom worden toegevoegd.

De laatste driver, de 'improved free-form modular display driver' wordt in Hoofdstuk 8 besproken. Deze heeft dezelfde eigenschappen als de vorige driver, zijnde dat de vorm van het display kan gedetecteerd worden en dat er modules kunnen toegevoegd en verwijderd worden na initialisatie. Hetgeen deze driver nog wat complexer maakt is dat de modules in een veel uitgebreider netwerk kunnen geconnecteerd worden. Ze kunnen op om het even welke manier verbonden worden, ongeacht de oriëntatie van de modules. Het algoritme laat ook toe om met driehoekige, vijfhoekige, zeshoekige, ... modules te werken. Deze extra vrijheidsgraad opent de weg naar een hele nieuwe reeks van mogelijke displays. Zo zouden ze verbonden kunnen worden om bijvoorbeeld een 3D gevormd display te maken. Deze extra vrijheidsgraad betekent ook dat de communicatie een stuk complexer zal zijn. Terwijl bij de vorige driver de adressen konden bepaald worden aan de hand van de adressen van de naburig modules, kan dit nu niet langer. De modules kunnen op geen enkele manier weten hoe ze precies verbonden zijn, en zijn dus niet in staat een zinnig, uniek adres te berekenen voor hun buren. De enige entiteit die adressen kan bepalen is de microcontroller zelf. Aangezien deze slechts één module tegelijk zal kunnen behandelen, is er een protocol nodig dat een point-to-point verbinding tussen elke module en de microcontroller voorziet. Aangezien nu de adressen geen informatie meer bevatten over de locatie in het display, zal er andere data nodig zijn om te detecteren hoe de modules verbonden zijn.

Elk van die vier vernoemde drivers is geïmplementeerd in VHDL en is getest op modules met een FPGA. De eerste driver is ook getest met een cholesterisch display. De GUI is geprogrammeerd in Visual C++ en een microcontroller voorziet de directe communicatie met de modules. De resultaten van deze testen zijn te vinden in de respectievelijke hoofdstukken. Het blijkt dat alle drivers en de geïmplementeerde protocollen correct werken. Zoals verwacht is de initialisatietijd bij de complexe tegenhangers groter dan bij hun eenvoudige tegenhangers. Deze initialisatietijd is sterk afhankelijk van het aantal modules enerzijds, maar anderzijds kan ook de precieze moduleconfiguratie deze tijd sterk beïnvloeden.

Na deze tests zijn de vier drivers geïmplementeerd in een chip, de FrIIDoM driver. Hiervoor is de C35 CMOS technologie van AMS gebruikt. Dit verloop kan gevolgd worden in Hoofdstuk 9. Deze driver is ontwikkeld om een LED display van maximaal 8×8 LEDs aan te sturen. Het heeft geïntegreerde LED drivers (stroombronnen voor de kolommen, schakelaars voor de rijen), een on-chip klok en een power-on reset. In de parameterdata is een byte voorzien om de LED stroom met 8-bit precisie aan te passen. De stroombron is berekend om maximaal 50mA te kunnen uitsturen. Dit betekent ook dat de schakelaar voor de rijen een stroom van 400mA moeten kunnen geleiden.

Om de FrIIDoM driver te kunnen testen zijn testbordjes en nieuwe modules gemaakt (zie Hoofdstuk 10). De stroombronnen en schakelaars werden op het testbord getest. Wanneer er parameters werden verzonden met een stijgende waarde voor de instelstroombyte, steeg de uitgangsstroom van de stroombronnen lineair, met een maximumwaarde van 50mA. Ook de schakelaars werden getest. Deze werden verplicht een steeds grotere stroom te geleiden. De stroom kon worden opgedreven tot 360mA (maximum waarde te bereiken met meetopstelling) zonder dat de schakelaars een krimp gaven. De functionaliteit van de drivers kon getest worden met de modules. Bij alle drivers werkte de initialisatiefase zonder problemen. Bij de eerste drivers werden de adressen correct doorgegeven, bij de laatste twee drivers kregen de modules het juiste adres en werd de displayvorm juist gedetecteerd. Wanneer er modules werden toegevoegd en verwijderd, werd dit correct weergegeven in de GUI. Modules die hun data verkregen via zo'n verwijderde module kregen een nieuw pad naar de microcontroller. Alle drivers reageerden ook op een goede manier op de verzonden parameters. De refreshrate, LED stroom en gebruikte rijen en kolommen konden worden aangepast. Buiten de derde driver konden ook de datasequenties goed worden verwerkt, zodat het beeld dat in de GUI gemaakt werd, kon worden afgebeeld op het display. De derde driver bleek wat problemen te hebben bij het detecteren van adressen in een datasequentie. Mogelijke oorzaken (en verdere aanpakken) zijn te vinden in dit hoofdstuk.

Summary

We are all aware that we live in a world that is filled with displays. They are present in every corner of our lives. The properties we require for those displays are becoming more and more stringent. High resolution, high contrast, high flexibility (physical flexibility as well as customizability). Modular displays can help provide for some of those properties. A modular display system is a type of display where the display is divided into several (whether or not completely) independent modules. Each module has its own private (small) display and all modules work together to give the impression of being one big display. Intelligent modular display systems provide some extra functionality to those displays. This Ph.D. describes four different versions of such an intelligent modular display system.

Chapter 2 gives an overview of some important display technologies and their properties. It also provides the basic insight in the difference of passive and active matrix display driving. It can be calculated that with passive matrix driving and with certain types of display material, there will always be a compromise between the achievable resolution and contrast (or overall brightness). If you want to increase the number of rows of such a display, you will have to decrease the contrast. A modular display system can overcome this. Every module has its own display, independent from the others. Increasing the number of rows can be done by simply adding some more modules. This will obviously have no effect on the contrast of the individual module displays.

Another aspect of modular display systems is that they can be used to create free-form displays. These are displays that can be created 'on the go'. They don't have a fixed shape, it can be changed. In its simplest form, this is just a scalable display, where modules are used to create one, very big display, or where the aspect ratio of the display can be changed. In its more complex form, this could mean that the precise shape of the display can be changed. Chapter 3 gives some examples of systems that allow the display to be created with an irregular shape. A modular display system can also be used for this. Modules can be connected to each other to create a random display shape, that works as one display. In order to achieve this, the modules will need to communicate with each other. Every module is exactly the same and they will be connected to the microcontroller with the same data line. To be able to distinguish the modules, they all need to be assigned an

address. We do not want to hardcode an address, so it will have to be assigned through software. This needs some intelligence. But then again, this is the topic of this book.

Since the connected and communicating modules are in fact a communication network, we'll take a look at some of the popular network protocols in Chapter 4. We will conclude that none of these protocols will fit our needs exactly, and that it is more efficient to create our own protocols. In general, the protocols consist of two phases. The initialization phase, where the addresses are being assigned, and the normal operation. The modules all have a bypass (connecting input and output), which is inactive during the initialization phase (modules can only communicate with their neighbors) and active during the normal operation (all modules are connected to the same data line). In the normal operation, all drivers are able to read parameter and image data. Parameter data includes refresh rate, number of rows and columns used in the display and specific display properties (like current strength in a LED display).

In Chapter 5 we start off by describing the first driver, the modular display driver. This driver is meant for modules connected in a bus network. They are connected as a daisy chain. Every module has one input and one output. The output of one module is connected to the input of another. This is a very simple network, and has a very simple protocol. Each module will receive an address of the previous module, and send the next address through to the next module. Due to the simplicity of this network, it can't really be used for free-from displays. It is helpful for the multiplexability problem of passive matrix displays, though.

The driver from Chapter 6, the improved modular display driver, is an extension of the previous driver. This driver is suited for mesh networks. In this network, every module has four inputs and outputs, one on each side. With this driver, it is possible to create a basic free-form display. The method for assigning an address is similar to the one above. A module receives an address on one of its gates, and will send calculated addresses to the other gates. We must take care, though. In a mesh network there can be data loops where there is more than one path between modules. This can cause data (e.g. addresses) to be stuck inside a loop. In this chapter we'll show a primitive way of data routing.

The first real free-form modular display driver is discussed in Chapter 7. It is also suited for mesh networks and the same displays as with the driver above can be created. But this driver adds some more complexity, some more intelligence to the system. It provides the means for the system to recognize how the modules are connected to each other, what the display looks like. The detected display shape is then represented in a Graphical User Interface (GUI) on the PC. Since we have the same network as above, modules can still calculate addresses for adjacent modules. For the GUI to know which modules are present, it suffices for all modules to send their own address to the microcontroller. This will have to happen in a controlled way of course. The representation in the GUI is real-time.

When modules are added or removed after the initialization phase, it is detected and the display continues working. A more complex routing protocol had to be implemented. The data loops are coped with by only looking at one module for data input, chosen during initialization. This creates a spanning tree, where every module has one parent node and none to three child nodes. But when a parent node is removed, it does not necessarily mean that its child nodes cannot receive data through another path. The data routing protocol should rebuild the spanning tree when that happens. Also, when new modules are added, they have to be added in the tree.

The last driver, the improved free-form display driver, is presented in Chapter 8. This driver has the same properties as the driver from Chapter 7. The created display shape can be detected and shown in a GUI and modules added or removed after initialization will be detected and will not interfere with the operation of the display. However, the modules are no longer limited to a mesh network. They can be connected any way you like, without regards of the orientation of the module. The algorithms also allows triangular, pentagonal, hexagonal, ... modules (they have to be of the same type within one display, though). This extra degree of freedom creates a whole new range of displays that can be created. Modules can be connected so that they create a 3D-shaped display for example. This extra degree of freedom also means that the communication protocol has become more complex. While all previous driver could receive their address from a adjacent module, this is no longer the case using this setup. There is no way for the modules to know how they are connected, so there is no means to calculate a meaningful, unique address. Every address should come from the microcontroller itself. The microcontroller can only handle one module at a time, so there is need for a connection set-up between every module and the microcontroller. Since the addresses will then no longer hold information about the location of the modules, extra information is needed to derive the way the modules are connected.

Every one of those drivers is implemented in VHDL and tested with FPGA-equipped modules and a LED display. The first driver was also used to drive a Cholesteric LCD. The GUI was created with Visual C++ and a microcontroller provided the immediate communication with the modules. The results of these tests can be seen in their respective chapters. It seems that every driver works as expected. It shows that, as expected, displays using the more complex drivers take longer to initialize, though their exact initialization time is dependent on the number of modules (obviously) but also how they are connected.

After the successful tests, the four drivers were implemented in one chip, the FrIIDoM driver. For this, the C35 CMOS technology from AMS was used. Chapter 9 shows this process. This driver is created for a maximal 8×8 passive matrix LED display. It has implemented LED drivers (current sources for the columns, switches for the rows), on-chip clock and Power-On Reset. In the parameter

stream, the LED current can be adjusted (8-bit precision). The current sources can output a 50mA current. This means that the switches on the row electrodes must be able to withstand 400mA.

To test the FrIIDoM driver, a new test board and modules were made (See Chapter 10). On the test board the current sources and switches were tested. Sending an increasing current-parameter byte shows a linearly increasing output current, reaching 50mA on its highest point. The row switches were tested up to 360mA (maximum using our test setup), without flinching. The functionality of the drivers was tested with the modules. The initialization phase of all drivers works without a problem. Using the first two drivers, all modules receive an address correctly. Using the last two drivers, the addresses are correctly distributed and the display shape detected. When modules were added they were added in the GUI. When modules were removed, they disappeared from the GUI and, importantly, the modules that depended on that module to receive data were rerouted. All drivers were successful in processing the sent parameter data, changing the refresh rate, LED current and display size. Apart from the third driver, all drivers could process incoming data correctly, displaying the data from the GUI on their screen. The third driver had some difficulties detecting its own address in a data stream. Possible explanations (and further approaches) for this are found in this chapter.

List of Abbreviations

Notation	Meaning
AC	Alternating Current
AM	Active Matrix
ASIC	Application Specific Integrated Circuit
ATPG	Automated Test Pattern Generation
CAD	Computer Aided Design
ChLC(D)	Cholesteric Texture Liquid Crystal (Display)
CMOS	Complementary MOS
CRT	Cathode Ray Tube
DC	Direct Current
DFT	Design For Testability
FPGA	Field-Programmable Gate Array
FPS	Frames Per Second
FTDI	Future Technology Devices International
GUI	Graphical User Interface
I^2C	Inter-Integrated Circuit
IP	Internet Protocol
ITO	Indium Tin Oxide
LC(D)	Liquid Crystal (Display)
LCoS	Liquid Crystal on Silicon
LSB	Least Significant Bit
MLA	Multi Line Addressing
MOSFET	Metal-Oxide-Semiconductor Field-Effect Transistor
MSB	Most Significant Bit
(O)LED	(Organic) Light-Emitting Diode
PDLC	Polymer Dispersed Liquid Crystal
PDP	Plasma Display Panel
PET	PolyEthylene Terephthalate
PM	Passive Matrix
POR	Power-On Reset
RMS	Root Mean Square
RS-232	Recommended Standard 232
SPI	Serial Peripheral Interface

(S)TN	(Super) Twisted Nematic
STP	Spanning Tree Protocol
TFT	Thin Film Transistor
USB	Universal Serial Bus
VHDL	VHSIC Hardware Description Language
VHSIC	Very-High Speed Integrated Circuit

Man's greatest asset
is the unsettled mind.

Isaac Asimov (1920-1992)

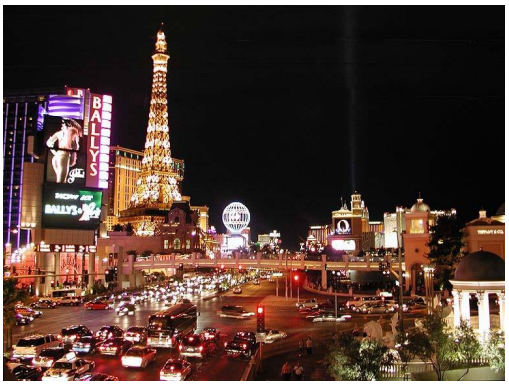
1 Introduction

1.1 The search for an intelligent modular display system

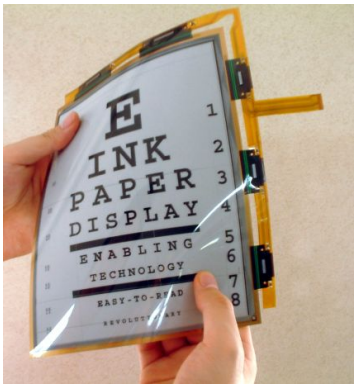
We live in a world where displays are becoming more and more important, and ubiquitous. Almost everywhere we go, we can see a display enlighten us with it's information (pun intended). Whether it's a LED display giving us the latest traffic information, an LCD screen showing the newest promotions in your favorite fast food restaurant or maybe an e-book showing the newspaper of today, displays are present in almost every aspect of our lives. This rise is far from over and in the coming years we'll see a lot more display applications seeping into our daily environment.

Our demands on these displays also keep increasing. We want some of them to be extremely big (Figure 1.1a), others we like to keep small, but with a good resolution and contrast (Figure 1.1b). These two applications don't seem to have an awful lot in common, but there is something that binds them.

When using certain display technologies, there is always a compromise between the achievable resolution and contrast. When you want to increase the one, you automatically decrease the other (See Chapter 2 for a detailed explanation). If you want to create a decent e-paper application (with properties comparable with regular paper) both resolution and contrast must be fairly high. A piece of paper printed with a standard printer, already has a resolution of 300 dpi or more.



(a) View of the strip in Las Vegas, by night



(b) Electronic paper by E Ink

Figure 1.1

Compare that to the 72-96 dpi a standard computer monitor has to offer. Also the contrast cannot be too low. A typical newspaper has a contrast of 5:1 to 8:1. E-paper is normally an inherent reflective display (emitted light is reflected ambient light, see Chapter 2), where the beautiful paper-like white can be hard to achieve. One of the solutions for the displays coping with this problem (resolution vs. contrast) is the main topic of this book: modular display system. If the display is divided in several modules, the resolution can be disconnected from the contrast (See Chapter 3).

And why this beautiful picture of Las Vegas, you ask? If you have been in Las Vegas, you were no doubt overwhelmed by the sheer size of some of the displays. It is quite obvious that these displays aren't made in one piece. You've probably heard me coming from a mile away, but again, the solution lies with modular displays. Several smaller displays are tiled to create one, very large, display.

Some other, new applications, could benefit from modular display systems. Especially in the range where the displays move away from the traditional rectangular shape.

Figure 1.2a shows a LED display embedded in clothing, by Lumalive, Philips. While this might as well be a rectangular display, things change if you also want to display something on the sleeves and front, if you want your entire jacket to behave as one display. Here, a modular display system could also come in handy. The shape of the display is that of the jacket, and can be created by connecting several modules together.

Modular display systems could also allow us to create almost any shape we want. Take the magnificent illuminated globe during the Beijing 2008 Summer Olympics for example (Figure 1.2b). This effect was generated with projection

(a) Display embedded in clothing, by Lumalive Philips

(b) Illuminated globe during the Beijing 2008 Olympics

Figure 1.2

systems, but (probably on a smaller scale) a similar display could be created using the correct modular display system.

But, in order for any of these systems to work properly, some intelligence is needed. What if every module is identical (no hardcoded address), how will the addresses be assigned? What if some modules are missing or broken? Is there need for some routing algorithm? What if modules are added or removed during operation? Can the display configuration be detected? And most of all, how can it be done with as little as possible intervention of the user? The main topic of this Ph.D. is to design some of those intelligent modular systems. In the end, four systems were designed, one more intelligent than the other. They are thoroughly discussed in four chapters (Chapter 5 to 8). They were all implemented in VHDL and tested on an FPGA. After proven successful, a custom chip was designed with the four drivers integrated (Chapter 9). Chapter 10 gives a detailed overview of the final results of the created display systems.

1.2 Publications

Papers published in a SCI-journal

- P. Bauwens and J. Doutreloigne, *Drivers for Free-Form Modular Displays*, Journal of the Society of Information Display, Vol. 18(3), 2010, pp. 235-239

- P. Bauwens, A. Monté, W. Christiaens, J. Doutreloigne and J. Vanfleteren, *Improved Passive-Matrix Multiplexability with a Modular Display and UTCP Technology*, Displays, Vol. 30(2), 2009, pp. 71-76

- A. Monté, P. Bauwens and J. Doutreloigne, *New driving scheme for intelligent power-efficient high-voltage display drivers*, Journal of the Society of Information Display, Vol. 16(11), 2008, pp. 1171-1180

Papers presented at international conferences listed as P1-publications

- P. Bauwens, J. Doutreloigne and A. Monté, *A Driver for Modular Passive-Matrix Displays*, Proceedings of the 14th International Display Workshops (IDW'07), 2007, Sapporo, Japan, pp. 1317-1320

- P. Bauwens and J. Doutreloigne, *A Free-Form Modular LED-Display Driver*, Proceedings of the 29th International Display Research Conference (IDRC'09), 2009, Rome, Italy, pp. 287-290

Papers presented at international conferences listed as C1-publications

- P. Bauwens and J. Doutreloigne, *A New Driver for an Intelligent Modular Display System*, SID2010 Digest of Technical Papers, 2010, Seattle, USA, pp. 1397-1400

- S. Maeyaert, B. Bakeroot, J. Doutreloigne, A. Monté, P. Bauwens et al., *Integrated Driver with Optical Compensation for Improved Uniformity of Emissive Displays*, Proceedings of the 8th International Meeting of Information Display (IMID'08), 2008, Seoul, Korea, pp. 692-695

- A. Monté, J. Doutreloigne and P. Bauwens, *A Completely Integrated Power-Efficient High-Voltage Driver for Bistable Displays*, Proceedings of the 27th International Display Research Conference (IDRC'07) , 2007, Moscow, Russia, pp. 428-431

- P. Bauwens, J. Doutreloigne and A. Monté, *A New Driving Technology for Passive-Matrix Displays*, Proceedings of the 27th International Display Research Conference (IDRC'07), 2007, Moscow, Russia, pp. 158-160

Papers presented at international conferences without proceedings

- P. Bauwens and A. Monté, *Driving a Modular Passive-Matrix Display*, SID-ME Chapter Spring Meeting, 2008, Jena, Germany

- A. Monté and P. Bauwens, *Design of a New Power-Efficient High-Voltage Bistable Display Driver*, SID-ME Chapter Spring Meeting, 2008, Jena, Germany

Papers presented at national conferences

- P. Bauwens, *Research on New Technologies for 'Electronic-Paper' Applications*, 8th FirW PhD Symposium, Faculty of Engineering, Ghent University, 2007

The whole purpose of education
is to turn mirrors into windows.
Sydney J. Harris (1917-1986)

2 It's all about displays

2.1 Introduction

In this chapter I'll give a basic introduction into the display world. We'll start off by describing some of the older and newer display technologies in Section 2.2. It is followed by a section where we elaborate on a specific classification in display technologies, which will help to clarify some vocabulary. In Section 2.4 we take a peak at a main division in driving a display, the difference between active and passive matrix driving, and some important consequences. Last but not least, we'll have a look at a special kind of display, namely the e-paper, since this was one of the main motivators for starting this Ph.D.

2.2 Display technologies

2.2.1 Cathode Ray Tube

The first Cathode Ray Tube or CRT saw its light somewhere in 1897 when the German physicist Karl Ferdinand Braun modified the Crookes tube (an experimental electrical discharge tube) with a phosphor coated screen [1, 2]. He conveniently called it the "Braun tube". It was only ten years later that the Russian scientist Boris Rosing used the CRT technology to display simple geometric shapes on the screen.

Cathode ray tubes use an electron beam. It was first called 'cathode ray', because the beam emanated from the cathode (negative electrode) of a vacuum tube [3]. This electron beam will hit the phosphorescent screen to light it up. There needs to be a very good vacuum in the tube, to avoid scattering of the electrons by collisions with air molecules. See Figure 2.1 for a cross section of a CRT.

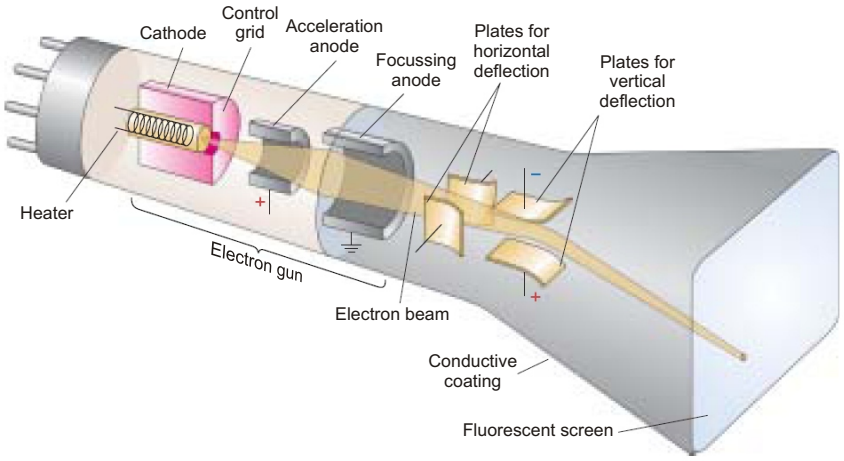


Figure 2.1 – Cross section of a CRT display

The electron beam source is an electron gun at the back of the monitor. The *cathode*, at the left hand side, emits electrons because it is raised to a high temperature by the *heater*, causing the electrons to evaporate from the surface of the cathode. The *accelerating anode*, with a small hole at its center, is kept at a high positive potential (1-20kV) relative to the cathode. This gives rise to an electric field between the cathode and accelerating anode, accelerating the electrons. Electrons passing through the hole in the anode form a narrow beam and travel with constant horizontal speed from the anode to the screen. The area where the electron hits, lights up.

The *control grid* regulates the amount of electrons that reach the anode, thus controlling the brightness of the spot on the screen. The *focusing anode* makes sure that every electron in the beam, hits the same spot on the screen. The beam can be deflected by an electric of magnetic field, to trace over the entire screen. This happens when it passes the two pairs of *deflecting plates or coils*. One pair controls the horizontal deflection, the other the vertical deflection. A picture is formed by very rapidly scanning the entire screen (left to right, top to bottom) and precisely controlling the amount of electrons for every point on screen.

One kind of phosphor with one electron beam will only produce one color. In color CRTs, three electron guns are used, aiming for three different phosphors,

emitting red, green and blue respectively. Those phosphors are packed together in stripes (aperture grille) or round clusters (shadow mask). These grilles or masks make sure that an electron does not hit the wrong phosphor.

2.2.2 Plasma Display Panel

The first, albeit monochrome, Plasma Display Panel or PDP was developed in 1964 at the University of Illinois by Donald Bitzen and Gene Slottow [1, 4]. However, the world had to wait for successful plasma televisions until after the rise of digital and other technologies. In 1983, IBM demonstrated a 19-inch (48 cm) orange-on-black monochrome display. In 1992, Fujitsu introduced the world's first 21-inch (53 cm) full-color display and later, in 1997 the first 42-inch (107 cm) plasma display with a resolution of 852x480. Philips had a PDP of the same size and resolution that year, it was the only PDP to be displayed to the retail public. Until the early 2000s, plasma displays showed some important benefits over LCDs. They provided better blacks, faster response time, greater color spectrum, and wider viewing angle. They could also be made much bigger. LCDs seemed to be a technology only suited for smaller displays. However, improvements in VLSI (Very-large-scale integration) fabrication technology have narrowed that technological gap. LCD displays can now be made in larger sizes and their lower weight, often lower power consumption and falling prices make them competitive with PDPs.

PDPs, as CRTs, use phosphors to create an image on screen. The way the phosphors are excited, however, is completely different. Figure 2.2 shows a cross section of a PDP.

Plasma is a gas made up of free-flowing ions (electrically charged atoms) and electrons. A plasma display consists of thousand of tiny cells with an inert mixture of noble gases (neon and xenon). Under normal conditions, the gas is mainly made up of uncharged particles. This changes very quickly when a voltage is placed across a cell. The introduced free electrons will collide with the atoms, knocking loose other electrons. When this happens, the atom becomes positively charged. It becomes an ion. Hence, a plasma is formed. The positively charged particles will be attracted to the negatively charged area, while the electrons will be rushing to the positively charged area. Particles are constantly bumping into each other. These collisions will excite the atoms in the plasma, causing them to release photons of energy. A gas with xenon and neon atoms, these will be *ultraviolet photons*. These are invisible to the human eye, but can be used to excite a phosphorescent material that can emit visible photons (see below).

The voltage across the gas in one specific pixel is established by having a special electrode structure and charging the correct electrodes (See Figure 2.2). *Address electrodes* (See also Section 2.4) are placed behind the cells, along the rear glass plate. The transparent *display electrodes*, surrounded by an insulating material and

coated with a protective layer, are placed on top of the cell, along the front glass plate. The electrodes run across the entire screen. The horizontal top electrodes and the vertical address electrodes form a *grid*. When a top and bottom electrode is charged, the cell at the intersection of both electrodes will be ionized.

As explained, this will stimulate the gas atoms to release ultraviolet photons. These photons will excite one of the phosphor atoms, coated on the inside wall of the cell. One of the phosphors electrons will jump to a higher energy level. When this electron falls back to its normal level, a visible light photon is emitted [5]. Just as with the color CRT, every pixel is made up of three separate subpixel cells, each with different colored phosphors: red, green and blue. While CRTs control the amount of electrons sent to adjust the brightness of a pixel, plasma panels use pulse-width modulation: by varying the pulses of current flowing through the different cells thousands of times per second, the control system can increase or decrease the intensity of each subpixel color to create billions of different combinations of red, green and blue. In this way, the control system can produce most of the visible colors. Plasma displays use the same phosphors as CRTs.

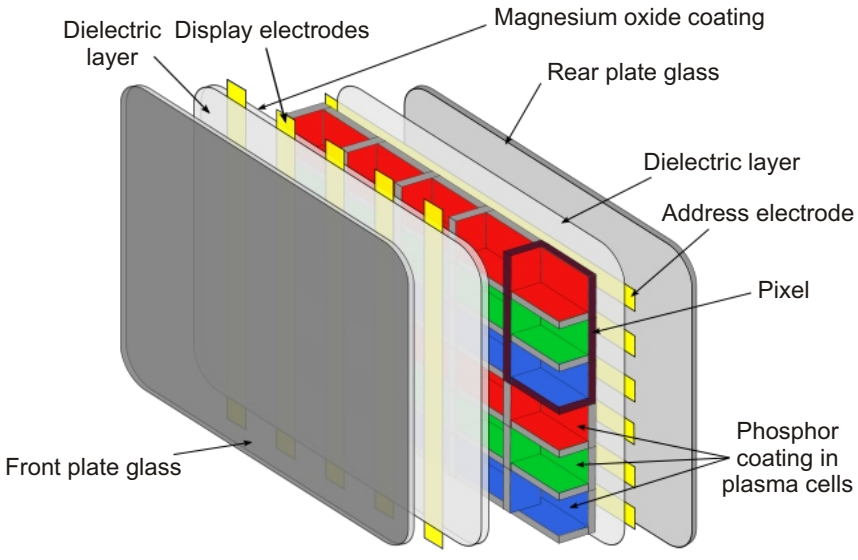


Figure 2.2 – Cross section of a PDP

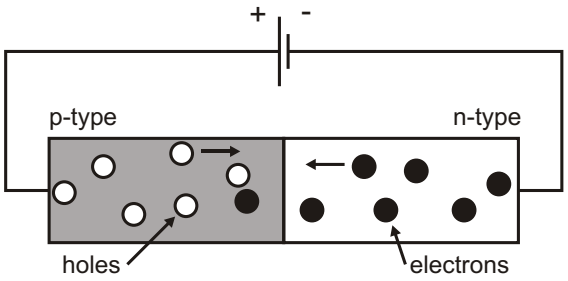


Figure 2.3 – The inner workings of a LED

2.2.3 LEDs and OLEDs

LEDs

While the phenomenon of electroluminescence (in which a material emits light in response to an electric current passed through it) was already discovered in 1907 by the British experimenter H. J. Round, and later (1927), independently, by the Russian Oleg Vladimirovich Losev, no practical use was made of the discovery for several decades [6]. It was not until 1961 that the first infrared Light-Emitting Diode or LED was created by the Americans Robert Biard and Gary Pittman.

The birth of the first visible-spectrum (red) LED was developed in 1962 by Nick Holonyak Jr. The first LEDs became commercially available in late 1960s. They were commonly used as indicators, and in seven-segment displays, first in expensive equipment such as laboratory and electronics test equipment, then later in such appliances as TVs, radios, telephones, calculators, and even watches. As the LED materials technology became more advanced, the light output was increased, and LEDs became bright enough to be used for illumination.

Figure 2.3 shows the inner workings of a LED. I will not go into the specifics of the physical processes that occur inside a LED, but I will skim the basic principles.

A *diode* is the simplest sort of semiconductor device. Most semiconductors are made of a poor conductor that has had impurities (atoms of another material) added to it. Adding impurities is called *doping*. When a semiconductor is doped with atoms with extra electrons, an *N-type* material is created. An N-type material has extra negatively-charged particles, free electrons can move from a negatively charged area to a positively-charged area. A *P-type* material is created when the semiconductor is doped with atoms with less electrons, creating electron holes in the material. Electrons can jump from hole to hole (towards a positively-charged area), creating the illusion of a hole moving towards the negatively-charged area. A diode consists of an N-type material bonded to a P-type material, with electrodes on each end. Without a voltage applied to the electrodes, the electrons from the N-type region will fill up the holes in the P-type region along the junc-

tion between the regions. This zone, where the semiconductor is returned to its insulating state, is called the *depletion region*. Applying a positive charge to the electrode on the N-type and a negative charge to the electrode on the P-type will only increase this depletion region, causing no current to flow. When, on the other hand, a negative charge is put on the N-type area and a positive charge on the P-type, the free electrons in the N-type area will be repelled by the negative electrode and drawn to the positive electrode. The opposite holds for the holes in the P-type area. If the voltage difference between the electrodes is high enough, the electrons in the depletion zone are pulled out of their holes. The depletion zone will disappear and current will flow. The free electrons moving through a diode can fall in empty holes in the P-type area, causing an energy drop which is released as a photon. The frequency of the photon (relative to the energy drop), and so the color, is dependent on the used materials.

But some colors are more difficult to make. It was not until the 1990s that low-cost efficient blue LEDs emerged, completing the red-green-blue color triad needed for a color LED display.

OLEDs

An organic LED (OLED) is, as implied by the name, a specific kind of LED. Tang and Van Slyke of Eastman Kodak first reported light emission from small-molecule organic systems in 1987. The company went on to patent this materials technology and has since licensed it to other companies, which have continued its development. In the past few years, many new small-molecule OLEDs have been discovered and refined, considerably diversifying the playing field. The first OLED product, a monochrome car stereo display, reached the market in 1997 [1, 7].

The operation of the OLED is basically the same as that of the regular LED (See Figure 2.4)

Between the metal cathode and the transparent anode, there is an emission layer and a conductive (or hole transport) layer, both made of an organic material. Sometimes more layers are added (electron transport layer, hole injection layer) to improve efficiency [8, 9, 10]. On the anode there is a glass substrate. One of the great advantages of the OLED is that it can be made flexible, although other substrate materials are needed [11]. The emission layer transports the electrons from the cathode, the conductive layer transports the holes from the anode. When, as with the regular LED, an electron and a hole combine they will emit a photon. In the materials used, the holes have a greater mobility then the electrons, so this will happen in the emission layer, hence the name. The frequency of the photon (the color of the emitted light) depends on the used material. So for a full color OLED TV, you need a layer that emits blue light, one that emits red light and one that emits green light.

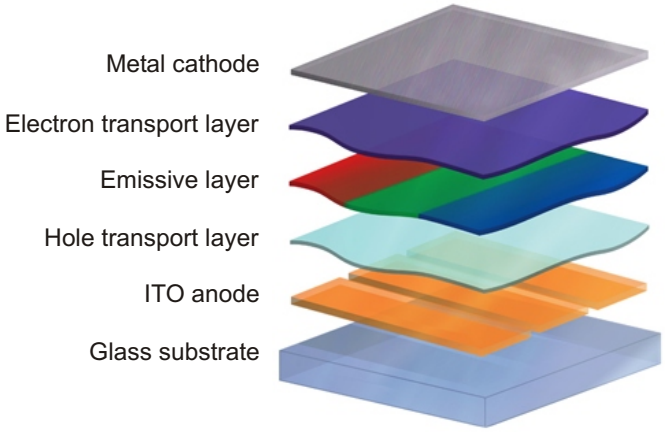


Figure 2.4 – The structure of an OLED

Different material technologies exist for creating OLEDs. They are typically a phosphorescent or fluorescent material. What these materials do is they can absorb energy (like high enery particles or photons) and re-emit the energy as a photon with a specific wavelength. Phosphorescence is similar to fluorescence except the light emission continues even after the source of energy is removed. It is said that the phosphorescent OLEDs are much more efficient than their fluorescent counterparts [12].

OLEDs have a lot of great advantages over both LCDs and regular LEDs. They can be made lighter, thinner and more flexible. They are brighter than LEDs and do not require backlighting like LCDs (see below), which makes them more power efficient. OLEDs are easier to produce and can be made to larger sizes. Because OLEDs are essentially plastics, they can be made into large, thin sheets. It is much more difficult to grow and lay down so many liquid crystals. OLEDs have large fields of view, about 170 degrees. Because LCDs work by blocking light, they have an inherent viewing obstacle from certain angles. OLEDs produce their own light, so they have a much wider viewing range.

There are still some problems though. One of the biggest problems OLEDs copes with, is the limited lifespan. While red and green OLED films have longer lifetimes (46,000 to 230,000 hours), blue organics currently have much shorter lifetimes (up to around 14,000 hours). Another problem is that water can easily destroy OLEDs.

2.2.4 Liquid Crystal Displays

One could probably write an entire book about liquid crystals (LC) and their use in liquid crystal displays (LCD). I will only cover some of the basic principles and give a few examples of LCD technologies that might pop up in later chapters of this book.

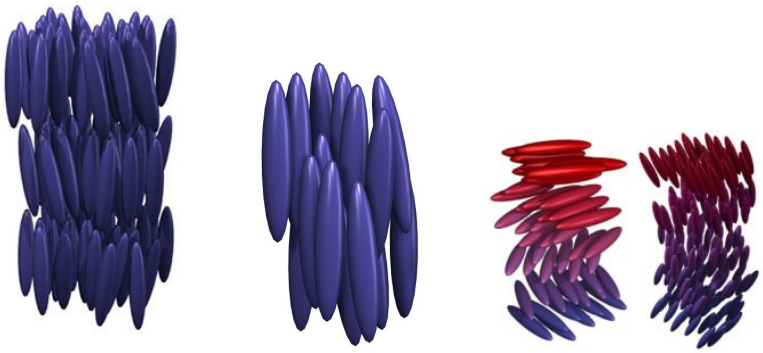
Liquid crystals

Liquid crystals are substances that can be in a phase between the liquid and solid phase. They were already discovered in 1888 by the Austrian physiologist Friedrich Reinitzer [1, 13]. Because it shared characteristics of both liquid and crystal, it got the name "fliessende krystalle". The name "liquid crystal" was born. He discovered it by varying the temperature of certain crystals. At a certain temperature, their state changed from crystals to liquid crystals. These types of LC are called *thermotropic*. There are also lyotropic LCs, based on a reaction with water or another solvent. These are mainly investigated in the fields of biochemistry and bionics and will not be discussed here.

Research on LCs made a slow start. In 1922, Georges Friedel describes the structure and properties of liquid crystals and classified them in 3 types (see below). The patent for the first practical application of the technology (the liquid crystal light valve) was filed in 1936 [14]. It was not until the 1960s, however, that serious studies of the materials and the effects of electric fields on these devices were carried out. The first use of a liquid-crystal device as a display dates back to Williams and Heilmeier in 1963.

As said above, Friedel divided the liquid crystals in three types. Liquid crystal molecules are often shaped like rods or plates that encourage them to align collectively along a certain direction. The division was made based on this natural ordering of the modules. The three types are the *smectic*, *nematic* and *cholesteric* type [15].

- *Smectic liquid crystals*: In smectic type liquid crystals, the cigar-like molecules are arranged side by side in a series of layers as shown in Figure 2.5a. The long axes of all molecules in a given layer are parallel to one another and perpendicular to the plane of layers.
- *Nematic liquid crystals*: In the nematic state, the molecules are not as highly ordered as in the smectic state, but they maintain their parallel order (See Figure 2.5b). On average, the nematic liquid crystals are aligned in one direction. Liquid crystals used in electronic displays are primarily of the nematic type.
- *Cholesteric liquid crystal*: The molecules in cholesteric liquid crystals are arranged in layers (See Figure 2.5c). Within each layer, molecules are



(a) Smectic LC **(b)** Nematic LC **(c)** Cholesteric LC

Figure 2.5 – The three types of LC

aligned in parallel, similar to those in nematic liquid crystals. The molecular layers in a cholesteric liquid crystal are very thin, with the long axes of the molecules parallel to the plane of the layers. A special aspect of the cholesteric structure is that the direction in each layer is displaced slightly from the corresponding director of the adjacent layer. The displacement is cumulative through successive layers, so that the overall displacement traces out a helical path, giving the liquid crystal some interesting properties.

Liquid crystal displays

After the invention of the LCD in 1963, things started to move faster. In 1970 the twisted-nematic (TN) field effect was discovered by Hoffmann-LaRoche [16], which is still employed in most of the active-matrix and direct-drive LCDs made today. A year later, the first LCD using this TN-effect was produced. It was a long road from the simple watch displays of the early 1970s to the full-color LCD desktop monitors of today. One by one, every technical challenge posed by the CRT standard has been met and in many cases exceeded. At the end of 2001, we could find excellent LCD monitors on the shelf in any major retail electronics store. In 2007 the LCD televisions surpassed CRT units in worldwide sales.

Liquid crystals find their use in display applications because of the difference in their properties in presence and absence of an electrical field. For example, light might be scattered, blocked or have its polarization changed by the LC in the absence of an electrical field, while the LC in the presence of an electrical field leaves the light unchanged. Color filters can be used to create the needed primary colors for a full color LCD. A lot of different LCD technologies exist, but

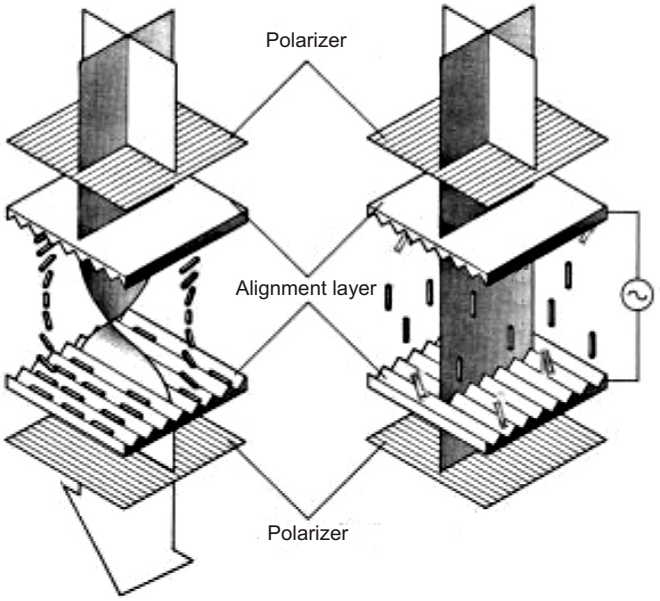


Figure 2.6 – The twisted-nematic liquid structure. In the absence of an electrical field, the light changes polarization (left). In the presence of an electrical field, the light remains unchanged.

I will only discuss the TN and its big brother the Super-Twisted Nematic (STN) because of their importance in the display world and Cholesteric displays and PDLC displays because they are mentioned further in this book.

(S)TN

When a nematic liquid crystal lies on a surface with grooves, the molecules of the LC will line up with the grooves. When the LC is sandwiched between two layers with grooves in directions perpendicular to each other, the LCs will be forced into a *helical twist* of 90°. The entire display structure consists of a polarizer, an alignment layer with grooves, glass with a transparent electrode, the liquid crystal, another transparent electrode on glass, the second alignment layer with perpendicular grooves and the second polarizer with a perpendicular polarization to the first polarizer. (See Figure 2.6).

As the light strikes the first polarizer, only one polarization is let through. The TN LC has such an effect on the polarized light, that it will rotate the polarization 90°. This way, it can safely pass the second polarizer. When this light comes from a light source at the back of the display, the pixel will emit light. Things change when a voltage is applied between the two electrodes. In that case, the LCs will

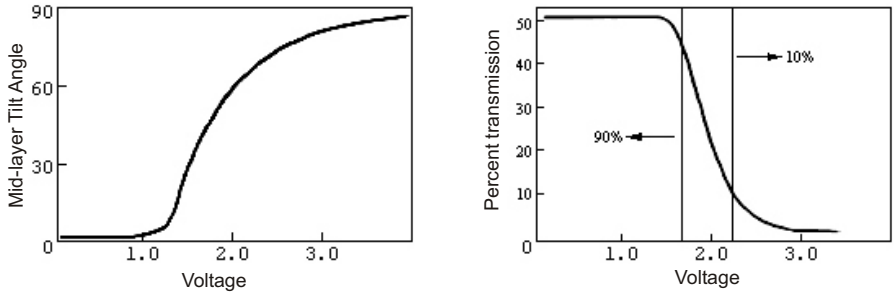


Figure 2.7 – The angle of the molecules versus the applied voltage (a). The resulting electro-optical response (b). (SHARP)

orient themselves more or less (depending on the strength of the electric field) parallel to the applied field. The polarized light is no longer completely twisted by the LCs and is partially blocked by the second polarizer. The pixel appears to be gray, or black if the light was completely blocked by a sufficiently strong field. By controlling the twist of the LCs in each pixel, the transmission of the exact amount of light that passes through can be varied. Figure 2.7 shows the response of a typical TN cell to an applied voltage. Figure 2.7a shows the angle of the molecules as a function of the applied voltage. Figure 2.7b shows the resulting electro-optical response, being the amount of light transmitted through the liquid crystal. A transmission of 10% can be considered black, a transmission of 90% can be considered white. This is 90% of the maximum light that can be transmitted. The polarizer filters 50% of the light source, so a white pixel corresponds with a total transmission of 45%. Color filters are used to create color pixels.

The biggest problem with a simple TN LCD is the limited viewing angle. This can be reduced by using other technologies like IPS (In Plane Switching) where the applied field is horizontal instead of vertical, leaving the LC molecules always parallel to the substrate, or VAN (Vertically Aligned Nematic) where the molecules will align themselves perpendicular to the field. The curve in Figure 2.7b is not really steep. The voltages for 10% (V_{OFF}) and 90% (V_{ON}) are quite far apart. In Section 2.4 we'll see that because of this, passive matrix driving (also discussed in that section) will put serious limitations on the achievable resolution. To increase the resolution, we'll have to move to active matrix driving, or make the curve steeper.

This is exactly the purpose of the Super-Twisted Nematic (STN) LC. In 1984, Terry Scheffer found that, when doping the TN LC with a cholesteric LC, cells could be made where the LC doesn't twist 90°, but 270°[17]. With some more adjustments, other twist angles were possible. Figure 2.8 shows how this affects the electro-

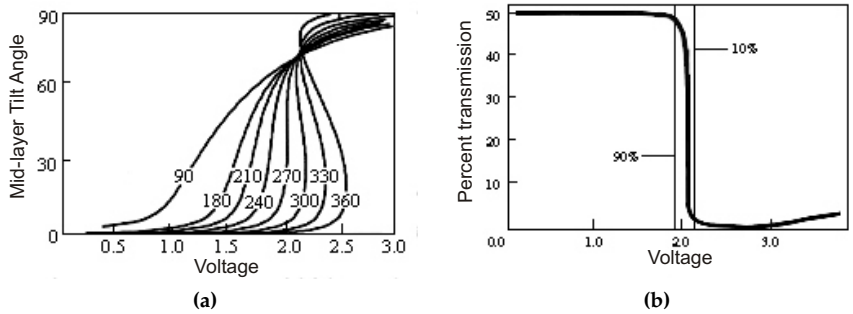


Figure 2.8 – The angular distortion versus the applied voltages for the different initial twist angles (a). The resulting electro-optical response (b). (SHARP)

optical/distortional responses.

The steeper the curve of the electro-optical response, the higher the resolution can be using passive matrix driving. However, grayscale images require intermediate points along the curve. For this reason, many commercial STN displays use a twist angle of 210°.

Of course, the STN LCDs also has its drawbacks. The shifted transmission spectrum of the device causes an undesirable coloration. This was solved by adding another STN layer (Double STN or DSTN), with a twist in the opposite direction, nullifying the shift. The contrast was further increased with the introduction of the Film Compensated STN (FSTN). An extra filter layer is placed between the STN display and rear polarizer to increase the sharpness and contrast. Also the Dual Scan STN increased the contrast. The screen is divided into halves which are scanned simultaneously, doubling the lines being refreshed per second.

PDLC

Polymer Dispersed Liquid Crystal is based on a different principle. It switches between a scattering state and a transparent state. In the *scattering state*, the incident light is scattered back and will be perceived as milky white. With a light absorber at the back, the *transparent state* will appear black [18]. PDLCs are thin polymer films with dispersed LC microdroplets. They do not require polarizers. The absence of polarizers greatly reduces light loss through the cell in the transparent state (polarizers usually block over 50% of the incident light in TN cells) [19].

Figure 2.9 shows a PDLC film in the presence and absence of an electrical field. When no electrical field is applied, the nematic LC in the droplets will align itself randomly per droplet. In the "off-state" the PDLC film appears milky white due

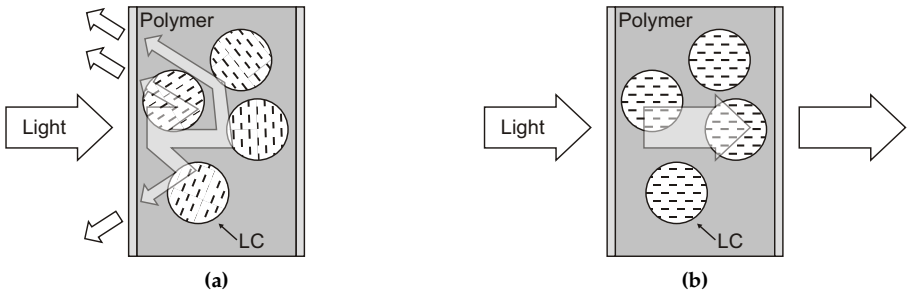


Figure 2.9 – A PDLC film in the absence (a) and presence (b) of an electrical field

to the scattering caused by the refractive index mismatch encountered by incoming light at the liquid crystal/polymer interface. When an electric field is applied across the film, the liquid crystal molecules within the micro-droplets align with the electric field. If the ordinary refractive index of the liquid crystal within the droplets is sufficiently close to the index of the polymer matrix material, the incoming light is no longer scattered and the PDLC film becomes clear [20].

To create a PDLC film, mixture of LCs and monomers (non-cured polymers) is created. This mixture has to be cured with UV to obtain a stable film. During the UV curing process, polymerization starts. This results in a separation between the polymer an the LC molecules. The size of the droplets is generally in the order of several micrometers. However, the size, and consequently some electro-optical properties, can vary a lot depending on the preparation of the PDLC. Smaller droplets means more scattering, but a higher voltage is needed to turn them transparent. Other variables are the materials used and the ratio they are mixed in. Parameters that control the way the photo-induced polymerization takes place are the temperature, UV intensity and UV cure time [18, 21, 22, 23]. Also the cell thickness plays an important role. The thicker the cell, the more the light will be scattered in the "off-state", but the higher the switching voltages will be. A general electro-optical response of a PDLC film of 20μm thick, is shown in Figure 2.10. Keep in mind that the voltages shown are not representative for all PDLC films, since these are, as said, highly dependent on the film parameters. From the graph it is also clear (or it will be clear after reading Section 2.4) that PDLC is not very multiplexable. For this reason it is mostly used in direct drive applications (like smart windows, switching between transparent and frosted).

With the correct color filter, it is also possible to create full color PDLC displays [24].

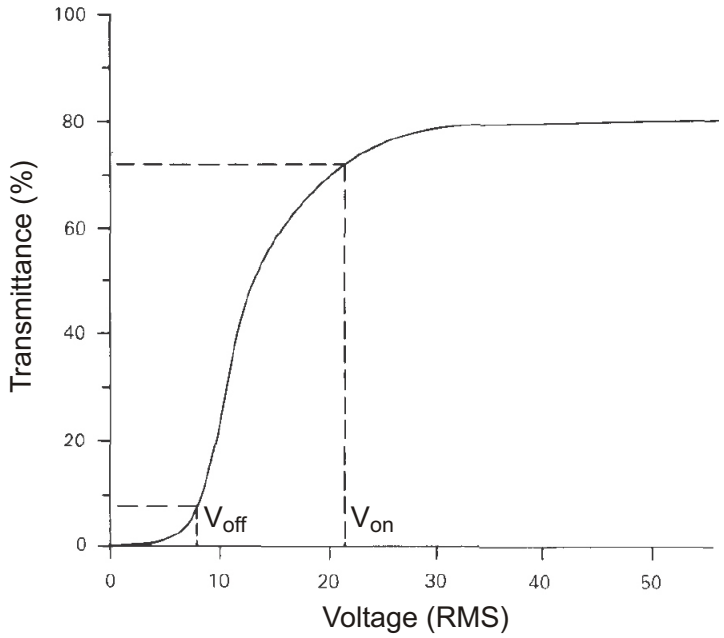


Figure 2.10 – A general electro-optical response of a PDLC film

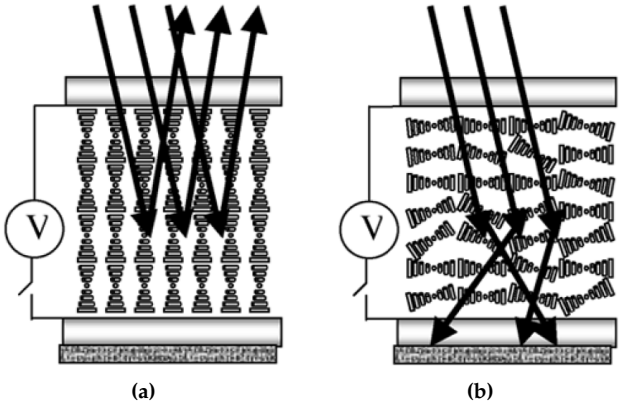


Figure 2.11 – In the stable planar state, the incident light is reflected (a). In the focal conic state, no light is reflected (b).

Bistable cholesteric liquid crystal

As said in the beginning of this section, cholesteric liquid crystal (ChLC) has a special helical structure in which the liquid crystal molecules are aligned along a common direction on a plane perpendicular to the helical axis but the molecules on different planes are twisted with respect to each other. The liquid crystal is a periodic optical medium along the helical axis; the refractive index oscillates periodically [25]. This creates a *Bragg reflector*, which reflects light within a certain wavelength band, depending on the twist of the helical structure.

Cholesteric reflective displays are made of two substrates with a ChLC sandwiched between them. They exhibit two stable states at zero field condition [26], meaning that no voltage is required to keep the crystal in a particular state, only to change state. One of them is the stable planar state (SP) in which the helical axis is more or less perpendicular to the cell substrates, as shown in Figure 2.11a. Incident light within the reflection band is reflected backward. The material has a bright appearance in this state. The other one is the focal conic state (FC) in which the helical axis is more or less parallel to the substrates, as shown in Figure 2.11b. Incident light is diffracted or scattered in the forward direction and absorbed by the absorption layer coated on the bottom substrate. The material in this state has a black appearance.

Switching between the two stable states is done by applying the correct voltages. The liquid crystal molecules tend to align parallel to applied electric fields. If an intermediate voltage is applied across the cell when the ChLC is in the planar state, all the molecules will be aligned perpendicular to the applied field. This makes the planar state unstable. At higher voltages the helical structures will be

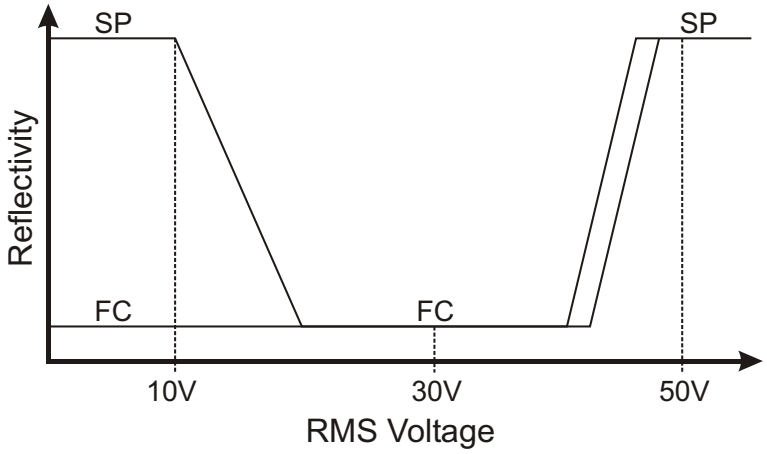


Figure 2.12 – A schematical representation of the response of ChLC to a voltage pulse

unwound, but if the applied field is not high enough, the helices themselves will turn and the ChLC will be switched to the focal conic state. Since this is a stable state, they will remain there even after the applied voltage is removed. When the voltage is high enough to unwind the helical structure, the ChLC is said to be found in the *homeotropic* state, where all molecules are aligned with the electric field. This is an unstable state, and when the high voltage is removed quickly the liquid crystal relaxes into the planar state. A schematic representation of the optical response to a voltage pulse can be seen in Figure 2.12.

Since the color of the reflected light depends on the twist in the helical structure, it is possible to design a ChLC for reflecting a certain color. Monochromatic displays can be made from a single layer of ChLC. Full color displays need a stack of three layers, reflecting blue, green an red light [27, 28].

2.2.5 Some other display technologies

Field Emission Display

A Field Emission Display (FED) is based on the same principles as the CRT. High velocity electrons are shot against a phosphorescent screen, which will light up. But, instead of an electron gun to emit the electrons a large array of fine metal tips or carbon nanotubes is used [29]. Every pixel will have its own set of nanotubes, which can be activated separately. Whereas the electron gun in the CRT relied on heating the cathode to emit electrons, the FED relies on cold emission. When a voltage is applied (in the order of kV) a very high electrical field will be generated between the cathode (nanotubes) and the anode (metal mesh at the screen side),

due to the very fine tips of the nanotubes. This field will pull the electrons from the tips and throw them against the phosphorescent screen.

Due to the cold cathodes, the emittors can be packed close together with their supporting electronics, without causing the entire display to overheat. The assembly of cathodes can then be placed close enough to the glass face of the display. As a result, the display can be made flat like the PDP.

Electrophoretic Display

Electrophoretic displays (EPDs) are based on the movement of charged particles in a fluid. When this mixture of particles and fluids is placed in a cell and a voltage is applied, the particles will move away from (or towards, depending of the charge) top of the cell. If the particles and the fluid have a different color, the top of the cell will obtain one of those colors, depending on the applied voltage.

Another option is using a transparent fluid and mixing it with two types of particles (e.g. black and white) with opposite charge. One voltage will pull the black particles to the top and the white to the bottom, the opposite voltage will pull the white particles to the top. Because of the large viscosity of the fluid, rather large voltages are needed to separate the black and white particles. The advantage though is that due to that viscosity, the particles remain in the same place when the voltage is removed, making the display bistable.

Electrowetting Display

Wetting is the ability of a liquid to maintain contact with a solid surface, resulting from intermolecular interactions when the two are brought together. A fluid with little wetting will be more like a sphere on the surface, while a fluid with more wetting will be more smeared out. Electrowetting is the modification of the wetting properties of a hydrophobic surface with an applied electric field.

In an electrowetting display, water and (colored) oil are the main actors [30]. With no voltage applied, the oil forms a flat film between the water and a hydrophobic (water-repellent), insulating coating of an electrode, resulting in a colored pixel. When a voltage is applied, the hydrophobic surface becomes hydrophilic and the oil is pushed aside, resulting in a white pixel if there is a white reflective layer beneath the element.

Electrochromic Display

Electrochromism is the phenomenon displayed by some materials of reversibly changing color when a burst of charge is applied. An electrochromic layer can be colored and turned transparent by the injection of ions (positively charged particles) and electrons [31]. The color change is persistent and energy need only

be applied to effect a change, which makes it a bistable technology. However, it is a very slow process, so it is not suitable for displaying video material. It is often used in smart windows or large-area information displays like advertising boards where high switching speed is not required.

2.3 Transmissive, emissive, reflective?

An important distinction in display technologies is the difference between reflective, transmissive and emissive displays.

Transmissive displays use a backlight. The image is formed by pixels blocking or transmitting the light. LCDs using technologies like (S)TN, VAN, ISP can be used as transmissive displays. They are usually low in power efficiency because the backlight is always emitting light, even when a black screen is to be displayed. Another consequence is the limited contrast ratio, since the 'blackness' of black is dependent on the blocking ability of the LC, which isn't 100%. Typically, an LCD has a contrast ratio of 1,000:1. There is a technique, called local dimming, which can improve this ratio. The uniform backlight is replaced with individual LEDs, which can be turned off in the areas where black is to be displayed. This way, ratios up to 10,000:1 can be achieved [32]. This might seem a good improvement, but compare this with the contrast ratio of a CRT or PDP. With these displays, perfect black is possible, which gives them theoretically an infinite contrast ratio, but is practically said to be about 2,000,000-5,000,000:1.

This is the difference with an *emissive* display, where light is only generated from the pixels when they are turned on. This is the case, for example, with CRT, PDP and OLED displays. In theory, these emissive displays should be more energy efficient, but due to low efficiency in the light generation process most emissive and transmissive flat panel displays have comparable efficiency. In retail, it is often claimed that a PDP consumes a lot more power than a LCD, however, these measurements are done with the display showing a completely white screen, which is not representable for real-life television.

A third category is that of the *reflective* displays. Ambient light is reflected (or not) to produce an image. Examples are an (S)TN display with a reflective surface at the back, a PDLC display, a ChLCD and electrophoretic/electrowetting/electrochromic displays. Reflective displays are most efficient. They are particularly good where ambient light is very bright, such as direct sunlight. They obviously do not work well in low-light environments.

There are also displays that are a combination of reflective and transmissive displays, called *transflective*. They use a semi permeable mirror that is able to reflect ambient light, and transmit light from the backlight. This is used in cell phones.

2.4 Driving the display: active and passive matrix driving

Another important distinction in displays is the way they are driven, how the voltage to turn a pixel on or off is applied. There is passive matrix driving, active matrix driving and the older direct drive.

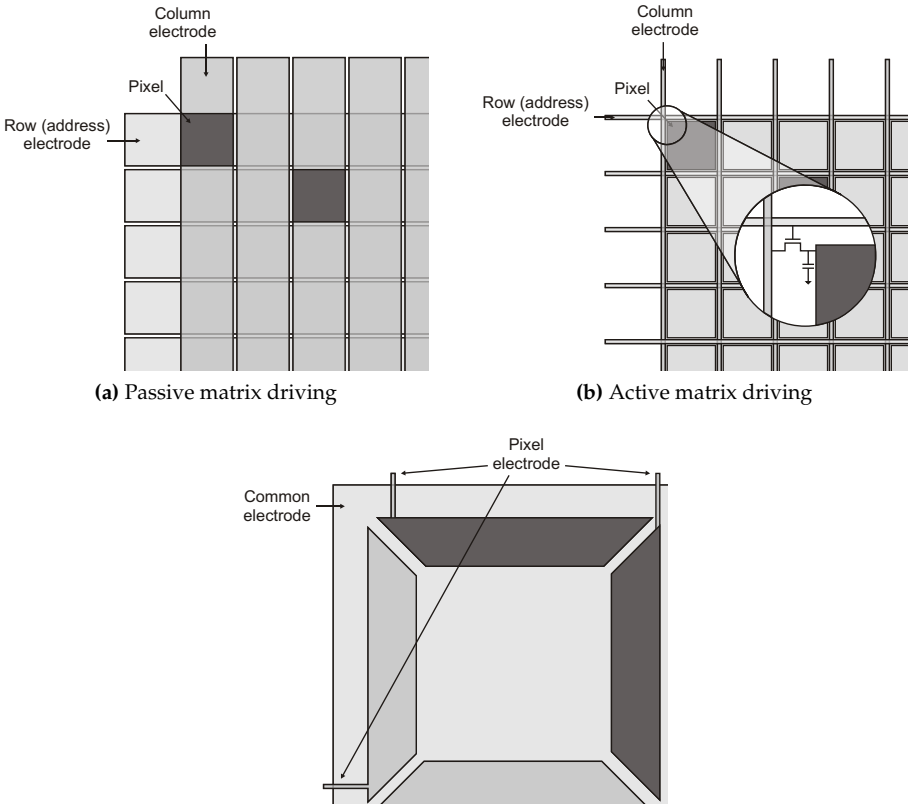
In *direct drive* displays, every pixel has its own electrode (with a common bottom electrode). Applying voltage to a pixel will have no effect on the other pixels. It is clear that, with an increasing number of pixels, the number of electrodes becomes unmanageable. For a display of N pixels, $N + 1$ electrodes are needed. This driving method was for example used in 8-segment alphanumeric displays you find in your old alarm clock (See Figure 2.13c).

A more efficient use of electrodes is found in so called matrix driving, where the display has a mesh of horizontal and vertical electrodes. There is a pixel at each intersection. A display with $(M \times N)$ pixels will only need $(M + N)$ electrodes. To display an image on the screen every row is selected one after another, these are called the *address electrodes*. During the selection of a certain row, the column electrodes are used to provide the pixels of that row with the correct voltage, they are often called *data electrodes*. Using *passive matrix* driving, the top and bottom electrodes of a pixel are directly connected to its row and column electrode. The voltage difference the pixel sees will be the difference between both electrodes (See Figure 2.13a). This has some important consequences. Since the same column electrode is used to provide the voltages for all pixels in that column, the pixels of a non-selected row can be influenced during the driving of the pixels of a selected row. They are no longer electrically separated, like with direct drive. The problem arises when using display materials that respond to the RMS (Root Mean Square) value of the applied signal. The luminance of a pixel in such a display, will not only depend on the voltage applied when its own row is selected, but will be influenced by every pixel value in that column. The more rows there are, the more difficult it is to make a large difference between the 'on' voltage (V_{ON}) of a pixel, and the 'off' voltage (V_{OFF}). In short, to achieve a certain contrast, there is a maximum number of lines that can be multiplexed. Alt & Pleshko [33] derived that, with N_{MAX} the maximum number of lines,

$$\frac{V_{ON}}{V_{OFF}} = \sqrt{\frac{\sqrt{N_{MAX}}+1}{\sqrt{N_{MAX}}-1}} \tag{2.1}$$

Or, when demanding a specific contrast, with $V_{OFF} = V_{th}$ and $V_{ON} = V_{th} + \Delta$ (See also Figure 2.14), for $\Delta << V_{th}$:

$$N_{MAX} = \left(\frac{V_{th}}{\Delta}\right)^2 \tag{2.2}$$



(a) Passive matrix driving

(b) Active matrix driving

(c) Direct drive

Figure 2.13 – Distinction between driving methods.

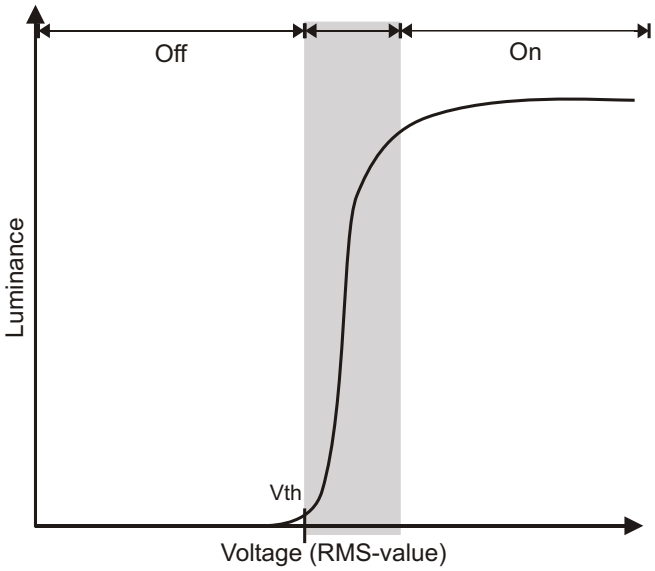


Figure 2.14 – General electro-optical characteristic of a PM-addressable liquid crystal.

Another issue arises during the passive matrix driving of fast responding displays. Take an (O)LED display for example. Every pixel in that display will only be emitting light when its row is selected. When the other rows are being selected, the pixel stays dark. This means that, to achieve a certain total brightness of the screen, the more rows there are, the brighter the pixels need to shine during their short row time. This is especially a problem with OLEDs where the lifetime is strongly dependent on how hard they are driven. Passive matrix displays are generally characterized with slow response time and poor contrast. They are easy and cheap to make, though.

Active matrix displays are developed to remove that limitation. The 'active' part denotes the presence of an active element, in this case the transistor. Every pixel has its own transistor and storage capacitor (See Figure 2.13b). In this case, when a row is selected, the pixel transistor is activated and the column electrode can charge the storage capacitor according the desired pixel value. When the row is not selected, the transistor is inactive and the pixel is electrically isolated from the column electrode (and the voltage over it remains the same). It is not influenced by the column electrode when it is applying voltages for other selected rows. This way, there is no inherent limitation on the number of lines that can be multiplexed. But because of this transistor (and extra ground line), active matrix displays are harder to make. The transistors can be made on glass like Thin Film

Transistors (TFT) or on silicon like LC on Silicon (LCoS). Currently there is also a lot of research on creating TFTs on flexible substrates, to create flexible, active matrix displays like AMOLED (Active Matrix OLED) displays.

2.5 Something about e-paper

Electronic paper or e-paper can be described as "any combination of display technology and application that provides a valid alternative concerning features and usability compared to traditional paper" [34]. This means that, ideally, e-paper should have a couple of properties. It should be thin, light and flexible. It should have a resolution comparable to printed paper. A budget printer already offers a resolution of 300 dpi (dots per inch), while the resolution of a standard PC monitor is about 72 to 96 dpi. E-paper should have at least the contrast ratio of a typical newspaper, around 5:1 to 8:1. It should have a large viewing angle and be clearly visible in the brightest sunlight. And last but not least, it should be ultra low power.

It is quite a challenge to create a display technology that can provide for all of these properties, but there are some technologies that can help. Technologies like ChLCD and EPD for example. These are bistable (low power consumption) LCs, with a contrast ratio that can go up to 20:1 and can be made flexible. They do require high voltages, though. A ChLCD may need up to 100V to switch states.

This Ph.D. was started with the aim of creating an easy to make, flexible, passive matrix e-paper display with a high resolution and good contrast. As said in the previous section, passive matrix displays and good contrast/resolution don't go well together, so the research towards a method to increase the resolution without touching the contrast was started. The following chapters will explain how this is done and how that same method was expanded to shift the main focus of this Ph.D. to modular display systems in general.

References

[1] D. E. Mently, "State of Flat-Panel Display Technology and Future Trends," *Proceedings of the IEEE*, vol. 90, no. 4, pp. 453–459, April 2002.

[2] Wikipedia. Cathode ray tube. [Online]. Available: http://en.wikipedia.org/wiki/Cathode_ray_tube

[3] P. S. Christaldi, "Cathode Ray Tubes and Their Applications," *Proceedings of the IRE*, vol. 33, no. 6, pp. 373–381, June 1945.

[4] Wikipedia. Plasma display. [Online]. Available: http://en.wikipedia.org/wiki/Plasma_display

[5] H. Bechtel, T. Jüstel, H. Gläser, and D. U. Wiechert, "Phosphors for plasma-display panels: Demands and achieved performance," *Journal of the SID*, vol. 10, no. 1, pp. 63–67, January 2002.

[6] Wikipedia. Light-emitting diode. [Online]. Available: http://en.wikipedia.org/wiki/Light-emitting_diode

[7] Wikipedia. Organic LED. [Online]. Available: http://en.wikipedia.org/wiki/Organic_light-emitting_diode

[8] S. V. Slyke, M. Hettel, M. Boroson, D. Arnold, N. Armstrong, and J. Andre, "Passive Matrix OLED Displays: Operational and Storage Stability," in *Proceedings of the* 20^{th} *International Display Research Conference (IDRC2000)*, 2000, pp. 341–345.

[9] S. Tokito, T. Tsuzukia, F. Satoa, and T. Iijima, "High-efficiency blue and white phosphorescent organic light-emitting devices," *Current Applied Physics*, vol. 5, no. 4, pp. 331 – 336, April 2005.

[10] T. Matsushima, M. Takamori, Y. Miyashita, Y. Honma, T. Tanaka, H. Aihara, and H. Murata, "High electron mobility layers of triazines for improving driving voltages, power conversion efficiencies, and operational stability of organic light-emitting diodes," *Organic Electronics*, vol. 11, no. 1, pp. 16 – 22, January 2010.

[11] R.-Q. Ma, R. Hewitt, K. Rajan, J. Silvernail, K. Urbanik, M. Hack, and J. Brown, "Flexible active-matrix OLED displays: Challenges and progress," *Journal of the SID*, vol. 16, no. 1, pp. 169–175, January 2008.

[12] M. Hack and J. J. Brown, "High-Efficiency AMOLEDs," *Information Display*, vol. 18, no. 7, pp. 16–19, Jule 2002.

[13] Wikipedia. Liquid crystal. [Online]. Available: http://en.wikipedia.org/wiki/Liquid_Crystals

[14] MarconiWireless Telegraph Company, "The liquid crystal light valve," British Patent 441 274, 1936.

[15] H. Kawamoto, "The History of Liquid-Crystal Displays," *Proceedings of the IEEE*, vol. 90, no. 4, pp. 460–500, April 2002.

[16] Hoffmann-LaRoche, "Lichtsteuerzelle," Swiss Patent 532 261, 1970.

[17] T. J. Scheffer and J. Nehring, "A new, highly multiplexable liquid crystal display," *Applied Physics Letters*, vol. 48, no. 10, pp. 1021–1023, November 1984.

[18] F. Bruyneel, "Introduction of Color in Reflective PDLC and PNLC Microdisplays," Ph.D. dissertation, Ghent University, 2002.

[19] G. Spruce and R. D. Pringle, "Polymer dispersed liquid crystal (PDLC) films," *Electronics & Communication Engineering Journal*, vol. 4, no. 2, pp. 91–100, April 1992.

[20] R. Karapinar, "Electro-optic Response of a Polymer Dispersed Liquid Crystal Film," *Turkish Journal of Physics*, vol. 22, no. 3, pp. 227–236, March 1998.

[21] Y. G. Fuh, K. L. Huang, C. H. Lin, I.-I. C. Lin, and I. M. Jiang, "Studies of the Dependence of the Electra-Optical Characteristics of Polymer Dispersed Liquid Crystal Fihns on Curing Temperature," *Chinese Journal of Physics*, vol. 28, no. 6, pp. 551–557, December 1990.

[22] S. A. Carter, J. D. LeGrange, W. White, J. Boo, and P. Wiltzius, "Dependence of the morphology of polymer dispersed liquid crystals on the UV polymerization process," *Journal of Applied Physics*, vol. 81, no. 9, pp. 5992–5999, May 1997.

[23] J. D. LeGrange, S. A. Carter, M. Fuentes, J. Boo, A. E. Freeny, W. Cleveland, and T. M. Millerd, "Dependence of the electro-optical properties of polymer dispersed liquid crystals on the photopolymerization process," *Journal of Applied Physics*, vol. 81, no. 9, pp. 5984–5991, May 1997.

[24] F. Bruyneel, D. Cuypers, H. D. Smet, and A. V. Calster, "Reflective Color PDLC Display using Color Filters," in *SID'02 Symposium Digest of Technical Papers*, 2002, pp. 534–537.

[25] D.-K. Yang, "Flexible Bistable Cholesteric Reflective Displays," *Journal of Display Technology*, vol. 2, no. 1, pp. 32–37, March 2006.

[26] D.-K. Yang, J. L. West, L.-C. Chien, , and J. W. Doane, "Control of reflectivity and bistability in displays using cholesteric liquid crystals," *Journal of Applied Physics*, vol. 76, no. 2, pp. 1331–1333, July 1994.

[27] M. Okada, T. Hatano, and K. Hashimoto, "Reflective Multicolor Display Using Cholesteric Liquid Crystals," in *SID97 Symposium Digest of Technical Papers*, vol. 28, 1997, pp. 1019–1022.

[28] D. Davis, K. Hoke, A. Khan, C. Jones, X. Y. Huang, and J. W. Doane, "Multiple color high resolution reflective cholesteric liquid crystal displays," in *Proceedings of the 17th International Display Research Conference (IDRC97)*, 1997, pp. 242–245.

[29] Y. Saito, K. Hata, R. Mizuchima, T. Tanaka, S. Uemura, T. Nagasako, J. Yotani, and T. Shimojo, "Field Emission from Carbon Nanotubes and its Application to FED elements," in *Proceedings of the 18th International Display Research Conference (IDRC98)*, 1998, pp. 173–181.

[30] B. J. Feenstra, R. A. Hayes, I. G. J. Camps, L. M. Hage, M. T. Johnson, T. Roques-Carmes, L. J. M. Schlangen, A. R. Franklin, A. F. Valdes, and R. A. Ford, "A Reflective Display Based on Electrowetting: Principle and Properties," in *Proceedings of the 23th International Display Research Conference (IDRC2003)*, 2003, pp. 322–324.

[31] Wikipedia. Electrochromic devices. [Online]. Available: http://en.wikipedia.org/wiki/Electrochromic_devices

[32] H. Chen, J. Sung, T. Ha, Y. Park, and C. Hong, "Backlight Local Dimming Algorithm for High Contrast LCD-TV," in *Proceedings of the 9th Asian Symposium on Information Display*, 2006, pp. 168–171.

[33] P. M. Alt and P. Pleshko, "Scanning limitations of liquid-crystal displays," *IEEE Transactions on Electron Devices*, vol. ED-21, no. 2, pp. 146 –155, February 1974.

[34] W. Hendrix, "Design of Low-Power High Voltage Driver Chips for Bi-Stable LCD's," Ph.D. dissertation, Ghent University, 2006.

[35] A. Monté, "Design of an Intelligent High-Voltage Display Driver to Minimize the Power Consumption in Bistable Displays," Ph.D. dissertation, Ghent University, 2008.

Reality is that which,
when you stop believing in it,
doesn't go away.
Phillip K. Dick (1928-1982)

3 Modular Displays

3.1 Introduction

In this small chapter I will give an introduction to modular displays, what they are, what they do and what they're trying to solve. We'll start out by taking a look at some display designs aiming to solve the issues that arise when driving with a passive matrix. In Section 3.3 we'll introduce briefly some existing systems that can create a free-form display, a display that can be shaped as desired. In the last section we'll see how modular displays cover those two areas.

3.2 Solving the issues with passive matrix driving

In Section 2.4 passive matrix driving was discussed. This type of displays is apparently very easy to make because of the lack of any TFT beneath the LC. But we all know there's no such thing as a free lunch, so this advantage comes with some disadvantages. For a certain contrast, the number of lines that can be multiplexed is limited. For a certain light output per pixel, the total brightness is reduced depending on the total number of lines. There are some technologies that can reduce these restrictions .

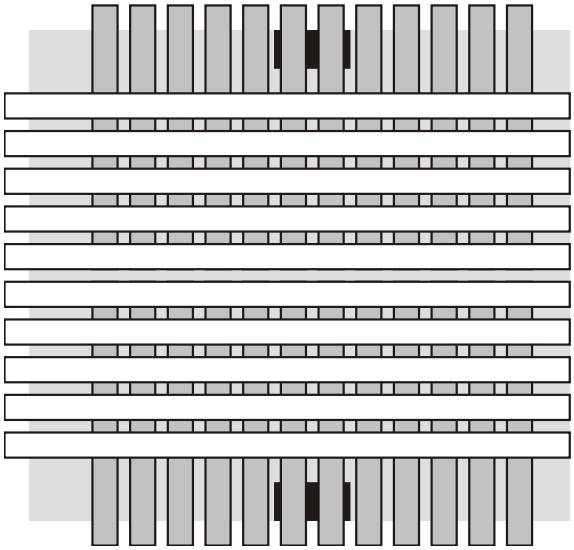


Figure 3.1 – Dual scan display

3.2.1 Limitation of multiplexability

Dual scan displays

The essence of this limitation is the fact that each column electrode can only be used to drive a limited number of pixels. Dual scan displays will not remove the limitation, but they reduce it [1]. Take a look at Figure 3.1. The display is split up in two sections, an upper half and lower half. The column electrodes run only across one half of the display. They are accessed both from the top as from the bottom. This way, when each set of column electrodes drives the maximum number of pixels, the total number of pixels across a vertical line is doubled. Using a dual scan display, twice as much rows can be multiplexed compared to single scan displays. Also, since both sections can be scanned at the same time, a dual scan display will be able to refresh faster.

Quad scan displays

Quad scan or multi scan displays take it a step further. Again, it is not the aim to completely remove the limitation but simply to reduce it enough. These types of display will try to divide the screen in more parts. The upper and lower part of the dual scan display can also be divided in two parts. The main problem encountered is the fact that the column electrodes of these new parts aren't easily accessible. The contacts also need to come from the top or the bottom of the dis-

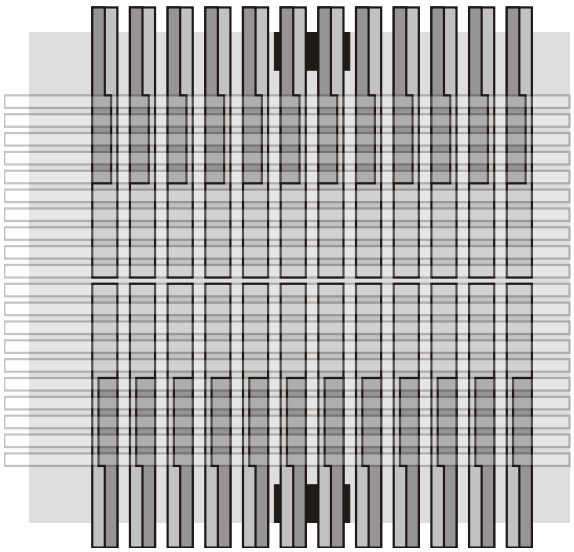


Figure 3.2 – Quad scan display

play, so special structures are needed [2]. An example is given is Figure 3.2. Each seperate part of the column electrodes can account for the maximum number of lines, meaning the total resolution can be quadrupled. Although one must realize that the connections to the column electrodes themselves can and will cause artifacts in the display.

3.2.2 Reduction of brightness

MLA

It's not really necessary to make a definitive distinction between the techniques for increasing the multiplexability and brightness, because in the end it all comes down to the Alt & Pleshko limitation from Section 2.4. The techniques discussed above could very well be used to increase the overall brightness of the display. But I would say that these techniques are more appropriate for slow responding LCs, while the techniques described below apply to fast responding LCs. This is also the reason why Multi-Line Addressing (MLA) was developed. The slow 'frame-responding' LCs caused a reduction in contrast and visible flickering. Multi-Line Addressing was initially developed to increase the contrast (and thus multiplexability) in fast responding STN LCDs. The concept of MLA stems from the early work on generalized matrix addressing by Nehring & Kmetz [3], which proved the validity of Alt & Pleshko's selection ratio limit.

What they proved is that, when using general matrix addressing (see below), the equality from Equation 2.1 is only an equality when the number of rows N is a perfect square. In the other case, it is possible for V_{ON}/V_{OFF} to be higher. They calculated that

$$\frac{V_{ON}}{V_{OFF}} = \sqrt{1 + \frac{N}{\sqrt{n_0(N-n_0)(N-1)} - n_0}} \qquad \text{, for } N \geq 6 \tag{3.1}$$

with n_0 the closest integer to ν_0:

$$\nu_0 = \frac{1}{2}(N - \sqrt{N}) \tag{3.2}$$

When N is a perfect square, Equation 3.1 can be simplified to Equation 2.1 from Alt & Pleshko.

The reason for this difference is that Alt & Pleshko started from the conventional time multiplexing of the display. Select a row, apply the data to the column, select next row. The applied voltage on the column electrode for a certain pixel, will only depend on the state of that pixel alone (e.g. 0V if pixel is off, 5V is pixel is on). Nehring & Kmetz proposed a more general matrix addressing, where multiple rows are being selected at the same time, hence Multi-Line Addressing. In this scheme, the voltage applied on the column electrodes isn't meant for one pixel alone, but will depend on the entire 'state' of that column.

I will not go to much into the mathematical details, but it might be interesting to see how this is done. Every column has a specific state, which is called λ. In a display with N rows, there are 2^N states possible (each pixel can be either on or off). The periodic potentials that will be applied to the row electrodes are called $F_i(t)$ with $(i = 1, \cdots, N)$, which are orthogonal functions. The potentials for the column electrodes, depending on the state λ of that column, are noted by $G_\lambda(t)$ and are determined by

$$G_\lambda(t) = \sum_{i=1}^{N} a_i^{(\lambda)} \cdot F_i(t) \tag{3.3}$$

When taking a look back at the conventional driving scheme, $F_i(t)$ would be strobe functions (high for a short time when te row is selected, low for the rest of the time when the row is not selected) and $a_i^{(\lambda)}$ would only be dependent on bit i (on the i^{th} row). But this does not provide the best results. In the general matrix addressing scheme the coefficients are dependent on the entire state of the column (with n the number of pixels in that row that are on):

$$a_i^{(\lambda)} = \begin{cases} b_n, & \text{when pixel } i \text{ is off} \\ c_n, & \text{when pixel } i \text{ is on} \end{cases} \tag{3.4}$$

These coefficients can be calculated by maximizing the minimal V_{ON}/V_{OFF} that will occur. Same goes for the RMS value $\overline{F}$ of the row potential functions $F_i(t)$.

Figure 3.3 shows examples for a display with $N = 2$ (normalized for the threshold voltage ($V_{th} = 1$)). The calculated values in this case are:

$$\begin{array}{lllll} \overline{F} & = & \sqrt{2} & & \\ b_0 & = & 1/2 & c_1 & = & -1/2 \\ b_1 & = & 3/2 & c_2 & = & -(2\sqrt{2}-1)/2 \end{array}$$

Figure 3.3b shows the results when using the strobe functions. There is one big disadvantage. When taking into account the reverse polarities needed to drive the display (netto no DC voltage), there is need for 9 different voltage levels, which is not so practical. Figure 3.3c uses a different set of orthogonal functions, and only needs 7 voltage levels. Figure 3.3d only needs 5.

The advantage of MLA is that with this technique, fast responding LCs can be used so the refresh rate can be increased. But, more importantly in this section, the maximum voltage supply can be reduced. For a constant RMS value, the maximum voltage will be lower if the voltage pulses are more distributed in time, instead of one big pulse. This is why this technique is extensively used in OLED displays [4]. Instead of providing a desired light output by giving the OLEDs one big pulse, which reduces their lifetime significantly, the lower pulses are more spread out over the entire frame time.

3.3 Creating free-form displays

On a somewhat lighter note, I will give some examples of systems aiming to provide a free-form display. A free-form display is a display without a fixed shape. It can be transformed and changed to suit the user's wishes. In its simplest form, this could just mean that the display is scalable. In this case the display retains its rectangular shape, only the size and aspect ratio can be changed.

3.3.1 Tiled displays

Tiled displays are in se not really a specific system, but more a collective noun of systems where several displays work together to display one large image. These can be direct view displays, although projection based systems, where it's the projectors that are being tiled, are also considered to be tiled displays [5]. In any case, these displays (or projectors) will not communicate directly to each other, but rather through a central server.

The main issues, and research topics, are to make the transition between the several tiles seamless. When using direct view displays, there is always an edge around the display. One of the solutions to work around this problem, is to use light guides [6]. The light of the pixels is guided to create a slightly larger display area, that overlaps with the edges of the underlying displays. This problem is not

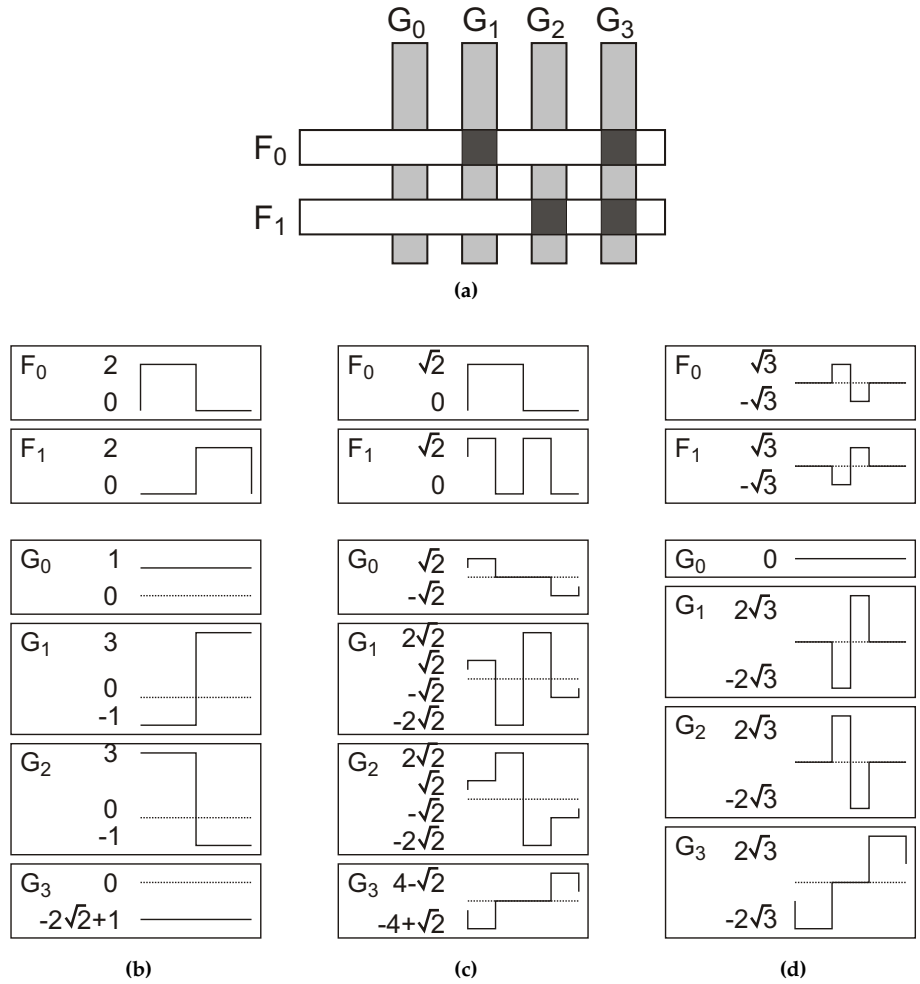


Figure 3.3 – MLA driving schemes for a 2-row display

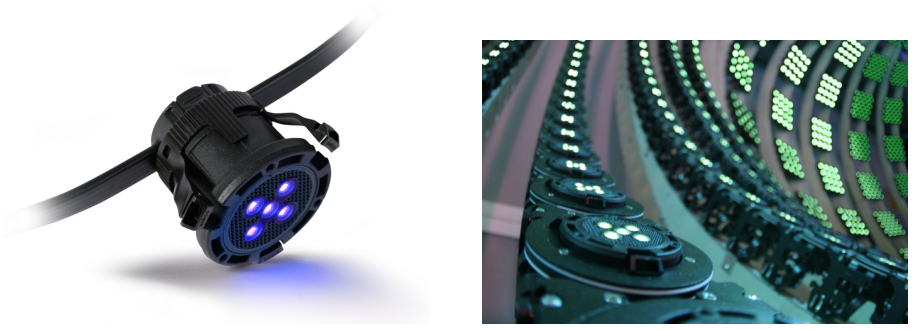


Figure 3.4 – Transformable LED from Barco

present using projection systems, however another issue is present. A display, or a projection, will always differ a little bit depending on the specific device that is used. Some will be a little brighter, or the colors are a little bit different, etc. A feedback loop is needed to try to circumvent this problem. A camera, connected to the central server, is placed in front of the screen to detect irregularities. This information can be used to adjust the images displayed, to create a seamless tiled display [7].

3.3.2 Transformable LED

The Transformable LED was developed by Barco. It consists of individual, compact, high quality LED pixel modules that can be used as building blocks to sculpt any design possible (See Figure 3.4). The pixel modules can be combined with a variety of specifically designed carriers (mechanical structures) into any shape customers desire [8].

3.3.3 CurveLED

CurveLED [9] is something a bit similar. It is a system that builds a display out of several LED ropes, a LED curtain. The ropes themselves can be separated after each pixel, so the length of the rope can be adjusted. They are mounted in a module that holds 8 ropes and the connection between two such modules can be bent 45°. Figure 3.5 shows the LED curtain in action.

3.3.4 FlyFire

Another amazing project (still a work in progress) is called FlyFire [10]. It's a project initiated by the SENSEable City Laboratory in collaboration with ARES Lab (Aerospace Robotics and Embedded Systems Laboratory) at MIT and says it

Figure 3.5 – CurveLED

"aims to transform any ordinary space into a highly immersive and interactive display environment".

Flyfire uses a large number of remotely controlled, self-organizing "micro helicopters". Each helicopter contains small LEDs and acts as a smart pixel. Through digitally controlled movements, the helicopters perform elaborate and synchronized choreographies, generating a unique free-form display in three-dimensional space. Using the self-stabilizing and precise controlling technology developed by the ARES Lab, the motion of the pixels is adaptable in real time. The Flyfire canvas can transform itself from one shape to another or bring a two-dimensional photographic image into an articulated shape. Today the researchers at MIT are able to simultaneously control a handful of micro helicopters, but they are aiming to scale up and reach very large numbers. Figure 3.6 shows one of the pixel helicopters, together with a (conceptual) image of a possible 3D free-form display.

3.4 Two birds, one stone: modular displays

3.4.1 What is it exactly?

Modular displays are displays that, as the name implies, consist of several modules. Each module is an independent display that can be connected to either a central server or microprocessor, or to another module. In contrast to tiled displays there is no point-to-point link to the central server, so we have to take in account some kind of routing algorithm to provide every module with data. Several network topologies are possible [11], each with its own advantages and disadvantages.

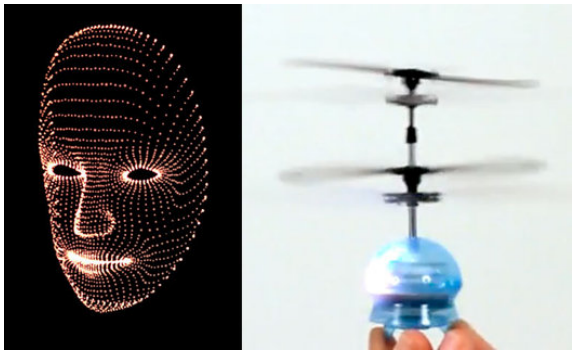


Figure 3.6 – Flyfire from MIT

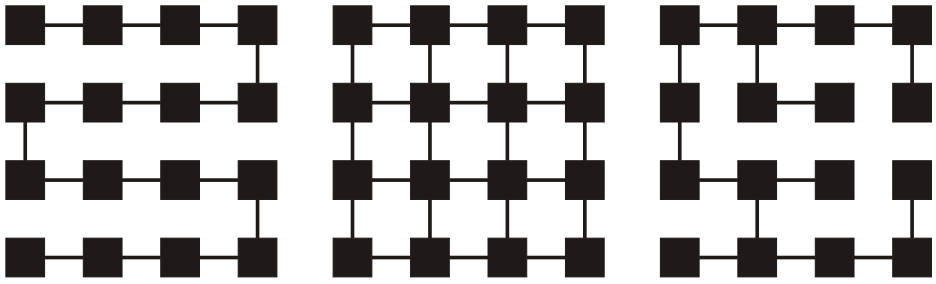


Figure 3.7 – Several network topologies for modular displays

The *bus* network (Figure 3.7a) provides a very simple solution of general connectivity for a small number of display modules, but does not scale well with a large number of display modules. The *mesh* network is the easiest to create with identical modules, but cannot route data efficiently. Data can get stuck in a loop, it keeps being sent from one module to another. The *tree* network has fewer links and no data loops, but is more susceptible to defects and errors in the nodes.

We'll see that, except for the driver discussed in Chapter 5, we'll start from a easy mesh network and transform it, through software, to a tree network optimized for that particular situation.

3.4.2 What can they solve?

Limitation of multiplexability

As explained, a passive matrix display can only have a limited amount of rows for a desired contrast. When the display is divided in several independent modules, this limitation only applies to the modules themselves, not to the display as a whole. It is an extension of the dual scan display, which can be seen as a modular display with two modules. A modular display has no theoretical limit in multiplexability. It is only a matter of having enough modules.

Reduction of brightness

Same goes for the limitation of brightness. You can add more rows to the display by adding modules, without having to reduce the time that one row is selected, i.e. without reducing the brightness.

Creating free-form displays

Free-form displays can be created by connecting the modules together to create a shape. Although routing algorithms need to be used to provide every module with data. More freedom in forms can be obtained by creating new ways for the modules to be connected. Can they be connected to form a 3D shape? Can the modules themselves have different shapes? In the next few chapters, a couple of modular display systems will be proposed.

References

[1] Wikipedia. Dual Scan. [Online]. Available: http://en.wikipedia.org/wiki/Dual_Scan

[2] C.-T. Lu, C.-I. Chao, K.-J. Ho, P.-Y. Chen, H.-M. Tsai, E.-C. Chang, and C.-K. Yen, "Multi-Scan-Line Passive-Matrix Organic EL Display," in *Proceedings of the International Display Manufacturing Conference (IDMC'07)*, 2005, pp. 730–731.

[3] J. Nehring and A. R. Kmetz, "Ultimate Limits for Matrix Addressing of RMS-Responding Liquid-Crystal Displays," *IEEE Transactions on Electron Devices*, vol. ED-26, no. 5, pp. 795–802, 1979.

[4] C. Xu, A. Karrenbauer, K. M. Soh, and C. Codrea, "Consecutive multiline addressing: A scheme for addressing PMOLEDs," *Journal of the SID*, vol. 16, no. 2, pp. 211–219, February 2008.

[5] Y. Yamaguchi, "Large Screen Display: Past, Present and Future," in *Proceedings of the* 17^{th} *International Display Research Conference (IDRC1997)*, 1997, pp. 278–280.

[6] A. C. Lowe, N. A. Gallen, and P. A. Bayley, "Tiling technology for large-area direct-view displays," *Journal of the SID*, vol. 14, no. 5, pp. 427–435, May 2006.

[7] A. Majumder, E. S. Bhasker, and R. Juang, "Advances towards high-resolution pack-and-go displays: A survey," *Journal of the SID*, vol. 16, no. 3, pp. 481–491, March 2008.

[8] Barco.com. Barco pushes creative boundaries with new transformable LED. [Online]. Available: http://www.barco.com/events/pressrelease/2257/

[9] CurveLED. CurveLED. [Online]. Available: http://www.curveled.net/

[10] Flyfire. Flyfire. [Online]. Available: http://senseable.mit.edu/flyfire/

[11] T. Ohkami, "Modular display: an approach to intelligent display systems," in *SID96 Symposium Digest of Technical Papers*, vol. 27, 1996, pp. 225–228.

I hear and I forget.
I see and I remember.
I do and I understand.

Confucius (551-479 BC)

4 Network and Communication Protocols

4.1 Introduction

This will be another small chapter. Last chapter introduced modular displays and showed that they form some sort of network. Each module is a node in the network and, depending of the functionality of the driver, these nodes need to communicate with each other. When this work was presented in conferences, some people asked why this or that protocol wasn't used instead of the one that was proposed. In this chapter, some network and communication protocols will be introduced. In the chapters about the drivers themselves (Chapters 5 to 8) we'll see why these protocols are or aren't applicable.

4.2 OSI 7 layer model

In 1984 ISO (International Organization for Standardization) developed the Open Systems Interconnection (OSI) model. It is a conceptual framework of standards for communication in a network across different equipment and applications [1]. It is an architectural model, used for inter-computing and inter-networking communications. In this model, the communication process is divided into 7 layers, each with clear characteristics. The goal is to have a communication process that

uses several protocols, residing on different layers. The layers and their protocols are independent from each other. A protocol on a higher layer doesn't know (and doesn't need to care) which protocols are used on the layers below. This simplifies implementation when a protocol on one of the layers is changed. Not all protocols can be allocated in one specific layer, though. Basically, layers 7 through 4 deal with end to end communications between data source and destinations, while layers 3 to 1 deal with communications between network devices. Since the communication between the modules mutually and between the modules and the microcontroller are low level, only the first two layers will be of any importance. Below is an overview of the layers with a short description [2].

1. *Physical Layer*
   The Physical Layer consists of the hardware transmission technologies of a network. It describes the relationship between a device and a physical medium, how the individual bits of information will be transferred. Examples are RS-232, I^2C, Bluetooth, DSL (Digital Subscriber Line, communication over telephone network), the physical layers of the USB and Ethernet protocol, etc.

2. *Data Link Layer*
   The Data Link Layer defines how data, as a whole, will be transferred between network entities. This includes to detect and possibly to correct errors that may occur in the Physical Layer. Data will be packaged in frames and the Physical Layer is used to send these packages to an adjacent node or put it on a shared medium. In this last case, the Data Link protocols will also specify how to deal with frame collisions. Frame collisions occur when two or more devices try to use the shared medium at the same time. Data Link protocols define how to detect and recover from those collisions. Since there are multiple devices on the shared medium, correct delivery of frames is done through the use of hardware addresses. A frame's header contains source and destination addresses that indicate which device originated the frame and which device is expected to receive and process it. These addresses are flat, they have no information about the logical or physical group to which the address belongs. Examples are the Ethernet protocol, IEEE 802.11 wireless LAN, Spanning Tree Protocol, parts of the USB, I^2C and RS-232 protocols, etc.

3. *Network Layer*
   While the Data Link Layer will only send a frame to the adjacent node, the Network Layer provides the functional and procedural means of transferring variable length data sequences from a source to a destination host via one or more networks. The Network Layer takes care of the routing of the frames sent by the Data Link Layer, while controlling the flow and congestion of packages. It uses normally a connectionless model. This means that

no connection is set up before sending. The package is just equipped with an address and sent off. This address will normally be assigned from a hierarchical system. As opposed to the addresses of the Data Link Layer, these addresses contain information about a logical group the address belongs to. This provides a way to route the packages. The Internet Protocol (IP) is an example of a Network Layer protocol.

4. *Transport Layer*
   Moving to the more application-oriented layers, the Transport Layer is responsible for delivering data to the appropriate application process on the host computers. For process-to-process communication, the Transport Layer uses port numbers. Together with the source and destination IP address, this constitutes a network socket, identifying the process-to-process communication. Usually, several processes on one device will access the network, so the Transport Layer protocols take care of the statistical multiplexing of data from different application processes, i.e. forming data packets, and adding source and destination port numbers in the header of each Transport Layer data packet. There can be either connection oriented communication (meaning that a connection is set up before sending) or connectionless communication. Using connection oriented communication, the data stream is divided into packages called segments. These segments are numbered making it possible to retrieve missing or erroneous segments and re-order out-of-order data, making it an end-to-end reliable communication. The best known example is the Transfer Control Protocol (TCP) used in the Internet Protocol Suite (TCP/IP). The connectionless communication doesn't offer a reliable end-to-end connection, but can be a lot faster. An example is the User Datagram Protocol (UDP), used for streaming video.

5. *Session Layer*
   The Session Layer provides the mechanism for opening, closing and managing a session between end-user application processes. Communication sessions consist of service requests and service responses that occur between applications located in different network devices. These requests and responses are coordinated by protocols implemented at the session layer. It provides authentications and permissions (e.g. SSH).

6. *Presentation Layer*
   The Presentation Layer manages the presentation of the information in an ordered and meaningful manner. This layer's primary function is the syntax and semantics of the data transmission. It converts local host computer data representations into a standard network format for transmission on the network. On the receiving side, it changes the network format into the appropriate host computer's format so that data can be utilized indepen-

dent of the host computer. Its main functions are conversion, encryption and compression.

7. *Application Layer*
   The Application Layer provides a means for the user to access information on the network through an application. This layer is the main interface for the user to interact with the application and therefore the network. This layer interacts with software applications that implement a communicating component. Such application programs fall outside the scope of the OSI model. Application layer functions typically include identifying communication partners, determining resource availability, and synchronizing communication. When identifying communication partners, the application layer determines the identity and availability of communication partners for an application with data to transmit. When determining resource availability, the application layer must decide whether sufficient network resources for the requested communication exist. In synchronizing communication, all communication between applications requires cooperation that is managed by the application layer. Some examples of application layer implementations include Telnet, File Transfer Protocol (FTP), Simple Mail Transfer Protocol (SMTP), HyperText Transfer Protocol (HTTP), etc.

4.3 Protocols

In this section, some protocols will be discussed. It would be useless (and impossible) to discuss them all, so I picked a few that might seem relevant to the network that will be generated with our modular display, keeping in mind that we want to keep everything as simple as possible. The protocols discussed here, will belong in one of the first two layers of the OSI model.

4.3.1 I^2C

I^2C (Inter-Integrated Circuit) is a multi-master serial single-ended computer bus. It uses two bidirectional open-drain lines, Serial Data Line (SDA) and Serial Clock (SCL), pulled up with resistors (See Figure 4.1). The used addresses are 7 bits long. With 16 reserved addresses, a maximum of 112 nodes can communicate on the same bus. A node can take one of two roles: master (node that issues the clock and addresses slaves) and slave (node that receives the clock line and address). The bus is a multi-master bus which means any number of master nodes can be present, although most systems only include one. Additionally, master and slave roles may be changed between messages (after a STOP bit is sent) [3].

The I^2C-compatible hardware slave device come with a predefined address. At the beginning of a transaction, the master node transmits the device address of

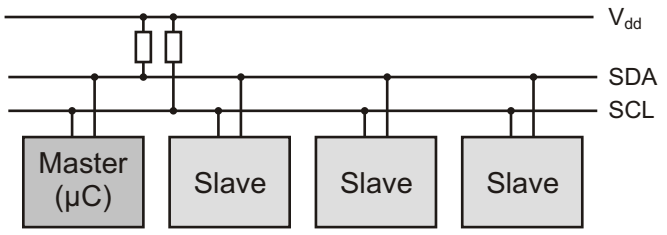


Figure 4.1 – A sample schematic of I^2C-use with one master (a microcontroller) and three slave nodes

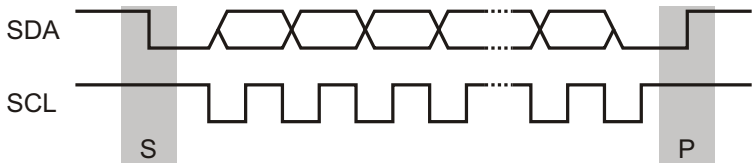


Figure 4.2 – Data transfer is initiated with the START bit (S). Then, the bits on the SDA line are sampled when SCL is high. When the transfer is complete, a STOP bit (P) is sent.

the intended slave. The slaves are responsible for checking and recognizing their own address. To be able to connect several identical devices on the line, some of the address bits are configurable at the board level. The limit is then set by the number of user-configurable address bits (typically two bits, allowing up to four identical devices).

Communication starts when the master sends a start bit followed by the address of the slave it wishes to communicate with. This is followed with a bit, indicating whether it wishes to write (0) to or read (1) from the slave. The slave, if it exists, will respond with an acknowledgment bit (active low). According to the read/write bit it send, the master continues in either transmit or receive mode. The slave chooses the complementary mode. Data is sent with most significant bit first. The start bit is indicated by a high-to-low transition of SDA with SCL high; the stop bit is indicated by a low-to-high transition of SDA with SCL high. When the master writes to a slave, it will send each byte separately with the slave sending an ACK bit after each byte. The same process happens when the master wishes to read. The slave will send the bytes, while the master responds with an ACK bit after every byte. Transmission is ended by the master with a stop bit. It may also send another start bit, if it wishes to retain control of the bus for another transfer (See Figure 4.2).

Name		Purpose
Data Terminal Ready	(DTR)	Tells modem that PC is ready to be connected.
Data Carrier Detect	(DCD)	Tells PC that modem is connected to telephone line.
Data Set Ready	(DSR)	Tells PC that modem is ready to receive commands or data.
Request To Send	(RTS)	Tells modem to prepare to accept data from PC.
Clear To Send	(CTS)	Acknowledges RTS and allows PC to transmit.
Transmitted Data	(TxD)	Carries data from PC to modem
Received Data	(RxD)	Carries data from modem to PC.
Common Ground	(GND)	Common return path for all signals.

Table 4.1 – Signals used by the RS-232 protocol, for example between a PC and a modem.

4.3.2 RS-232

The RS-232 (Recommended Standard 232) serial communication protocol is a standard protocol used in asynchronous serial communication. It is commonly used in computer serial ports. Its original purpose was to connect a terminal with a modem. The serial communication link between two devices comprises several components [4]. There are the devices themselves, obviously, with an UART (Universal asynchronous receiver/transmitter) possibly accompanied with a voltage converter circuit, and the serial channel itself. The UART takes bytes of data and transmits the individual bits in a sequential fashion. At the destination, a second UART re-assembles the bits into complete bytes. The voltage converter circuit will translate the voltage levels generated by the UART (normally TTL/CMOS voltages) to voltages defined by the RS-232 protocol. The standard requires the transmitter to use +12V and -12V, but requires the receiver to distinguish voltages as low as +3V and -3V. Some manufacturers therefore built transmitters that supplied +5V and -5V and labeled them as "RS-232 compatible". The serial channel itself can have up to 22 wires. Besides the two data wires (one for each flow of direction, RxD and TxD) and the common ground wire (GND), there can be a lot of control wires, but not all of them are always needed. See Table 4.1 for the most used signals. In its minimal form, RS-232 communication only requires 3 wires: RxD, TxD and GND (or even two if data only flows in one direction).

In the control signals, no clock signal is available. The devices communicate with each other at a previously established bit rate. The bits are arranged in 7 or 8 bit words, encapsulated between a start and stop bit. It's also possible to include a parity (control) bit. There are a couple of ways to implement a parity bit. You can

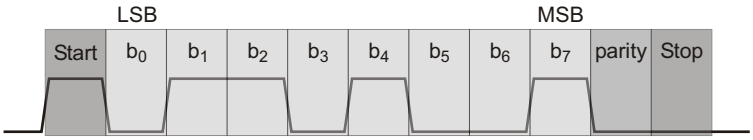


Figure 4.3 – RS-232 communication

set it to be always 1, or always 0. If the parity bit is wrong, there's a good chance there are more errors in the frame. You can also set the parity bit to a value that makes the total number of 1's or 0's in the frame odd or even. Figure 4.3 shows an example.

4.3.3 SPI

The Serial Peripheral Interface (SPI) circuit is a synchronous serial data link. It is basically "three-wire plus slave selects" serial bus, with one master and several slaves. [3]. Each device on the bus acts simultaneously as a transmitter and receiver. Two of the three lines transfer data (one line for each direction: MOSI (Master Out, Slave In) and MISO (Master In, Slave Out)) and the third is a serial clock (SCLK) (See Figure 4.4). Each slave has a slave select input (chip enable), which has a dedicated line to the master node (hence "three-wire plus slave select"). When the master wishes to communicate with one of the slaves, it will enable the slave using its slave select wire. The master will generate the clock signal.

The SPI bus employs a simple shift register data transfer scheme: Data is clocked out of and into the active devices in a first-in, first-out fashion. It is in this manner that SPI devices transmit and receive in full duplex mode.

4.3.4 USB

While previous protocols were mostly situated in the Physical Layer of the OSI model, the Universal Serial Bus (USB) protocol can be considered also to be part of the Data Link Layer. It provides the means to communicate with several devices through a single medium (using addresses). The USB protocol allows for 127 devices to be connected to a host controller. These devices can be connected in a tiered-star topology (See Figure 4.5). In a star topology, all devices are directly connected to the controller (no common line). A tiered-star topology is a star topology where every "ray" of the star can be another star (by using a USB hub). Sometimes devices are also functioning as a hub. In this way, some sort of tree structure can be created with several levels (tiers). However, the USB protocol specifications tell us that only five tiers can be used [5].

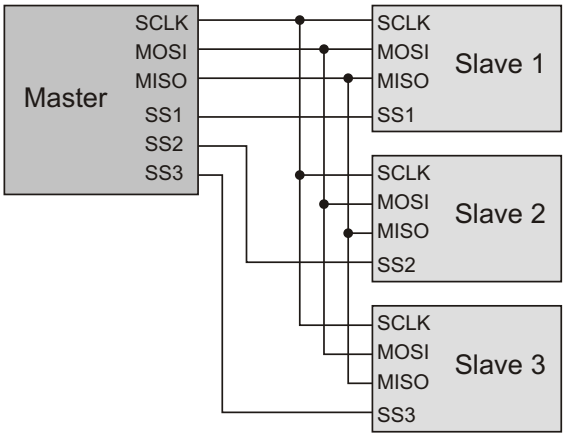


Figure 4.4 – Configuration for SPI communication.

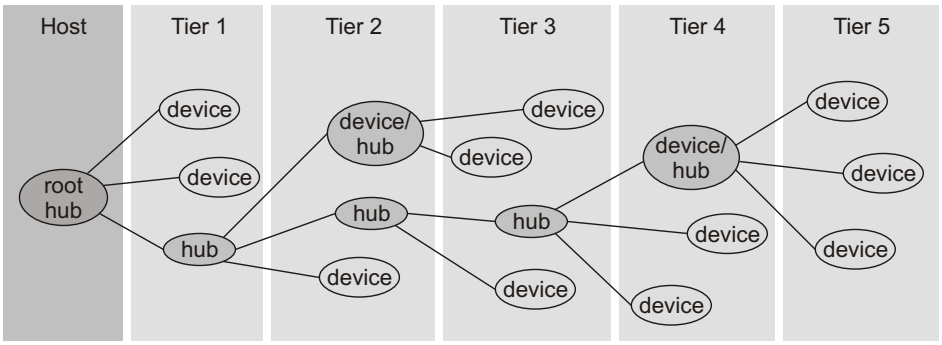


Figure 4.5 – The tiered-star topology for the USB protocol.

At the Physical Layer, the USB consists of 4 wires. V_{DD} (5V), ground and two data wires (D+ and D-). The data wires are used differentially (both wires take opposite values). The data itself is sent using NRZI (Non-Return to Zero, Inverted) encoding. NRZI-encoding means that a logical '0' causes a transition on the data wires, while a logical '1' causes no transition. To make sure that there are at least some signal transitions, USB also uses bit stuffing. When bit stuffing is used an extra 0 bit is inserted after any series of six 1 bits. This means that seven consecutive 1 bits always results in an error. Data is sent in packages of different lengths, depending on the type of package. Each package is preceded with a synchronization sequence, a package identifier (PID) and ends with error checking bits (CRC) and an end-of-package identifier (EOP). At the host (or hub) side, D+ and D- are pulled down with a 15kΩ resistor. A device has a 1.5kΩ pull-up resistor, which will pull up the line. This way a connected device can be detected. A low-speed device will pull up the D- wire, a full-speed device pulls up the D+ wire. When a new device is connected, initialization (or enumeration) is started, where the device will receive its address. But more about this later. First, let's take a look at how USB operates at a higher level.

A connection (called pipe) is made between the host and an endpoint on a device. Each endpoint represents a part of a device that fulfills one specific purpose for that device, such as to receive commands or transmit data. A full speed device can have up to 16 endpoints, though low speed devices can have only three. Several endpoints can account for one function of the device, and a device can consist of several functions. The endpoints are numbered (set at design time). To set up a pipe with one endpoint, both the address of the device itself and the desired endpoint number are used. Endpoint 0 is supported by all devices when powered up. This endpoint is the target of the default pipe. After the attachment of a device has been detected, endpoint 0 is used to initialize the device.

I will not go into the details about the different types of packages and pipes, because that would take us too far. I will, however, mention briefly the initialization process and data transmission because of its relevance to the next chapters. In short, there are handshake packages (e.g. ACK as response to a data package), token packages (only sent by the host, could be seen as control packages, always contains address and endpoint number) and data packages (containing the actual data). For example, the host wants to send some data to a device with address A (endpoint E). First, a token package is sent (OUT) indicating that the host wants to send something. This package will contain the address A and endpoint number E of the specific device. The next package will be a data package (max. 1024 bytes in full-speed devices, 8 bytes in low-speed devices), which will only be processed by device A, which will send it to its endpoint E. This device will respond with a handshake package (ACK). For every new data package, a new token package needs to be sent. Initialization works with the same principle. As said, when a device is connected, it will pull one of the data wires high. When this is detected,

the host put the device in the reset state by pulling both data wires low. When hubs are used, it's the hub that will detect the new device. It will use its endpoint 0 to let the host know about the new device. The host will then command the hub to put the new device in the reset state (pulling both data wires low). After the device is reset it will respond to address 0. In other words, during configuration, the host will always set up a pipe with (address 0, endpoint 0). The host can now communicate with this device and by using special tokens (setup token) can ask for the device description and give it a specific address.

4.3.5 Ethernet

Another technology that can find its place in both the Physical Layer and the Data Link Layer is Ethernet. It is actually not one specific technology, but rather a family of frame-based computer networking technologies for local area networks (LANs). Systems can be connected with twisted pair cabling (UTP), coaxial cable, optic fiber, etc. It was originally based on the idea of computers communicating over a shared coaxial cable acting as a broadcast transmission medium. To let this happen in an orderly fashion, the CSMA/CD (Carrier Sense Multiple Access with Collision Detection) protocol is used [6]. Basically this protocol follows the same protocol that we use (when we're being polite) when talking in a group. Before we begin to talk, we check if nobody is talking. If it is quiet, we begin to talk and everybody who was planning on talking waits. If two people start talking at the same time, they both stop, wait a while, and one of them will start talking again first. CSMA/CD uses the same principle. Each system continuously listens for traffic on the medium to determine when gaps between frame transmissions occur (carrier sense). Systems may begin transmitting any time they detect that the network is quiet (multiple access). If two or more stations in the same CSMA/CD network (collision domain) begin transmitting at approximately the same time, the bit streams from the transmitting stations will interfere (collide) with each other, and both transmissions will be unreadable. If that happens, each transmitting station must be capable of detecting that a collision has occurred before it has finished sending its frame. Each must stop transmitting as soon as it has detected the collision and then must wait a quasi-random length of time (determined by a back-off algorithm) before attempting to retransmit the frame.

The advantage of CSMA/CD is that, all nodes can "see" each other directly. All "talkers" share the same medium - a single coaxial cable - however, this is also a limitation; with only one speaker at a time, packets have to be of a minimum size to guarantee that the leading edge of the propagating wave of the message gets to all parts of the medium before the transmitter stops transmitting. If not, not all collisions can be detected. This also means that the minimal size of a packet is relative to the total length of the medium.

The topology of an Ethernet network can take different forms. Originally only a

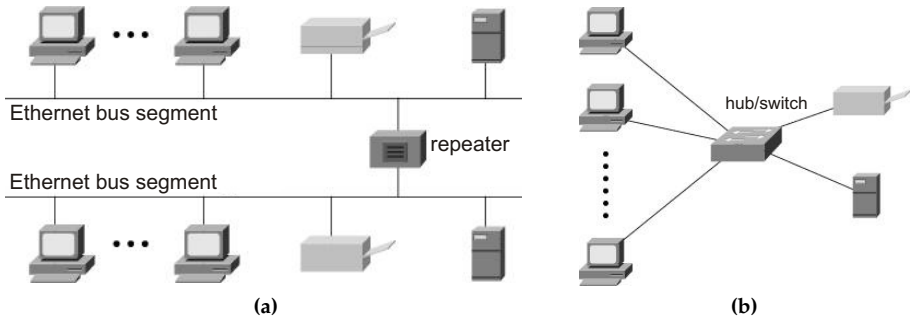


Figure 4.6 – Network topologies using Ethernet: (a) a bus network. (b) a star network

bus topology was possible (See Figure 4.6a). All devices are directly connected to each other with a single connection. As said, this single connection has a maximum length for the collision detection to work properly. The maximum length was also related to signal degradation: the farther the signal has to travel, the more transformed it gets, the more difficult it is to read. Later, *repeaters* were introduced to increase this maximal length. This solution tackles both the problem of collision detection and signal degradation. The repeater will amplify the signals, so it can travel farther. Collision detection is also implemented in the repeaters. When it detects a collision, it will send a jam signal, causing both transmitters to stop sending. This does not mean that you can expand the LAN indefinitely. The rule said one should only connect 5 segments on a single network (maximal 4 repeaters), and only 3 segments should be populated (i.e. have hosts attached).

Another improvement came when these repeaters had more than two ports. Repeaters with multiple ports are called *Ethernet hubs*. With hubs, a star-connected topology can be created (See Figure 4.6b). They reduce the complexity in cabling and provide a point-to-point cabling (one wire between two devices). When one of the cables is damaged, the rest of the network isn't affected. It is important to notice that the collision domain hasn't changed by introducing hubs. Hubs will simply forward everything they receive. This changed with the introduction of *Ethernet switches*. These switches can 'learn' where systems are located, and will only forward data for a certain system to a port where this system is connected to.

Packages are recognized, by the system and the switches, by a unique address: the LAN address (Ethernet address or MAC (Media Access Control) address). This is a 6 bytes long address which is given to every Ethernet adapter by its manufacturer. A complete Ethernet package starts with a preamble (8 bytes), followed by the MAC address of the destination (6 bytes) and the MAC address

of the source (6 bytes). The next two bytes indicate the Ethertype, the protocol that is used on the higher Network Layer. Then the data follows (46-1500 bytes) with 4 bytes for error checking. At the end, there is an interframe gap. This is the minimal time between frames, the minimal time a node has to wait and check the line before sending.

With a topology like in Figure 4.6b there is a considerable problem. When one of the switches breaks down, it's possible that a whole part of the network is disconnected from the rest. For this reason it is wise to provide redundant paths between nodes, so when one of the paths fails the other can take over. However, this too has a disadvantage. By inserting redundant paths, loops are created in the network. This can cause packages to wander around in the network. If this goes on unchecked, the network gets flooded. To solve this the Spanning Tree Protocol (STP) is developped [7]. In this protocol, the switches will communicate with each other to create a spanning tree. A spanning tree of a network is where all nodes are connected through exactly one path. If one of the connections would fail, the switches can be rerouted to create a new spanning tree. How this protocol works is said best in Radia Perlmans own words [7]:

Algorhyme

I think that I will never see
A graph more lovely than a tree.

A tree whose crucial property
Is loop-free connectivity.

A tree that must be sure to span
So packets can reach every LAN.

First the root must be selected
By ID, it is elected.

Least-cost paths from root are traced
In the tree, these paths are placed.

A mesh is made by folks like me
Then switches find a spanning tree.

–Radia Perlman

References

[1] Javvin Technologies, *Network Protocols Handbook*. Javvin Technologies, 2004.

[2] Wikipedia. OSI Model. [Online]. Available: http://en.wikipedia.org/wiki/OSI_Reference_Model

[3] *Application Note: I^2C Manual*, Philips, 2003.

[4] Wikipedia. Rs-232. [Online]. Available: http://en.wikipedia.org/wiki/RS-232

[5] Beyond Logic. USB in a NutShell. [Online]. Available: http://www.beyondlogic.org/usbnutshell/usb1.htm

[6] Cisco Systems, *Internetworking technologies handbook*. Ciscopress, 2004. [Online]. Available: http://www.cisco.com/en/US/docs/internetworking/technology/handbook/Ethernet.html

[7] R. Perlman, *Interconnections: bridges, routers, switches, and internetworking protocols*. Addison-Wesley, 2000.

[8] J. F. Kurose and K. W. Ross, *Computernetwerken, een top-down benadering*. Pearson, 2003.

Research is what I'm doing
when I don't know what I'm doing.
Wernher Von Braun (1912-1977)

5 A first modular display driver

5.1 Introduction

In this chapter, we'll talk about the design of a first, simple, modular display driver. The prime objective of this driver was to increase the multiplexability of passive-matrix displays, as discussed in Chapter 2 , but, as will be shown in Section 10.5, other applications can also benefit from this design. The first couple of sections deal with the functionality of the driver. Afterwards we'll take a look at some first testing results.

5.2 Requirements

5.2.1 The display

Before trying to determine what the driver is supposed to do, we should take a look at the display itself. As said, we want to use the modular structure to increase the multiplexability of passive-matrix displays. The multiplexability will be increased when the row electrodes (or groups of row electrodes) become independent. We can, for example, create long rectangular modules which run over the entire width of the display. This is actually the same approach the dual and quad scan displays take. Only now, the number of 'row groups' can be expanded indefinitely. Another option is to create square modules and also group the col-

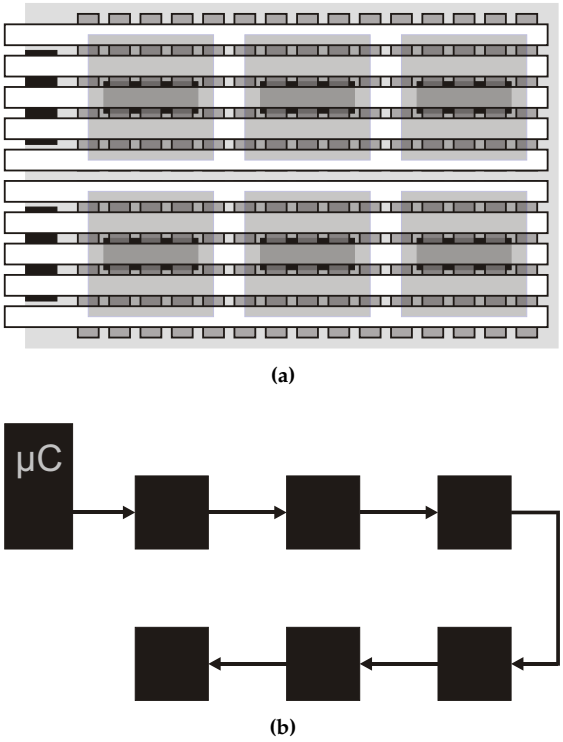


Figure 5.1 – A display configuration (a) and the corresponding driver configuration (b).

umn electrodes. The modules in one row can have common row electrodes that are driven by a simple row driver, or they can have completely independent row electrodes. In this last case, each module is completely independent and has an integrated row and column driver. The size and resolution of the modules will depend on the used display material.

In any case, the modules are in a fixed position. In order to minimize the number of connections on the display, each module has one input and one output. They are connected in a daisy-chain: the input of a module is the output of the previous module (See Figure 5.1). Only the first module is directly connected to a microcontroller. This microcontroller provides the communication between the display and the PC. To connect every module with the microcontroller, each module has an internal bypass. When activated, all incoming data will be forwarded to the output. When all bypasses are activated, all modules appear to be connected to one data line. The created network will have a bus topology (Chapter 3).

5.2.2 The driver

Since all drivers are connected to the same data line, there is need for some kind of addressing. We do not want to hardcode an address in each driver. We want every module to be identical. The assigning of an address will have to happen through software. Furthermore, we want the driving parameters, such as timing information, current or voltage levels,... to be adjustable, since these can be dependent on external factors (e.g. temperature, used material). The microcontroller will send the info on how to adjust the driving parameters to all the chips at once. The parameter info is meant for every module, so it will not be accompanied by an address.

In the case were the row electrodes are not independent, there is also need for some kind of synchronization between the row drivers on the side and the modules of the corresponding row. When a row driver selects a row, all the column drivers on that row of modules have to put their data on the columns at the exact same time. In short, the column drivers will need to be able to distinguish and interpret image data, parameter data and (possibly) synchronization signals.

At first glance it seems that the I^2C protocol would be helpful. However, as said in Chapter 4 this protocol does not provide dynamic addressing (I^2C assumes a given, fixed address for each device). We also want to limit the number of connections on the display. The I^2C protocol needs an extra line to provide a clock signal to the devices. We're probably better of basing the protocol on RS-232. Since data only needs to flow in one direction (from PC to modules) this can be achieved by using only two wires: the data wire and the ground wire. Since RS-232 operates purely on the Physical Layer level, we will need to develop a protocol that takes care of the addressing of the modules.

5.3 Implementation

The block diagram of this driver is shown in Figure 5.2. In short, *Rx* and *Tx* are in charge of respectively the correct receiving and sending of the data streams. *Main Control* is, as expected, the backbone of the driver. It consists of a small state register and generates the control signals for the other blocks. The address, image data and parameter data are stored in the *Memory*. The *Sequencer* is the only part of the driver that is dependent on the used display material. It will generate the signals for the row and column electrodes depending on the information in the memory. In the next sections, we will take a look at the different blocks.

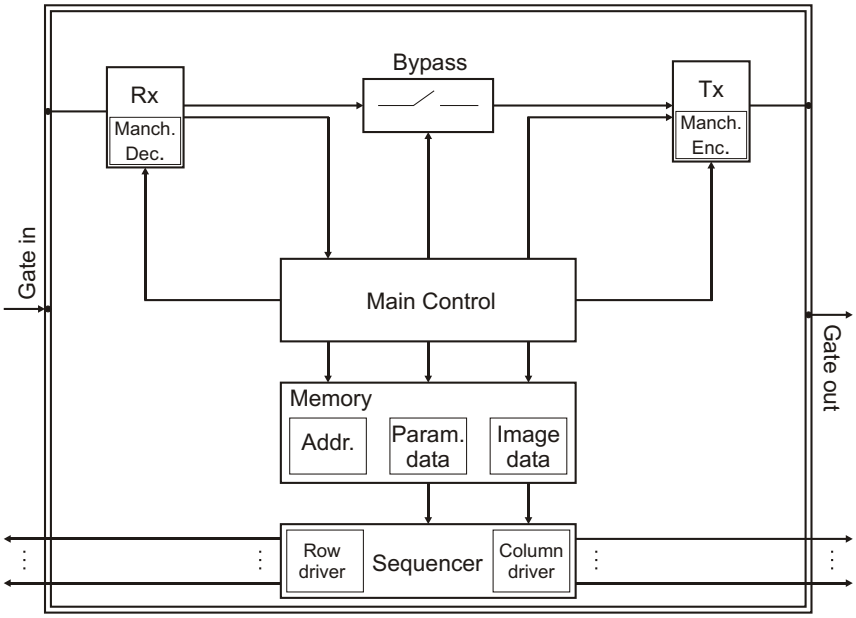


Figure 5.2 – The block diagram of the first modular display driver.

5.3.1 Communication protocol

First let's take a look at the used communication protocol. Data will only flow from the microcontroller to the modules, so there won't be any problems with data collisions on the data line. The microcontroller can just send data as he sees fit. Start and stop bits will indicate the beginning and end of a data stream. As with the RS-232 protocol, the start bit is a logical `0`, the stop bit a logical `1`. The data lines are equipped with pull-down resistors, so that, when nothing is driving the line, it will be read as a logical `0`. This does not really matter for this version of the modular display driver, but the reason for this will become clear in later chapters.

The data stream itself starts with a control bit, which indicates whether the following data is image data (`1`) or parameter data (`0`). In the case of image data, the following 8 bits are the address of the module the image data is meant for. Addresses are sent with least significant bit (LSB) first (See Figure 5.3).

A parameter sequence is 4 bytes long. The first byte controls the refresh rate of the display. The second byte tells the module the number of rows and columns used in the display. The module may be designed for an 8×8 display, but can adjust its waveforms to adequately drive a 7×5 display for example. The last two bytes can be used for display specific properties, e.g. the current level in a

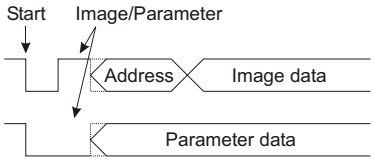


Figure 5.3 – Communication protocol for the first modular display driver.

LED display.

The length of the image sequences is dependent on the size of the display and determined by the second parameter byte. Each row in the display will be represented by one byte.

As with the RS-232 protocol, the modules communicate on a fixed frequency. Each module will need its own internal clock. When the data sequences are not very long, or when the clocks are very precise this doesn't create problems. When this is not the case, and a module reads the data sequence at a slightly different speed than the microcontroller sent it, the read data will be wrong. Since the timing errors will accumulate throughout the data sequence, the longer the sequence, the more chance of misreading the data. Very precise clocks can be created using crystal based clocks or by laser trimming the capacitors and resistors in an integrated clock to very precise values (See Chapter 9). These options are not always possible or are too expensive. A way to cope with this problem is to embed a clock signal in the data signal. The resulting signal is called a Manchester coded signal [1].

Instead of using a fixed voltage level per data bit (high level for `1`, low level for `0`), voltage transitions are used. A falling edge represents a logical `0`; a rising edge represents a logical `1`. This way, there is an edge in every data bit to which we can adjust the on-chip clock. As an example, 'Data' in Figure 5.4 represents the 3 bit long bit stream `011`. On the downside, the needed bandwidth is doubled and an extra action is required to decode the Manchester-coded signal.

Coding and decoding the Manchester code

Coding Manchester code is as simple as XOR-ing the data signal with the clock signal (See Figure 5.4). This also makes it clear what is meant by 'embedding the clock signal in the data signal'.

Decoding the code is a little bit more difficult. We have to look for transitions in the signal. If we can read the signal right after the transition in the middle of every bit, we're decoding the signal. The problem lies with the fact that not every transition is a transition in the middle of a bit. Sometimes there are transitions between bits (Manch(`11`) $\Rightarrow$ `0101`), sometimes there aren't (Manch(`10`) $\Rightarrow$ `0110`).

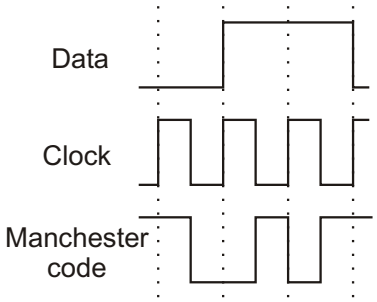


Figure 5.4 – An example of a manchester coded signal. The signal reads 011

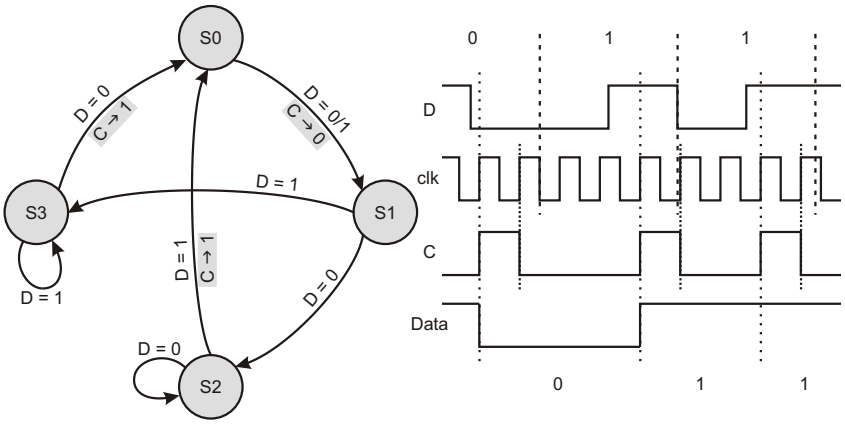


Figure 5.5 – Manchester decoder state diagram with corresponding signals.

There are a couple of ways to deal with this.

A first way is using the state machine from Figure 5.5. The state machine uses the coded signal D and a clock signal *clk* as input, and generates signal C, which has a rising edge somewhere after the transition in the middle of the bit of the coded signal. When D is read with a flip-flop clocked with signal C, the correct data is received. In the example in Figure 5.5, we start in state $S3$. At the rising edge of *clk*, D is changed to `0`. We move to state $S0$ and C is set to `1`. At *clk*'s next rising edge, we find ourselves in state $S1$ (independent of D) and C is reset to `0`. The next rising edge should lie in the first half of the next bit (This puts some restrictions on the frequency of the *clk* signal. This will be addressed below). According to D, we move towards $S2$ ($D = 0$) or $S3$ ($D = 1$). C remains unchanged as long as D keeps its value. When D changes (transition in the middle of the bit), C will be set to `1` again. Since *clk* has to have at least three rising edges per data bit to work correctly (three states per data bit: $S2/S3$, $S0$ and $S1$), the frequency of the *clk*

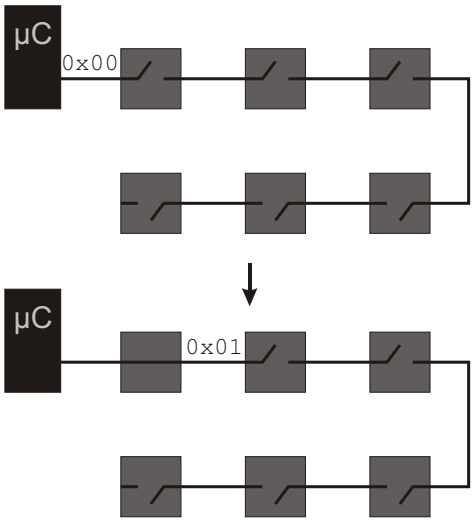


Figure 5.6 – The initialization process in the first modular display driver

signal f_{clk} should be at least $3f_{bit}$. As said above, the third rising edge of *clk* after the transition in the middle of the bit should lie in the first half of the next bit. If f_{clk} is too high ($> 5f_{bit}$), this rising edge can still occur in the same bit, causing the decoder to fail. This means the precise frequency of *clk* is not important. As long as $3f_{bit} < f_{clk} < 5f_{bit}$, the decoder will function properly. When we create an on-chip clock with a frequency set to $4f_{bit}$, there will not be any decoding problems, even with a frequency variance of 25% (assuming that the microcontroller uses a crystal based clock with a precise frequency).

When the frequency of the processing clock of the driver is much higher than the bit frequency, another option becomes available. Now it is just a matter of counting the time between two transitions. In a Manchester coded signal, there are 'long' (a whole bit) and 'short' (half a bit) periods between transitions. When a 'long' period has passed, the data must be read after the transition. When a 'short' period has passed, we must wait for the next transition to read the data (See also Figure 5.5). Differentiating between a 'long' and a 'short' period can be done by claiming that any period longer than 0.67 times the nominal bit time (t_{bit}) is a 'long' period. This way a clock variance of 33% can be dealt with ($0.67t_{bit} = (0.5 + 33\%)t_{bit} = (1 - 33\%)t_{bit}$).

5.3.2 General principles

When the display is turned on, all modules have their bypass inactive. The modules are in a wait state, waiting to receive an address. The microcontroller sends

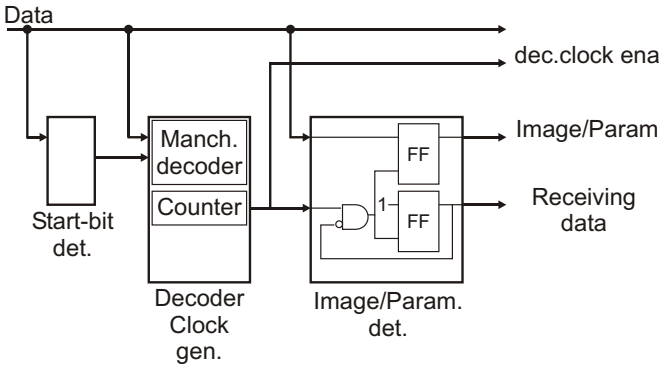


Figure 5.7 – Block diagram of *Rx*

out an address (e.g. `00000000`). This will only be seen by the first module. This module will now send an address to the next module, by adding `1` to the received address. This module will have address `00000001` and will send out address `00000010` (See Figure 5.6). When a module received and sent out an address, it will activate the bypass. When every module went through this initialization process, all modules will have an address and are directly connected to the microcontroller. They are now ready to receive data. If we need synchronization signals, we can use for example a specific address that we're sure that is unused (e.g. `00000000` if the microcontroller sends `00000001`).

5.3.3 *Rx* and *Tx*

As said, *Rx* and *Tx* will be used to send and receive data, controlled by *Main Control*. They have an internal Manchester coder/decoder. They are clocked at a speed of 20 times the bit rate, so, if necessary, we can use the second way of decoding the Manchester code (counting clock edges between transitions).

Rx

When *Rx* sees a falling edge on the data line after a while of inactivity (start bit), the *decoder clock generator* is activated (See Figure 5.7). This will generate the control signals (enable signals for the flip-flops) for correctly reading the data. This also includes the case where the signal is Manchester coded. In that case, the *decoder clock enable* will count the time between transitions so the signal is decoded as described above. *Rx* also presents two control bits, *Receiving data* (high when receiving data) and *Image/Param* (the first bit of the data sequence, indicating whether the data sequence is parameter or image data)

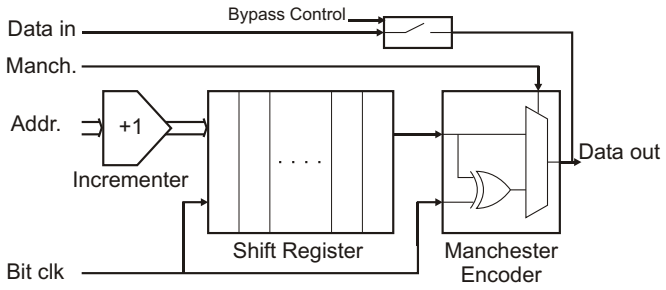


Figure 5.8 – Block diagram of *Tx*

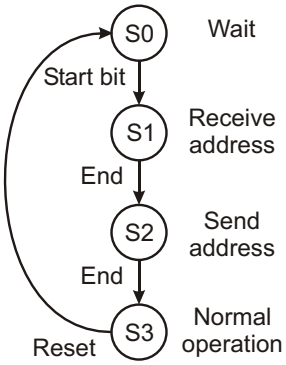


Figure 5.9 – The state diagram of the first modular display driver

Tx

Tx is nothing more than a simple shift register with some extra functionality (See Figure 5.8). When the address is received it is loaded in *Tx* which adds `1` and shifts it out the output at the correct speed (of course while making sure that the correct start and stop bits are used). When we're using a Manchester coded signal, it will XOR its output with a clock. After the initialization, the shift register is bypassed and the incoming data is directly forwarded to the output.

5.3.4 *Main Control*

Main Control generates the control signals for the other blocks. It can be seen as a (albeit very small) state machine. It might be interesting to look at, to see how it evolves over the next few drivers. There are four distinct states (See Figure 5.9).

We start in state *S*0 when the display is turned on. When a start bit sequence is detected (*Rx*) we move to state *S*1 where the address is received. *Main Control*

enables the address register and the *decoder clock enable* signal from *Rx* is used to read the address. When the address is received, we enter state *S*2 where the address for the next module is sent. *Main Control* loads the address in *Tx* and enables its shift register to send data. When the address is sent, we arrive at the final state *S*3. Here, the bypass is activated and the *Sequencer* is started. From this point on data can be received. After a reset, the state machines goes back to state *S*0.

Parallel with this state machine is the control block for the receiving of image and parameter data. This will be the same block in all four drivers. *Main Control* will enable the registers based on the data control bit and *decoder clock enable* signal from *Rx*. Before enabling the registers for image data, it will check if the address that was sent with the data stream matches its own address.

5.3.5 *Sequencer*

This block will be dependent on the used display material. It will also be the same in the four drivers. The *Sequencer* reads one byte from the image memory and presents it on the column electrodes while selecting the corresponding row. The unused rows and columns (depending on the information about the display size in the parameter memory) are left low. After a while the next byte is read and the next row selected. The speed of this process is determined by the refresh rate stored in the parameter memory. In Section 5.5 some more specific examples are elaborated.

5.4 A simple example

Figure 5.10 shows a simple example of a display configuration with the first modular display driver. The framed waveforms show some important signals within one module. The microcontroller sends out an address (in this case `0x00`) and the incrementing addresses propagate through the display. Address `0x01` reaches module *M*2 and is clocked by the *decoder clock enable*, after which *Tx* is enabled to send out the following address. When finished, the bypass is activated. At the end, some data (which is immediately seen by every module) enters. Since the first data bit is `0`, we're dealing with incoming parameters.

5.5 Setting up the test environment

We implemented the driver as described above in VHDL (VHSIC Hardware Description Language). This code is used to program an FPGA (Field-Programmable Gate Array). For reference, the source code of the most important block, the *Main*

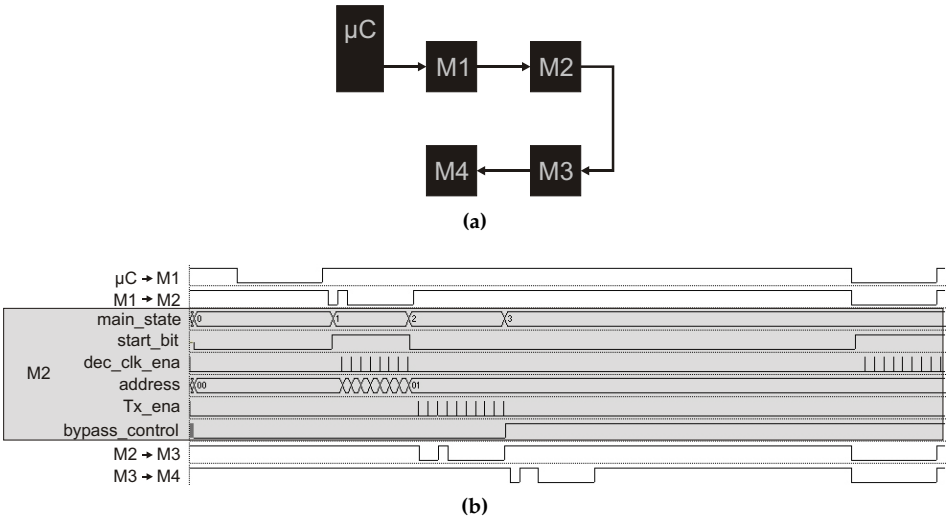


Figure 5.10 – An example display configuration (a) with the corresponding waveforms (b) using the first modular display driver.

Control is listed in Appendix A. FPGA's come in all sizes, with varying speed, number of logical elements, number of input and output pins, etc. The Altera Cyclone EP1C3T144C6N seemed to fit our purposes. It has 144 pins of which 104 can be used as input or output pins. With it's 2910 logical elements, it provides enough calculating power for our needs. It also has about 7.5 kB RAM memory. The design was optimized to work at a speed of 20MHz. The corresponding bit rate is 1MHz (See Section 5.3). Since this FPGA is a volatile device (programming is lost when power is cut), it's often combined on the test board with a non-volatile configuration device. In this case, the EPC2 device [2]. After programming this device, it will on its turn configure the FPGA every time the board is turned on.

This first driver was tested on a couple of displays. It was used to drive a cholesteric display (See Chapter 2) and a LED display.

5.5.1 Driving a ChLCD

Driving the display with the conventional minimal-swing drive scheme

As explained in Chapter 2, a ChLCD is a bistable display. The two stable states are the Focal Conic (FC) and Stable Planar (SP) state. A typical electro-optical response is shown in Figure 2.12. The electro-optical response of the ChLCD we used is shown in Figure. 5.11. It is a Polymer Stabilized Cholesteric Texture

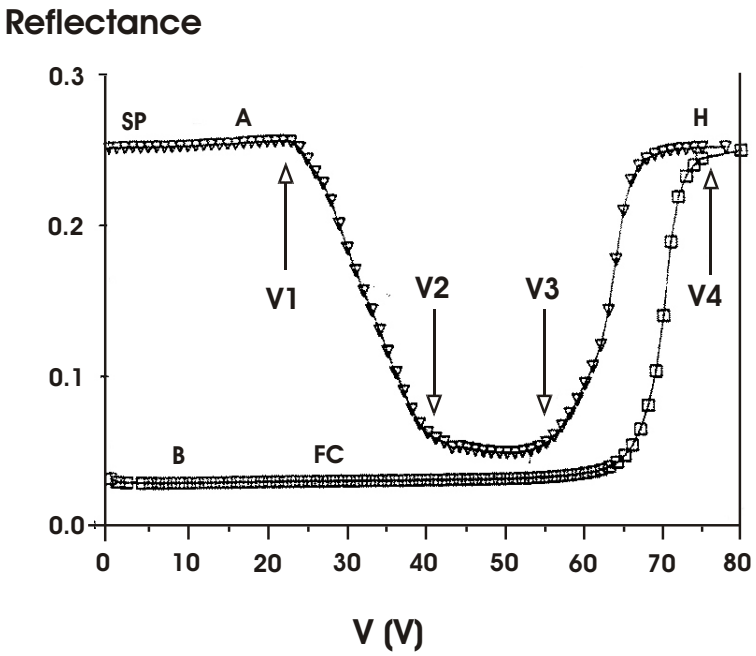


Figure 5.11 – The electro-optical response for a PSChT cell to a singe AC voltage pulse.

(PSChT) LC used by Kent Displays [3]. This kind of LC has the merit of a wide viewing angle due to the dispersed polymer in the LC that disturbs the orientation of the helical axes. The reflectivity of the cell was measured after a time delay of approximately 1 second following the pulse.

When the voltage level over the cholesteric liquid crystal cell is lower than V_1 the state of the liquid crystal will not change. The reflectivity of a cell in the SP state will start to decrease if the voltage level over the cell surpasses V_1. It decreases approximately linear with increasing voltage. When V_2 is reached, the liquid crystal will have changed to the stable FC state. If the voltage would be released now, the LC would remain in this state. Further increasing the voltage (above V_3) makes the reflectivity rise again linearly with increasing voltage. A voltage of V_4 forces the LC back into the SP state. Voltage levels on this last slope can also be used to acquire gray scales in the LC [4]. An important issue is that the ChLC can't sustain a DC voltage. The permanent polarization would pull the molecules apart. To solve this, every applied voltage has to be followed by a voltage of the same absolute value and duration, but with opposite sign.

The evolution of the LC is not only dependent on the voltage level, but also on the duration of the applied pulse. Figure 5.12 shows its influence. From Figure 5.12a

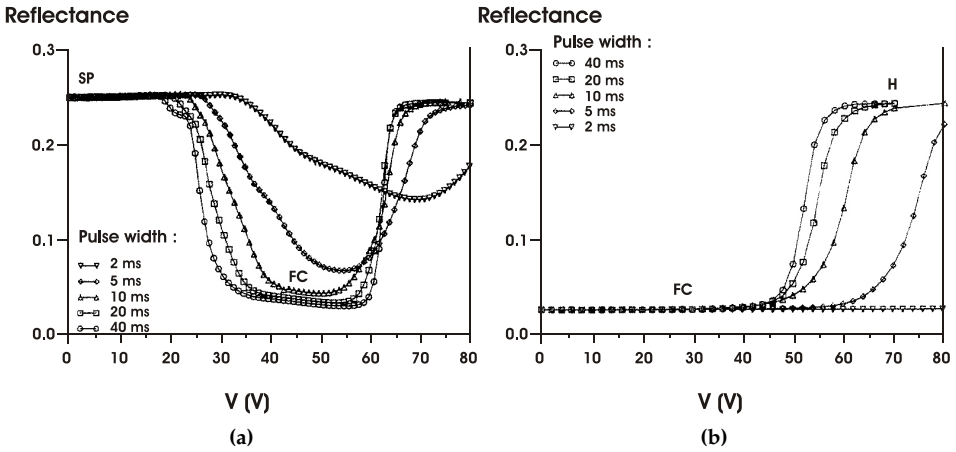


Figure 5.12 – Influence of the pulse width on the reflectivity of the PSChT cell. Starting from a cell in the SP state (a) and the FC state (b)

we learn that with pulses below 10ms , it becomes difficult to get a sufficient contrast. Figure 5.12b tells us that with faster pulses the voltage levels need to be higher for the cell to change to the SP state.

There are several driving schemes to drive a ChLCD [5]. We will use the conventional minimal-swing driving scheme, shown in Figure 5.13 (See also Chapter 2 on driving a display). With this driving scheme, the needed voltage swings are minimal and identical for the row and column electrodes. As you can see in Figure 5.13b, there will be no DC component over the pixel. The voltage levels to get into the FC and SP state are achieved in a selected row (S) while the voltage levels on a non-selected row (NS) remain low enough not to change the state. Grayscales can be achieved by dividing the line time in a portion with voltage levels for driving to the SP state, and a portion with voltage levels for driving to the FC state. This will force some cells in the pixel to the SP state, while others remain in the FC state, creating a grayscale. The longer the portion for driving to the SP state, the more cells are forced into the SP state, the more reflective the display becomes. For a better performance, the display needs a double reset (reset to SP, reset to FC) before writing the frame.

Figure 5.14 shows the used ChLCD. It is a 16×16 display. It is not really divided into modules. As said the column electrodes should be cut when grouping the row electrodes to make them electrically independent. But since this test is simply to check the implemented driver, we assumed that the display is divided in four modules in one row. Each module has 16×4 pixels.

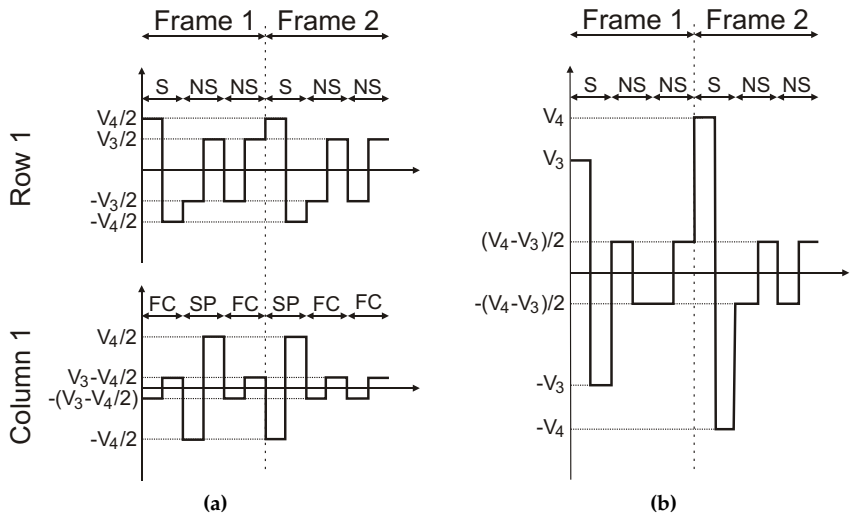


Figure 5.13 – The conventional minimal-swing driving scheme. (a) shows the voltage levels on the row and column electrodes. (b) shows the resulting voltage levels over the pixel.

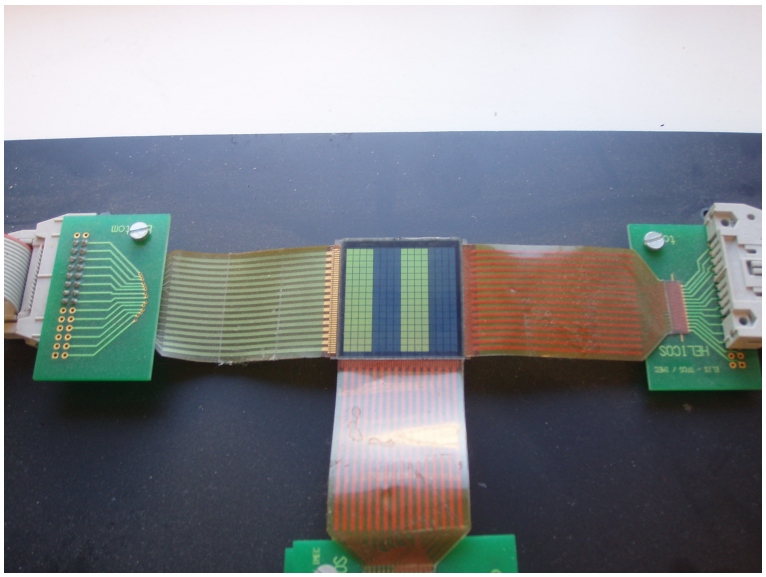


Figure 5.14 – The used ChLCD. It has four modules in one row, each with 16×4 pixels.

Design of the test boards

As is clear from Figure 5.11 and Figure 5.13, we need several and fairly high voltage levels to drive a ChLCD. It is not possible to directly drive the display from the FPGA. Instead, the *Sequencer* of the modular display driver will generate control signals for high-voltage multiplexers which, in their turn, will drive the display. Two boards were created. One board has the FGPAs with the necessary electronics, the other board is equipped with the high-voltage multiplexers. The reason these two are separated is because this way the board with FPGAs could still be used for other purposes. Since we created a modular display of the type where the modules are not completely separated (row electrodes for modules in one row are connected), the *Sequencer* in the driver can only play the part of the column driver. The row driver is common for the four modules.

The FPGA board is depicted in Figure 5.15. The four FPGAs each represent one modular display driver. At startup, they are all configured at the same time with the same configuration device (EPC2), ensuring that every driver is exactly the same. Each driver has a separate clock (Osc.), making them completely independent from one another. However, the clocks are generated by precise crystal oscillators (SG531, clocking at 20 MHz) , so no Manchester coding was needed. The DS89C420 from Dallas takes the role of the microcontroller. There is room for external memory and communication with the outside world is possible using RS-232 communication with the PC (MAX232 and a female RS-232 connector). The outputs of the modular display drivers (i.e. the outputs of the *Sequencer*) are accessible on pins on the board. These will be connected to pins on the multiplexer board.

The multiplexer board is shown in Figure 5.16. There are six high-voltage multiplexers [6] on this board. These are the DILA chips, created by Dr. Ir. Ann Monté at CMST. It has 8 multiplexers, each capable of multiplexing 8 voltages up to 100V. Each multiplexer has a 4-bit selecting input [5]. Two of the DILA chips are used to drive the row electrodes. They are controlled by a microcontroller, functioning as the row driver. Because of the limited number of ports on the microcontroller, latches are used to provide the DILA chips with the correct control signals. The other four DILA chips drive the row electrodes. In total, we can drive a display with 16×32 pixels. We will only use two out of those four DILA chips, though, to drive the display from Figure 5.14.

The control signals for the column-driving DILA chips come from the FPGA board. From an external source, the 8 high-voltage inputs (See Figure 5.13) are presented to the DILA chips.

Since there is one row driver and four column drivers, I told that there would be need for some synchronization between the drivers. Since driving a ChLCD is very slow (order of 10-20ms per line) there is no real need for synchronization between the column drivers. The *Sequencer* of the first driver will still be long

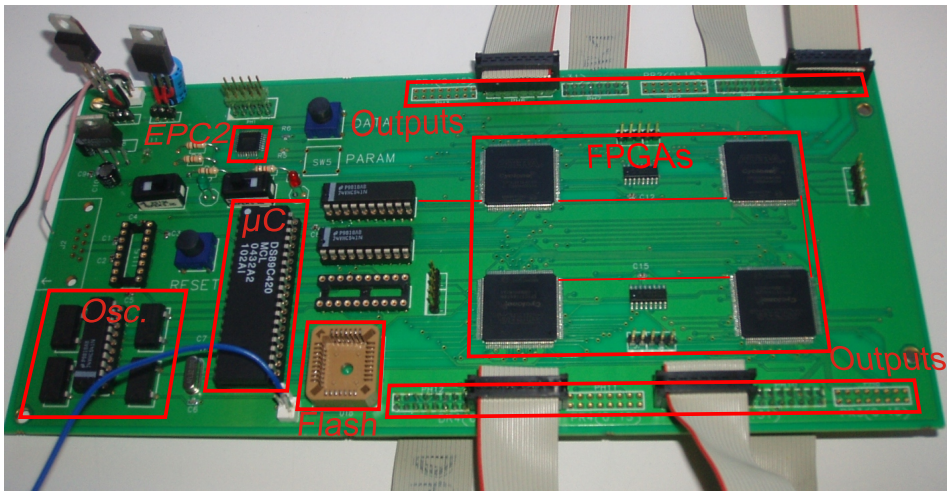


Figure 5.15 – The FPGA board. The four FPGAs each represent one modular display driver. The outputs of the FPGAs are visible on the pins on the board.

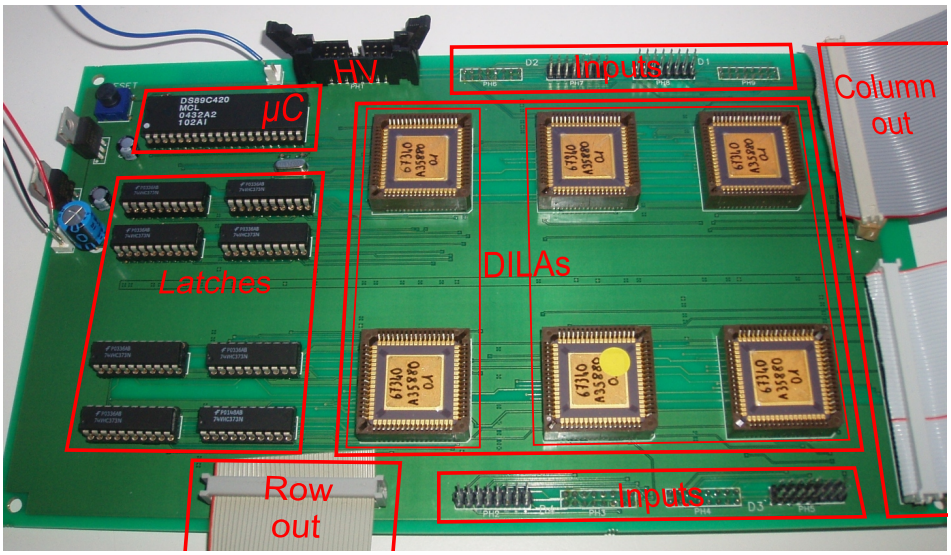


Figure 5.16 – The multiplexer board. There are six multiplexer chips (DILA). Two are used to drive the row electrodes, four are used to drive the column electrodes.

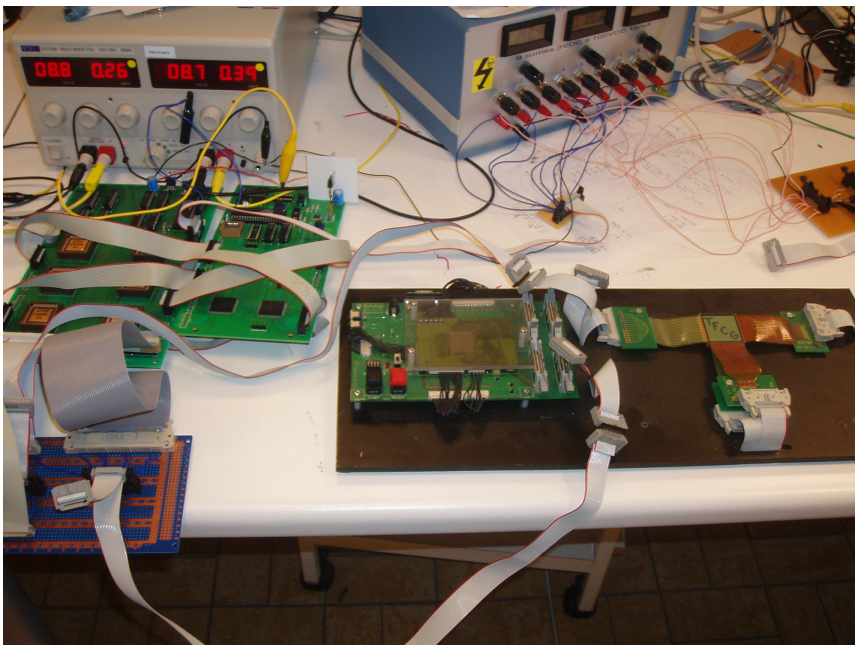


Figure 5.17 – Test setup for driving a ChLCD with the first modular display driver. High-voltage source is seen above. Left are the FPGA and multiplexer board. Right is the ChLCD.

from finished driving the first line when the last driver receives its data. The synchronization of the row driver with the column drivers happens through the pass-pulses the DILA chip needs to read the selecting input [5]. These are controlled by the row driver, so the row driver synchronizes the row signals by itself, with the column signals from the column drivers. The entire setup is shown in Figure 5.17.

5.5.2 Driving a LED display

Driving the display as a passive-matrix display

I use the term 'as a passive-matrix display' because technically, LEDs are active components (See Chapter 2). It just means that there aren't any transistors used to keep the LED on during the frame time. In a passive-matrix LED display, the cathodes of the LEDs in one row are connected to one row electrode, the anodes of the LEDs in one column are connected to one column electrode (See Figure 5.18). LEDs are current driven, rather than voltage driven. In this case, a selected row

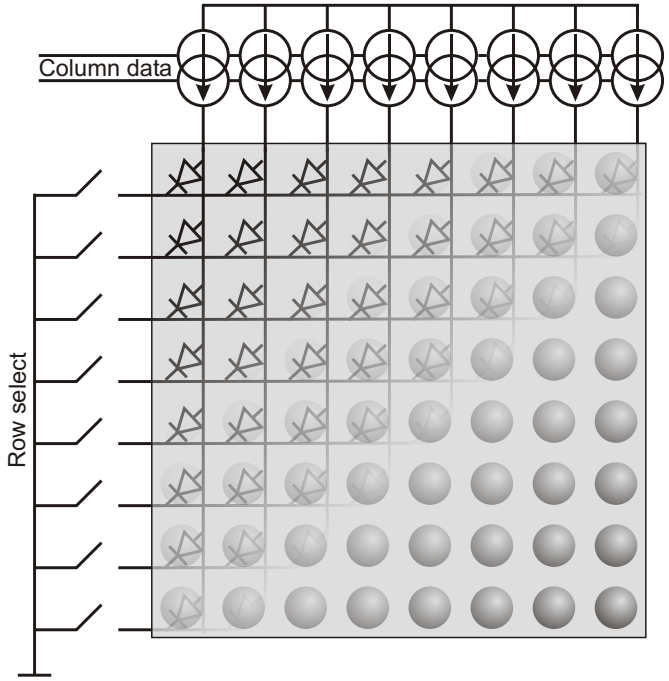
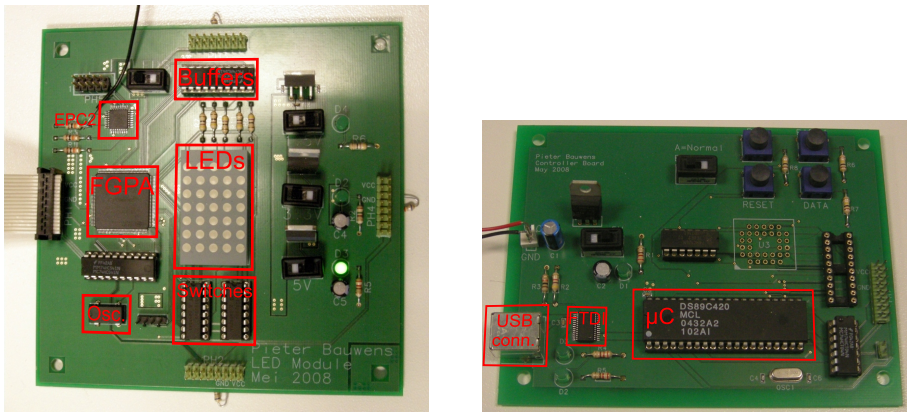


Figure 5.18 – A passive-matrix LED display.

is a row where the current can flow away , whereas a non-selected row is a row where no current can flow. When current is pushed into the column electrodes (depending on the desired image), only the LEDs on the selected row will emit light. Controlling whether in a row can flow current or not can be accomplished by placing switches on the row electrodes and activating (selected row) or deactivating (non-selected row) them.

Design of the test boards

As opposed to above, where the modules in the display were not completely independent (common row electrodes), we wanted to create a display with completely separable modules. For this reason we created some LED-display modules, shown in Figure 5.19a. We chose a LED display because they are easy to drive and easy to create. They will also be used to test the other versions of modular display drivers. Each module is equipped with an FPGA and its configuration device. The *Sequencer*s column-electrode outputs are connected to buffers (MM74HC541N) that can be used in this case as current sources when loaded with resistors. The row-electrode outputs are connected to switches (DG202BDJ).



(a) (b)

Figure 5.19 – A LED-display module (a) and the controller board (b). Each module has an FPGA, with the *Sequencer* acting as a row and column driver.

The LED display itself has 7×5 pixels (TA12-11 from Kingbright). Each FPGA has a 20MHz clock (SG531). The modules can be connected to each other by a series of pins. There are four groups of pins, making connection to any of the four sides possible. Two of each group of pins go to the FPGA (input and output), although for this modular display driver only one input and one output is necessary. The other pins are used for power routing and some extra pins from and to the FPGA as reserve.

With LED-display modules alone, of course, not much will happen. We need a controller board that is able to communicate with the modules and, if needed, with the PC. The controller board is shown in Figure 5.19b. It uses the Dallas DS89C420 microcontroller with the possibility of using external Flash memory. The board provides a USB interface to communicate with the PC. An FTDI chip (FT232RL) translates the signals from the USB to the RS-232 protocol the microcontroller uses. Through this interface, it is also possible to program the microcontroller.

5.6 Some first results

5.6.1 Results from the ChLCD

Using the test setup as described in previous section, we were able to successfully drive the ChLCD and, in doing so, we proved that the modules correctly received their address and that the image data reached the desired modules. The *Sequencer*

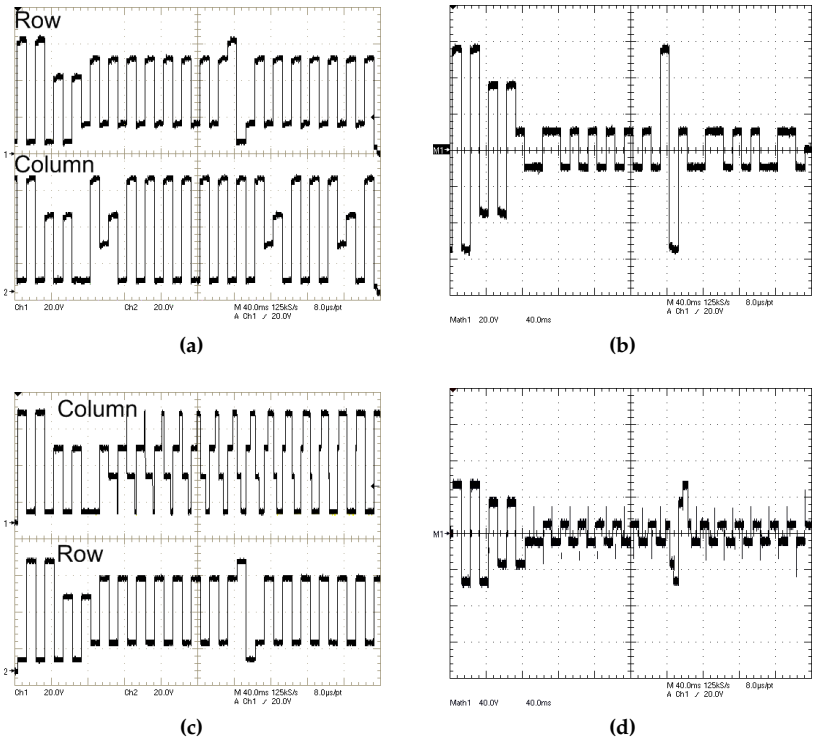


Figure 5.20 – Voltage levels on a row and column electrode of the ChLCD (a and c (with grayscale)) and over a pixel (b and d (with grayscale)), using the conventional minimal-swing driving scheme.

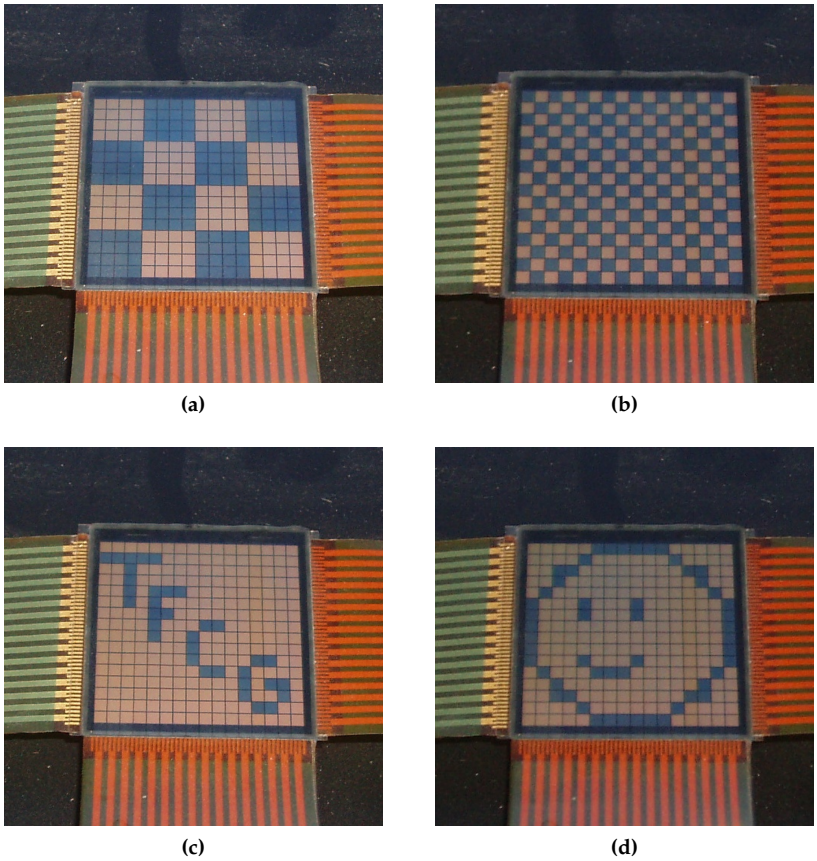


Figure 5.21 – Images on the ChLCD using the first modular display driver

designed to drive the ChLCD provided the control signals for the multiplexer board. The outputs of those multiplexers (column and row electrode) are shown in Figure 5.20a. The next figure (5.20b) shows the voltage levels as seen by the pixel. This corresponds with the desired waveforms as discussed in previous section, preceded by a double reset sequence. Figure 5.20c on the other hand, shows the voltage outputs when driving the display with a grayscale driving scheme, with the resulting voltage over the pixel in Figure 5.20d.

In Figure 5.21 you can see some images on the ChLCD.

5.6.2 Results from the LED display

Also the modular LED display worked as expected. To control the display, I created a Graphical User Interface (GUI) in Visual C++. A screen shot can be seen in Figure 5.22a. It allows the user to choose the size of display in a module (which will be sent as a parameter to the drivers) and the total size of the display (number of rows and columns of modules). The refresh rate, which is also sent as a parameter to the drivers, can also be adjusted. The desired image can be created by drawing with the mouse on screen. After choosing the correct COM port, the image (and parameters, if they have changed) can be sent to the drivers. With these independent modules, we can create a display configuration as depicted in Figure 5.1b. It is shown in Figure 5.22b. With this setup, we were able to control the displayed images. The drivers also responded correctly when the refresh rate and the number of used rows and columns changed.

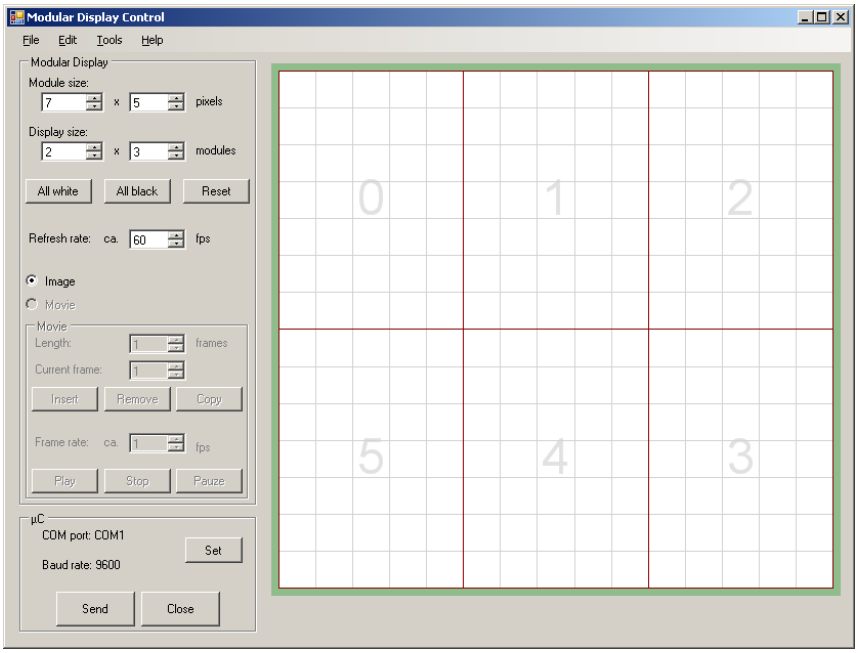
An interesting characteristic is the time needed to initialize the display. There is not a lot of information to be sent between modules (only the address), so this will be quite fast. Since the modules are connected in a chain, the total initialization time will obviously be dependent on the number of modules. With a data rate of 1Mbit/s, it takes a display about $9,6\mu$s per module to be initialized.

Section 10.5 will give some insight in the significance and possible applications of this driver.

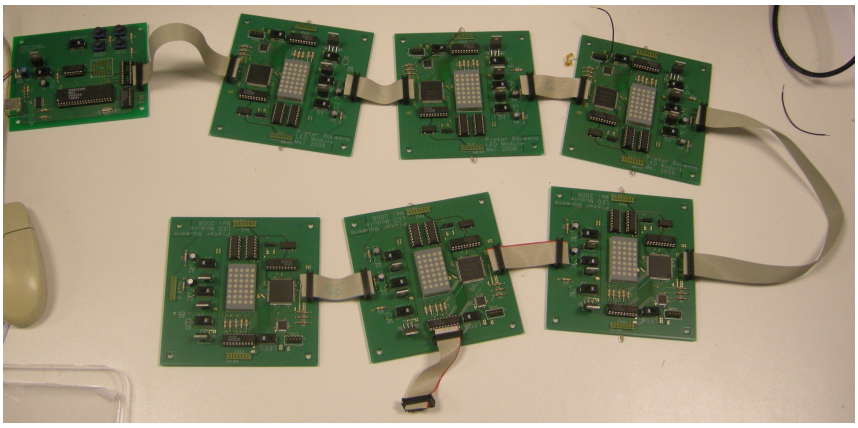
5.7 Can we do better?

Section 5.6 told us that this modular display driver can be used to remove the limitation in multiplexability in passive-matrix displays. Section 10.5 shows some other applications. While these are interesting results, we might do more with modular displays.

The first modular display driver, is designed for fixed displays. Displays where the drivers would be integrated and where the connections between the drivers cannot be changed. But what if we make the modules completely independent. What if we let the user create its own display by connecting modules to each other? And if we do that, we might as well allow him to create a more irregular shaped display.



(a)



(b)

Figure 5.22 – Configuration of a modular LED display, with the first modular display driver (b). The display can be controlled with a GUI (a).

References

[1] W. Stallings, *Data and Computer Communications*. Pearson, 2007.

[2] *Cyclone Device Handbook*, Altera Corporation, 2005.

[3] J. W. Doane, D. K. Yang, and Z. Yaniv, "Front-lit flat panel display from polymer stabilized cholesteric textures," in *Proceedings of the* 12^{th} *Int. Display Research Conf.*, 1992, pp. 73–76.

[4] A. Khan *et al.*, "Reflective cholesteric LCDs for electronic paper applications," in *Proceedings of the IDMC'05*, 2005, pp. 397–399.

[5] A. Monté, "Design of an Intelligent High-Voltage Display Driver to Minimize the Power Consumption in Bistable Displays," Ph.D. dissertation, Ghent University, 2008.

[6] J. Doutreloigne, H. D. Smet, and A. V. Calster, "A new architecture for monolithic low-power high-voltage display drivers," in *Proceedings of the* 20^{th} *International Display Research Conference (IDRC2000)*, 2000, pp. 115–118.

[7] P. J. Ashenden, *The VHDL Cookbook*. University of Adelaide, 1990.

A pessimist is one who makes difficulties of his opportunities.
An optimist is one who makes opportunities of his difficulties.

Harry Truman (1884-1972)

6 Improved modular display driver

6.1 Introduction

In this chapter we will take a look at an improved version of the first modular display driver. It can still be used to increase the multiplexability in passive-matrix displays, but it has some other applications as well (See Section 10.5). It provides more freedom to the user when creating a display. As we'll see in Section 6.2 we'll move away from the daisy chain structure, which will make the modules less dependent on one specific other module if this were to fail initialization. Some other issues will be lurking around the corner, but they will be dealt with in Section 6.3.

6.2 Requirements

6.2.1 The display

As opposed to the display for the first modular display driver, we want to have a display where the modules are not fixed in position. The display now consists of completely independent modules, each with its own separate driver. To let the user create a display as he sees fit, each module will have four input/output gates, so they can be connected to each other by any side (See Figure 6.1a). As was the case with the first modular display driver, only one of the modules will be directly connected to the microcontroller, which will provide the communication

with the PC. The modules have internal bypasses which, when activated after the initialization, will connect every module to the same data line.

6.2.2 The driver

The driver requirements explained in Section 5.2 have not changed. Every module needs an address and these addresses need to be assigned completely through software. There still needs to be a distinction between parameter data and image data. Since every module is independent, synchronization signals won't be necessary for now.

Since every module can have four connections, we will have created a mesh network which has some advantages and disadvantages [1]. The advantage over a bus (or daisy chain) network is that the display can have modules missing (or malfunctioning) without interfering with the rest of the display. The disadvantage, in our case, is that when the bypasses activate, there will be several open paths between modules. And if we're lucky (and we mostly are) there will be loops in the network. Since the bypasses have small delays, it's possible that data is stuck in a loop and keeps being sent from one module to another. For this reason, a primitive network protocol needs to be developed to cope with this.

We can, as in previous chapter base our communication protocol on RS-232, but since our network has become a little bit more complex, maybe other (network) protocols are more applicable. The USB protocol does provide our needed dynamic addressing, but the problem lies with the possible network topologies. Section 4.3 explains the tiered-star topology of the USB protocol. Theoretically we could create our display in such a way that it would fit in this topology, but we'll have to avoid the extra connections between modules that would make it into a mesh network. But, above all, the USB protocol specifications tell us that only five tiers can be used. This poses an unacceptable limitation on the possible display configurations. So, we will slightly change the network protocol from the first modular display driver and try to incorporate solutions for the issues raised above. This time, we'll need a two-way communication channel between the modules. So three wires are needed.

6.3 Implementation

As you can see in Figure 6.2, the block diagram of the driver resembles the previous driver for a great deal. Only *Rx*, *Tx* and the state machine in *Main Control* are slightly different. The communication protocol (i.e. start bit and stop bits, control bit for distinction parameter/image data) will also be the same.

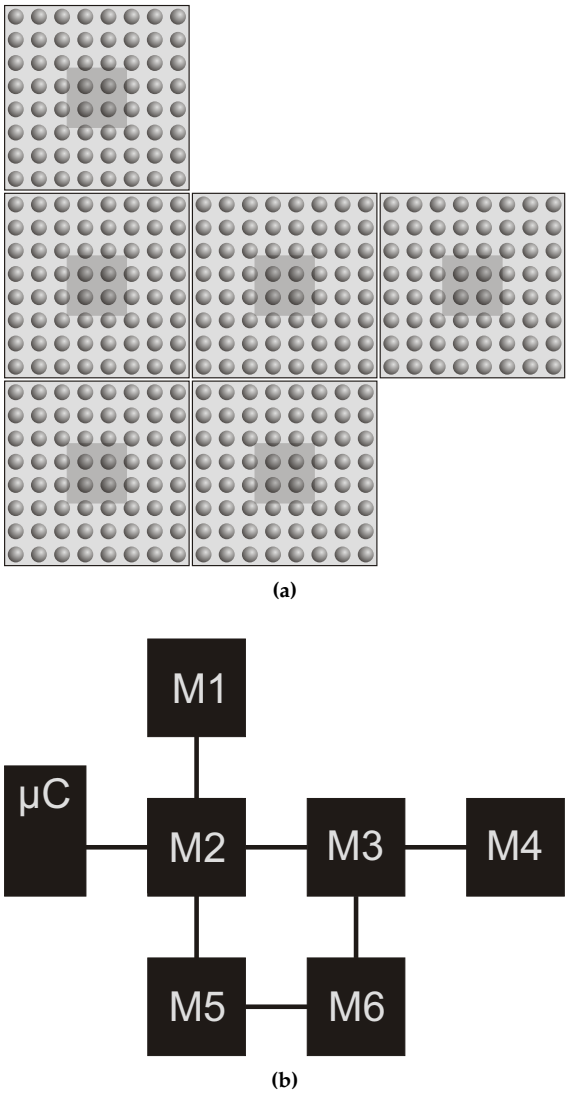


Figure 6.1 – A possible display configuration for the improved modular display driver (a) and the corresponding driver configuration (b).

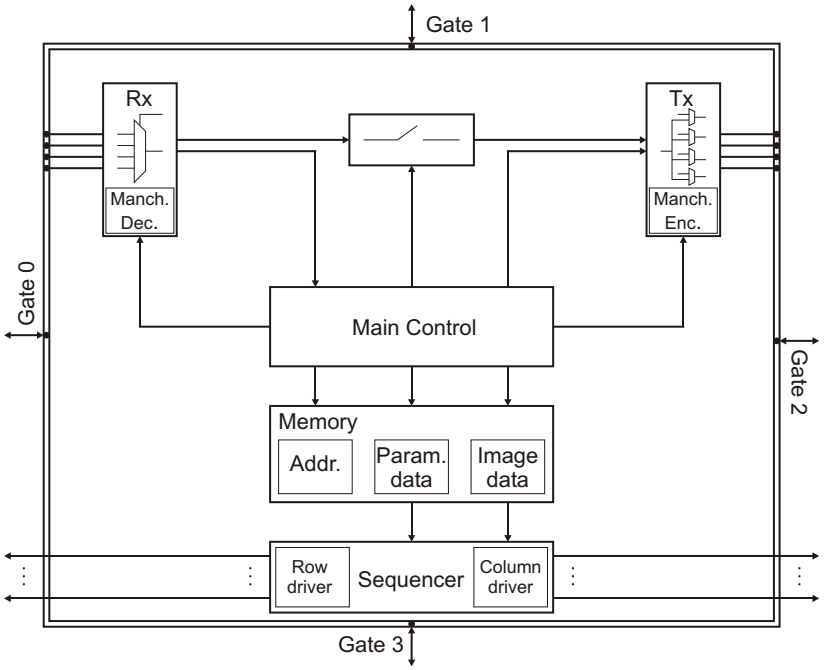


Figure 6.2 – The block diagram of the improved modular display driver.

6.3.1 General principles

When the display is turned on, all bypasses are inactive. The microcontroller will again send the first address (`0x00`) to the connected module. As with the previous driver, the module will calculate the addresses for the neighboring modules. The address byte used here is divided in two parts. The first nibble (4 bits) indicates the number of the column the module is in, the second nibble indicates the number of the row. Determining the addresses for neighboring modules is as simple as adding or subtracting 1, according to the gate the address is sent to (See Table 6.1). The addresses are cyclic, so `0x0` subtracted by 1 gives `0xF`. The gates are numbered from 0 (`00`) to 3 (`11`), starting at the left side and turning clockwise around the module.

gate	column	row
00	-1	$=$
01	$=$	-1
10	$+1$	$=$
11	$=$	$+1$

Table 6.1 – The addresses that need to be sent out , corresponding the gate numbers

From Figure 6.3 its clear that if there are several paths from the microcontroller to a module, this module will receive this address –it will of course always be the same address– multiple times. But the module may only send it once, and best not to a module that already has an address. For this reason, only one of the gates will be chosen as input. The gate where a module first receives an address, will be the input gate. If two addresses are received at the same time, the gate with the lowest gate number gets the highest priority. This makes also sure that every module is connected to the microcontroller through the shortest path. The other gates will be output gates, but will be closed when a start bit is detected on that gate (during initialization). This would mean that the module connected to that gate already receives data from another module, so there's no need to send data there. This is actually a very crude implementation of the Spanning Tree Protocol (See Chapter 4).

By only looking at one (dynamically chosen) input gate, and cutting away unnecessary connections, a tree structure is created. When the bypasses are closed after receiving and sending the addresses, there won't be any loops left to cause problems.

6.3.2 *Rx* and *Tx*

As said *Rx* and *Tx* are very similar to the one from the first modular display driver (See Section 5.3). I will just cite the differences here.

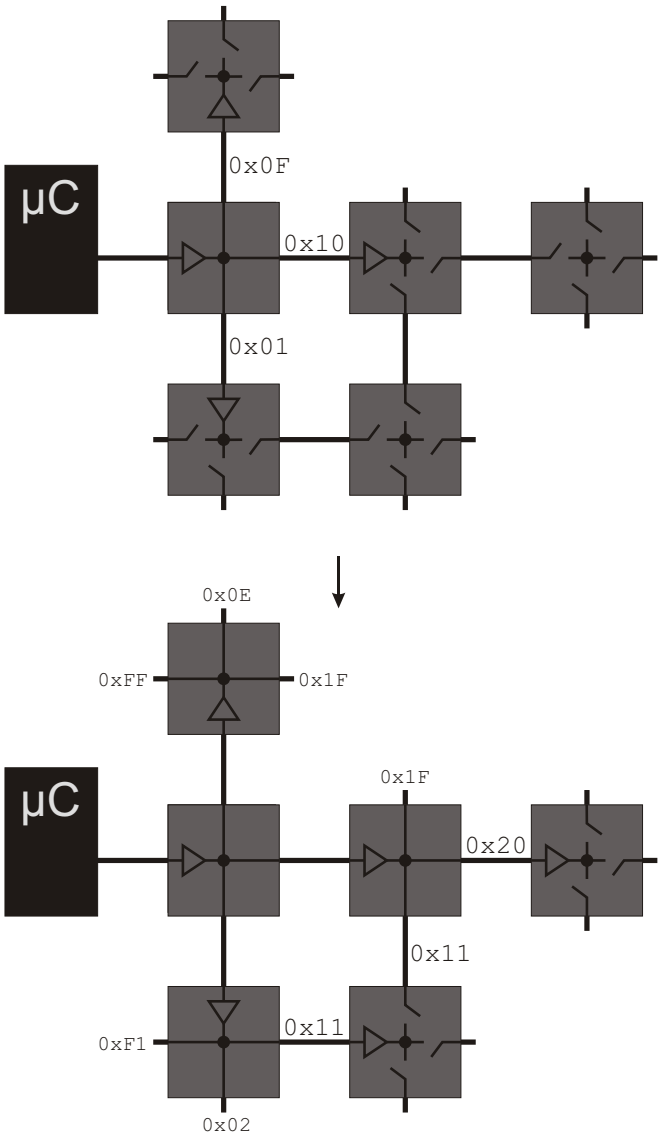


Figure 6.3 – The initialization process in the improved modular display driver. When a module receives an address from two gates (lower right module), it chooses one gate as input gate. No data will be forwarded to the other gate.

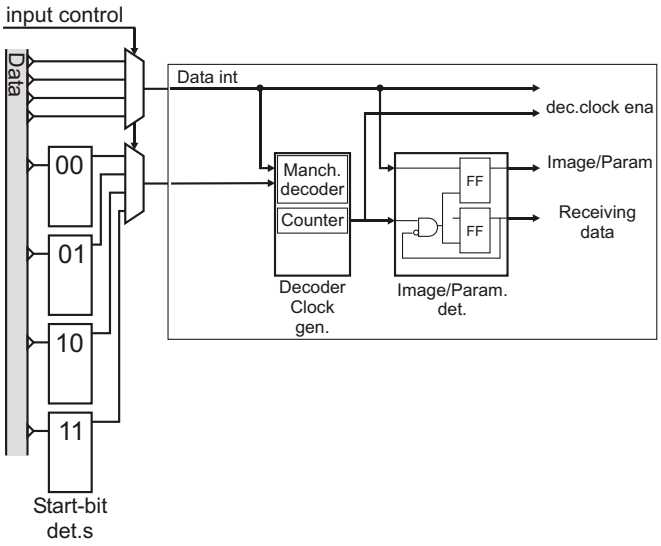


Figure 6.4 – Block diagram of *Rx*

Rx

Rx now has to monitor four gates instead of one. There are four start bit detectors, which present their outputs to *Main Control*. There will the decision be made which gate will function as the input gate. A multiplexer selects the chosen data signal and from then on the functionality of *Rx* is the same as in Chapter 5 (See Figure 6.4).

Tx

Analogous to above, *Tx* now has four shift registers. Addresses are calculated and sent simultaneously. Each shift register can be disabled when *Main Control* decides not to send data to a specific gate (*Out control*). After the initialization, bypasses are activated and the input data is forwarded to the necessary gates (See Figure 6.5).

6.3.3 *Main Control*

The state machine in *Main Control* has exactly the same states as in the first modular display driver (See Figure 5.9). There are only some small changes in functionality. When the display is turned on (state *S*0), *Main Control* waits for an incoming start bit. On an incoming start bit, a signal `in_control` will be adjusted

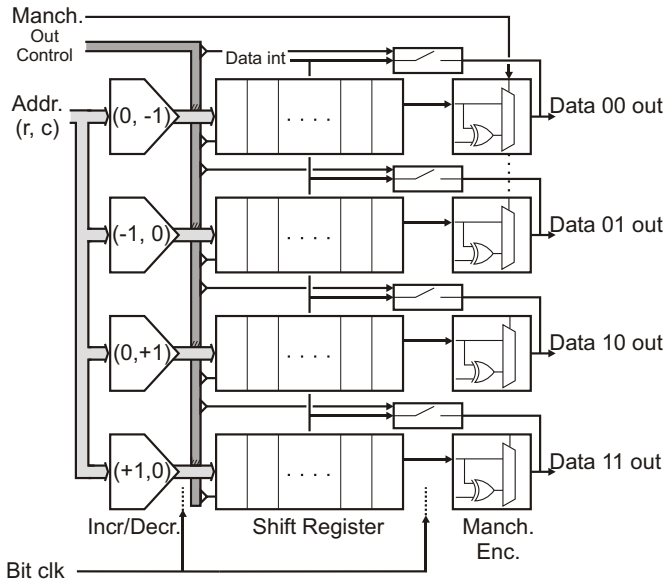


Figure 6.5 – Block diagram of *Tx*

according the gate number this start bit originates from. This signal controls the multiplexer from *Rx*. If two gates would receive a start bit at the same time, the lowest gate number gets the highest priority. The address itself is received in state *S*1. When, in this state, *Main Control* sees start bits on another gate, a signal `out_control` is adjusted, to disable the shift registers and bypasses that would send data to that gate. Note that in the ideal case, given the fact that a module will not send data to a module from which it received data and assuming that the time to calculate and send addresses is exactly the same for each module, a module will only receive addresses on maximum two gates and these addresses will arrive at exactly the same time.

6.4 A simple example

Using the same structure as previous chapter, let's take a look at a simple example. The considered display configuration is the one shown in Figure 6.1b. The waveforms in Figure 6.6 show the corresponding initialization process.

The microcontroller starts by sending out address `0x00` to *M2*. After it received the address *M2* sends the addresses `0x0F`, `0x10` and `0x01` to *M1*, *M3* and *M5* respectively. Keep in mind that addresses, just as everything, are sent with the LSB first. Since *M2* has row number `0x0`, *M1* in the row above receives number

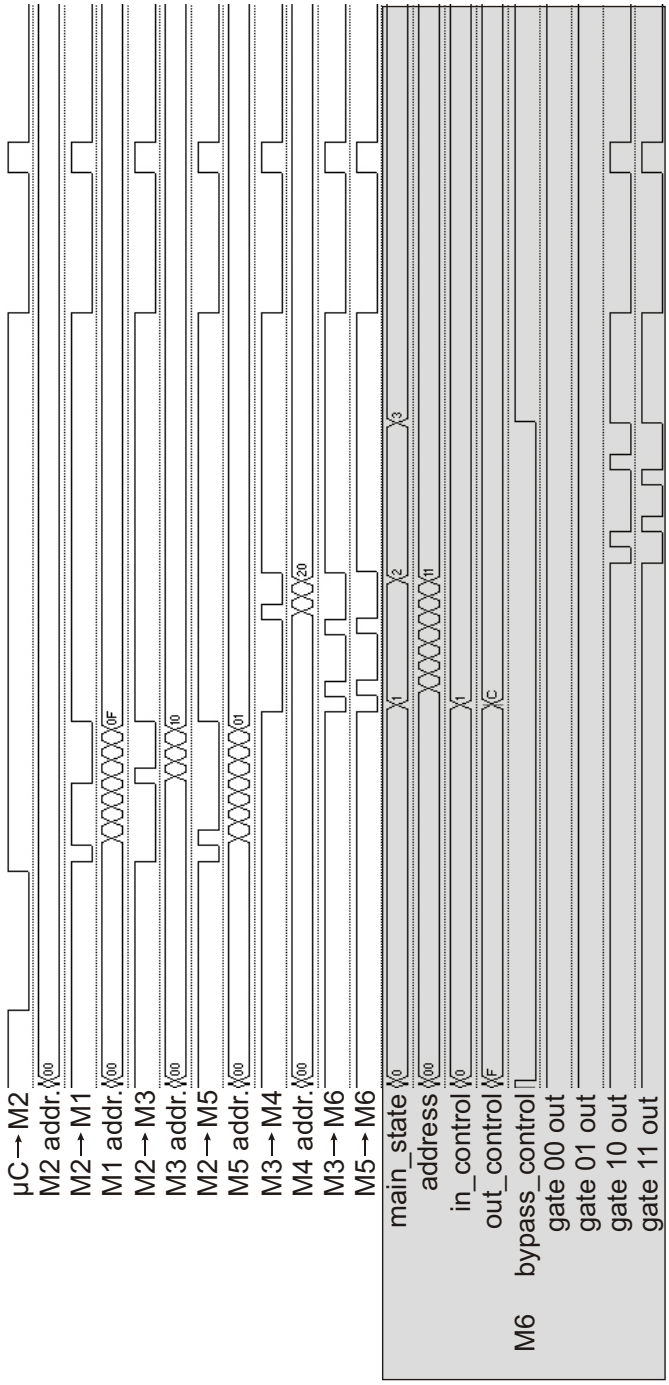


Figure 6.6 – Example of the initialization process for the improved modular display driver. The considered display is shown in Figure 6.1b.

`0xF`. *M3* and *M5* received their address at the same time, and will send the next address (`0x11`) at the same time to *M6*. Taking a closer look at *M6*, we see that at the moment the addresses start, the decision about which gate to use as input, is made. In this case, `in_control` is `0x01`, meaning gate `0x01` is chosen. I said earlier that gates with a smaller gate number have a higher priority, so normally gate `0x00` should have been chosen. It is not visible in the picture, but to simulate this setup I added some small variations to the clocks of the different modules. If you would zoom in, you would see that the start bit of *M3* arrives a little bit earlier. Early enough to be seen as *M6* as the first arriving address. The 4-bit vector `out_control` is immediately adjusted not to send any data to the input gate. When the address from *M5* arrives, this gate is also disabled for output, resulting in `out_control = 0xC = 1100`. The two lowest gates are disabled, the rest is enabled. As a consequence, the addresses that *M6* calculated will only be sent to those two highest gates. At the end, parameter data is coming in, immediately seen by every module, but not necessarily forwarded to all the gates.

6.5 Setting up the test environment

Since the main goal here is to be able to build up a display from independent modules, we can use the LED modules from Section 5.5.2. The first modular display driver only had one input and one output gate, but the LED modules were designed to provide four input/output gates (See Figure 5.19a). The VHDL implementation is printed in Appendix B. Some parts are exactly the same as in the implementation of the first modular display driver. These were not printed again.

The GUI also hasn't changed much (See Figure 6.7). Since the GUI has no way of knowing how the user built the display, the user has to tell it to the GUI. We chose to let the user enter the maximum dimensions of the display, rather than to let him enter the exact configuration. But when something is drawn on the display, the data will be sent for the entire display, also for modules that aren't actually there.

6.6 Some first results

We built some displays and looked at how they reacted at the commands from the GUI. Again, they all worked fine. Every module received an address and processed the data from the PC correctly. Since the *Sequencer* hasn't changed from previous driver, changing the refresh rate and number of used rows and columns went without a problem. Figure 6.8 shows one of the display configurations, with could be controlled with the GUI from Figure 6.7.

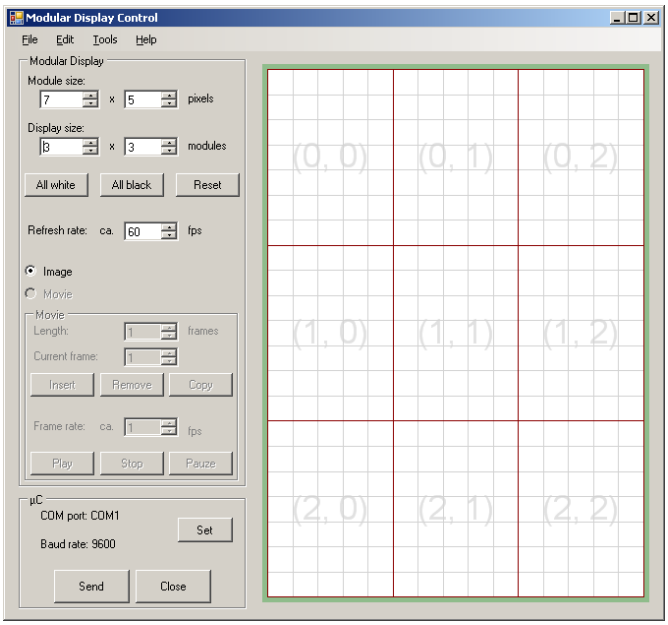


Figure 6.7 – GUI for the improved modular display driver

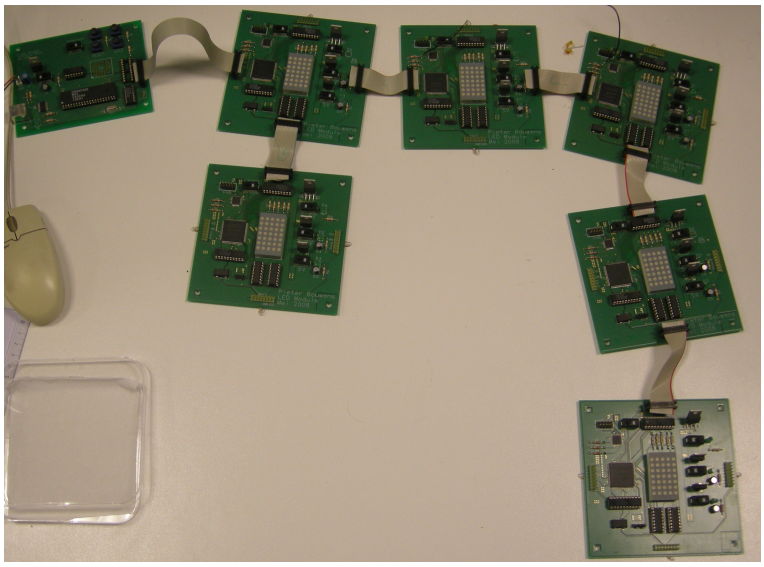


Figure 6.8 – Display configuration. The corresponding GUI is shown in Figure 6.7.

Since the addresses can be handed out in parallel, the total initialization time of a display will not be only dependent on the number of modules, but more on the way they are connected. More precisely, it is dependent on the longest Manhattan distance from the microcontroller [2]. The name alludes to the grid layout of the streets in Manhattan, where the shortest path between two locations, created by walking along the streets, is the sum of the (absolute) differences of their coordinates. Every module is connected to the microcontroller through the shortest path, so the initialization time will be dependent on the number of modules in the longest path. Since the information that is sent between modules is the same as with the first modular display driver, the total time is $9,6\mu$s per module in that path.

6.7 Is there still room for improvement?

The driver explained here, allows us to create our own displays using simple, independent modules. The modules themselves will distribute the addresses and they will make sure that there aren't any problems with data loops. It is a system that works and that could be used in a number of applications.

But, of course, there are some shortcomings. The main shortcoming being that the user has to enter the exact configuration of the display, manually in the GUI. To make it easier (which we did) the user could also just enter the maximum dimensions of the display, but then there is a lot of unnecessary data transmission. Ideal would be that the GUI would somehow be able to detect the exact configuration of the display. When the user creates a display, the configuration shows up in the GUI. No manual entering, no unnecessary data transmission.

Another limitation applies. Since the communication is unidirectional, the microcontroller has no way of knowing if something went wrong during the initialization. Above that, it would be more efficient if the microcontroller knows exactly *when* the initialization is finished, instead of just waiting for the maximal initialization time.

A third shortcoming is that after the initialization no modules can be added or removed without interfering with the display operation. A true free-form modular display driver would allow the user to add and remove modules, while the display keeps working. To combine this with the issue raised above, the GUI should also have an real-time view of the configuration of the display. When a module is added or removed, it should show up (or disappear) in the GUI.

References

[1] T. Ohkami, "Modular display: an approach to intelligent display systems," in *SID96 Symposium Digest of Technical Papers*, vol. 27, 1996, pp. 225–228.

[2] Wikipedia. Manhattan distance. [Online]. Available: http://en.wikipedia.org/wiki/Manhattan_distance

[3] P. J. Ashenden, *The VHDL Cookbook*. University of Adelaide, 1990.

I'd rather be a "could-be",
if I cannot be an "are";
because a "could-be" is a "maybe"
who is reaching for a star.
I'd rather be a "has-been"
than a "might-have-been", by far;
for a "might-have-been" has never "been",
but a "has" was once an "are".

Milton Berle July (1908-2002)

7 A free-form modular display driver

7.1 Introduction

In this chapter, a free-form modular display driver will be described. This driver can, as his name indicates, be used to create free-form displays. Free-form displays are built from independent modules (well, we are creating modular display drivers, aren't we?) that can be connected to create a desired shape. This sounds an awful lot like what was said about the improved modular display driver, however, this driver will have a lot more functionality. A lot of the 'freedom' of this free-form modular display driver comes from the fact that the actual configuration of the display is detected and shown in a GUI. More so, this driver makes it possible to add and remove modules while the display is running. The requirements these properties impose on the driver are explained in Section 7.2. The implementation (Section 7.3) still resembles the previous two drivers, but will be somewhat more complex due to the added functionality. As always, we'll take a look at a (maybe slightly less) simple example in Section 7.4 and check out some test results in Section 7.6.

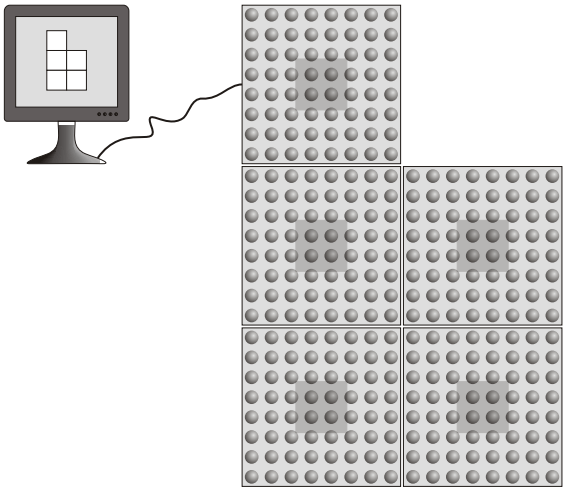


Figure 7.1 – The possible display configurations are the same as with the improved modular display driver, but with the free-form modular display driver, the GUI shows a real-time view of the configuration.

7.2 Requirements

7.2.1 The display

The display setup itself doesn't differ much from the setup for the improved modular display driver (Section 6.2). Every module is completely independent and can be connected by four sides to create a desired display. One of the modules is connected to the microcontroller, which provides the communication with the PC. Communication with the PC will now play a bigger role in the operation of the display, since the display will be in constant contact with the PC to 'tell' it how its modules are connected. This not only needs to be done during initialization, but during the entire time the display is turned on, to provide a real-time view of the configuration to the user.

7.2.2 The driver

The basic requirements for the improved modular display driver still apply here (Section 6.2), but with some extras. The address of a module can still be determined by the surrounding modules. We'll still start out with a mesh network, from which we need to create a tree structure. Since we want to be able to detect the exact configuration of the display, it's important that every module knows which of its output gates are used by other modules. Every module will have

to keep track of one parent node and $0-3$ child nodes. This resembles the tree structure from the improved modular display driver (except the fact that the improved modular display driver didn't care if there was a module connected to its gates), but it needs one major difference. This tree needs to be fully dynamic. As said in the introduction, it should be possible to add and remove modules while the display is running, after the initial tree is already created. When a module is added, it should be added in the tree. When a module is removed, it should be removed from the tree. But there is a problem with the latter. Since the tree makes sure that there will be only one path from the microcontroller to any module [1], the modules in a branch will stop receiving data when one of the parent nodes in that branch is removed. In some occasions, however, those modules might have another path to the microcontroller. In other words, when a module is removed, the tree should be (partially) rebuilt.

Last but not least, the modules need to be able to communicate with the microcontroller in order to give it the necessary information to determine the configuration of the display. Again, the dynamic addressing of the USB protocol seems interesting, but for the same reasons as in previous chapter, it's not applicable in our situation. This time there are even more reasons. Even if we were satisfied with the limited topology that can be created, this topology is completely fixed. It is not the dynamic tree that we need. The SPI protocol (Section 4.3) would allow the microcontroller to communicate directly with a module, without the need for an address. But, as explained, the protocol uses $3 + n$ wires (with n the number of modules). For larger displays, this becomes impossible. Again, it's probably more efficient to create our own, dedicated protocol.

7.3 Implementation

The block diagram in Figure 7.2 still has most of the features from the previous two drivers. The *Memory* and *Sequencer* are exactly the same, and everything is controlled by *Main Control* with a state machine. The simple bypass structure has become a little bit more complex, though, and is replaced with *Out Control* since now we need more than just data being forwarded from parent node to child node. The modules will also need to be able to talk to each other or sometimes even the microcontroller itself. This will become clear in following sections. Also, because of the fact that the modules can independently talk to each other, every gate needs its own *Rx*. We have a lot of different sources that could drive the outputs, so multiplexers are used to choose the appropriate source.

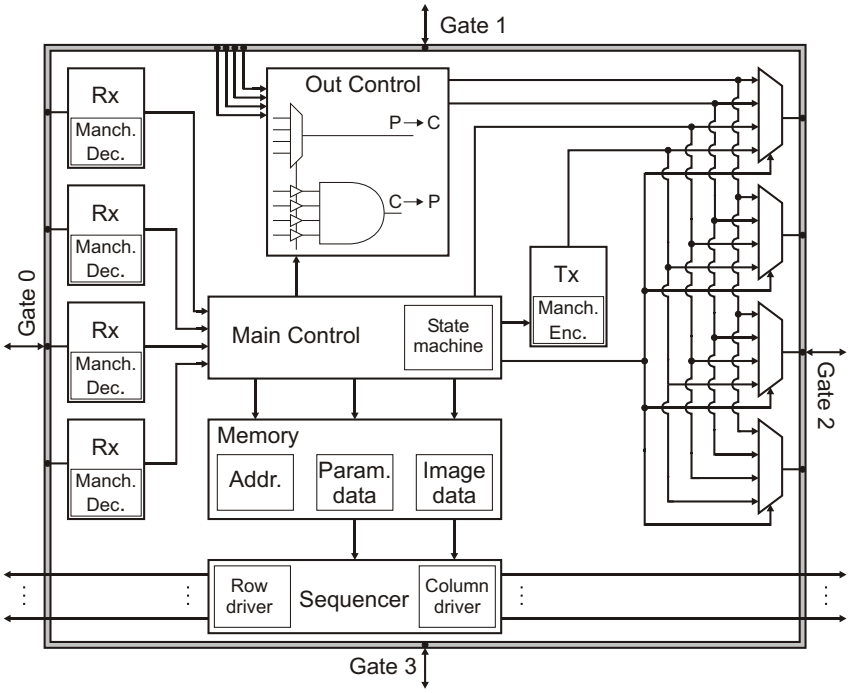


Figure 7.2 – The block diagram of the free-form modular display driver.

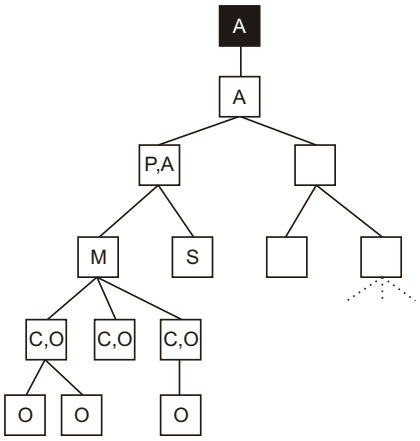


Figure 7.3 – The genealogical tree created using the free-form modular display driver. A module (M) has a parent node (P), child nodes (C), sibling nodes (S), ancestor nodes (A) and offspring nodes (O).

7.3.1 A little terminology

To simplify the explanation of some situations that will occur, I've defined some new terms. As said, the goal is to create a tree structure, where a module (*M*) has a parent node (*P*) and several child nodes (*C*). This already shows some hints towards a genealogical tree, so why not expand that idea? An example is shown in Figure 7.3. Every module might also have one or two sibling nodes (*S*). These are the nodes with the same parent node as the module. The chain of parent nodes of a module (parent node, parent node of the parent node, etc.) are called the ancestor nodes (*A*) of that module. Ancestor nodes provide the path from the microcontroller to the module. The closer an ancestor is to the microcontroller, the older it is. A module can also have a lot of offspring nodes (*O*). These are the nodes that see that module as an ancestor node. When a module breaks down, all offspring nodes are affected.

7.3.2 Communication protocol

For this driver, the communication protocol will be a little bit different. Data sequences will still be preceded with a start bit (`0`) and ended with a stop bit (`1`) (Well, most of the time, but this will become clear later). The biggest difference is that every data sequence will also have a command sequence. A lot of different information can be sent between modules, so the modules (and the microcontroller) need to know what to do with it. Each command is three bits long.

The exact meaning of the command is fixed most of the time, but sometimes the meaning changes according to the state of *Main Control*. Some of the commands are followed by data, others are not. The command sequences and their meaning can be found in Table 7.1. The commands, as everything, are sent with LSB first.

Command		Info	
000	adr_ack	Send address	+ address
001	adr_req	Request address	
010	sht_req	Request shout permit	
011	sht_ack	Shout permit	
100	sht	Shout address	+ address
101	sht_end	Shout last address	+ address
110	param	Send parameter data	+ data
111	data	Send image data	+ data

Table 7.1 – Command sequences used by the free-form modular display driver.

7.3.3 General principles

When we want to create a display that needs to keep working when modules are added or removed, every module needs to be aware of its surrounding neighbors. This is where the pull-down resistors on the data line come in. As said in Section 5.3, we didn't really need them at that point, but now, things have changed. When a module sees a high input on one of it's gates, it can be sure that there is a module on the other side, driving the line. We can expand this a little more. A module doesn't *need* to drive the line. When a module only drives its outputs high when it has a path available to the microcontroller, other modules will know that a low line means that at that gate there is either no module, or a module that is not (yet) connected to the microcontroller. In either case, not a module that could function as a parent node. When a module sees a high line, it knows connecting to that module will provide it with a path to the microcontroller. How this works will be clear after the next paragraphs.

Detection of the configuration

One of the most significant changes of this driver is that it is supposed to detect the exact configuration of modules in the display. Since the address of a module, which will be calculated in the same way as the previous driver (See Section 6.3), is in fact the row and column number of that module in the display, this address can be used as coordinates for that module to build up the display at the PC side. When the PC knows all the used addresses in the display, it can show us the exact configuration of the display. The only issue is how to get all addresses to

the PC (or microcontroller). This has to happen in a controlled way, to avoid data collisions and to avoid flooding the microcontroller with addresses.

Initialization

When the display is turned on, every module will look out for a high input. As said above, as long as a gate has a low input, there is (for now) no path to the microcontroller available through that gate. When a module sees a high input on a gate, it knows it can get an address through that gate. The module connected to that gate will become its parent node. The module immediately requests an address from its parent node by sending an `adr_req` and waits for an address (`adr_ack`).

When the address is received, the module will drive all its outputs high. This way the neighboring modules are notified that a path is available through this module. They should start asking for addresses immediately, so the module waits for incoming `adr_req`s. An incoming `adr_req` is answered with an `adr_ack` followed with the correct address. The address is calculated according to the gate the request originates from (See Table 6.1). It's obvious that at every gate where a `adr_req` comes in, a module is connected that will be a child node of this module. The `adr_req`s are used to determine where the child nodes are, which is stored in a vector (`children`). `adr_req`s are handled one at a time, with the lowest gate number having the highest priority. The other modules will have to wait their turn. Using this technique, every module will be connected to the microcontroller through the fastest (i.e. shortest) path. In the end, a minimum spanning tree with a fixed root (i.e. module directly connected to the microcontroller) will be created[2]. In se this is a simple implementation of the Spanning Tree Protocol discussed in Section 4.3. When all `adr_req`s are handled (or when no `adr_req`s were received) the module will try to send its address to the microcontroller. This action will be called *shouting*.

A shouting routine works as follows. Since there isn't a direct connection with the microcontroller yet (which is a good thing), the module will try to send its address to its parent node. The parent node receives this address and will try to send it to its own parent node, and so on. The address will propagate through the path of ancestor nodes until it reaches the microcontroller. To send an address to the parent node, the module asks for a shout permit (`sht_req`). When the parent node is ready (meaning it has already sent its own address or any other address stored in the shout buffer), it sends a `sht_ack` and waits for the incoming address. The module will then shout the address using either `sht` or `sht_end`. The difference between those two will be explained later (See Figure 7.4 for an illustration). The incoming address will be stored in the parent node's shout buffer.

After sending its own address, the module will look for incoming `sht_req`s from

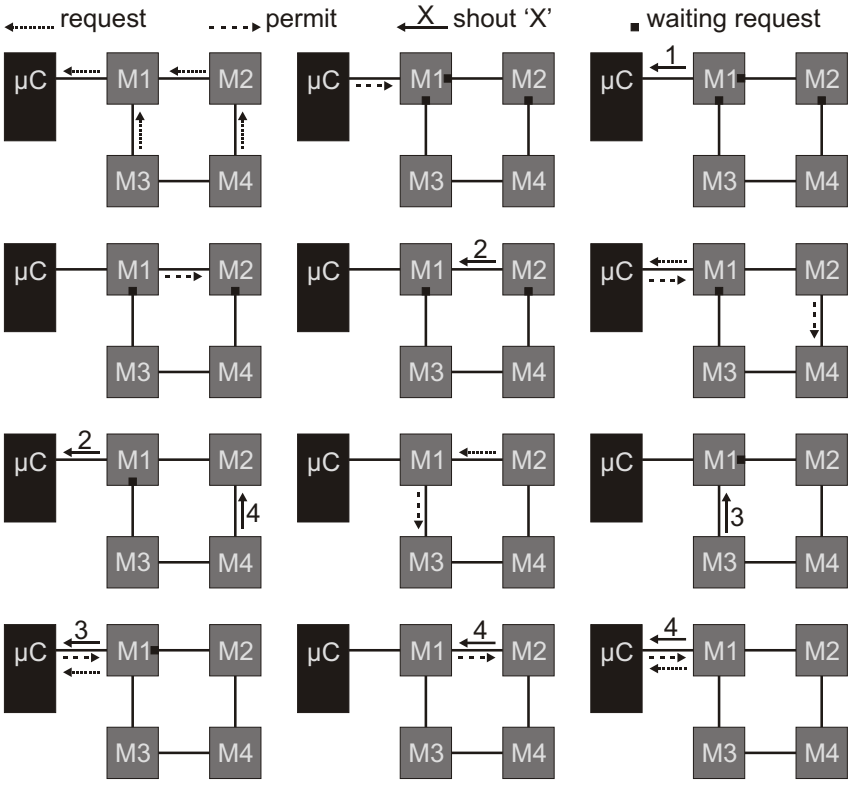


Figure 7.4 – The shout routine in the free-form modular display driver

its child nodes. The first addresses that will be shouted will be the addresses from the child nodes itself, but in the end all addresses from all offspring nodes will have to pass through this module. This poses a small problem. How does a module know how many offspring nodes there are, how many addresses to expect, when to stop waiting for incoming `sht_req`s? When a module wants to shout its own address, it checks if it has child nodes or not. If it doesn't, this means that the only address it needs to shout will be its own address. It will shout this address using `sht_end`. Any module that does have child nodes will only shout an address using `sht_end` when every child node has sent a `sht_end`. In other words, shouting using `sht_end` simply means that the attached address is the last address that will be sent from this module. All addresses from all the offspring nodes have been sent. When the microcontroller receives a `sht_end`, every address has reached the microcontroller and the initialization is finished.

After the initialization, the 'bypasses' are activated. Only now, the bypasses work in two ways. Data from the parent node is forwarded to the child nodes (and only to the child nodes). But also, data coming from the child nodes is combined (see paragraph on *Out Control*, Section 7.3.6) and sent to the parent node (and thus to the microcontroller). This last bit is a bit tricky, since it might cause problems when two or more modules are trying to send something to the microcontroller. Modules trying to send something to the microcontroller will not happen very often. This is explained in next paragraph.

Adding modules

One of the reasons why `adr_req`/`adr_ack` is used instead of just sending the addresses to all gates, is that not every module is necessarily present at that time. We want to create a driver which makes it possible to add modules to the display after initialization. Let's see what happens when a new module is attached to an already present module. Since this last module has a path available to the microcontroller, its outputs are driven high. The new module will immediately see a high input and will send an `adr_req` to its new parent node. Since, at this point, the new module isn't a registered child of the parent node yet, the `adr_req` will only be seen by the parent node and not forwarded to the microcontroller. The new module receives an address (`adr_ack`) and will try to shout the address to the parent node. We still don't want to activate the bypass towards this new module, though. For several reasons, first, the new module doesn't realize the initialization for the rest of the display is already over and that it could cause problems if it would simply perform a `sht_req` (which would be directly sent to the microcontroller). Second, if the microcontroller is sending data and the bypass is activated mid-sequence, the new module would appear to receive faulty data. So, for now, the bypass stays closed.

As I was saying, the new module will try to shout the address to the parent node

by sending a `sht_req`. The parent node lets it shout (`sht_ack`) and the new module will shout its address. If only one module is added, it uses a `sht_end` of course. Now comes the tricky part. The parent module needs to send the address to its own parent node, but since all other modules have their bypasses activated, it will be directly sent to the microcontroller. The parent node checks if the line is free (no incoming parameter or image data) and if it is free, sends a `sht_req` to the microcontroller. The data line will be free in between two frames. The microcontroller sends a frame by sending image data to every module, then waits before sending the next frame. The time to check if the data line is free must simply be longer than the time between the data sequences for two modules, but shorter than the time between the end of one frame and the start of the next frame. This method poses one restriction on the adding of modules, though. When two new modules would be added on different places at the same time, the two new parent nodes would try to shout at the same time, which would cause collisions. This could be avoided by using collision detection and multiple-access protocols explained in Chapter 4 [3], but this was not implemented. We simply prohibit that two modules are added at the same time on a different place. That works too. Multiple modules *can* be added at the same time, but it should be one group of modules, all connected to only one module. Another thing that we need to be careful about is that the `sht_req` and following `sht(_end)` will be seen by all ancestor nodes, and the returning `sht_ack` by *all* modules. All modules will have to follow whats going on, without intervening, to avoid confusion.

Removing modules

A couple of problems can arise when removing a module. This module was probably a child node of another module. Since there is no-one left to drive the line, the line will be pulled down. But since the corresponding gate was considered a child gate, this low signal will be forwarded to the ancestor nodes and the microcontroller. So when a module sees a low signal on one of its child gates, it could be that its own child node is gone, or one of its other offspring nodes is gone. The precise method to check whether an offspring node is gone (own child or not) will be explained in Section 7.3.4. When *Rx* notifies that an offspring node is missing the bypass from that node is deactivated. If the signal remains low, then it is one of the own child nodes that is gone and the `children` vector is adjusted. If the signal returns to a high voltage then it was one of the other offspring nodes and no action is required (other than reactivating the bypass, of course).

As said in Section 7.2, another problem can make removing modules a little bit more difficult. When a module is removed, its entire offspring would stop receiving data notwithstanding there might be an other path through which they could receive data. The tree has to be rebuilt so solve that problem. When *Rx* notifies that the parent node is gone, *Main Control* performs a full reset. Since data from the parent node is forwarded to all offspring nodes, *Rx* actually noti-

fies when an ancestor is gone. Or, in other words, when a module disappears, the entire offspring performs a full reset. After a full reset, the memory is still intact (address and data are remembered), but *Main Control* will have returned to the begin state. Bypasses are deactivated, outputs are pulled low, `children` vector is emptied, `in_control` is reset. The entire offspring will start looking for a high input, just as during the initialization. When one of those modules is connected to another module that wasn't part of the affected offspring, it will see a high input on that gate. The new parent node is chosen. The following initialization will be a little bit different, though, for two reasons. First, since the module still has an address, it does not have to receive one. But, more important, second, it's very possible that multiple modules of the former affected offspring find a new parent node at about the same time. If all those new parent nodes would start to shout, there would be a lot of data collisions, as explained above. And anyway, a shouting routine isn't necessary because the microcontroller hasn't forgotten about the modules that were in the offspring. There is no need to shout the addresses.

So, when a module finds a new parent, it will not send an `adr_req` but an adopt request (`adpt_req`). The careful reader will have noted that the `adpt_req` command isn't present in Table 7.1 (It isn't). This is because the `data` command will be used instead. The `data` command could only come from a parent node, so seeing `data` on one of the gates that isn't the parent node, means there is a module there that wants to be adopted. The parent responds with an `adpt_ack` (= `adr_ack` without address) to complete the adoption. The module is now reconnected to the tree and will drive its outputs high. Other modules from the former offspring can now perform an `adpt_req` to this module. Note that this way, a module that was once a parent node of another module, can now become its child node. After this procedure, the affected part of the tree is rebuilt and every module that can connect to the microcontroller, will be connected.

Keeping the GUI updated

When the initialization is finished, all used addresses have been sent to the microcontroller. With these addresses, the GUI can show the current configuration of the display. When modules are added or removed after the initialization, the GUI needs to be updated. When modules are added, the new addresses are also sent to the microcontroller, so the GUI can adjust the configuration. When removing modules, however, it becomes a little bit more difficult. A module cannot 'announce' that it has disappeared. And even if that could (e.g. parent node telling that one of its child nodes is gone), there is no way of knowing if the entire offspring of the removed module was able to reconnect to the microcontroller.

The only option is to periodically check which addresses are still used, and which have disappeared. This is done through a poll. Every 1-2 seconds, the microcontroller performs a poll request. This will be the `sht_req` command. It's still an

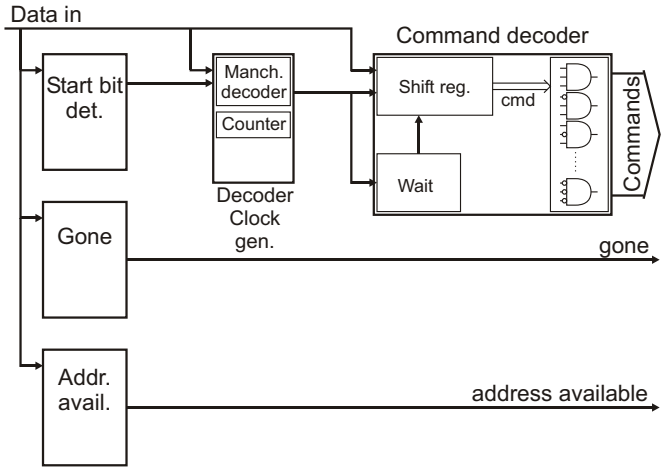


Figure 7.5 – Block diagram of *Rx*

unambiguous command, since it cannot be sent by a parent node during initialization. When the modules see `sht_req` from their parent nodes, they deactivate all bypasses and start the polling routine, which is actually just a complete shout routine from during initialization. Every module will try to shout their address to the microcontroller, but since the bypasses are inactive, it will only reach to the parent node, which will shout it to its own parent. When a module hasn't received any `sht_req`s by the time it received a `sht_ack`, it will assume that it has no children (in the case the missing child hasn't been noticed yet) and will shout using `sht_end`, otherwise using `sht`. As during the initialization, the polling routine is over when the microcontroller gets a `sht_end`. The microcontroller again knows all currently used addresses and the GUI can be updated.

7.3.4 *Rx*

Since the communication protocol has changed and become a bit more complicated, *Rx* has some more functionality. Figure 7.5 shows the block diagram. The *Start bit detector* and *Decoder clock generator* (with the Manchester decoder) are the same as in previous drivers. The *Command decoder* replaces the *Image/Parameter detector*. The extra functionality is provided by the *Address available* block, which looks for high inputs while filtering out glitches. The *Gone* block detects when the attached module might have disappeared.

Command decoder

This block actually consists of three smaller blocks. A *wait* block, a *shift register* and the actual *decoder*. When a start bit is detected, the *wait* block (which is a counter) is activated and counts the incoming command bits. The command bits are shifted into the register. After three bits, when the command is finished, the shift register is disabled and the decoder enabled. The decoder has nine outputs, one for each command and one to indicate that a command is received (cfr. *Receiving data* from Section 5.3). As long as *Rx* is not reset, the signals remain available. This is necessary because a module can for example receive multiple `sht_req`s, but can only process one at a time. The others are kept available in their respective *Rx*s.

Address available

The *Address available* block will look for a high input. A high input indicates that an address (and path) is available through that gate, and will be presented to the *Main Control*. Possible glitches on the data line are filtered out by demanding that the data line remains high for a specific time. But there is another issue that we need to be aware of. Let's assume two adjacent modules that aren't siblings. Call them *M*1 and *M*2. When *M*1 looses its parent node it will try to find a new parent node, which will be *M*2. At the moment *M*1 looses its parent, it goes into full reset, pulling its outputs low. But this is seen by *M*2 as the start of a command (modules need to listen to unused gates for commands, because there might be a new module attached there). If *M*1 sees the high output from *M*2 and if it would almost immediately send an `adpt_req`, this command would be misread by *M*2 because, according to him, the command started when *M*1 pulled its outputs low. Therefor, *Address available* must at least wait for 5 bits (1 start bit, 3 command bits, 1 bit to let the *Rx* be reset) before presenting the signal to *Main Control*.

Gone

Gone will present a high signal to *Main Control* when it suspects that the attached module has gone missing. As said in Section 7.3.3, we can not be sure at this time whether a parent node or one of the ancestor nodes is gone (not that it matters, because the same action is required), or whether a child node or one of the offspring nodes is gone (this time it does matter). But the *Gone* block doesn't need to care, it just tells when he suspects a module is missing and *Main Control* will do the rest. Detecting when a module is missing is dependent on whether Manchester code is used or not. If you see a signal that remains low for longer than one bit period, there is definitely something wrong when Manchester code is used. When no Manchester code is used, we can only know if a module is missing when no data is being received. In both cases of course, the *Gone* signal is only relevant if there

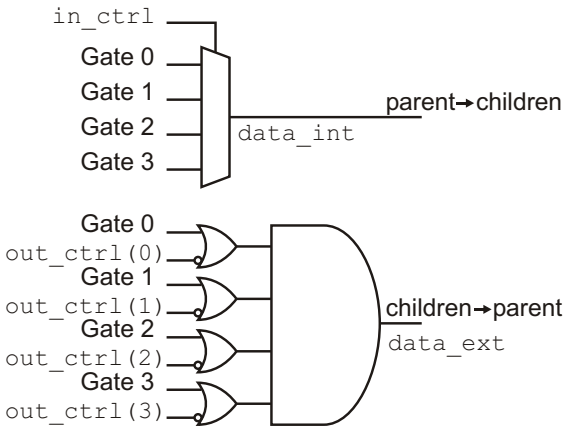


Figure 7.6 – *Out Control* chooses the parent signal to be sent to child nodes and combines the child signals to be sent to the parent node.

should be a module in the first place (i.e. high *Address available* signal).

7.3.5 *Tx*

Tx, on the other hand is a little bit simpler than before. It is now only a simple shift register (with the possibility of Manchester coding its output). The contents, which can be a calculated address for a child node, are loaded by *Main Control*. No calculation of any kind is done in *Tx*. The shift register here is 12 bits long: 1 start bit, 3 command bits and 8 bits for sending an address if needed. There is also no more bypass present in *Tx*. *Tx* is now only one of several sources that can be used for output.

7.3.6 *Out Control*

As said before, *Out control* will take the function as bypass, connecting the parent node and child nodes. Figure 7.6 shows how this is done. From parent node to child node a simple multiplexer suffices. The data signal from the parent node is called `data_int`. The child signals need to be combined to one signal, before it can be sent to the parent. When a child node is present (`out_control=1`) its default output is `1`. So should the output towards the parent node be. Only when a child node sends data, should the output towards the parent node be changed. A child node that is not present (`out_control=0`) should not interfere with the process. Figure 7.6 shows the logic that accomplished this. The combined signal from the child nodes is called `data_ext`. Which gates can participate and which can't are controlled by `out_control`. This will be the same as the `children`

vector from before for the most of the time. But not always. When it is suspected that a child node drops out, the bypass needs to be temporarily disabled (`out_control`) but the child node must not be removed from the `children` vector. Or, when a new child node is connected after the initialization, the bypass to this child cannot be immediately activated, notwithstanding the parent node already listed it as its child.

7.3.7 Output multiplexers

These are simple 4-to-1 multiplexers. Each gate can output one of the following signals. The `default` output. This is the value that will be sent to the gates not connected to either child or parent node (`0` when no address is available, `1` when address is available). The command or *Tx* output, for when a command needs to be sent. The `data_int` signal for the child nodes and the `data_ext` signal for the parent node.

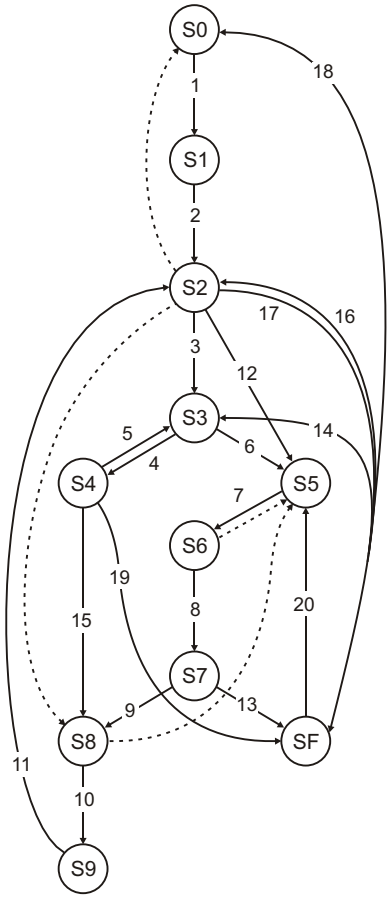
7.3.8 *Main Control*

Because of the added functionality, *Main Control* will be a lot more complicated than in previous drivers. It can still be represented as a state machine, which is shown in Figure 7.7. There are 11 states, they are enumerated in Table 7.2. There is also a time-out counter present in *Main Control*, if it takes too long for a command to be received (timer is set to 13ms) action needs to be taken. The dotted lines in Figure 7.7 show what happens if there is a timeout or if a wrong command is received.

State	Info
*S*0	Begin state
*S*1	Send address request or adopt request
*S*2	Receive address or `adpt_ack`
*S*3	Receive address requests or adopt requests
*S*4	Send address or `adpt_ack`
*S*5	Send shout request
*S*6	Receive shout permit
*S*7	Send shout
*S*8	Receive shout request
*S*9	Send shout permit
SF	Normal operation

Table 7.2 – The states from the state register in the free-form modular display driver explained

When the display is turned on, or after a full reset, we start out in state *S*0. When



#	Info	#	Info
1	Address available	11	Shout permit sent
2	Address request sent	12	Own address shouted
3	Own address not yet shouted	13	Shout using `sht_end`
4	Address request received	14	Address request received
5	Address sent	15	New child after initialization
6	Address requests finished	16	Pass address (return 17)
7	Shout request sent	17	Adopt permit received
8	Shout permit received	18	Parent node gone
9	Shouted using `sht`	19	Adopt permit sent
10	Shout request received	20	Start poll

Figure 7.7 – The state machine of the free-form modular display driver

a high input is detected, we go to state *S*1 to send an address or adopt request (`adr_req` or `adpt_req`), depending on whether we already have an address or not (See Section 7.3.3, 'Removing modules'). In state *S*2, we wait for an address (`adr_ack`) or an adopt acknowledgment (`adpt_ack`). If the latter is received, we can immediately go to the final state *SF*. *S*2 is also used to receive addresses from child nodes when these are shouting their addresses. If something went wrong or a timeout occurred, we go back to state *S*0, to ask for a new address, or to state *S*8 if we are in the shout routine, to process the other shout requests (see below). So, from *S*2 we can go in several directions, but I'll try to follow it chronologically. After receiving an `adr_ack` from the parent node in *S*2, we first need to provide the possible child nodes with an address. This happens in *S*3, where we wait for incoming `adr_req`s.

If an `adr_req`s is received, state *S*4 is used to calculate and send the corresponding address to the needy child, after which we go back to state *S*3 to process the other `adr_req`s. States *S*3 and *S*4 are also used to process incoming adopt requests in the same way. When a `adr_req` or `adpt_req` enters in state *S*3, the `children` vector is adjusted. After sending an `adpt_ack` as response to an `adpt_req`, state *S*4 is followed by the final state *SF*. When all address requests are handled or when no address requests were received (we'll wait for 20 clock cycles for an address request to come in), we are ready to shout our own address in state *S*5. After the initialization process, we could arrive in state *S*4 because of a new module that has been added. In that case, we don't need to shout our own address and can go immediately to state *S*8 to receive shout requests.

State *S*5 is used to send a shout request. During the initialization, a shout request can simply be sent, but when initialization is over and we find ourselves in this state because of a new module that has been added, we have to make sure the data line is free. This is explained is Section 7.3.3, 'Adding modules'. If the request is sent, we can go to state *S*6 to wait for a permit. If this permit doesn't come or is received incorrectly, we go back to *S*5 to request it again. The shouting itself happens in state *S*7. Here we will make the decision whether to shout with `sht` or `sht_end`. A vector (`adr_end`) keeps track of the gates from which a `sht_end` has arrived. If this vector equals the `children` vector, a `sht_end` is used. So, if there are no children, `sht_end` will be used when shouting our own address. After shouting with `sht_end`, we can go to the final state *SF*. In the other case, we go to state *S*8 to process the shout requests from our child nodes.

Since we can only arrive in state *S*8 when there are still child nodes that didn't shout using `sht_end`, we should definitely receive at least one request. If this doesn't happen when the timeout timer runs out, we will have to assume that an offspring node malfunctions. We have to send the last received address again, but now using `sht_end`. We adjust the `adr_end` vector and go back to state *S*5 to send a shout request. If every module follows this procedure, it will only be the module that has a child node malfunctioning that will have to adjust its

adr_end, which is what we want. The other modules (the ancestor nodes that would also be waiting for a sht_req) would receive the sht_req just in time. (If the module with the child node malfunctioning doesn't have sibling nodes, the sht_req would probably come just too late. The requests would always come in just as the timer runs out, which makes it dependent on the exact clock frequency. But when that module does have a sibling node, the ancestor nodes would still be busy processing the info on the sibling node when the timer of the affected module already started running.) When everything is OK, though, and a shout request is received in state *S*8, a shout permit is sent in state *S*9. After this, we go back to state *S*2 to receive the shouted address.

We are now in the shout routine where *S*2 is used to receive the shouted addresses. When an address is shouted using sht_end, the adr_end vector is adjusted. If no address is received or the wrong command is received from the expected child, we go back to state *S*8 to wait for the child to request to shout again. Since state *S*5 (send shout request) is the fall-back state for when something goes wrong while trying to shout, this should happen. If the address is correctly received, we go to *S*5 to perform a shout request ourselves. During the shout routine, we stay in the same 6 states: *S*5 (send shout request), *S*6 (receive shout permit), *S*7 (shout), *S*8 (receive shout request), *S*9 (send shout permit), *S*2 (receive address). The loop will be broken when every child has shouted using sht_end (adr_end = children) at which point we'll move from *S*7 to the final state *SF* (after shouting using sht_end ourselves). The initialization is over. Out Control is activated and the *Sequencer* is started.

A couple of things can happen in this final state. When the parent (or ancestor) node is gone, everything is reset and we go back to the begin state, *S*0. When an offspring node is gone, we can stay in the same state, but we do need to check if it is our own child node that is removed or not. This is explained in Section 7.3.3, 'Removing modules'. Another important command that can be received, here, is the polling request from the microcontroller. When this happens, *Out Control* is deactivated, the adr_end and children vector are reset and we go to state *S*5 to send a shout request, to shout our own address. From there, we are back in the shout loop (*S*5, *S*6, *S*7, *S*8, *S*9, *S*2), the children vector will now be adjusted according the received shout requests in state *S*8. After the polling (shouting) routine, we will be back in *SF*. When a new module is added, or when a module needs a new parent, we receive an address request or an adopt request. In this case, we go to state *S*3 and handle it from there (*S*3, *S*4, *S*3, *S*5, *S*6, *S*7, *SF* for a new module, *S*3, *S*4, *SF* for an adopted module). A final thing that needs to be done in this state is keeping track of other transmissions. If, for example, there is a new offspring node which needs its address shouted to the microcontroller, there will be some data passing through not meant for us. More precise, there will be a shout request coming from one of the child nodes, followed by a shout permit by the microcontroller, followed by a shout by one of the child nodes. For the shout

request and permit, we can stay in the SF, but for the shout itself, we go briefly to state $S2$ to let the address pass. Any other command that is not recognized or not expected to be received in state SF, whether from a child node or the parent node, is ignored.

7.4 A slightly less simple example

Figures 7.8 and 7.9 show some driving signals from the free-form modular display driver driving the display from Figure 7.8a. The initialization process is seen in Figure 7.8. The display is turned on and $M1$ asks the microcontroller for an address. After the address is received, $M1$ puts its outputs high so that $M2$ can do the same thing. Since there are no more pending address requests, $M1$ can already start shouting. $M2$ processes the address requests from $M3$ and $M4$, in that order, while adjusting the `children` vector. $M1$ was already finished shouting its own address, so $M2$ can immediately start shouting after asking. $M1$ received the address from $M2$ and shouts it to the microcontroller. In the mean while, both $M3$ and $M4$ requested a shout permit from $M2$. $M3$ is granted first and since $M3$ has no child nodes, it shouts using `sht_end`. `adr_end` is adjusted. $M4$ does not get a permit before $M2$ cleared its shout buffer by shouting the address from $M3$ to $M1$. When $M4$ can finally shout, it also uses `sht_end` and `adr_end` is again adjusted. All children have now finished shouting, so $M2$ also uses `sht_end` to shout the address from $M4$. Finally, $M1$ also shouts using `sht_end` and the initialization is finished. It is only now that `out_control` is adjusted, to activate the bypasses of the modules.

Figure 7.9 shows what happens if we add and remove modules. First $M5$ is added in the system (See Figure 7.9a). The lowest gate number is connected to $M4$, so this will become its parent node. $M5$ asks and receives an address, and shouts it to $M4$. $M4$ shouts the address directly to the microcontroller, as the bypasses of all other modules are activated. The state of *Main Control* of $M2$ changes to $S2$ as the shouted address passes. A little bit later, $M4$ is disconnected. After $M2$ detects the missing child node (`gone = 1`), it quickly adjusts the `out_ctrl` vector. Since $M4$ is $M2$'s own child node, the `children` vector is adjusted. $M1$ also adjusted its `out_ctrl` at the same time, but realized that it was not its own child node that has disappeared, and changed it back to what it was. $M4$ was also the parent node of $M5$. The missing parent node is detected and the state goes back to $S0$. $M5$ tries to reconnect to the system by sending an adopt request to $M3$. $M3$ now has an extra child node and $M5$ a new parent node. At the end of the waveforms, a polling routing is started (`sht_req` from microcontroller). Every module resets the `children` vector, deactivates the bypasses and performs a shout request. The shout routine can commence and the missing $M4$ will be noticed.

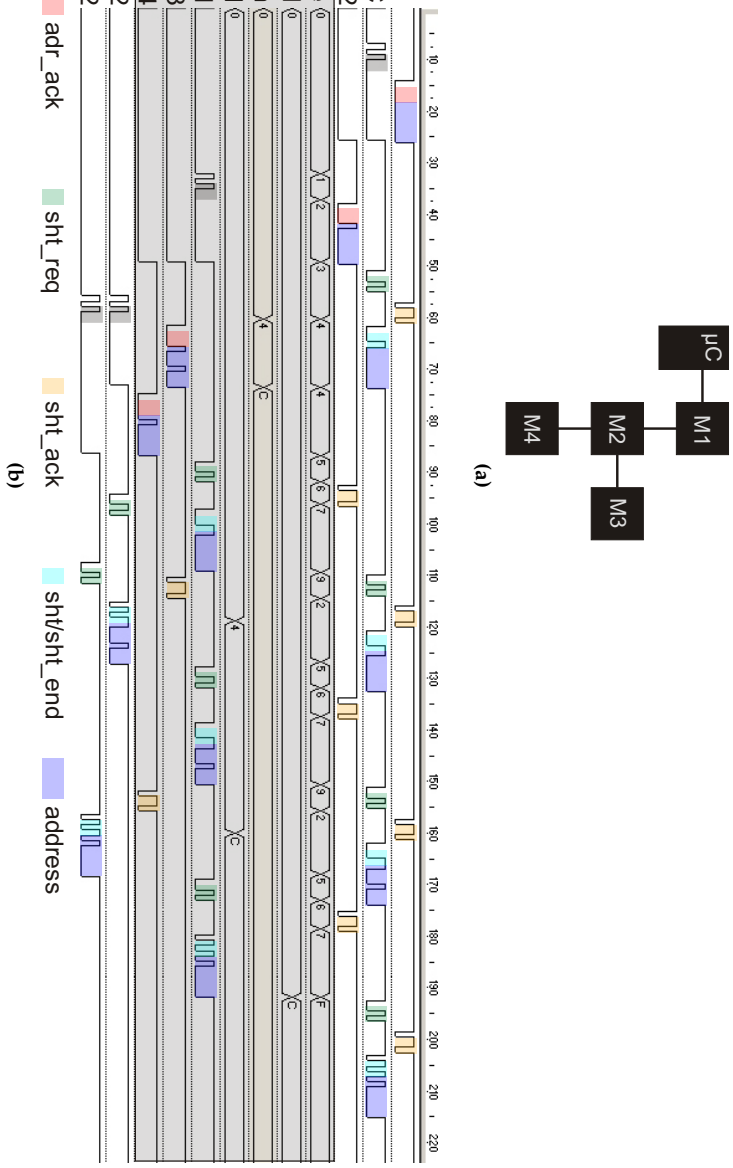


Figure 7.8 – Example of the initialization process for the free-form modular display driver, according to the display configuration shown above.

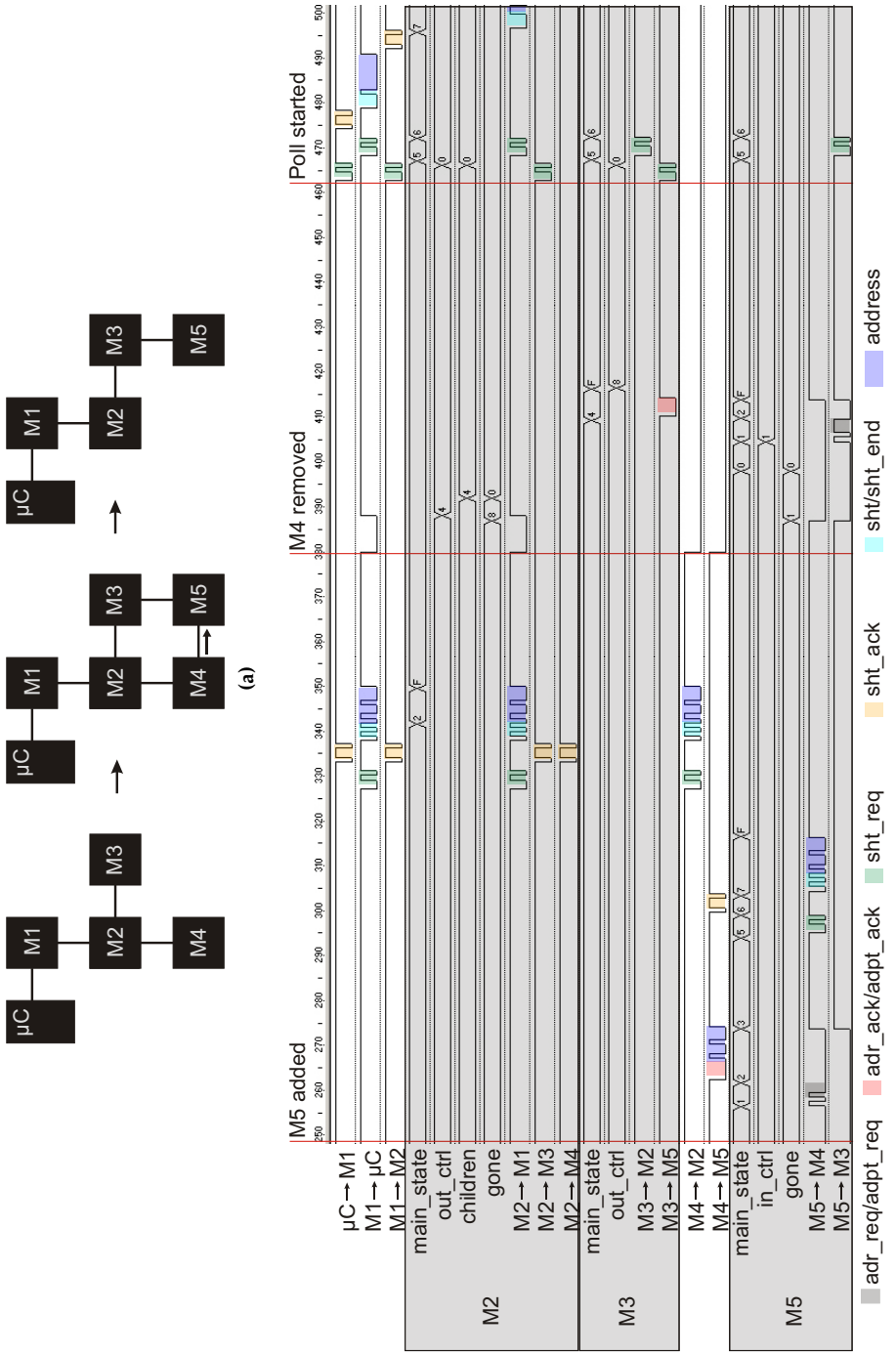


Figure 7.9 – Example of the driver signals when a module (*M5*) is added after the initialization, when a module (*M4*) is removed, and during the polling routine.

7.5 Setting up the test environment

I used the LED modules again to test this free-form modular display driver. The VHDL implementation of *Main Control* is included in Appendix C. Since there are a lot more states than in previous drivers, the implementation is quite long. Some less important or repetitive parts have been excluded.

This time the GUI has changed a bit. The outside looks pretty much the same, although the option to enter a display size has disappeared. The biggest changes are in the functionality of the program, since now it also has to process incoming information from the microcontroller. There are two types of data streams can be received. One is a stream with addresses of modules that need to be added to the system. This can either be at the end of the initialization, or when modules are added after the initialization. The other stream is the result of a polling, where the addresses from the stream are compared with the existing addresses in the GUI.

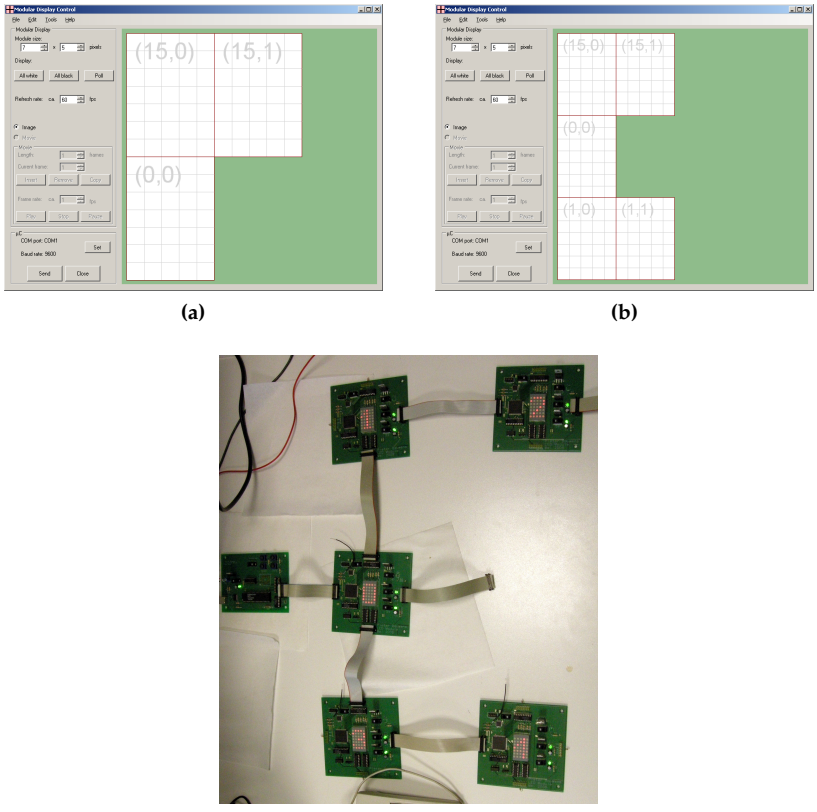
7.6 Some first results

I created some test displays and after turning them on, the configuration was detected correctly. I was also able to add and remove modules without interfering with the operation of the display itself. The microcontroller was programmed in such a way that the module shouting the first address would display the number 0, the second number 1 and so on. This continues also after initialization. In Figure 7.10 first the top three modules were turned on (Figure 7.10a). After the initialization, the bottom two modules were turned on (Figure 7.10b). The resulting display can be seen in Figure 7.10c. The top three modules have numbers 0, 1 and 2, according to their distance to the microcontroller. Later, the bottom two modules received numbers 3 and 4.

Figure 7.11 shows that the display still responded as expected after those two modules were added.

Let's take a look at the initialization time. In this system we have a bottleneck situation. While the addresses might be distributed in parallel (just like with the improved modular display driver), every address needs to be send back to the microcontroller, which happens serially. The oldest node can only send one address at a time to the microcontroller. The total initialization time will again only be dependent on the number of modules in the display. With a bit rate of 1Mbit/s, the display needs on average about $40 - 50\mu$s per module to initialize. There are some variations depending on the exact display configuration. In a denser connected network (as opposed to, let's say, one long chain), some things (like some address requests) can happen in parallel.

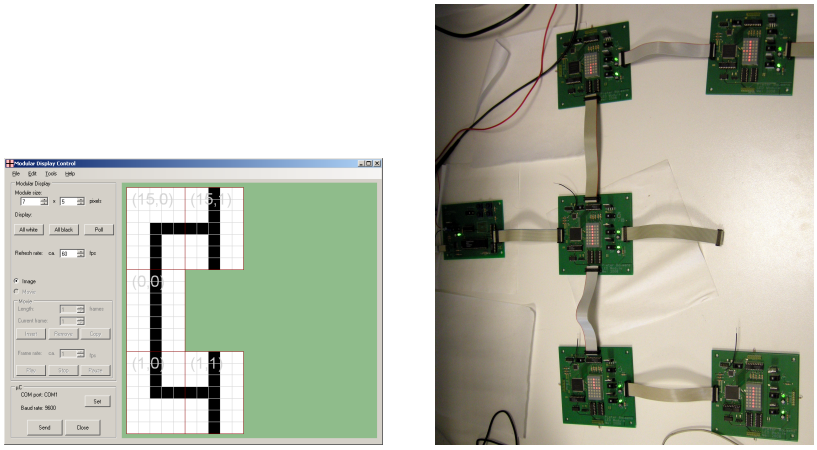
Next to the initialization time, the time to complete a polling routine is also im-



(a) (b)

(c)

Figure 7.10 – Images of the GUI and the corresponding display setup for the free-form modular display driver. First the top three modules are turned on (a), then the lowest two (b). The result is shown in (c).



(a) (b)

Figure 7.11 – After the modules were added, the display still works as required.

portant. For larger networks, this time will be about the same as the initialization time (around $40\mu s$/module), since during the initialization, the shouting of the addresses overlaps with the distribution of the addresses for the largest time.

7.7 But maybe we could do something more?

With our free-form modular display driver, we already created a lot of freedom. We can not only create a display as we see fit, but the display configuration is detected automatically. We can also add and remove modules while the display is running without interfering with the operation of the display. But, as always, there are some limitations.

We said that a display of any shape could be created, but this is not entirely true. We are still limited in creating a flat matrix of modules. If we were to create a flexible display and used flexible modules (See also Section 10.5) we could create a lot of problems. With flexible modules you could easily create a display, connect the modules in such a way that they don't fit in a flat matrix anymore. Addresses would be wrong, or worse, two modules could receive the same address. And on top of that, the address itself would be irrelevant to the location of the module.

Another annoyance is that every module is supposed to be oriented in the same direction, in order for the correct addresses to be distributed. When creating square modules that look the same from every direction, why not make it possible that the modules can be oriented either way? Let them sort it out.

And last but not least, when we're moving away from the strict matrix formation and predefined orientation anyway, why limit ourselves to *square* modules? If we could use triangular modules, we could create a lot more display shapes, especially when looking at creating 3D shaped displays.

References

[1] R. Diestel, *Graph Theory*. Springer-Verlag, 2005.

[2] Wikipedia. Minimum spanning tree. [Online]. Available: http://en.wikipedia.org/wiki/Minimum_spanning_tree

[3] J. F. Kurose and K. W. Ross, *Computernetwerken, een top-down benadering*. Pearson, 2003.

[4] P. J. Ashenden, *The VHDL Cookbook*. University of Adelaide, 1990.

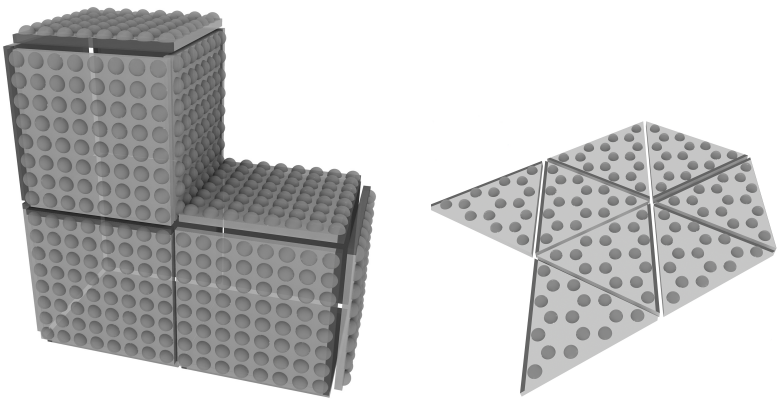
I may not have gone
where I intended to go,
but I think I have ended up
where I needed to be.
Douglas Adams (1952-2001)

8

Improved free-form modular display driver

8.1 Introduction

In the conclusion of last chapter, some improvements were proposed for the free-form modular display driver. These improvements are implemented in the driver described in this chapter. This driver will conveniently be called 'the improved free-form modular display driver'. It will make it possible to create displays with flexible modules, where the modules don't have to be placed in a matrix pattern, or to create 3D-shaped displays. There will be even more freedom in display shapes, since the driver allows the modules to be something other than simple squares. See Section 8.2 for more details. Previous drivers were, in the end, all based on the same principle, namely that modules were able to calculate the addresses for their neighbors. We'll see that with the improved free-form modular display driver this will no longer be possible. The new algorithm is built up from scratch and explained in Section 8.3. After the accustomed example in Section 8.4, we'll show some results in Section 8.6.



(a) (b)

Figure 8.1 – Displays that can be used with the improved free-form modular display driver. The dark edge represents gate 0.

8.2 Requirements

8.2.1 The display

We want to give the display some more options, some more variety in shapes, in applications. Take Figure 8.1a for example. The modules are used to create a 3D-shaped display. If we were to use the previous driver, a first problem arises: how do you orient the modules? In a flat matrix there is a clear 'up' and 'down', but in this case there isn't. And since the calculation of the addresses relies on the fact that every module is directed 'up', this could cause problems. It's obvious that in such a 3D-shaped display, the address of a module would be dependent on what it happens to choose as parent node. This is not necessarily a problem, except when it happens that, and this is not unlikely, two modules would receive the same address. Even more so for displays where the modules are made flexible. Flexible modular displays could for example be used in clothing. See also Section 10.5 for more information. The amount of ways that flexible modules can be connected is so large that it becomes extremely likely that the addressing of the modules would fail miserably. In other words, if we want to be able to create such displays, we'll have to find another way to provide every module with an address.

Since the addresses won't be provided from the modules themselves, the orientation of a module in the display won't be so relevant. Well, as we'll see, it will be relevant, but it can be taken care of, as opposed to with the previous drivers. And since the orientation of the modules can be taken care of, the display could also work with modules that aren't square (See Figure 8.1b). Creating a decent dis-

play with triangular modules is simply impossible if every module would need to be oriented the same way. Being able to use triangular modules creates a whole new degree of freedom in display shapes. Actually, the algorithm is also valid for pentagonal, hexagonal, ... modules.

It has probably become clear that, since the addresses are distributed in another way, the addresses will not be able to be used as an indication of the location of the modules anymore. We still want to be able to detect the configuration of the display, so this information will have to be retrieved another way. Also adding and removing modules after the initialization should remain possible. The representation of the created display in the GUI will not be an exact replica of the display, meaning that if a 3D-shaped display is created, the GUI will show a flat, unfolded representation.

8.2.2 The driver

Most of the requirements for the driver are the same as for the previous driver (See Section 7.2), but there are some differences. The most important difference is the fact that the address of a module cannot be determined using the addresses of the neighbors. A module does not know how the other modules are connected. The only entity that can have a complete view over the entire display is the microcontroller. If we want to make displays like these possible, every module will have to ask the microcontroller personally for an address. At startup there will be a lot of modules that want an address at the same time, so the algorithm will have to make sure this happens in a controlled way.

Another big change is that the addresses, now distributed by the microcontroller itself, hold no information on the location of a module. Other information to determine the display configuration is needed, and this information needs to reach the microcontroller in a safe way. In previous chapters, some network protocols were proposed and rejected. The arguments still apply for this driver. But now the modules can be connected in a more complex network, where the only option to receive an address is by asking it to the microcontroller. A more complex protocol may be suitable. An idea would be to base it on the Internet Protocol (IP). This was not discussed in Chapter 4 because it is entirely situated in the Network Layer and needs a protocol from the Data Link Layer to operate. With our network, this would be the Ethernet protocol, where the modules could act as switches and end-systems at once. Almost any network topology is possible and routing could be fairly simple. Also, if a node/module would break down, the system would find a way to recover itself. There is one major, fatal problem. Ethernet is based on fixed MAC addresses, so we would need another protocol anyway to provide the modules with a unique address. Again, it is more efficient to develop our own protocol that is able to do everything at once.

The other requirements remain the same. In order to avoid data loops, the created

mesh network needs to be transformed to a tree network. And this tree needs to be completely dynamic. So again, a similar implementation of the Spanning Tree Protocol can be used. If modules are added or removed after the initialization, the tree needs to be updated to provide every active module with an address and data. And of course, the changes need to be noted by the GUI.

8.3 Implementation

Notwithstanding the algorithm will be completely different form previous driver, the block diagram is exactly the same (See Figure 7.2). The only differences are found in *Main Control*. *Out Control*, *Rx* and *Tx* operate mostly in the same way (See also Section 7.3 and below).

8.3.1 Communication protocol

Since the communication between the modules is a little bit more complex than with the previous driver, we need to add an extra bit to the command sequences. Some of the commands resemble those from the free-form modular display driver, but most of them have a completely different function. They can be found in Table 8.1. A couple of commands are followed with 'information', which will be used to determine the display configuration. What this information is and how it is used is explained in the next section.

Whereas the modules using previous driver only needed to communicate with their neighbors, these modules need to be able to communicate with the microcontroller itself. This can be organized in following way.

During initialization (see below) all bypasses are still inactive, so a communication line needs to be set up between a module and the microcontroller. If a module wants to talk to the microcontroller, it sends a send request (`send_req`) to its parent node. This node will send a `send_req` to its own parent and so on, until the request reaches the microcontroller. If the microcontroller is ready to receive data, it sends a send permit (`send_ack`) to the first node. This will send the permit to the child that requested it and activates the bypass between its parent node and concerning child node. This child node will do the same. When the send permit reaches the module that first requested it, there will be an open path between that module and the microcontroller. They are now free to communicate with each other. After the communication all bypasses are deactivated again. If a module were to receive two `send_req`s, the later received `send_ack` will go to the child that requested to send first (or the child with the lowest gate number). The other child has to wait until the communication is finished. That module will then again perform a send request, and the following `send_ack` now goes to the other waiting child node. This process will be called the *send routine* and is

Command		Info	
0000	error	Disappearing neighbor	
0001	send_req	Request a send permit	
0010	send_ack	Send permit	
0011	send_end	No more sending is needed	
0100	ready	Ready to receive new data	
0101	adr_req1	Preliminary address request	+ info
0110	adr_req2	Complete address request	+ info
0111	adr_ack	Send address	+ address
1000	poll_req	Polling request	
1001	poll_ack1	Preliminary polling answer	+ info
1010	poll_ack2	Complete polling answer	+ info
1011	adpt_req	Adopt request	
1100	adpt_ack	Adopt permit	
1101	param	Send parameter data	+ data
1110	data	Send image data	+ data
1111	reset	Completely reset driver	

Table 8.1 – Command sequences used by the improved free-form modular display driver.

illustrated in Figure 8.2.

8.3.2 General principles

The main principle remains the same as that from the free-form modular display driver, namely that the mesh network needs to be transformed to a tree network. Every module will choose a parent node, and can have 0 to 3 child nodes (using square modules). The choice of the parent node is based on which adjacent module puts its outputs high first. When a module found its parent, it can start initialization. The important vectors like `children`, `in_control`, `out_control` and `adr_end` are used and adjusted in the same way as with the free-form modular display driver, so I will not mention them here (See Section 7.3.3). Another point is that, while the algorithm itself makes displays from pentagonal and hexagonal modules possible, the driver is designed for triangular and square modules. This is because the identification of the different gates was only made two bits long. For modules with more gates, extra bits are required, some registers and timers need to be adjusted, and so on. But the algorithm itself wouldn't change.

Detection of the configuration

We still want to have a display where the display configuration can be detected. Since the addresses don't hold any information any longer, we need to derive it

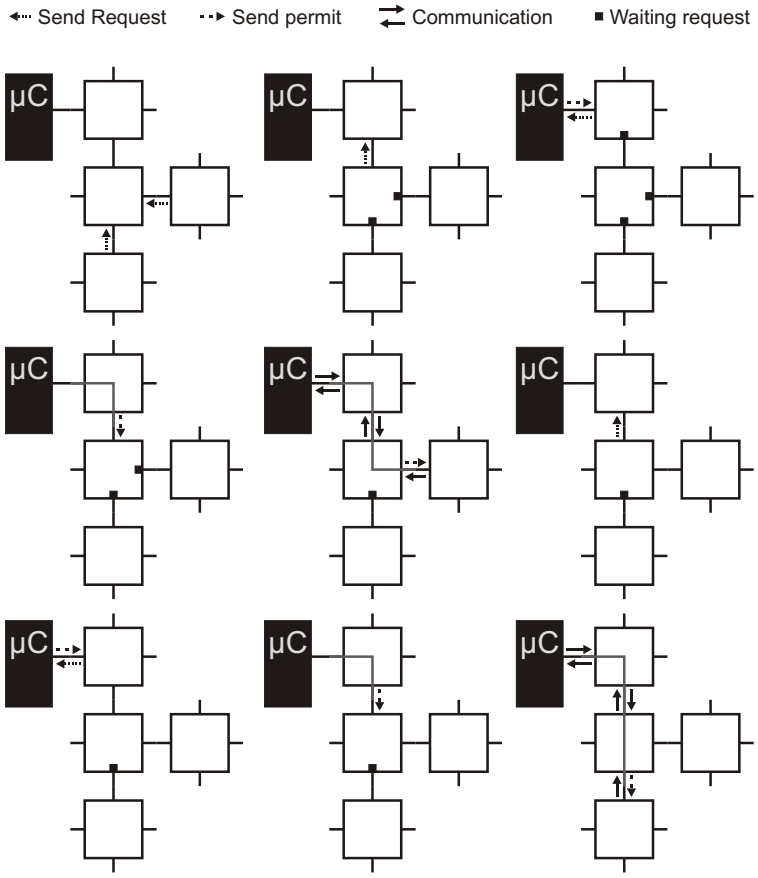


Figure 8.2 – The send routine in the improved free-form modular display driver

in another way. Before going into the initialization itself, let's first take a look at what information is needed to determine the configuration. When I say 'configuration' in this context, it actually means two things. Not only do we need to know *where* a module is located, but also how it is oriented. If a module is upside down, the image data that needs to be sent to that module will obviously be different.

Let's assume we know the orientation and location of every module in the display, and we want to detect the location and orientation of a new, unknown module. The location of this new module can be determined if we knew the address of the parent node and the gate number this new module is connected to, since we know the location and orientation of this node. The orientation of the new module, on the other hand, is known if we know the gate number of the new module the parent node is connected to. So in short, we need three pieces of information: the address of the parent node, the child gate number of the parent and the parent gate number.

It is clear that, opposed to the previous driver, this is a recursive algorithm. Location and orientation are only known when the location and orientation of the parent node is known. For the location and orientation of the first, oldest parent (directly connected to the microcontroller) only the gate number the microcontroller is connected to (parent gate number) is required.

Initialization

The display is turned on and every module will wait for a high input on one of its gates. As said, the adjacent module that first provides a high input will be chosen as parent node. When the parent is chosen, the module wants to receive an address. It performs a preliminary address request (`adr_req1`) to its parent. This request is followed by the number of the gate the parent is connected to. The parent stores this number in one of the receive buffers (See *Rx*). It will now try to perform a complete address request (`adr_req2`) to the microcontroller. For this, a connection need to be set up with the microcontroller via the send routine explained above. The parent node sends a `send_req` and waits for a `send_ack` from its parent. There is now a free path to the microcontroller to send an `adr_req2`. This complete address request is followed by some information. First follows the address of the parent itself (only known by the parent itself), then the gate number of the requesting child node (also only known by the parent) and finally the gate number the child node sent with its preliminary address request (parent gate number). This is exactly the information the microcontroller needs to determine the location and orientation of the address-requesting module (see above). After the parent node sent this complete address request, it activates the bypass between the open communication line and the requesting child node. Everything the microcontroller sends will now be seen by this node. Since the microcontroller keeps track of every module, it can search for the next available

address and send it through the communication line directly to the module that sent the preliminary address request. After seeing the address pass by, the participating modules deactivate their bypasses again. From the viewpoint of a new module, following happens: chooses parent, ask address (`adr_req1`), receive address (`adr_ack`) and the necessary information has reached the microcontroller. This also applies for the module directly connected to the microcontroller. It chooses the microcontroller as parent and sends a preliminary address request with the parent gate number, which is the only information needed for this first module. After a module receives its address, the outputs are driven high, so adjacent, waiting modules can start initializing.

The same problem as with previous driver occurs: how do we know when the initialization is finished, when can we be sure that there won't be anymore incoming address requests? The solution to this problem is also similar. After receiving an address, the module waits for incoming address requests, if these don't come, this module assumes it has no children and a `send_end` will be sent to the parent node, indicating that no more address requests will come from this node. A module that does have children will send a `send_end` when it received a `send_end` from all of its children. It has the same function as the shouts with `sht_end` in the free-form modular display driver. If a `send_end` should arrive from a certain gate, and it doesn't before a timeout occurs, the `adr_end` vector is adjusted to be equal to the `children` vector, so the module can send a `send_end` anyway. This way the total initialization process doesn't stall. After sending the `send_end` command, the module will activate the bypasses between parent and child nodes (See *Out Control*). When the microcontroller receives a `send_end` the initialization is finished, and all bypasses are activated.

Adding modules

This same process can be applied to modules that are added after the initialization of the display. Just as with the previous driver, a freshly added module is not connected to the microcontroller yet, but only to its adjacent module(s). After choosing a parent, its send a preliminary address request together with the parent gate number. This parent will now try to set up a connection with the microcontroller to perform the complete address request. However, since all bypasses are activated, the `send_req` from the send routine will arrive directly at the microcontroller. We only have to make sure that the data line isn't occupied with other data when sending. Since this is the same situation as with the previous driver, the solution is the same (See Section 7.3.3). The module will send the `send_req` in between frames. After receiving the `send_ack` from the microcontroller, the complete address request is performed (with parent address, child gate number and parent gate number). The microcontroller sends the next available address which arrives at the new module. If no other modules are attached, the new module sends a `send_end` to its parent, which will send a `send_end` to the mi-

crocontroller. The actions of the microcontroller are halted until the `send_end` is received, so the data line will be clear. After the `send_end` the necessary bypasses are activated and the new child is added in the tree.

Removing modules

We told that when removing a module, two problems arise. The first being that the dropping line from a removed child node is forwarded all throughout its ancestors. The second being that removing a parent node causes its offspring to stop receiving data.

The solutions are again very similar to those proposed for the previous driver. *Rx* will notify *Main Control* that it suspects that a module has gone missing. This time we can also use the extra `error` command (not in Manchester mode, of course). When the suspected node is a child node, the bypass to this child is quickly deactivated. When the signal is still low after a couple of bits, the child node is really gone and the `children` vector is adjusted. If the input signal rises again, the missing node is not an own child node and the bypass is activated again.

Also the problem of the missing parent node is solved in the same way as before. If the parent node is gone, *Main Control* performs a full reset. Bypasses are deactivated and the modules go looking for a high input. When found, an adopt request (`adpt_req`) is performed to the new parent, which will respond with an adopt permit (`adpt_ack`). Just as with the previous driver, the tree is rebuilt and every active module can again receive data.

Keeping the GUI updated

To update the GUI, a polling needs to be done once in a while. If we want to be able to rebuild the tree at the GUI side, we again need the information that was sent during initialization. After all, when a parent node has disappeared and its child nodes changed parents, this has to match the information that is still in the GUI. A polling request (`poll_req`) is sent by the microcontroller. Upon receipt, every module deactivates the bypasses and respond by sending a preliminary polling answer (`poll_ack1`) to their parents. A `poll_ack1` is followed by the address of the module and the parent gate number. This information is temporarily stored in the parent node. When the parent node is ready, it will send a complete polling answer to the microcontroller (`poll_ack2`) using the send routine. The complete polling answer is followed by the address of the parent node, the child gate number and the information sent by the child itself (address and parent gate number). With this information, the microcontroller can check which modules are gone and whether the other modules have changed parent node or not. When the checking is finished, the microcontroller responds by

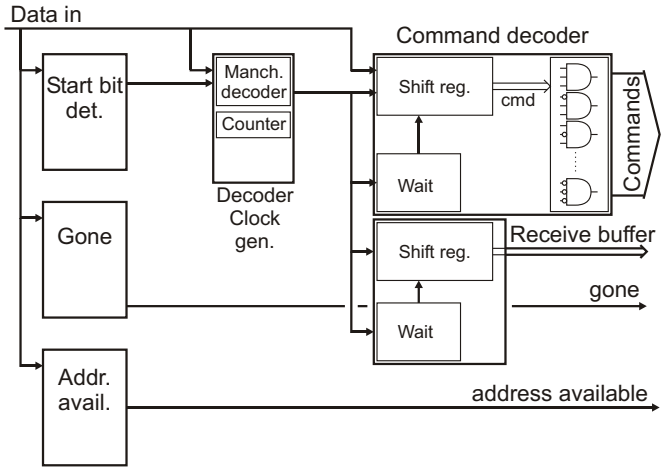


Figure 8.3 – Block diagram of *Rx*

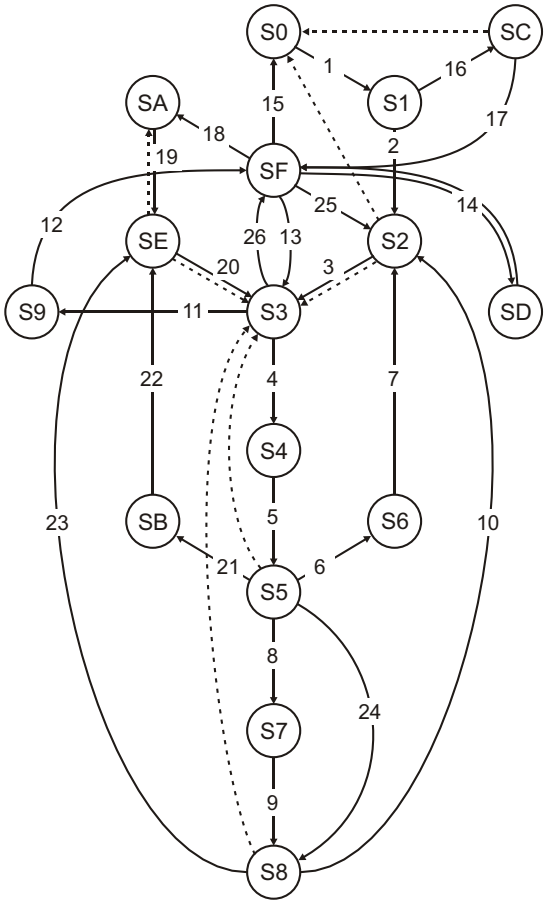
sending `ready`. This `ready` command is also forwarded to the child that performed the `poll_ack1`, just like the `adr_ack` during initialization. After receiving the `ready` command, a module can process the `poll_ack1`s from its child nodes. When no `poll_ack1`s are received, the module sends a `send_end`. The other modules send a `send_end` when every of its child nodes sent a `send_end`. Polling is finished when `send_end` reaches the microcontroller.

8.3.3 *Rx*, *Tx* and *Out Control*

Rx, *Out Control* and *Tx* have the same functionality here then in the previous driver. Only *Rx* differs a little bit. Since the preliminary address requests and polling answers are accompanied with some data, this data needs to be able to be stored per gate. The maximum of data is 8 bits (address) plus 2 bits (gate number), so a shift register of 10 bits suffices. The commands are now four bits long, so the *Command decoder* will have 17 outputs (16 commands + 1 'command received' signal). The *Gone* block also hasn't changed, except that now the `error` command can be used to detect missing modules.

8.3.4 *Main Control*

The state machine for the improved free-form modular display driver is shown in Figure 8.4. There are now 16 states (See 8.2). The dotted lines show what happens if a wrong command is received or if a timeout occurs. The fall-back state here is either $S3$ where the commands from the child nodes are processed, or $S0/SA$ if



#	Info	#	Info
1	Address available	14	send `adpt_ack`
2	Prel. address request sent	15	Parent node gone
3	Address received	16	Parent was gone
4	Command received	17	Adopt request sent
5	Send request sent	18	Start poll
6	Command was `adr_req1`	19	Prel. polling answer sent
7	Compl. address request sent	20	`ready` received
8	Command was `send_req`	21	Command was `poll_ack1`
9	Send permit sent	22	Compl. polling answer sent
10	`adr_req2` passed	23	`poll_ack2` passed
11	All children sent `send_end`	24	No own new child nodes
12	`send_end` sent	25	`adr_ack` from μC
13	Command received	26	No own new child nodes

Figure 8.4 – The state machine of the improved free-form modular display driver

you sent a preliminary address request or polling answer.

State	Info
*S*0	Begin state
*S*1	Send preliminary address request or adopt request
*S*2	Receive/pass address
*S*3	Receive command
*S*4	Send send request
*S*5	Receive send permit
*S*6	Send complete address request
*S*7	Send send permit
*S*8	Pass complete address request/polling answer
*S*9	Send `send_end`
SA	Send preliminary polling answer
SB	Send complete polling answer
SC	Receive `adpt_ack`
SD	Send `adpt_ack`
SE	Receive/pass `ready`
SF	Normal operation

Table 8.2 – The states from the state register in the improved free-form modular display driver explained

As always, we start in state *S*0 when the display is turned on, or after a full reset. The parent node is chosen, and we move to state *S*1 to send a preliminary address request (`adr_req1`) or an adopt request (`adpt_req`) depending of whether we're here after startup or after losing our parent node. In case of the latter, the `adpt_ack` is received in state *SC*, after which we can go to the final state *SF*. In the other case, state *S*2 is used to receive our address (`adr_ack`). If anything were to go wrong in the reception of either `adpt_ack` or `adr_ack`, we return to state *S*0, so the `adpt_req` or `adr_req1` can be sent again. State *S*2 will also be used to let an address pass (See below). If the address is correctly received, the outputs are driven high.

State *S*3 is used to receive commands from the child nodes. These can be preliminary address requests, send requests (`send_req`), preliminary polling answers (`poll_ack1`) or `send_end`s. After receiving the address in state *S*2, we arrive in state *S*3 to process the incoming address requests. If, after a while, no `adr_req1`s are received, we go to state *S*9 to send `send_end` and end in the final state *SF*. If `adr_req1`s do arrive, the `children` vector is adjusted accordingly and in state *S*4 the send request is sent. We wait for the send permit (`send_ack`) in state *S*5. After the receipt, we have a direct communication line with the microcontroller. If the `send_ack` is not received (wrong command or timeout), we go back to state *S*3. If we were in state *S*5 because of, for example, an `adr_req1` from

a child node, this child node will resend the adr_req1 after the timeout, so we have to be in state *S*3 to be able to process this. If we are in state *S*5, on the other hand, because of a send_req of one of the child nodes (an offspring node wants an address), this child node will also go back to state *S*3, waiting for the resending of the adr_req1 of that offspring node. This child node will then resend the send_req. Again, we need to be in *S*3 to be able to process this.

Where we go from state *S*5 depends on the command that was received in state *S*3 (adr_req1, poll_ack1 or send_req). Logically, the first commands that we'll see in state *S*3 will be the adr_req1s, from child nodes that want an address. If this command was received, a complete address request (adr_req2) is performed in state *S*6. After this request, the bypass to the child node that performed the preliminary address request is activated. When the microcontroller answers, it is directly forwarded to this child node. The receiving (more like passing through) of the address occurs in state *S*2, the same state were the own address was received. Afterwards, the bypass is deactivated again and in state *S*3 the next commands are processed. Also when something goes wrong during the address receipt, we should go back to state *S*3 since the child that needed the address will eventually ask it again.

In state *S*3 the other received adr_req1s are processed in the (*S*3, *S*4, *S*5, *S*6, *S*2)-loop. After a while other commands will start to poor in. Either send_ends from child nodes that noticed they don't have child nodes of their own to which the adr_end vector is adjusted, or send_reqs from child nodes that have other child nodes asking for an address (or child nodes that received a send_req themselves). These offspring nodes will perform a send_req in order to be able to do the complete address request. If a send_req is received, a send_req is sent in state *S*4 and a send_ack is received in state *S*5, just as before. Only now we go to state *S*7 to send a send_ack to the child node. The bypass between parent and concerning child are activated to provide for the open communication line. State *S*8 is used to let the adr_req2 pass. Again, if something goes wrong, we return to state *S*3. Since an address request was performed, the microcontroller will respond with an address. We can let it pass in state *S*2. After the address has passed, all bypasses are deactivated again. We run through the (*S*3, *S*4, *S*5, *S*7, *S*8, *S*2)-loop until all offspring nodes have an address. At that time, the send_ends will start coming in. If the send_end is received from every child, we send one of our own in state *S*9. After that we can go to the final state *SF*, where the bypasses are activated and where we enter normal operation.

The polling routine follows a similar pattern. If a poll_req is received in state *SF*, the bypasses are deactivated and the preliminary polling answer is sent in state *SA*. In state *SE*, we wait for the ready command from the microcontroller. If this command is not (or wrongly) received, the poll_ack1 is resent (cfr. adr_req1). In *S*3 the incoming poll_ack1s are processed. If none are received, send_end is sent in *S*9. After a poll_ack1, we send a send_req in state

*S*4 and receive the `send_ack` in state *S*5. This time we follow the path to state *SB*, where the complete polling answer (`poll_ack2`) is sent. The bypass to the child is opened to let the `ready` command pass in state *SE*. Back in *S*3 the other `poll_ack1`s are processed in the (*S*3, *S*4, *S*5, *SB*, *SE*)-loop. As during the initialization, `send_req`s will start to come in after a while, these will be processed in the (*S*3, *S*4, *S*5, *S*7, *S*8, *SE*)-loop. The same loop from the initialization, only now we have to wait for the `ready` command instead of the `adr_ack`. When the entire offspring has answered the polling request, every child will have sent the `send_end`. We send our own `send_end` in state *S*9 and finish in state *SF*.

The reason why the `ready` command is used is twofold. First of all, checking and updating the module info in the microcontroller might take some time, so you don't want to send the following `send_req` to quickly after the `poll_ack2`. It's safer if the microcontroller indicates when it is ready to receive new data. But secondly, a module can receive multiple `poll_ack1`s, but can only process one at a time. The others have to wait. If a child node would not have to wait for a `ready` command after sending the `poll_ack1`, it would start processing its own received `poll_ack1`s and start to send a `send_req`. This would get lost, because its parent hasn't even processed the `poll_ack1` it sent yet. So the `ready` command doesn't only mean that the microcontroller is ready, but that your parent node is also ready processing your polling answer.

Some other things can happen in the *SF* state. When a new child arrives, it can either send an adopt request or an address request. The `adpt_req` is answered with an `adpt_ack` in state *SD*. For the address request, we have to send the complete address request to the microcontroller, so the (*S*3, *S*4, *S*5, *S*6, *S*2)-loop is needed. We only have to make sure that, while sending the `send_req` in state *S*4, the data line is free, since this request will go directly to the microcontroller. The adding of the new module is finished with a `send_end`.

Another thing that we need to be aware of is that, after the initialization, we do not always have to participate in the communication. If there is a new offspring node, which is not an own child node, we will receive a `send_req` in state *SF*, this command will be processed in state *S*3. This `send_req` will already be forwarded to the microcontroller, so coming from state *S*3, we cannot send a `send_req` ourselves in state *S*4. Same goes for the `send_ack` from the microcontroller. This will be forwarded through the bypass, so we'll have to skip state *S*7. After state *S*8 and *S*2, we're back in *S*3. Normally we would send a `send_end` before going to *SF*, but since we have no own new child nodes, we'll have to skip that. If the new module is not an offspring node, we will only see what the microcontroller sends. This is a `send_ack`, which can be simply ignored, and an `adr_ack` for which we'll go to state *S*2 to let it pass. As usual, we'll arrive in state *S*3 afterwards, but since there are no commands waiting and we have no new own child nodes, we'll go directly to state *SF*. Any other command from the parent or child nodes will be ignored.

8.4 Another example

Figure 8.5 and 8.6 show some relevant signals during the initialization of the display from Figure 8.5a, the addition of module *M*3 and removal of module *M*2. The `mux_sel_x`-signals are the selection signals for the output multiplexers, explained in Section 7.3.7. A summary is given in Table 8.3.

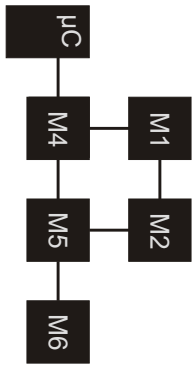
sel	**value**
00	default
01	Command/*Tx*
10	data_int
11	data_ext

Table 8.3 – Output sources

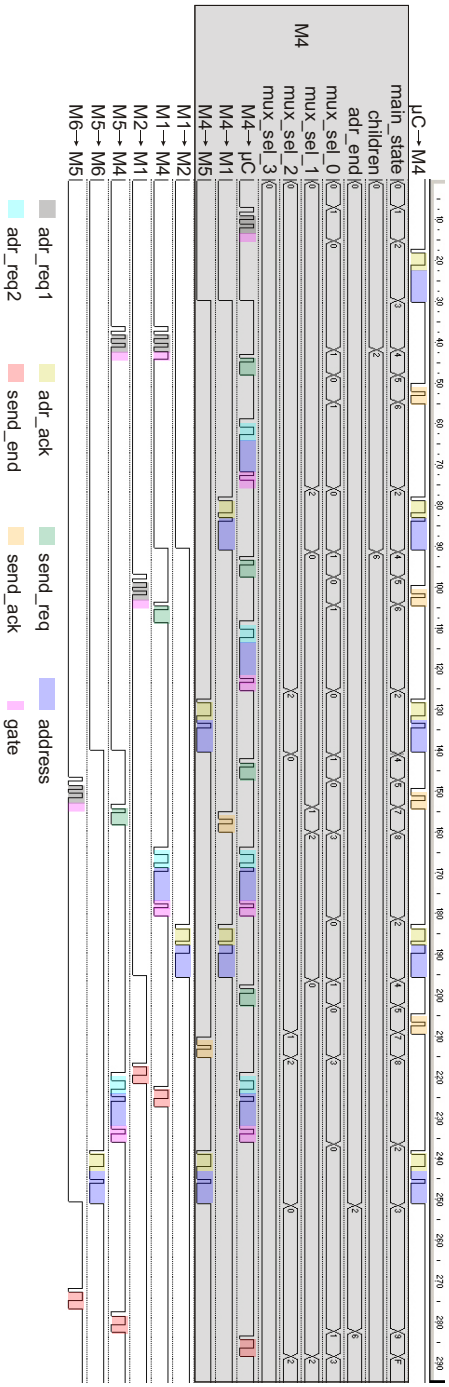
In this display, *M*4 is connected to the microcontroller, so when the display is turned on, this module performs a preliminary address request (`[adr_req1 + gate]`) to the microcontroller, which responds with an address (`adr_ack`). After *M*4 put its outputs high, both *M*1 and *M*5 send `[adr_req1 + gate]`. First *M*1 is served (lower gate number). *M*4 sends a `send_req` and the microcontroller responds with `send_ack`. *M*4 then performs the complete address request for *M*1 (`[adr_req2 + address + gate + gate]`). *M*4 opens the bypass towards *M*1. *M*1 is connected to gate 1, so `mux_sel_1` is set to 2 (= `data_int` = data from parent node). When the microcontroller sends the address, *M*1 receives it immediately. The same process is done for module *M*5.

In the mean while, *M*1 has already put its outputs high, so *M*2 can perform the preliminary address request. *M*1 can send a `send_req` to *M*4, but has to wait since *M*4 is busy with providing *M*5 with an address. As soon as this is finished, *M*4 continues the `send_req` to the microcontroller. After the `send_ack` from the microcontroller, the `send_ack` is sent to *M*1 and the corresponding bypasses are activated. The multiplexer for gate 0 (microcontroller) is set to 3 (= `data_ext` = data from child node), the multiplexer for gate 1 (*M*1) is set to 1 (`data_int`). *M*1 can now communicate with the microcontroller. The complete address request is sent and the returning address is directly forwarded to *M*2. Since module *M*5 also has a child node (*M*6), the same process is repeated for *M*5. A connection between *M*5 and the microcontroller is established, and *M*6 receives the address. By that time module *M*2 has realized that it has no child nodes and has sent a `send_end` to which *M*1 does the same. A little bit later *M*6 and consequently *M*5 send the `send_end` command. *M*4 now knows the initialization is finished and sends the `send_end` to the microcontroller. All bypasses are activated ('3' for the parent gate, '2' for the child gates).

Figure 8.6 shows what happens when a new module (*M*3) is added. *M*3 performs



(a)



(b)

Figure 8.5 – Example of the initialization process for the improved free-form modular display driver, according to the display configuration shown above.

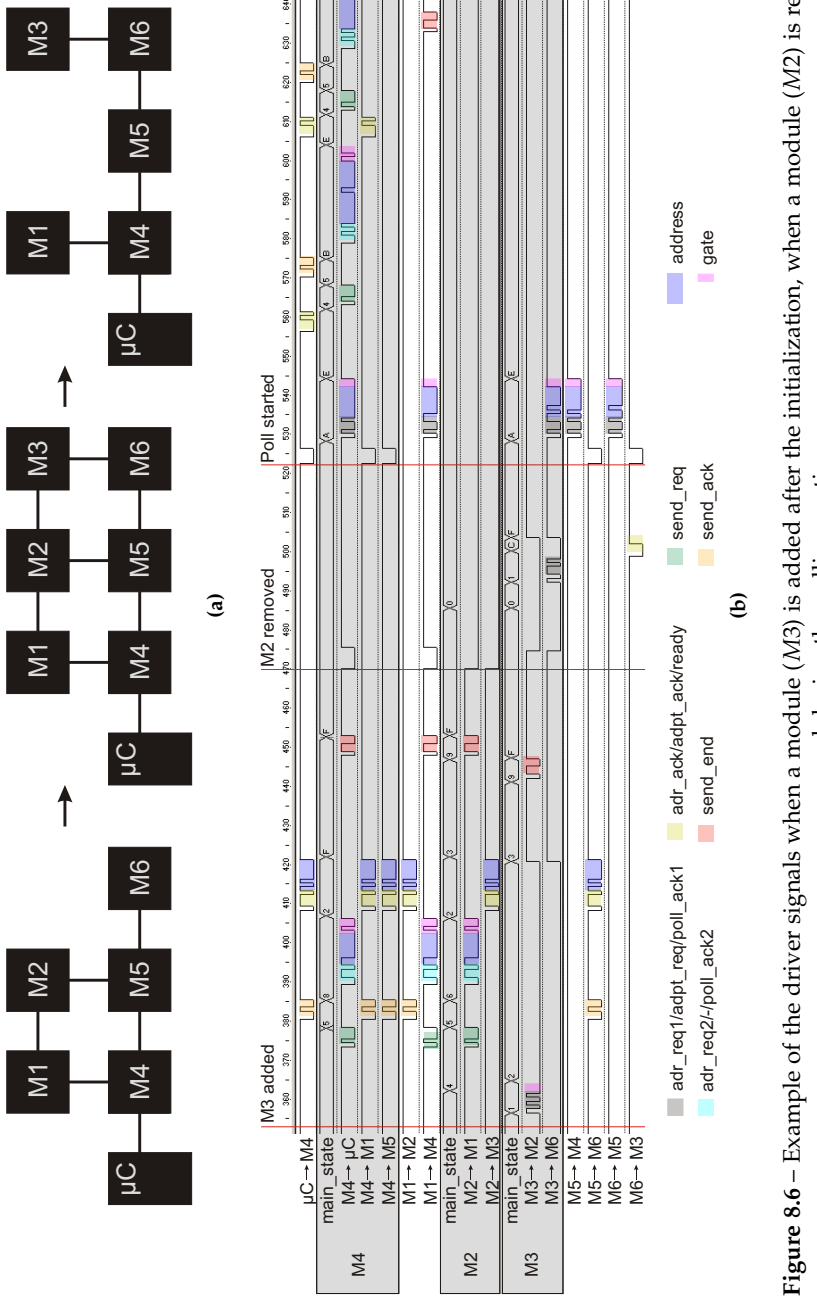


Figure 8.6 – Example of the driver signals when a module (*M3*) is added after the initialization, when a module (*M2*) is removed, and during the polling routine.

the preliminary address request to *M*2. Before sending `send_req` to the microcontroller, *M*2 has to wait a bit to check if the data line is free. The `send_req` is directly forwarded to the microcontroller and the responded `send_ack` is seen by every module. Same goes for the complete address request and returned address. While *M*4 does not participate in this process it still has to follow it (see `main_state` of *M*4). After the `send_end`, module *M*3 is included in the tree.

A little bit later, *M*2, the parent node of *M*3 is removed. After the removal is detected, *M*3 goes back to the first state and asks *M*6 for an adoption (`adpt_req`). *M*6 accepts (`adpt_ack`) and *M*3 again has a connection to the microcontroller. Just in time for the polling request. All bypasses are closed and every module performs the preliminary polling answer (`[poll_ack1 + address + gate]`). *M*4 is served first, being sent the `ready` command. It will now process the `poll_ack1`s of its child nodes. After the `send_ack` from the microcontroller, it sends the complete polling answer (`[poll_ack2 + address + address + gate + gate]`) with one address and gate sent by *M*1. The `ready` command is now directly forwarded to *M*1. Same goes for *M*5. Since *M*1 has no child nodes anymore, it sends the `send_end` command. Later, *M*6 and finally *M*3 will have answered the polling, using *M*4 (and *M*5) as bypass. The poll is ended with `send_end`.

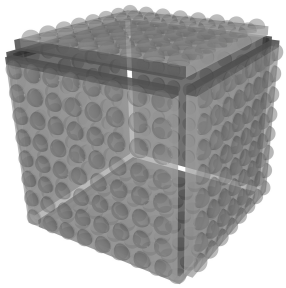
8.5 Setting up the test environment

Also this final driver was implemented in VHDL. The code for *Main Control* can be found in Appendix D. Again, this is not the complete code. Some repetitive parts have been left out.

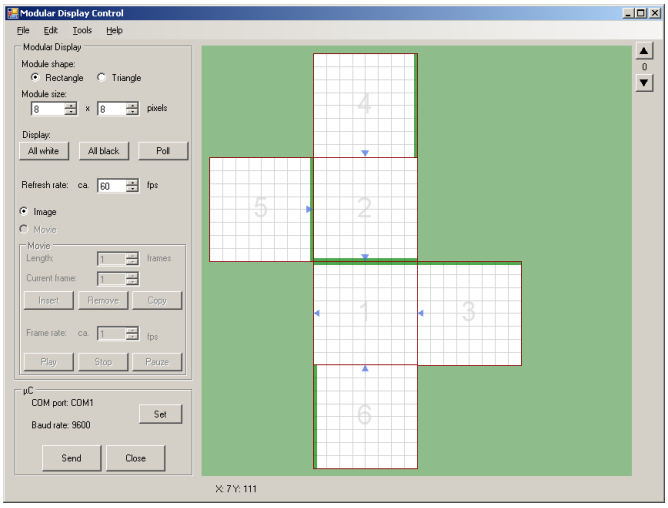
The GUI needed some extra functionality. First of all, we needed to implement the algorithm to determine the display configuration (or check the configuration after a poll) based on the information sent by the modules, both for square and triangular modules. We did not really create triangular modules, but since the driver is able to deal with them, so should the GUI.

While the created display does not have to be limited to a flat matrix of modules, if seemed sufficient that the display representation in the GUI was. The GUI will simply reflect how the modules are connected to each other (See Figure 8.7). In some occasions this can cause an 'overlap' of two or more modules, though. This is solved by creating different 'levels' in the representation. The button on the right lets you skip between the levels. So, each module can be assigned coordinates. The x and y coordinate represent the location on screen, the z coordinate the level it should be drawn in (See Figure 8.8). The dark arrows point to the parent node, the lighter arrows indicate where there is a level change. The dark bar on each module shows the gate 0.

After the initialization is finished, the microcontroller sends one data stream with

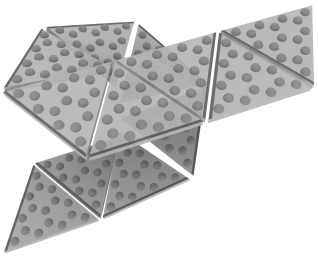


(a)

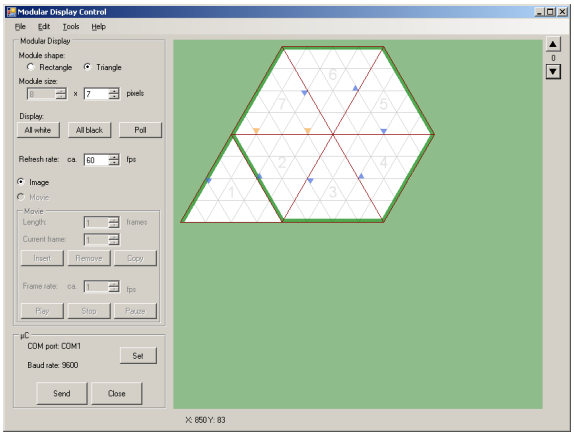


(b)

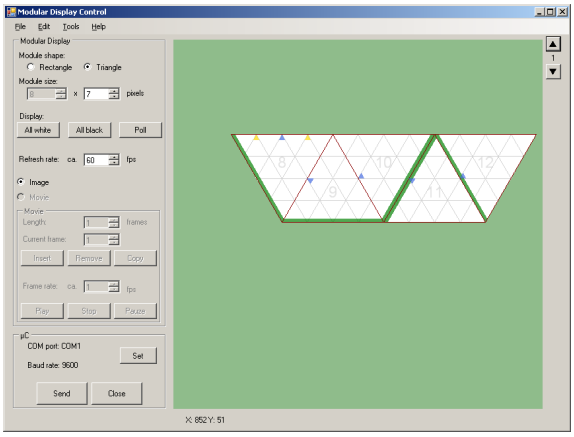
Figure 8.7 – Flat representation of a display configuration with square modules



(a)



(b) level 0



(c) level 1

Figure 8.8 – Flat representation of a display configuration with triangular modules with overlap.

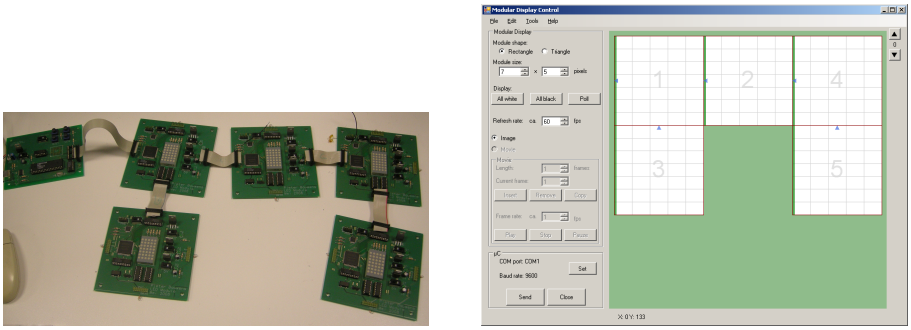
for each module the address (1 byte), the address of the parent node (1 byte) and 1 byte with the gate numbers (parent gate and child gate of the parent node). The microcontroller sends the information of the modules in the order of the addresses (as organized in the microcontroller memory). After an initialization, this will also be the natural order of the modules (first parent node, then child node). This is important to know, since as we told, the algorithm to determine the configuration is a recursive one. You cannot process a module if you don't have the information of its parent node. After a normal polling, however it's possible that a module was removed and that the tree has been rebuilt. The location of the modules in the microcontroller will not change, so the natural order of the modules might be scrambled. When a new module is added, the microcontroller will only send the information for this one module.

The processing of the information stream starts with the first module, which will always be the oldest parent node. From its parent gate number its orientation can be derived. In the case of square modules there are only four possible orientations. With triangular modules, however, there can be 6 possible orientations. The parent address and child gate number of the parent node will of course be irrelevant here. The next module will probably be one of its child nodes. Based on the orientation of the parent node and the supplied gate numbers, the orientation and coordinates can be derived. For each of the successive modules, the parent node is searched in the already processed modules and the orientation and coordinates are derived. If the coordinates are already used, the z coordinate is incremented. If the parent node is not yet processed, the module is moved to the end of the row, to be processed later. Following this procedure, every module will have received an orientation and coordinates after a while. The display can be drawn at this point.

8.6 Some first results

Let's take a look at how this driver works in real life. Figure 8.9a shows the display configuration when it was turned on. Figure 8.9b shows what display configuration was detected. Note that the modules here were not created to be connected in any orientation. The voltage supply and ground are supplied by the same flatcables as the data signals. Connecting the flatcable to another gate would create shorts. When a module is only connected to one other module, two orientations are possible. This was tested and proven to be working. But during these tests here only the 'up' orientation is used.

After the initialization, a new module was added, which was correctly detected and represented in the GUI (See Figure 8.10). The arrows in the GUI indicate that the new module chose the upper module as its parent (obviously because there was no connection made to the left module). Finally, we removed the upper



(a) (b)

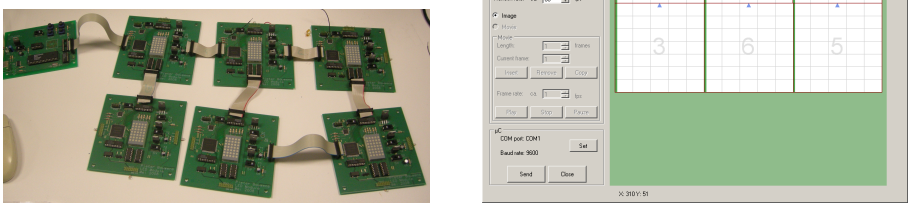
Figure 8.9 – Display was turned on looking like (a), the GUI (b) shows the corresponding representation

right module. Figure 8.10b tells us that this is the parent node of the lower right module, so the tree will have to be rebuilt. After the polling (which happens once per second) the GUI shows Figure 8.11b. You can see that the concerned module has changed parent node.

The initialization time (and polling time) for this driver will be highly dependent on both the number of modules *and* the exact display configuration. Using the free-form modular display driver, a lot of things could happen in parallel during the shout routine. A node could shout its address at the same time some ancestor nodes shout theirs. So there were only small variations on the initialization time. With this driver, this is not always the case. Every module needs to establish a connection with the microcontroller separately (send routine) and only *after* that, an address is received so the possible child nodes can start asking for addresses. In the previous driver, the addresses could still be distributed in a tree branch even if the shout routine was halted there.

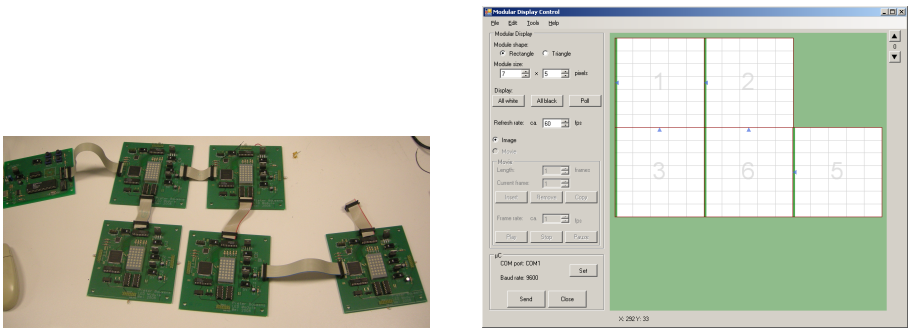
The worst possible scenario is when all modules are connected in one long chain. Nothing can happen in parallel. Every module has to wait until its parent has gone through the send routine. And this send routine will progressively grow longer and longer since there will always be an extra `send_req`/`send_ack` to be sent. The best case scenario is when we have a full tree structure, where every module has multiple child nodes. In this case, some things can happen in parallel. A lot of `send_req`s can already be sent while the microcontroller is communicating with another module. Luckily for us, a random display configuration will be closer to the best case scenario. A module will always connect to the first module that is ready, so a more evenly distributed tree will be created.

Figure 8.12a shows the simulation results for the initialization times (in μs) for the best and worst case scenarios. The data rate is 1Mbit/s. Figure 8.12b shows



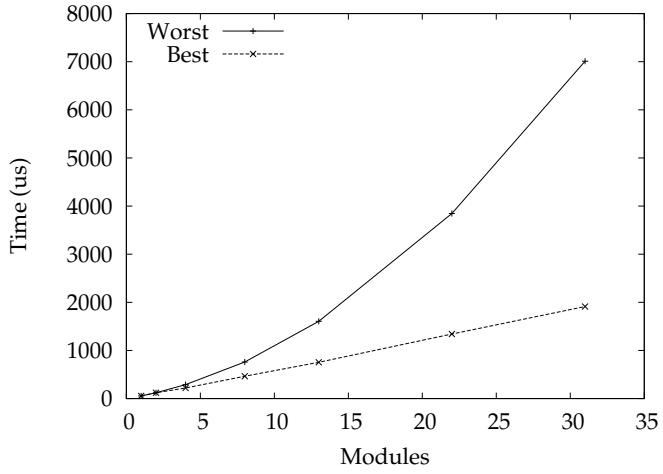
(a) (b)

Figure 8.10 – After adding a module (a), the GUI (b) shows the corresponding representation

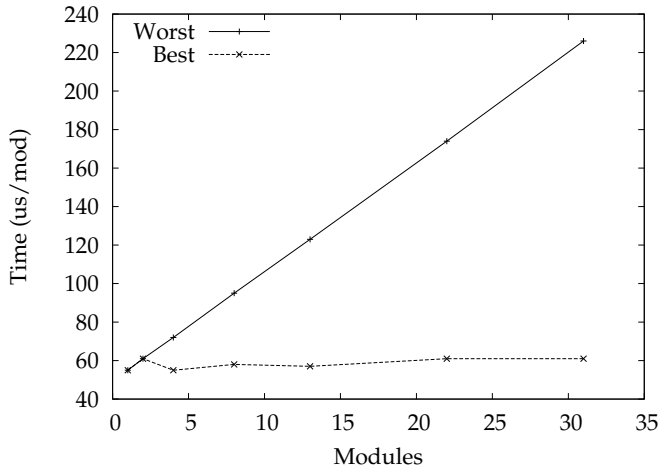


(a) (b)

Figure 8.11 – After removing a parent node (a), the GUI (b) shows the corresponding representation



(a)



(b)

Figure 8.12 – Simulation results of the initialization times (a) (in μs) for the best and worst case scenarios. (b) shows the average initialization time per module (in μs.)

# modules	Best case (ms)	Worst case (ms)
4	0.24	0.29
8	0.49	0.76
16	0.97	2.25
32	1.95	7.42
64	3.89	26.52
128	7.79	99.72
255	15.52	383.27

Table 8.4 – Extrapolation of the simulation results

how that reflects on the average initialization time per module.

In the worst case scenario, every module has to wait for the previous one to be finished. Therefor, the average initialization time per module is linearly dependent on the total amount of modules. In the best case scenario, the send routine can happen completely in parallel. While the microcontroller communicates with one module, the other modules have the time to partially set up the communication line. Also, the amount of `send_ack`s to be sent is limited to the amount of levels in the tree. The initialization time per module is almost constant (about 60μs). This time will still slowly rise, because the amount of levels is increased with the amount of modules. But the extra `send_ack` time will be divided among the total amount of modules in that level, and even with a full display of 255 modules, there are only 6 levels. The problem with this is that, when extrapolating the data from Figure 8.12, the worst case scenario provides very long initialization times for very large displays. Table 8.4 gives an overview. While a display of 255 modules with the best case scenario needs about 16ms to initialize, the worst case scenario needs 383ms. During the initialization this is not really a problem, but the same thing occurs during the poll. The time needed will be about the same (375ms), so if you want one or two polls per second, there is not much time left to send the actual image data. Luckily the used data rate is not yet very high. If you want to use a large display, the data rate will have to be augmented a lot.

8.7 This is the end, isn't it?

Yes. With this improved free-form modular display driver you can connect the modules any way you like, add and remove modules when you like. They will find a way to get an address and the configuration will be detected. The ability to use other shapes of modules gives a lot of additional freedom. There probably isn't a lot that can be added that could create more freedom in the display shapes. Some additions could make the system itself somewhat more user friendly, though.

Maybe the modules could communicate wireless with each other. But we would have to make sure that the different gates don't interfere which each other. This could also complicate the configuration detection. Or maybe the modules are connected to each other with wires, but there is a wireless connection to the microcontroller. The building of the tree will need to be revised, there will have to be a way to determine the 'oldest ancestor'. Another interesting option would be to be able to determine the physical orientation of the module (with respect to gravity), to better represent the display configuration. This would require some extra components on the modules, such as accelerometers, gyroscopes and magnetometers if we want to incorporate the earth's magnetic field in the calculation. We could also try to implement some sort of collision detection in the drivers (collisions on the data line, that is, not physical collisions). This would enable several modules being added at the same time, in different places, after the initialization. But most of this added functionality would probably be outweighed by the extra cost of providing them.

References

[1] P. J. Ashenden, *The VHDL Cookbook*. University of Adelaide, 1990.

In theory, there is no difference
between theory and practice.
But in practice, there is.

Yogi Berra (1925 -)

9 Design and Layout of the FrIIDoM Driver

9.1 Introduction

After implementing and rigorously testing the designs, it is now time to poor them into an ASIC. The aim is to create a device that can drive a module without any other components. This is not possible with an FPGA. It at least needs a programming device because of its volatility. Above that, the outputs of an FPGA cannot be used to drive a LED display. In itself there is no reason not to use a different display technology, but keeping in mind that we want to create our own modules, a LED display is preferred because of its ease in creating and driving. Section 9.2 gives the overview of the driver. The subsequent sections will each elaborate on one specific aspect. Section 9.6 will also provide some insight on the VHDL-to-ASIC design flow.

9.2 Four drivers in one chip

'FrIIDoM' is an acronym that stands for, with some poetic license, Four really Interesting Intelligent Drivers for Displays out of Modules. As the name implies, the four (really interesting) drivers from previous chapters are integrated in this one chip. The schematics and layout of the FrIIDoM driver are shown

in Figure 9.1 and Figure 9.2 respectively. The layout is about $12.5mm^2$ in size. The design was made in the C35 CMOS technology from AMS. The reason for this choice is the simple fact that, for this technology, there were very good tools available for the synthesis of VHDL to layout (See 9.6). The four distinct blocks are the four drivers. On the top, there are the eight current sources, on the bottom (and one on the side) are the eight row switches (see below). The extra logic, clock generator and Power-On Reset reside between the two left drivers.

Since we want to be able to test every driver separately, the drivers are completely independent and can be selected using the selection inputs on the chip (SEL(0:1)). As was clear from previous chapters, the drivers differ strongly in complexity. This is reflected in the size of the drivers. The small ones are the first and improved modular display driver (top left and bottom right corner respectively). In the bottom left and top right corner lie the regular and improved free-form modular display driver.

There is only one set of inputs and outputs for all four drivers. Multiplexers are used to guide the inputs to the selected driver. Same goes for the outputs. To be able to test and check as much as possible, a lot of extra signals are outputted: the control signals for the row and column electrodes, the state of the state machine in *Main Control* and the two last parameter bytes (the first two bytes are used in each driver internally). The (de)multiplexers that accomplish this are placed in between the drivers.

At that same location there is also an on-chip clock generator, providing the selected driver with a 20MHz or 10MHz clock signal. Close to it lies the Power-On-Reset, which generates a reset signal for all drivers when the chip is turned on. This makes sure that all registers are set to a known state.

To be able to test each digital driver for structural errors, scan chains are inserted (SE, SI, SO). See Section 9.6.1 for more information.

Last but not least there is the part that will drive the actual LED display. As you remember from Figure 5.18 in Chapter 5, we need two things. Current sources at column electrodes and switches at the row electrodes. The current sources can be found at the very top of the FrIIDoM driver, while the switches are found at the bottom. Each current source is designed to source up to 50mA, so the switches need to withstand 400mA. To minimize the influence on each other, the analog and digital parts have separate supply and ground lines.

The input and output pads are predefined AMS cells. All input pads have the needed pull down resistor. With the FPGA design, these had to be added externally. The digital output pads are able to source 16mA for signals that have to leave the board (communication signals), and 8mA for signals that stay on the board (test signals). The analog input/output pads are able to withstand only 100mA. This is enough for the current source outputs, but for each switch, several pads need to be combined.

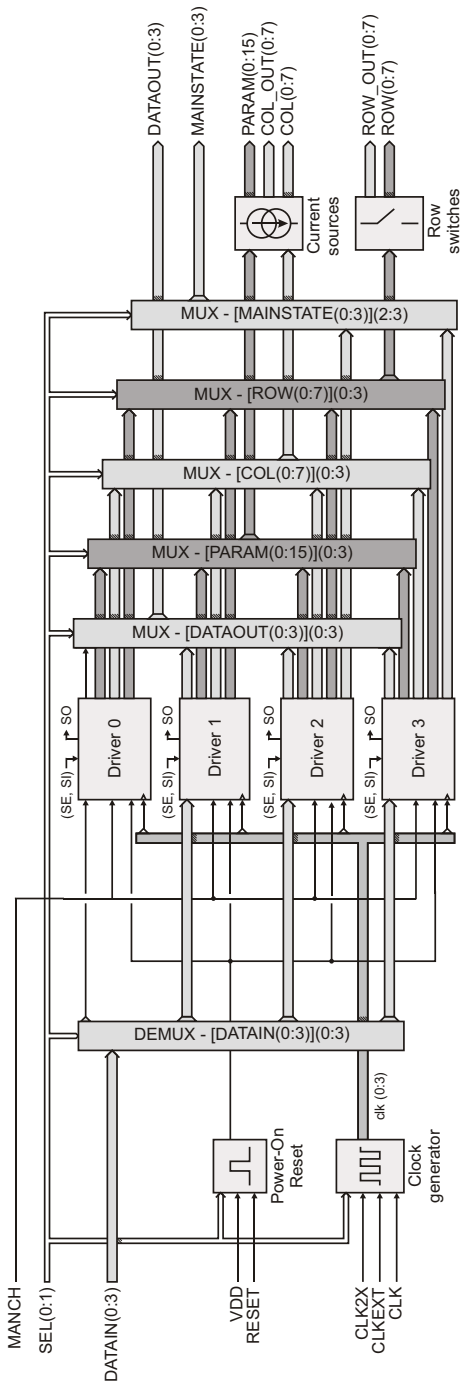


Figure 9.1 – Schematic of the FrIIDoM driver

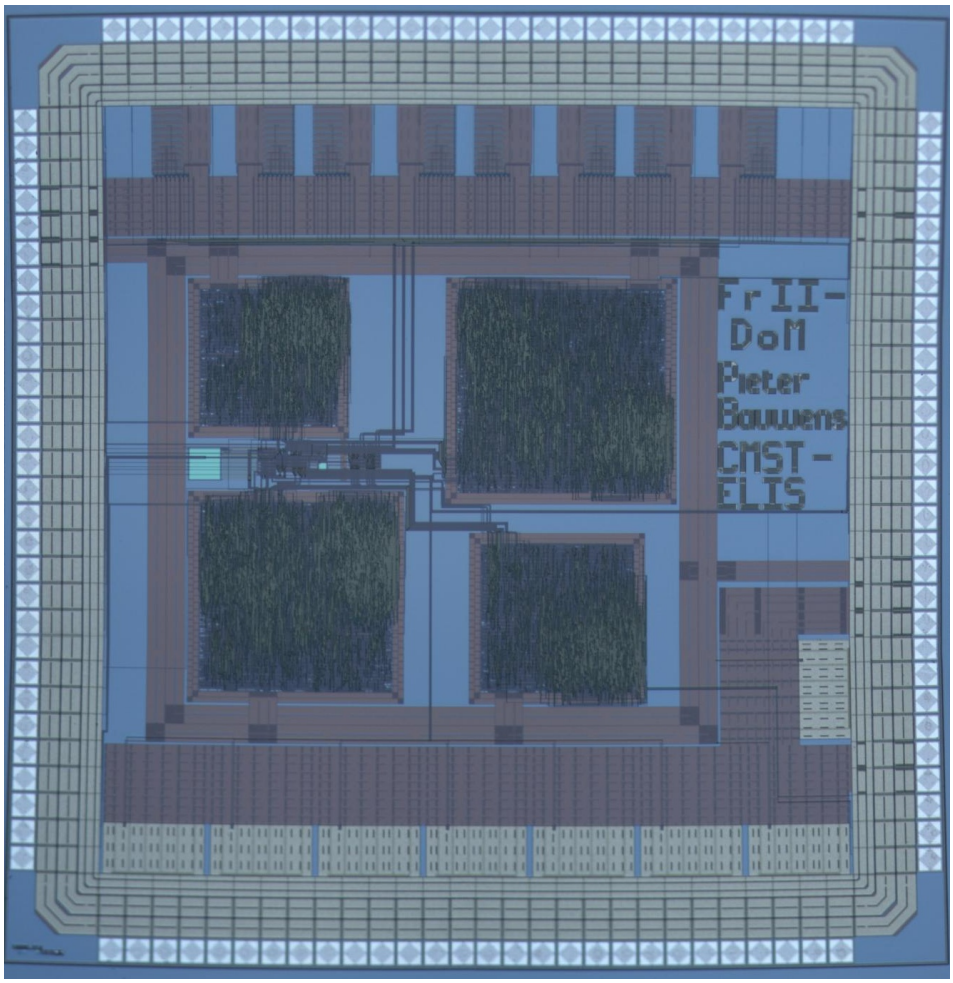


Figure 9.2 – The layout of the FrIIDoM driver. The four distinct blocks are the four drivers (left to right, top to bottom: driver 0, driver 3, driver 2, driver 1). On top and at the bottom you can see the 8 current sources and 8 switches respectively.

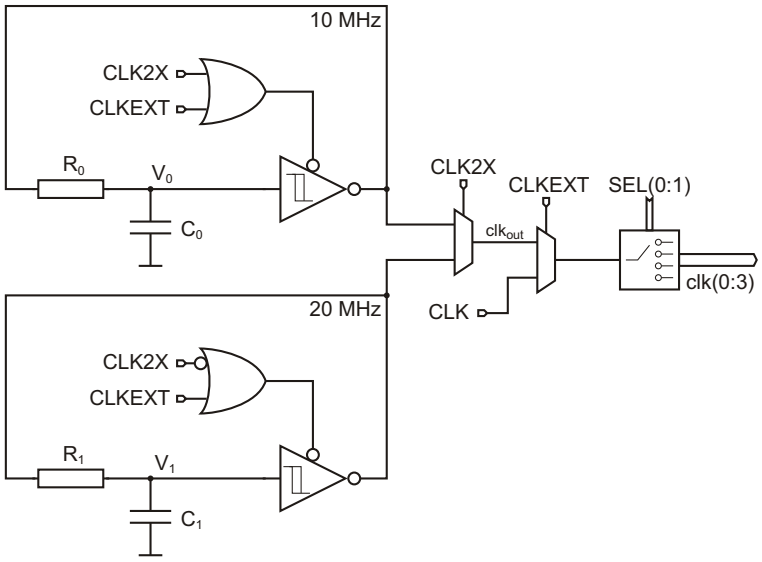


Figure 9.3 – Schematic of the clock generator

9.3 On-chip clock generator

The schematic of the clock generator is shown in Figure 9.3. There are some options that can be set. The most important option is the used clock frequency. As with the FPGA implementation, the driver is designed to work with a clock of 20MHz. However, when Manchester code is used during communication the needed bandwidth doubles (See Chapter 5). This is not so much of a problem for the driver itself, but the used microcontroller would have problems with the increased bandwidth. For this reason, the clock frequency can be lowered to 10MHz by pulling the CLK2X pin of the chip low. This way, the microcontroller can operate at the same speed when Manchester code is used. Since size wasn't really

Device	Parameters
R_0	100kΩ
R_1	97kΩ
C_0	740fF
C_1	345fF

Table 9.1 – Component parameters of the clock generator

an issue (the final size of the chip is entirely determined by the drivers, current sources and switches), I opted to create two independent clock generators so they

could be fine tuned for each frequency separately. But even then, there can be still a lot of variance on the produced clock frequency. The design parameter datasheet of AMS tells us that the final frequency can vary up to about 24%. This is the reason why Manchester coding is introduced.

At the heart of each generator resides a Schmitt-trigger. The schematics are shown in Figure 9.4 and Table 9.2 specifies the used transistor dimensions. With these dimensions, the threshold voltages of the Schmitt trigger to 2V and 1V. The tristate buffer is a standard AMS C35 cell (dimensions of the output stage are mentioned in Table 9.2). Transistor $T_{n,3}$ will make sure that, with a high EN signal, the Schmitt trigger produces a low output.

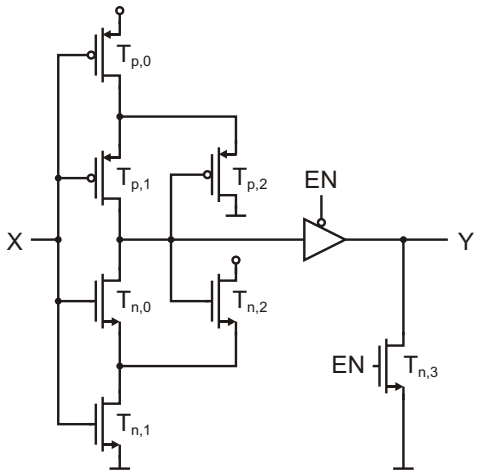


Figure 9.4 – Schematic of the Schmitt trigger

Device	Parameters
$T_{p,(0,1,2)}$	$W = 0.8\mu$m,$L = 0.35\mu$m
$T_{n,(0,1,2)}$	$W = 0.5\mu$m,$L = 0.35\mu$m
$T_{n,3}$	$W = 1.0\mu$m,$L = 0.35\mu$m
$T_{p,buf}$	$W = 3.2\mu$m,$L = 0.35\mu$m
$T_{n,buf}$	$W = 2.0\mu$m,$L = 0.35\mu$m

Table 9.2 – Component parameters of the Schmitt trigger

If the clock frequency would deviate too much from the desired frequency, and even the Manchester coding can't help out, there is also an option to use an external clock. Each driver has it's own clock and will only be clocked when selected. Figure 9.5 shows some simulation results. The CLK2X signal high starting from 1μs.

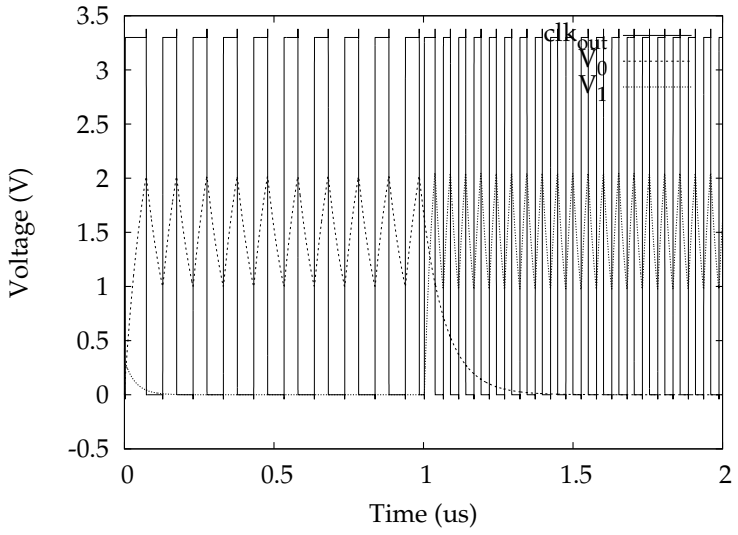


Figure 9.5 – Simulation results of the clock generator.

9.4 Power-On Reset

The POR circuit generates a reset pulse (active high) when the chip is turned on. This makes sure that the drivers are in a known state at start-up. The POR should work irrespective to the slew rate of the rising supply voltage. Figure 9.6 shows the used schematic.

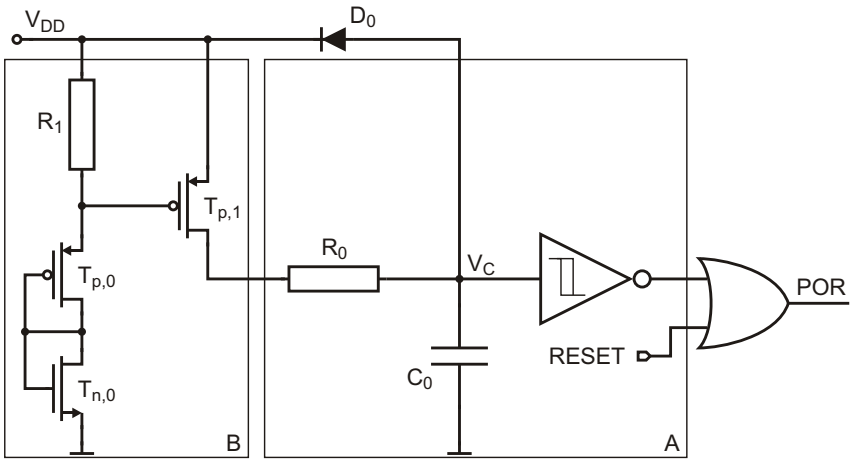


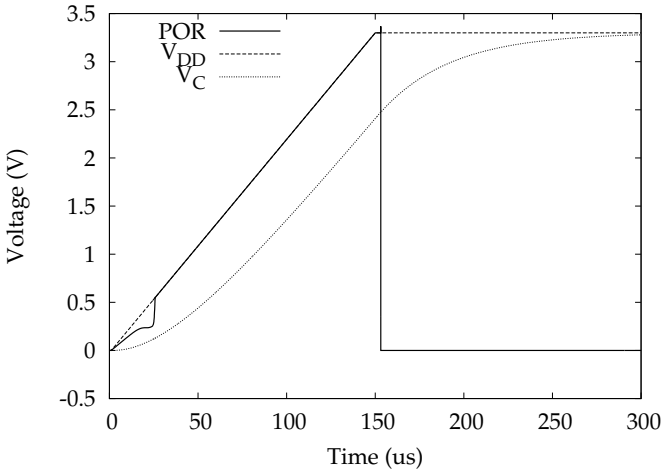
Figure 9.6 – Schematic of the POR

Device	Parameters
$T_{p,0}$	$W = 0.8\mu$m, $L = 1.0\mu$m
$T_{n,0}$	$W = 0.5\mu$m, $L = 1.0\mu$m
$T_{p,1}$	$W = 0.8\mu$m, $L = 0.35\mu$m
R_0	3.2MΩ
R_1	250kΩ
C_0	12pF

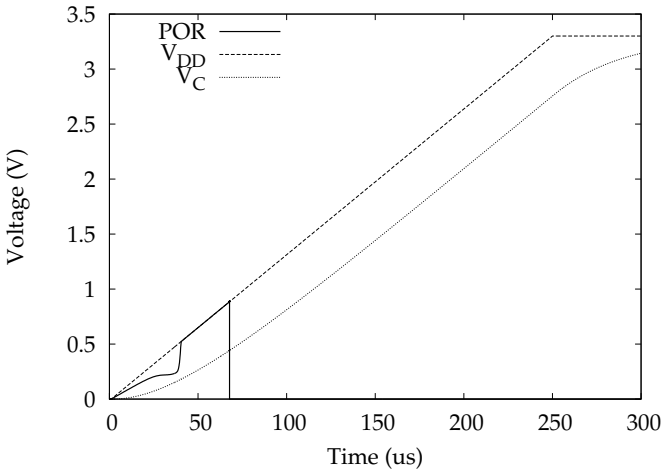
Table 9.3 – Component parameters of the POR

Part A makes sure that a pulse of at least 60μs is generated if the supply voltage raises very quickly ($<10\mu$s/3.3V). For slower rising supply voltages, the pulse will be expanded. However, when the slew rate is too low ($>200\mu$s/3.3V), the voltage over the capacitor (V_C) rises too quickly in respect to the voltage supply. V_C already reaches the threshold voltage of the Schmitt-trigger at $V_{DD} = 650m$V so the POR isn't generated at all (See Figure 9.7).

Part B in Figure 9.3 takes care of that by 'delaying' the supply voltage for the capacitor. Transistor $T_{p,1}$ will only present the voltage supply to C when it has



(a)



(b)

Figure 9.7 – The POR isn't generated correctly if the supply voltage slew rate is too low. Rise time of (a) is 150μs, rise time of (b) is 250μs.

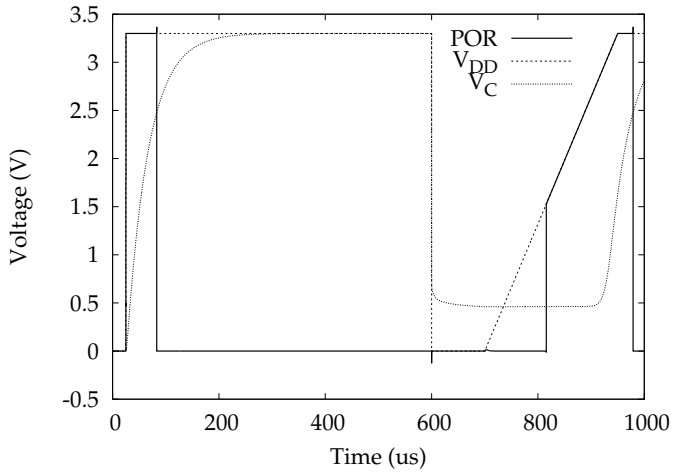


Figure 9.8 – With the extra circuitry, the POR is correctly generated for both fast and slow rising supplies.

already reached about 2.7V. This way V_C remains well below the threshold voltage of the Schmitt-trigger in the beginning, and a POR is generated. Part B has no effect on fast rising supplies. Figure 9.8 shows the generated reset as a result of a fast rising V_{DD} and a slow rising V_{DD} starting at 750μs.

Diode D_0 ensures that C can be quickly discharged when V_{DD} drops. Severe but short power glitches that could result in putting the device in an unknown state are responded with a reset pulse. The POR can also be activated externally by pulling the RESET pin of the device high.

9.5 LED drivers

As said in Section 9.2, we need current sources and switches to drive a LED display. The current sources will force current in the column electrodes according to the received image data. The switches connected to the row electrodes are used to select the row the data is meant for.

Since the driver is to be able to drive a whole range of LED displays, we made the current source adjustable. With 8 bits, sent with the parameter data, the current can be adjusted from 0 to 50mA. FrIIDoM is designed to drive a 8×8 display, so the switches at the row electrodes should be able to cope with currents up to

Device	Parameters
$T_{p,ref}$	$W = 8.0\mu$m, $L = 2.0\mu$m
$T_{p,0}$	$W = 8.0\mu$m, $L = 2.0\mu$m
$T_{p,1}$	$W = 16.0\mu$m, $L = 2.0\mu$m
$T_{p,2}$	$W = 32.0\mu$m, $L = 2.0\mu$m
$T_{p,3}$	$W = 64.0\mu$m, $L = 2.0\mu$m
$T_{p,4}$	$W = 128.0\mu$m, $L = 2.0\mu$m
$T_{p,5}$	$W = 256.0\mu$m, $L = 2.0\mu$m
$T_{p,6}$	$W = 512.0\mu$m, $L = 2.0\mu$m
$T_{p,7}$	$W = 1024.0\mu$m, $L = 2.0\mu$m
$T_{p,(8-23)}$	$W = 3.2\mu$m, $L = 0.35\mu$m
R	5.25kΩ

Table 9.4 – Component parameters of the current source

400mA.

9.5.1 8-bit adjustable current source

The schematic of the current source can be found in Figure 9.9. It is basically a group of current mirrors where the current through transistor $T_{p,ref}$ is mirrored to transistors $T_{p,(0-7)}$. Each column electrode has one of these. Assuming that the channel ratio $\frac{W}{L}$ of $T_{p,ref}$ is N, Table 9.4 tells us that $T_{p,(0-7)}$ have ratios N, $2N$, $4N$, ... $128N$. If the current through $T_{p,ref} = I_{ref} = \frac{V_{DD}-V_{th,pmos}}{R}$, the respective currents generated through $T_{P,(0-7)}$ will be I_{ref}, $2I_{ref}$, $4I_{ref}$, ... $128I_{ref}$.

The gates of $T_{p,(0-7)}$ can be controlled by the programming bits B_i, turning them on and off. This way we can create a programmable current I_{prog} given by Equation 9.1, ranging from $0I_{ref}$ to $255I_{ref} = I_{max}$. I_{prog} can of course be turned off according to the data bits for that particular column.

$$I_{prog} = I_{ref} \sum_{i=0}^{7} B_i 2^i = \frac{V_{DD} - V_{th,pmos}}{R} \sum_{i=0}^{7} B_i 2^i \qquad (9.1)$$

In our case, where we want I_{max} to be about 50mA, I_{ref} needs to be set to 196μA, resulting in the resistor value from Table 9.4.

The aim is to be able to drive different types of LEDs that require different driving currents. Since these different LEDs can (and will) have different threshold voltages, we don't want the driving current to change too much when an other LED is used. The threshold voltage of a LED affects the V_{DS} voltage of the driving transistors, which affects the current through *channel length modulation* [1]. With λ the channel-length modulation parameter, the drain current I_D (in active mode)

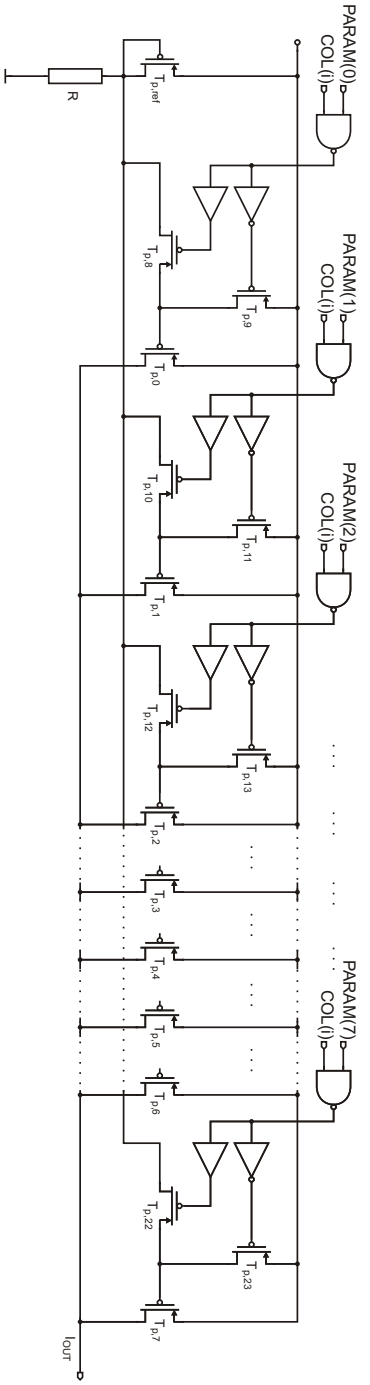


Figure 9.9 – Schematic of the current source

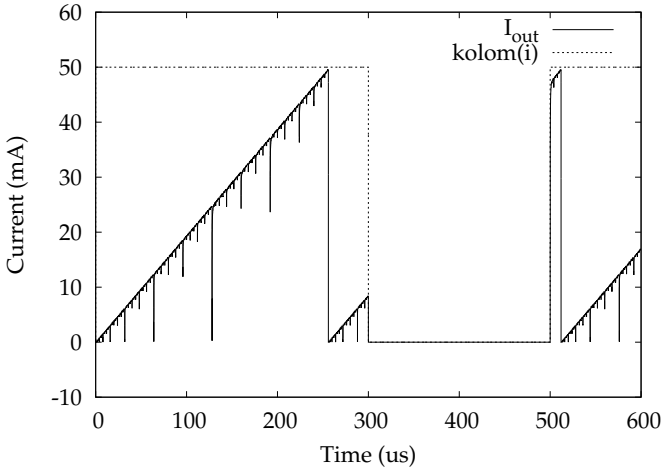


Figure 9.10 – Simulation results of the current source

is given by

$$I_D = K_n \frac{W}{L} (V_{GS} - V_{th})^2 (1 + \lambda V_{DS}) \quad (9.2)$$

K_n is a technology specific parameter and V_{GS} and V_{th} are the gate-source voltage and threshold voltage of the MOSFET respectively. λ is usually taken to be inversely proportional to the channel length L, so for short channels, it becomes difficult reaching the 50mA if the threshold voltage of the LEDs increases. For this reason, a channel length of 2μm was used instead of the minimal length. This way we can still reach about 45mA if the threshold voltage is 2V, where for a channel length of 1μm (with the same channel ratio $\frac{W}{L}$), I_{max} would already be limited to 37mA. Figure 9.10 depicts the simulation results. The current is increased gradually, showing a linear incline. The current is switched off when the data bit is zero. Speed is not really an issue here, because the current sources need to be turned on and off at the refresh rate times the number of rows, which is in the order of milliseconds. Another thing to take in consideration is how to layout this current source. We're dealing with fairly large transistors and these transistors need to be matched as good as possible. The layout can be seen in Figure 9.11. The transistors have a fingered structure with a base width of 8μm. $T_{p,ref}$ is found in the very center and the rest of the driving transistors have their fingers intertwined to provide a close matching. To further improve transistor matching, dummy transistors are placed on the sides, top and bottom[2].

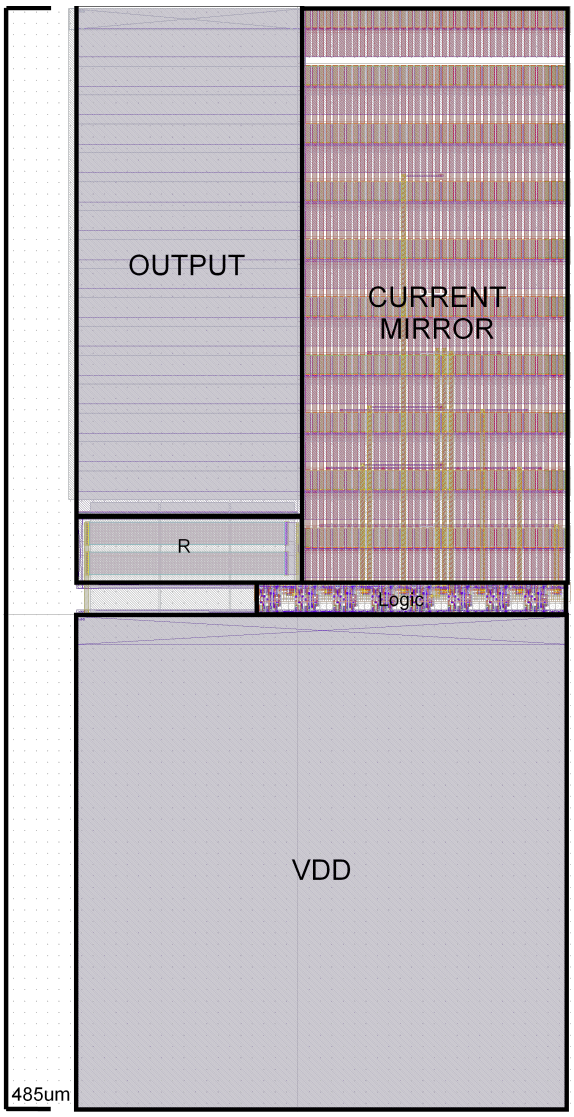


Figure 9.11 – Layout of the current source

The current towards the column electrode will be maximal 50mA. Since the bondpads can withstand up to 100mA, this poses no problem. We just have to limit the output resistance by making the metal path as wide as possible. This produces a series resistance of only 100mΩ [3]. If all current sources are working at full capacity, however, 400mA will be drawn from the voltage supply. We need 4 bondpads to provide this current. There are two supply bondpads on each side of the chip to minimize the average distance of the current sources to those pads (See Figure 9.1). The supply line itself is composed of two interconnected metal layers in parallel, resulting in an average resistance of only 85mΩ.

9.5.2 Switch

The switches at the row electrodes are just very large n-type MOSFETs, with some extra buffers to drive the large gate. An NMOS with a channel with a minimal length and a width of 1600μm is able to sink the needed 400mA [3]. As said, each bondpad can only withstand 100mA, so for each switch, four bondpads are needed. The connections to the bondpads are made as wide as possible and again two parallel metal layers are used to minimize the series resistance, reducing it to 20mΩ. Luckily, only one of the switches can be turned on at a time, so the ground line also only needs to guide 400mA (Even if multiple switches would be activated, there is only 400mA generated, so evidently only 400mA needs to flow away). The layout of the row switch is depicted in Figure 9.12.

9.6 From VHDL code to ASIC layout

A final task that needs to be done is to transform the VHDL code that was created in the previous four chapters, into an ASIC layout. Luckily for us, this does not have to happen manually. There are some powerful CAD tools to help us with that. Figure 9.13 gives an overview over the work flow. After the VHDL-code is tested and shown to be free of functional errors, it can be converted to a gate-level netlist (e.g. with Synopsys Design Vision). This is where the behavioral model in VHDL is translated to a schematic using basic logical elements (AND, OR, NOT, etc.) and memory elements. This is also the place where the scan chain will be inserted for the Design For Testability (DFT), which will be explained later on (See Section 9.6.1) [4]. Which basic blocks are available, which blocks that will be used and the properties of these blocks are technology dependent, so the generated netlist will only be applicable to that specific technology.

After the creation of the netlist, the blocks need to be placed and routed. AMS already provides full layouts of those blocks, so this can also be done without manual intervention (See Section 9.6.2) [5].

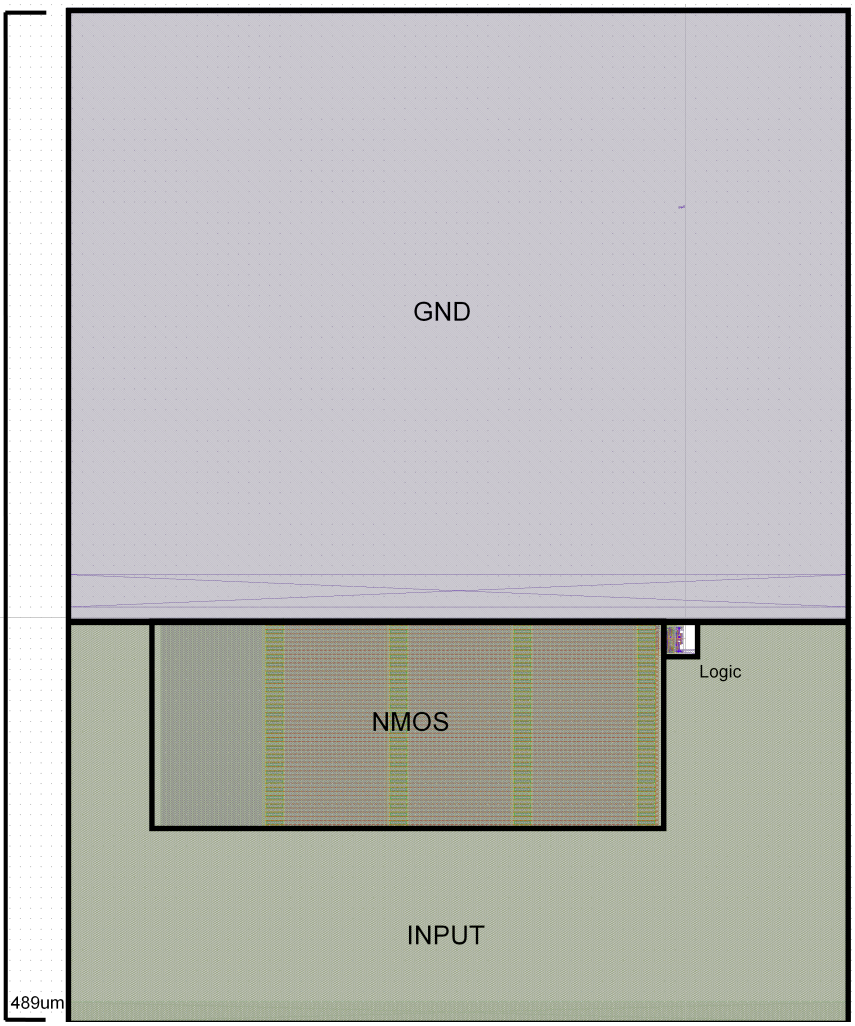


Figure 9.12 – Layout of the row switch

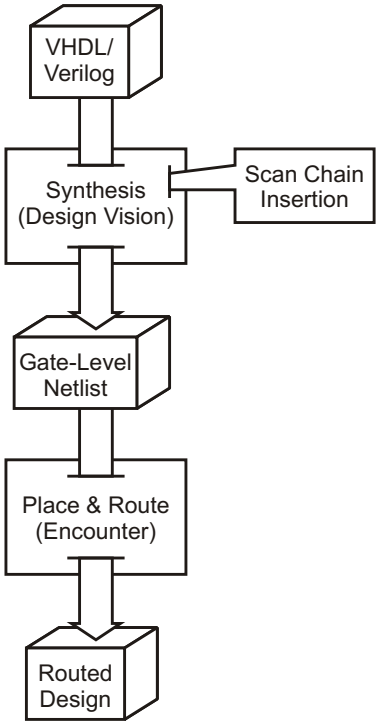


Figure 9.13 – The VHLD-to-ASIC workflow

9.6.1 Gate-Level Netlist and DFT

Besides functional errors, an ASIC can also fail because of manufacturing or structural errors, errors that occur during the processing of the layout mask or silicon wafer itself. For example the output of a flip-flop or AND gate is stuck to a fixed value. Notwithstanding the functional simulation was completely correct, this (obviously unforeseen) error will impede the eventual functionality. Of course it is always possible that a functional error slips through the cracks of the simulations, so it is important to know if the error is caused by a functional error (which means revising the code) or a structural error (which means revising the layout or using another ASIC). This is where DFT comes in.

DFT or Design For Testability is a design technique that adds certain testability features to the design [6]. The aim is not to determine if the functionality of the design is correct, but merely if the created circuit matches the gate-level netlist. Are all building blocks present? Are they functioning properly? Are they connected the way they should? The problem is that there are a massive amount of building blocks, heavily interconnected, and that the only test points available are the inputs and outputs of the chip. To make the nodes deep in the design more accessible, scan chains are inserted.

Scan chain insertion

To see how this works, take a look at Figure 9.14. During the creation of the gate-level netlist, there are memory (sequential) elements produced, connected to each other through a logic 'cloud'. We can make those sequential elements accessible by inserting a multiplexer in them. The input of the flip-flop can now be the output from the logic 'cloud' or the output of the previous flip-flop in the chain. If the scan mode is enabled, the flip-flops become one big shift register in which data sequence can be shifted.

There are several ways to insert a scan chain. If every flip-flop is connected to the chain, we have a full scan design, which makes it easier to generate the test patterns (See below). In a partial scan design, some of the flip-flops are not connected to the chain. A multiple scan design has several parallel scan chains [7].

Detecting the faults

Now how can this inserted scan chain help us to detect structural errors in the design? The basic procedure goes as follows (See also Figure 9.15). Shift a scan sequence in the scan chain (SI) to force the design into a known state (with the scan chain enabled (SE)). Then clock your system once (in normal mode, with the scan chain disabled). The scan chain is now filled with the results of the values of the previous state that went through the logic cloud. We can now shift the results

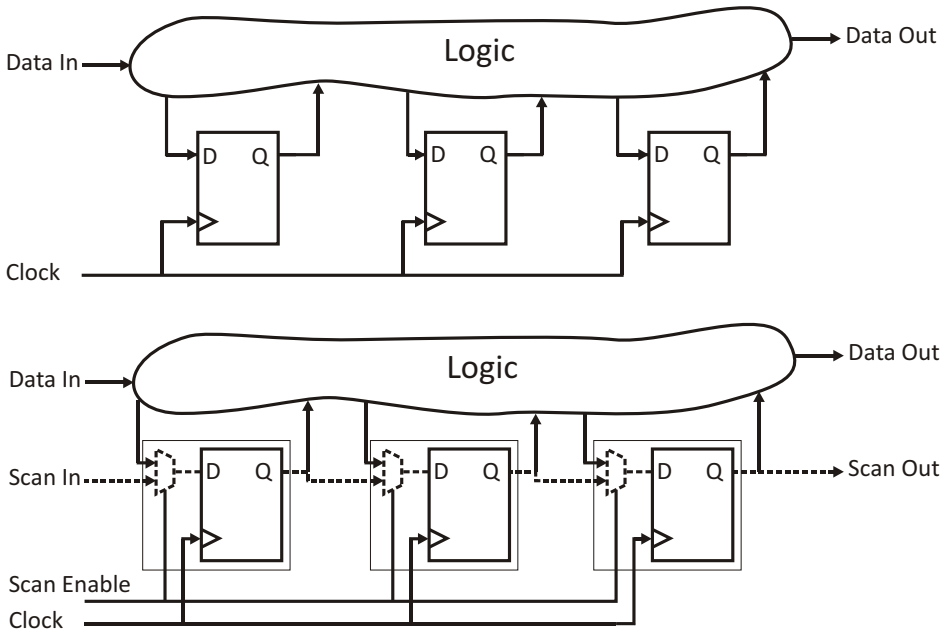


Figure 9.14 – Example of scan chain insertion. Flip-flops are replaced with flip-flops with an internal multiplexer.

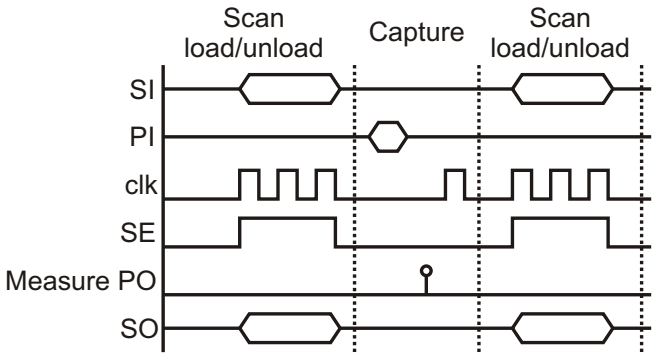


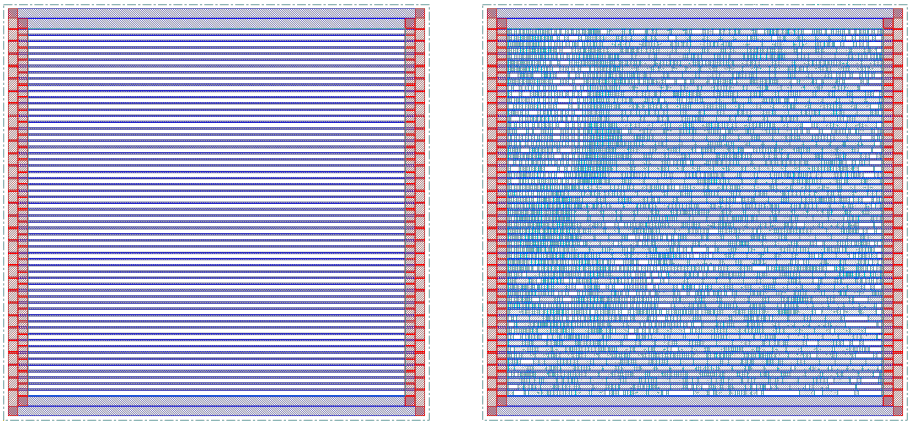
Figure 9.15 – A typical tester cycle in a full scan design.

out of the scan chain (SO) to look at them. To have more information from one scan sequence, there are also inputs (PI) forced before the clocking of the system, to see how the outputs (PO) change. In short:

1. Shift in scan sequence (*load/unload scan*)
2. Force inputs, measure outputs (*capture*)
3. Clock the system
4. Shift in next sequence while previous sequence is shifted out (*load/unload scan*)
5. Repeat from item 2.

The measured results can then be compared with the expected results, calculated from the netlist. Of course it is not enough to simply choose random test patterns for the scan sequence and input vector. If you want to be able to detect as much faults as possible, these patterns need to be intelligently created. This is done with an Automatic Test Pattern Generation (ATPG) tool (e.g. Synopsys TetraMax). This tool will create those patterns and calculate the fault coverage, the percentage of faults that can be detected using those test patterns. Some faults are intrinsically undetectable (e.g. if there is a fault on the enable of a tristate gate, the output of that gate is unknown), other faults are detectable, yet the ATPG tool hasn't found the pattern that could find it.

The tester cycle described above is only valid for full-scan designs. In partial-scan designs, where not all flip-flops are connected to the scan-chain, there are multiple *captures* needed per *load/unload scan* sequence. This allows the data to propagate to the non-scan elements. This *sequential ATPG* is a lot more complicated than the *combinational ATPG* described above.



(a) (b)

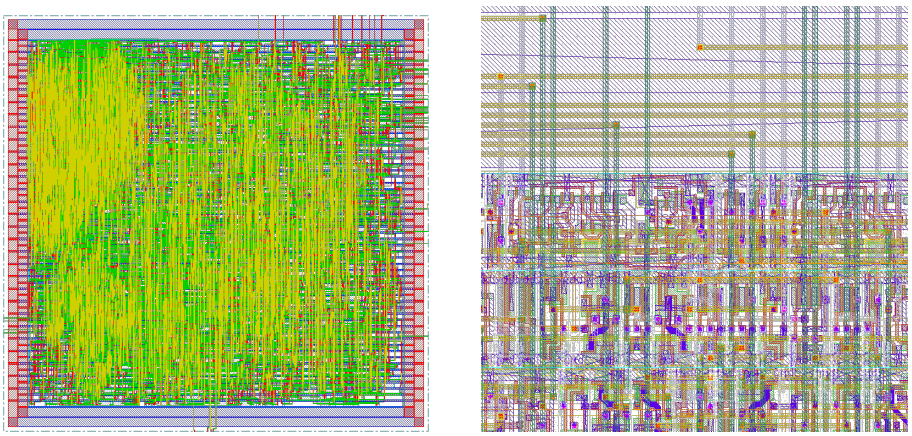
Figure 9.16 – (a) shows the empty floorplan with added power lines. (b) shows the floorplan after the standard cells are inserted

9.6.2 Place and Route

After the generated gate-level netlist is simulated again to check if none of the functionality got lost in translation, it can be read into a Place&Route tool to build the actual layout. The tool that was used here is SoC Encounter 7.1. Several options can be set. You start with an empty floorplan, the size of which is dependent on the read design and how 'dense' you plan to make your layout, how much of the floorplans surface will actually be used for functioning cells. In a highly interconnected design, you might want to choose a lesser density, to provide room for routing. Afterwards, the power lines are added (See Figure 9.16a). The standard cells of the AMS library all have the same height and fit perfectly between the alternating power and ground strips. The power strips can be terminated with capacitors on each side.

The standard cells can now be placed (See Figure 9.16b). The cells are already positioned with the routing in mind. The locations of the outputs of the chip are based on their driving cells, so if you want to give your outputs a fixed position, this needs to be done before the standard cells are placed.

When the cells are placed, the design can be routed. This happens in a couple of stages. First the clock tree is routed, after which a timing analysis is done. When the clock tree is fully optimized, the open spaces between functional cells are filled with filler cells. These cells have no functionality. They simply form a connection between matching layers (e.g. n-well) of the functioning cells. Now the design is ready for the final routing. See Figure 9.17a for the result and Fig-



(a) (b)

Figure 9.17 – (a) shows the layout after final routing. (b) shows a close-up.

ure 9.17b for a closup.

References

[1] D. Johns and K. Martin, *Analog Integrated Circuit Design*. Wiley, 1997.

[2] A. Hastings, *The Art of Analog Layout*. Pearson, 2004.

[3] *AMS C35 CMOS Process Parameters*, Austria Microsystems, 2008.

[4] A. Miczo, *Digital Logic Testing and Simulation*. Wiley, 2003.

[5] E. Brunvand, *Digital VLSI Chip Design with Cadence and Synopsys CAD Tools*. Pearson, 2010.

[6] Wikipedia. Design For Test. [Online]. Available: http://en.wikipedia.org/wiki/Design_For_Test

[7] Wikipedia. Scan chain. [Online]. Available: http://en.wikipedia.org/wiki/Scan_chain

[8] V. De Gezelle, "Design of a Switching xDSL Line Driver in a Submicron High Voltage Technology," Ph.D. dissertation, Ghent University, 2009.

[9] W. Hendrix, "Design of Low-Power High Voltage Driver Chips for Bi-Stable LCD's," Ph.D. dissertation, Ghent University, 2006.

[10] J. Rabaey, A. Chandrakasan, and B. Nikolic, *Digital Integrated Circuits, a Design Perspective*. Pearson, 2003.

A common mistake that people make when trying to design something completely foolproof is to underestimate the ingenuity of complete fools.

Douglas Adams (1952-2001)

10 Results and Applications

10.1 Introduction

Of course, designing the chip is not the end. We still have to check how the created driver operates in real life. In this chapter we will discuss the results of the measurements on the FrIIDoM driver. We start by elaborating on the test environment (Section 10.2), followed by the actual measurement results (Section 10.3). First we'll have a look at some general properties of the FrIIDoM driver, then we'll check each driver in detail.

In Section 10.5 we talk about some of the applications of the drivers. It also shows some of the first attempts in creating a flexible display.

10.2 Setting up the (final) test environment

10.2.1 Design of the test boards

A couple of different test boards have been created. Besides the traditional module and accompanying controller board, there is now also a specific test board. This test board will be used to test the current sources and row switches of the FrIIDoM driver. It can also be used to test the functionality of the chip and can, if necessary, be used to test the driver using the inserted scan chains.

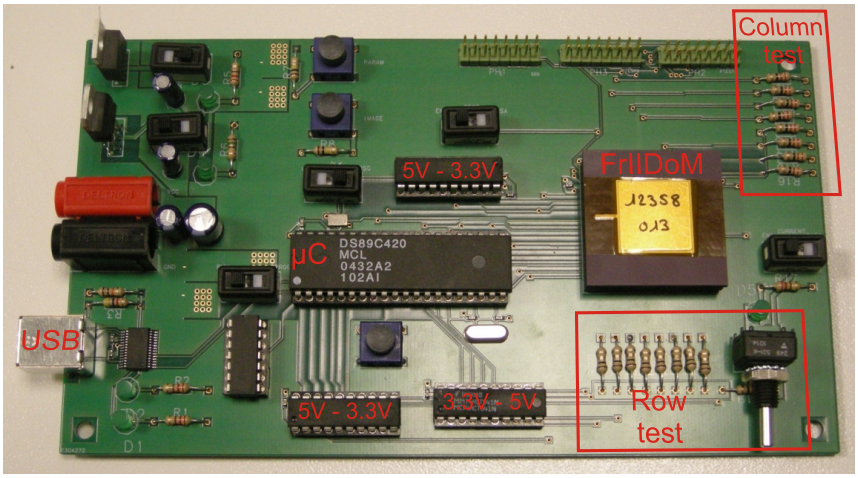


Figure 10.1 – FrIIDoM Test board

FrIIDoM Test board

The test board can be seen in Figure 10.1. It has room for both the Dallas DS89C420 microcontroller (or the newer, pin compatible DS89C450) and the FrIIDoM driver. The microcontroller can be programmed in system using the USB interface with the FTDI chip (FT232RL) which provides the translation to RS-232. Through this setup, the microcontroller can communicate with the PC.

Since the microcontroller works with 5V and FrIIDoM with 3.3V, we need the proper conversion. This is done with the MM74HC541 buffer which can be used with a wide variety of supply voltages (2-6V). Supplied with 3.3V, the 5V input from the microcontroller is outputted as a 3.3V signal. For the conversion from 3.3V to 5V, the MM74HCT541 buffer is used. This buffer can only be supplied with 5V, but the inputs are TTL compatible, meaning that the high level of the 3.3V devices is high enough to be used with this buffer.

Most of the control signals for the driver are directly controlled by the microcontroller: `SE(0:3)` to enable the scan chain, `CLKEXT` to select the external clock, `CLK2X` to select the clock frequency, `SEL(0:1)` to select the driver, `DATAIN(0:3)` to control all input gates, `RESET`, `MANCH` to enable Manchester coding and `SI`, the scan input of every scan chain. The external clock can be either a crystal oscillator (20MHz) or a microcontroller output, selectable by one of the switches on the board.

To test the current sources, each column output gets a small resistor (22Ω) to the ground. We do not want to take a resistor that is too large, where the voltage drop along the resistor would be too great, causing the internal current mirrors to stop

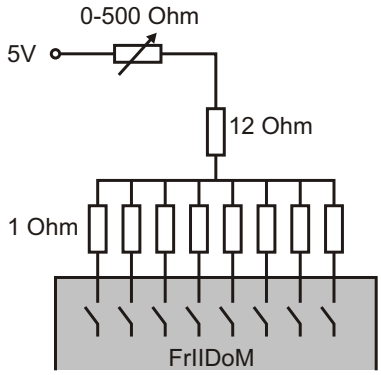


Figure 10.2 – Test structure for the row switches.

functioning properly. The voltage drop over the suitable LEDs will also be about 1-2 Volts.

The row switches need to be checked if they can bear the full 400mA. For this, the test structure in Figure 10.2 is used. The potentiometer, connected to the 5V supply line, has a resistance ranging from 0Ω to 500Ω. It is connected to a 12Ω resistor, which in its turn is connected to 8 parallel 1Ω resistors, one for each switch. In normal circumstances, only one switch will be activated at a time. If the potentiometer has full resistance, a current of 9.7mA will flow. With a zero resistance potentiometer, about 385mA will flow. The reason for doing it this way, instead of only using 8 12Ω resistors in parallel besides the potentiometer is because of the fact that if, for some reason, two switches would be active at the same time, the maximum current would become about 800mA. Since the internal ground line is calculated for only 400mA, this could cause some serious damage. Two switches can become active at the same time due to a functional error, or maybe during DFT testing. In any case, the current can also be completely disabled using one of the switches on board.

The pins on top provide easy access to the digital outputs of the FrIIDoM driver.

FrIIDoM Module

One of the LED module boards is shown in Figure 10.3. The main components are of course the FrIIDoM driver and the LED display. Again, to simplify assembly, the option was made to use a simple 5×7 LED display.

Switches allow to choose between the on chip and external clock (20MHz oscillator), and between the general reset (originating from the controller board, for all modules) and a local reset (push button). To connect or disconnect the module from the system, a power switch can be used, cutting the power to the module.

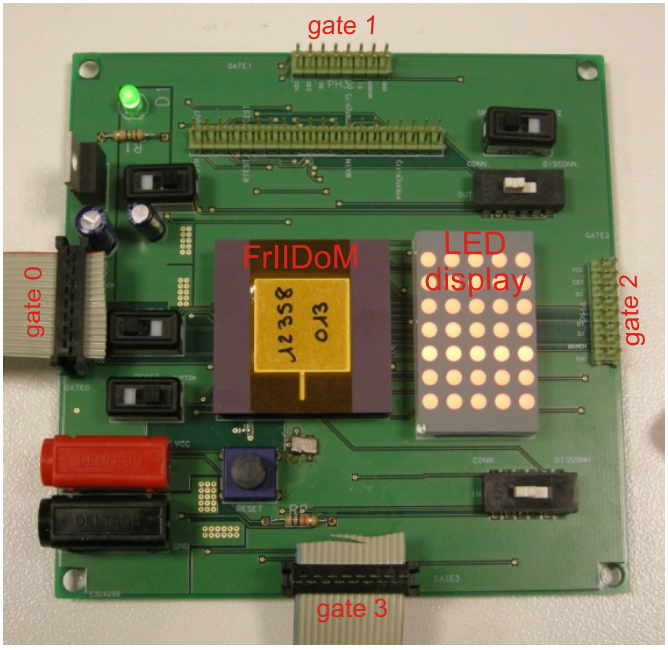


Figure 10.3 – A FrIIDoM LED module

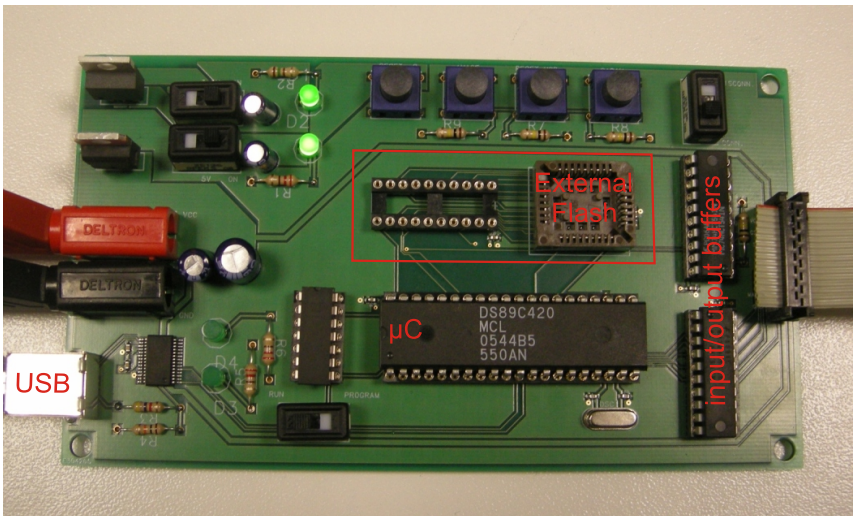


Figure 10.4 – The FrIIDoM controller

Another option is to use the switch that disconnects all outputs with one throw (idem ditto for the inputs). In some modules, the local reset push button has been replaced with a switch, to provide another way to deactivate the module. During a reset, all outputs are pulled low. The outputs and inputs are then connected to either the ground, or an open net (using the internal pull down resistors).

The connections towards the surrounding modules are configured in such a way that every side of a module can be connected to every side of another module without causing shorts. This way the full functionality of the last driver can also be tested.

FrIIDoM Controller

The controller board that will act as an interface between the modular system and the PC is printed in Figure 10.4. It doesn't differ much from the controller board described in Section 5.5.

Again the Dallas microcontroller (DS89C420/450) is used, programmable in system through USB, with external memory (AT29C010 with the DM74LS373N latch) available. As opposed to the controller board in Section 5.5, the 5V to 3.3V conversion (and back) happens on this board, with the same buffers as the FrIIDoM test board (MM74HC541 and MM74HCT541). Since these buffers don't have pull down resistors, a 10kΩ resistor is added at the DATAIN input.

Buttons allow for some input for the microcontroller and include a reset button for the entire display. A switch enables (disables) the buffers, which will connect

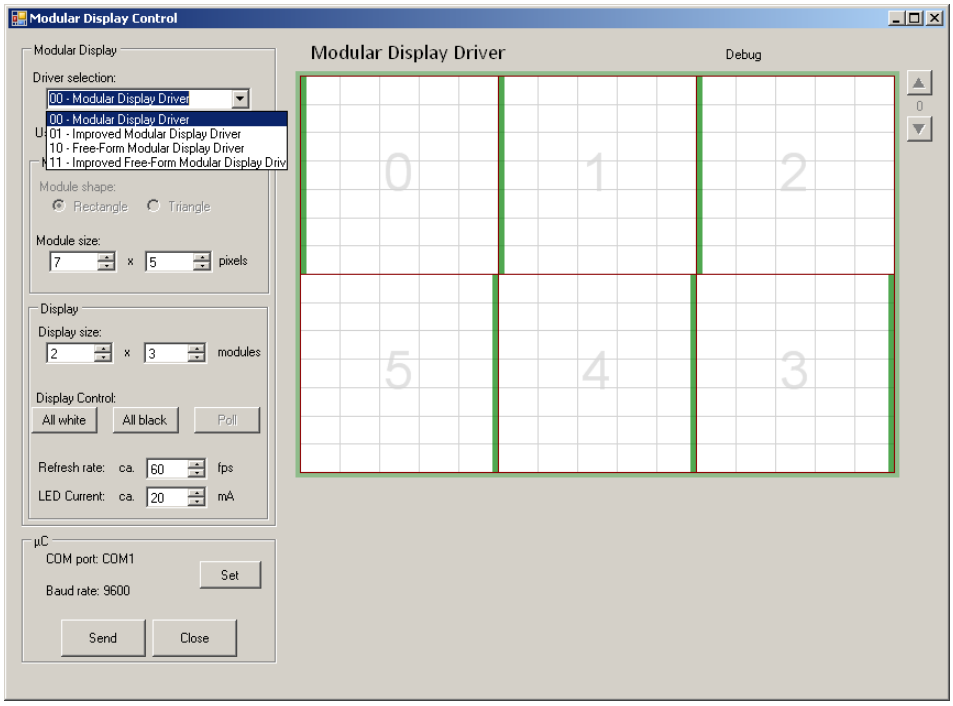


Figure 10.5 – GUI for the FrIIDoM driver.

(disconnect) the controller from the modular system.

10.2.2 Design of the GUI

Just a small note on the designed GUI, since it is basically the same program used during the test phase of the individual drivers. Figure 10.5 shows the interface.

The main difference is that now there is one program for the four drivers, instead of one for each individually. The same goes for the microcontroller code. The program allows to choose between the four drivers and will enable/disable some components according to the choice. For example the module shape (triangle vs. rectangle) will only become available when the Improved Free-From Modular Display Driver is selected. Further, the display size ((Improved) Modular Display Driver) and module size can be chosen. You can now also indicate that Manchester code has to be used. The available display parameters are the refresh rate and the desired LED current.

The green bars on the modules in the display area indicate how the modules are oriented (gate 0), which will only be of real importance using the final driver. The

location of the parent node and transitions to lower or higher levels will also be shown (See Section 8.5 for more details).

10.3 Measurement results

In the first few paragraphs, we'll have a look at the general properties of the FrIIDoM driver. This includes the performance of the current sources, the row switches and on-chip clocks. Since the Manchester (de)coder and Sequencer is the same in all four drivers, it will also be discussed here.

The last four paragraphs focus on the performance of the four drivers separately. Measurements were done using a Tektronix Logic Analyzer, designed to measure digital data. Instead of showing the exact voltage of the data signal, a logic analyzer will translate the signal to logic bit values to create a clear digital signal. Just as a reminder, data is sent with LSB first.

10.3.1 8-bit adjustable current sources

The 8-bit adjustable current sources were designed to generate up to 50mA (See Section 9.5). One byte of the parameter data sent to the modules controls the current source. They were tested on the test board by rapidly sending parameter data with an increasing current control byte. This should result in a linearly increasing current. As said in previous section, the currents were measured over a 22Ω resistor.

The results are shown in Figure 10.6. As you can see, the required 50mA is reached nicely. Note that the rate that the parameters are sent is much higher than the frame rate (and thus much higher than what would be needed for an actual display application). Figure 10.6 also shows the control signals for that current source. It is set to drive all but one lines high in an 8-row display. This shows how the current sources respond when being turned completely on or off.

Figure 10.7 shows how this affects the LED display. The LEDs are driven with 20mA, 10mA, 5mA and 2mA. The higher currents are not tested on those LEDs, because they can only withstand 25mA.

10.3.2 400 mA row switches

The switches were designed to be able to withstand a current up to 400mA ($8 \times$ 50mA) (Section 9.5). This was tested by controlling the current that flows through the switches with a potentiometer. This way a current range from about 10mA up to 370mA could be reached.

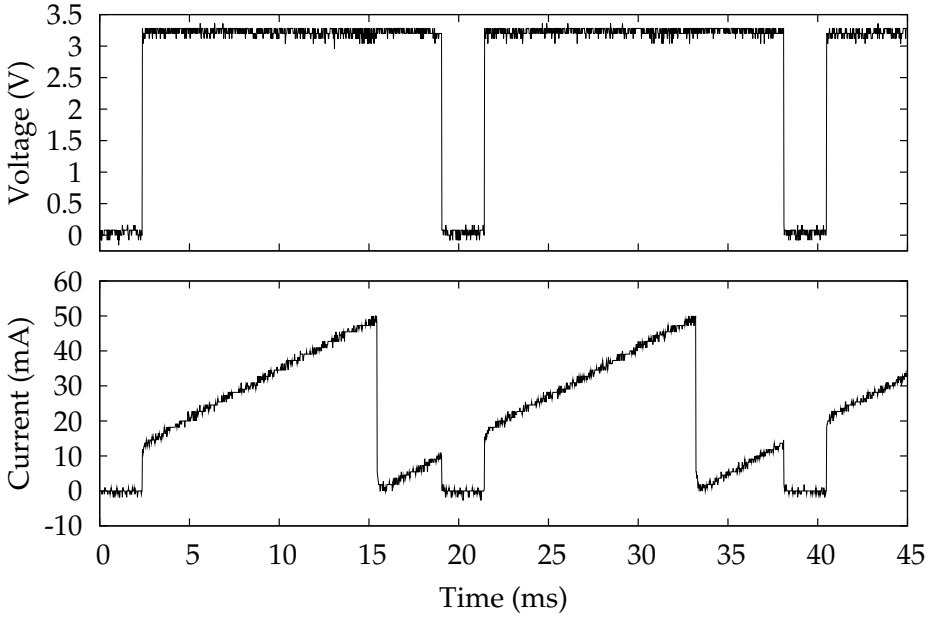
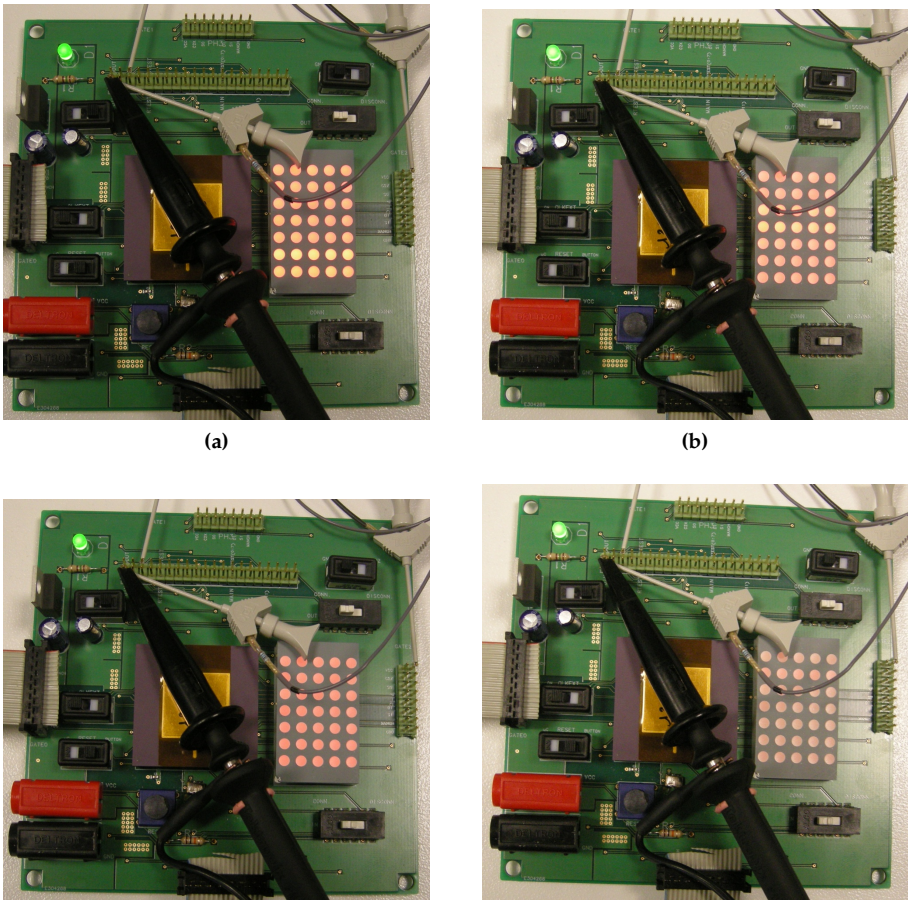


Figure 10.6 – The outputs of a current source with the corresponding control signal.



(a) (b)

(c) (d)

Figure 10.7 – A module with all LEDs on. LEDs are driven with 20mA (a), 10mA (b), 5mA (c) and 2mA (d).

Figure 10.8 shows the results of the measurements on the row switches. The potentiometer was set to create a current of about 100mA (Figure 10.8a), 200mA (Figure 10.8b) and the maximum 370mA (Figure 10.8c).

The switches (and the chip as a whole) don't seem to have much difficulty sinking the current.

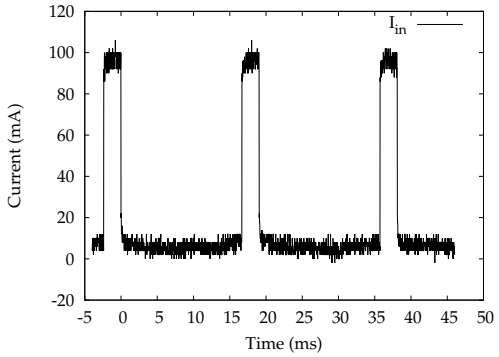
10.3.3 Clocks and Manchester (de)coding

There are two on-chip clocks, one is set to 10MHz, the other to 20MHz (See Section 9.3). The 20MHz clock was to be used during normal communication, the 10MHz clock could be used if the communication happened with Manchester coding. As said, Manchester coding doubles the required bandwidth which is too much for the used microcontroller. The clocks are not directly outputted, but we can measure them indirectly, though, by looking at the data outputs of the drivers.

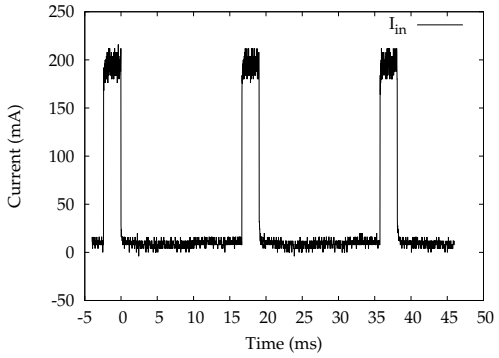
The 20MHz clock was measured by looking at the data output of the first modular display driver during initialization. This driver will send an 8-bit long address (plus one start bit) after receiving his own address (irrespective of whether this address was received correctly). So at a bit rate of 1MHz, we should see a 9μs long sequence. Every bit lasts 20 base clock cycles. The measured sequences where always slightly below 9μs, meaning that the designed frequency was slightly underestimated. Of the 20 available FrIIDoM drivers, the average frequency was 20.95 MHz with a standard deviation of 0.39MHz (See Figure 10.9 for the distribution).

The 10MHz clock was measured in a similar way. Since this clock is used when sending with Manchester coding, there are enough transitions in the data signal to derive the clock frequency. Again, the measured bit times where slightly smaller than expected, indicating an underestimation of the clock frequency. Over the 20 drivers, the average clock frequency was 10.80MHz with a standard deviation of 0.16MHz (See Figure 10.9 for the distribution).

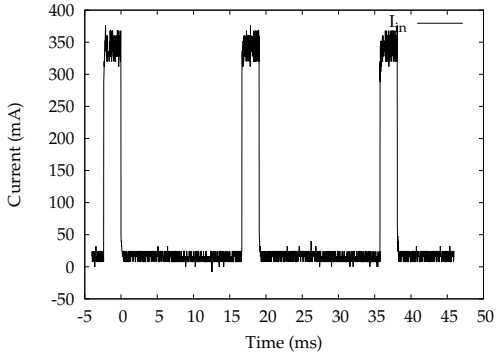
The 1MHz deviation on the 20MHz clock proves to be too large for regular communication. Using the first modular display driver, for example, the address might be received correctly, but the image and parameter data are scrambled. The initialization of the more complicated drivers fails altogether. Things improve when we invoke the Manchester coding. Notwithstanding there is also a 1MHz deviation on the 10MHz, it is possible to communicate. See Figure 10.10 for an example. It shows the initialization process of a display of two modules using the free-form modular display driver. To refresh your memory: using this driver, the first module asks an address to the microcontroller, which will send an address (`000 + 0x00`). Then the second module asks the first module for an address, which replies by sending `000 + 0x10`. After that the addresses are being shouted to the microcontroller.



(a)



(b)



(c)

Figure 10.8 – The performance of the row switches when having to sink 100mA (a), 200mA (b) and 370mA (c)

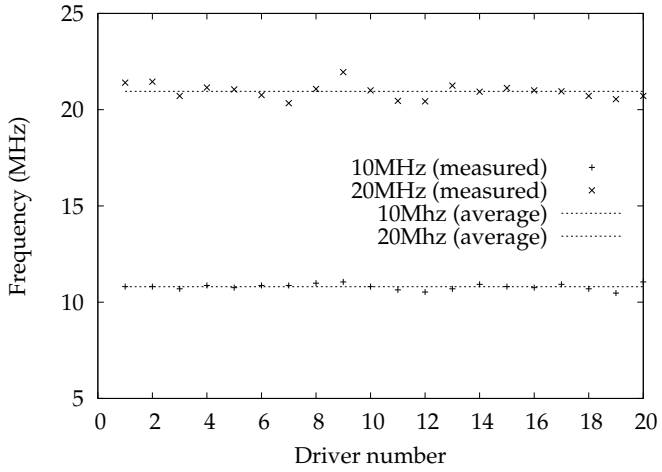


Figure 10.9 – Distribution of the on-chip clock. 20MHz and 10MHz

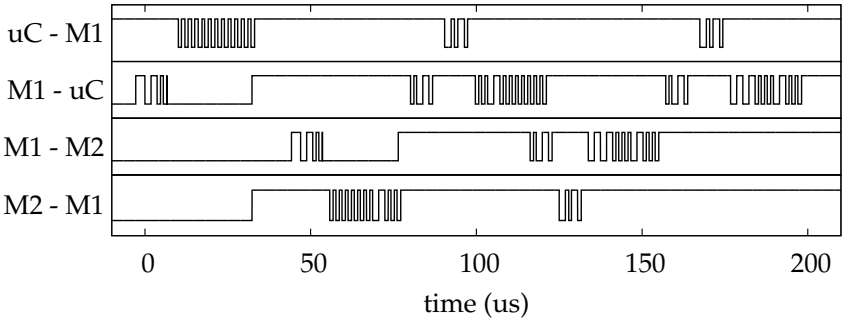


Figure 10.10 – The initialization process of a display of two modules with the free-form modular display driver using Manchester code.

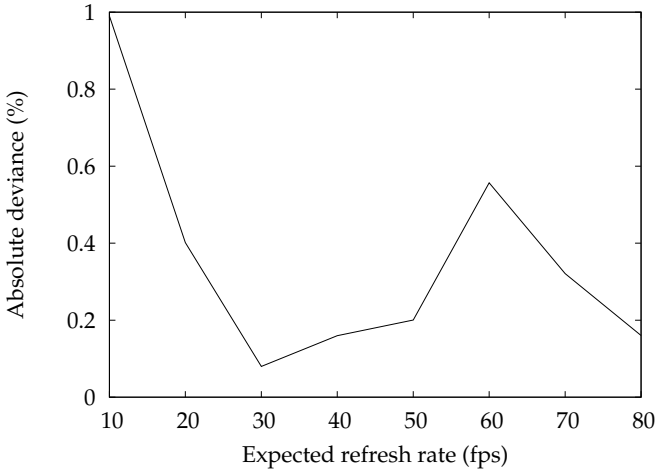


Figure 10.11 – Expected and measured refresh rates

Although communication using Manchester code works for most drivers, there still can be a lot of miscommunication, as well as between the drivers mutually as between the drivers and the microcontroller. For this reason, it was opted to use the external clock for the functional testing of the drivers.

10.3.4 Refresh rates

Since we're using the external clock from now on, the measured refresh rates should be very close to the desired refresh rates. Figure 10.11 confirms this assumption. In the GUI refresh rates of 10 fps, 20 fps, 30 fps, $\cdots$, 80 fps were requested and the resulting refresh rates were measured. These matched the desired rates with a precision of about 0.4%

10.3.5 Modular Display Driver

Using the first modular display driver, each module receives an address from the previous module and sends the next address to the following module. After the module sent the address, the internal bypass activates (See Chapter 5). The modules are connected in a bus network.

Figure 10.12a shows the measured signals of the initialization process of a sim-

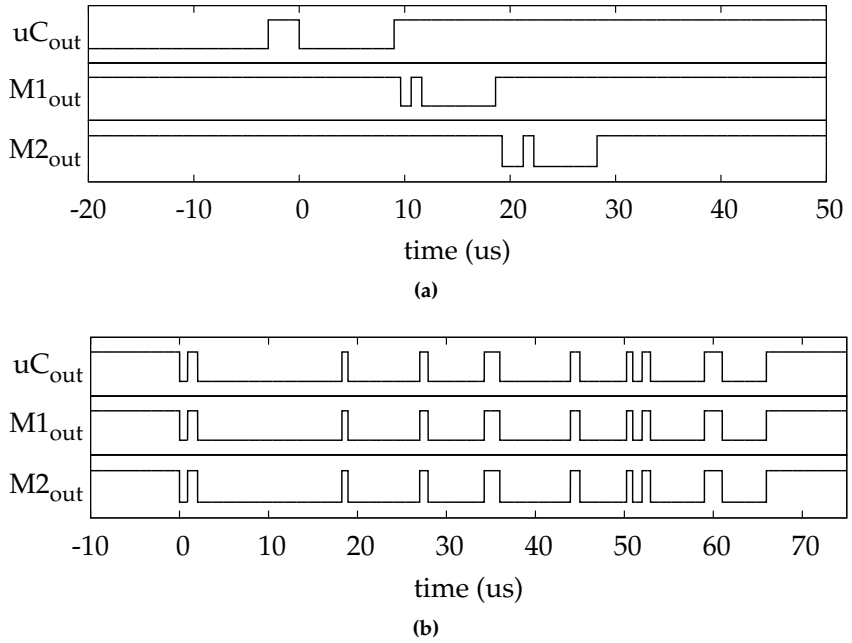
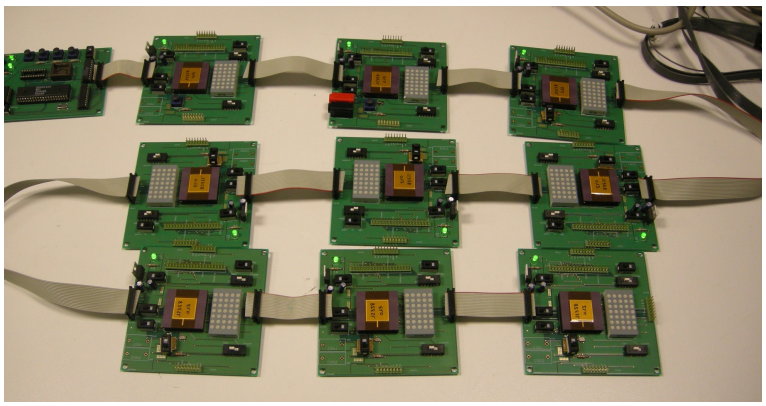


Figure 10.12 – The initialization process in a display of two modules, using the first modular display driver (a). After the initialization, the bypass is activated (b)

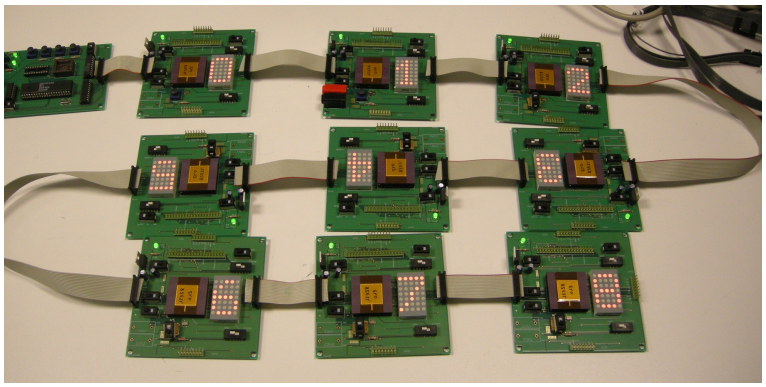
ple display of two modules. The microcontroller sends address `0x00`, module 1 sends `0x01` to module 2, which outputs address `0x02`. Figure 10.12b indicates that after the initialization, the bypass is activated. The data that is sent from the microcontroller, is immediately seen by both modules.

The driver was also tested on the display in Figure 10.13. The bus structure can be clearly seen. Also note that the modules in the middle row are upside down. Figure 10.13a shows the display before initialization. After the initialization is completed successfully, the microcontroller will already send default parameters (refresh rate: 60 fps, display size: 7×5, current: 20mA) and default data. In this case, the default data for each module will be the address of that module. So, after a successful initialization process, every module should display its own address. This is shown in Figure 10.13b.

We can now use the GUI to draw images and send it to the display. First the display size has to be selected (in this case it's a 3×3 display). The GUI will calculate which data belongs to which module and will take into account that some modules (the modules in the odd rows) are upside down. You can see some examples in Figure 10.14.

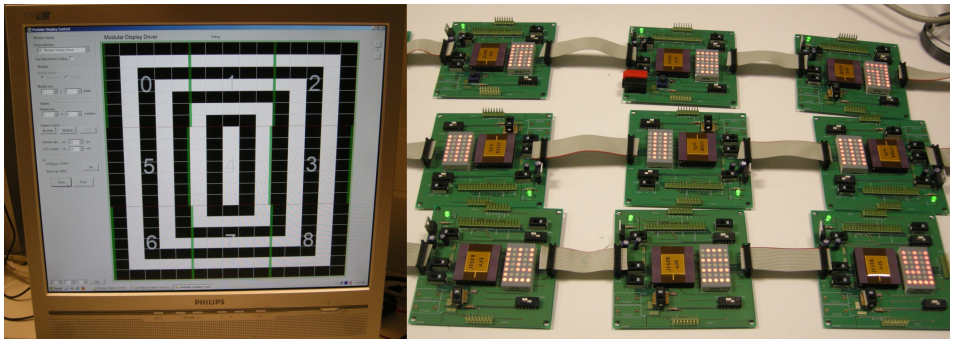


(a)

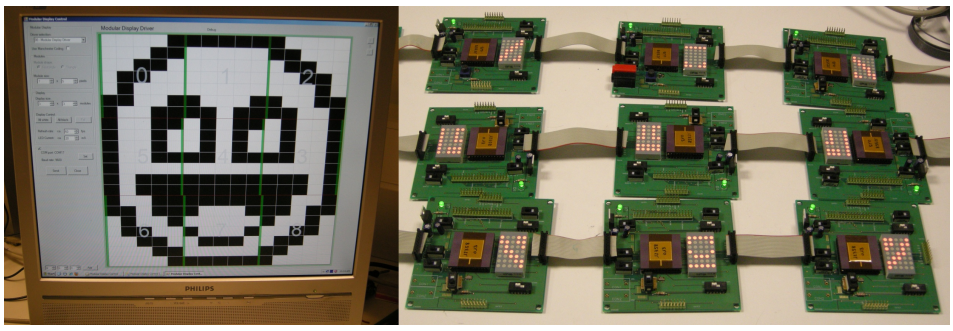


(b)

Figure 10.13 – A larger display with the first modular display driver. Before initialization (a) and after initialization (b).



(a)



(b)

Figure 10.14 – The image drawn in the GUI is represented correctly on the display.

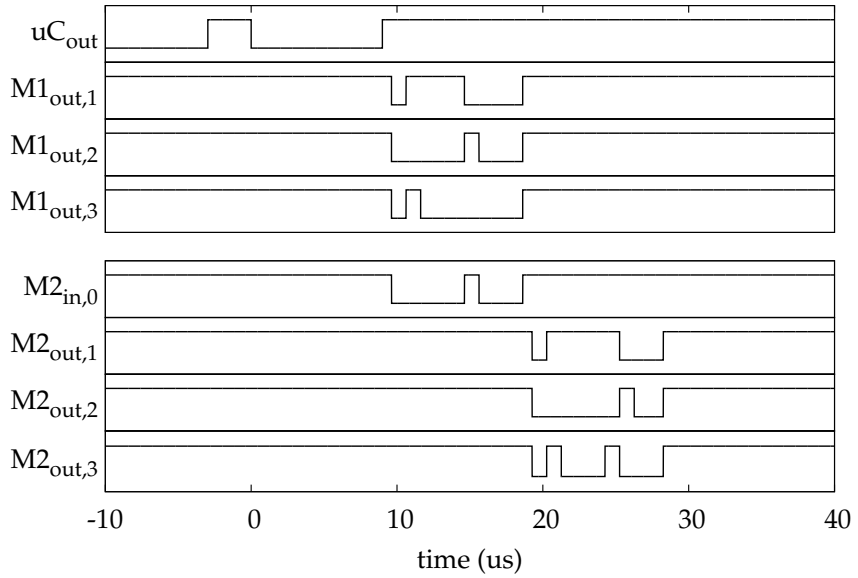


Figure 10.15 – The initialization process of a display of two modules using the improved modular display driver.

10.3.6 Improved Modular Display Driver

The improved modular display driver resembles the previous driver a lot. The modules are now ordered in a mesh network. Every module receives an address from another module, but instead of sending the next address to just one other module, it sends the corresponding addresses to the three other outputs (See Chapter 6). Addresses are made up of a row and column number: (row, column).

In Figure 10.15 you can see the measured initialization signals in a display of two modules (next to each other). The microcontroller sends address `0x00` to the first module. This will output address `0x0F` to the gate above (row -1, column), `0x10` to the right gate (row, column $+1$) and `0x01` to the gate below (row$+1$, column). Since the second module is located at the right gate, it will receive address `0x10` and send address `0x1F` upwards, `0x20` to the right and `0x11` downwards.

For a bigger example, you can look at Figure 10.16. The display before initialization (Figure 10.16a) shows the mesh network with possible data loops (well, one, in this case), and the fact that not every module needs to be present in this 3×4 display. After a successful initialization, the microcontroller will again send default parameter and data values. The default parameters are the same as before. After the default data is sent, the first three modules in the first column will dis-

play the numbers 0-2, the first three in the seconds column will display 3-6, etc. And only the numbers 0-9 are used. Figure 10.16b gives the result of a completed initialization on our display.

Again, the GUI was brought in to draw some images and display them. In this case, we have to select a 3×4 display. Not all modules are present in this display and both the GUI as the microcontroller are unaware of this. So they will both process data for modules that aren't there. But the modules that are there, will receive their data irrespective of the missing modules. Figure 10.17 shows a couple of examples.

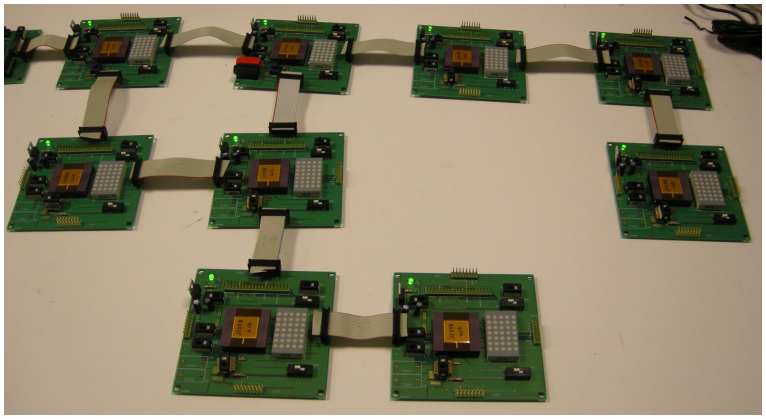
10.3.7 Free-Form Modular Display Driver

We proceed to the more complex drivers. Using the free-form modular display driver, every module will find another module from which it will request an address. When this address is received, it is ready to hand out addresses itself. The received addresses are shouted to the microcontroller according to the shout routine explained in Chapter 7. This way, the microcontroller knows which modules are present. Modules can be added and removed without interfering with the display operation.

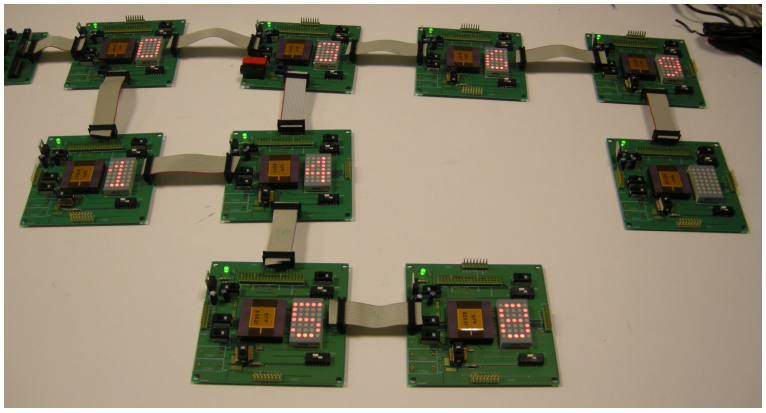
We can again look at an example of a small display of two modules (connected in the same way as before). The measured initialization signals are shown in Figure 10.18a. The microcontroller comes online (high output) and the first module asks an address. The second module will ask an address when the first module indicates it is ready (high output). Then the modules start the shout routine. The first module shouts its address to the microcontroller (request, permit, shout), the second module shouts the address to the first module (request, permit, shout end), which in its turn will shout it to the microcontroller (request, permit, shout end).

Figure 10.18b shows the similar procedure during polling. After the poll request from the microcontroller (seen by both modules at the same time), the shout routine is initiated and both addresses are correctly sent to the microcontroller.

Notwithstanding the modules can receive and remember their addresses correctly (during the polling, they respond with their given addresses), there seems to be a problem with comparing this address with the addresses sent in an image data stream. It appears that the modules have some issues recognizing their own address. The module with address `0x00` will also respond to data for module `0x01`, `0x10`, `0x02` and `0x20` for example (but not to address `0xF0`), while module `0x01` will not respond at all to its own address. These problems did not occur in the simulations (on behavioral VHDL code or on the generated gate-level netlist, see Section 9.6) or even in the FPGA implementations. Furthermore, the VHDL block that takes care of receiving image data and checking the addresses is the same in all four drivers, which seem to be working there. So it must be a

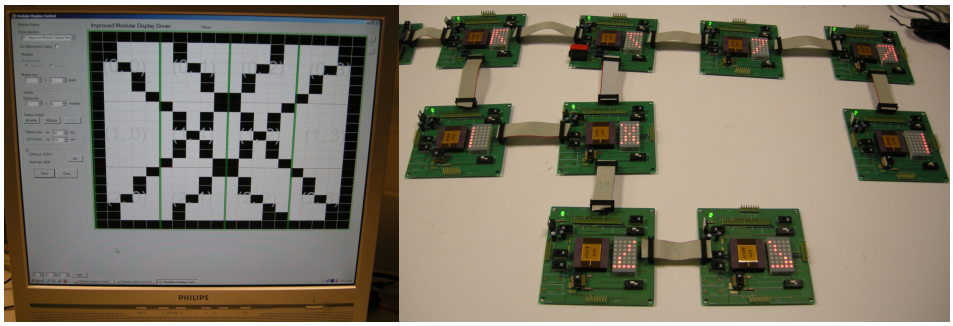


(a)

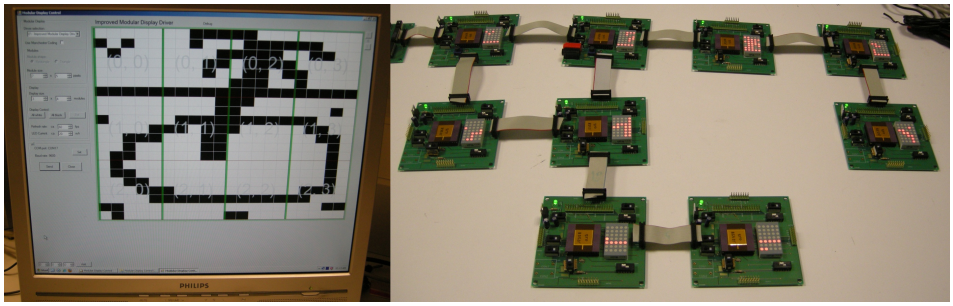


(b)

Figure 10.16 – A larger display using the improved modular display driver. Before initialization (a) and after initialization (b).



(a)



(b)

Figure 10.17 – Examples of how the improved modular display driver performs on a larger display, controlled by the GUI.

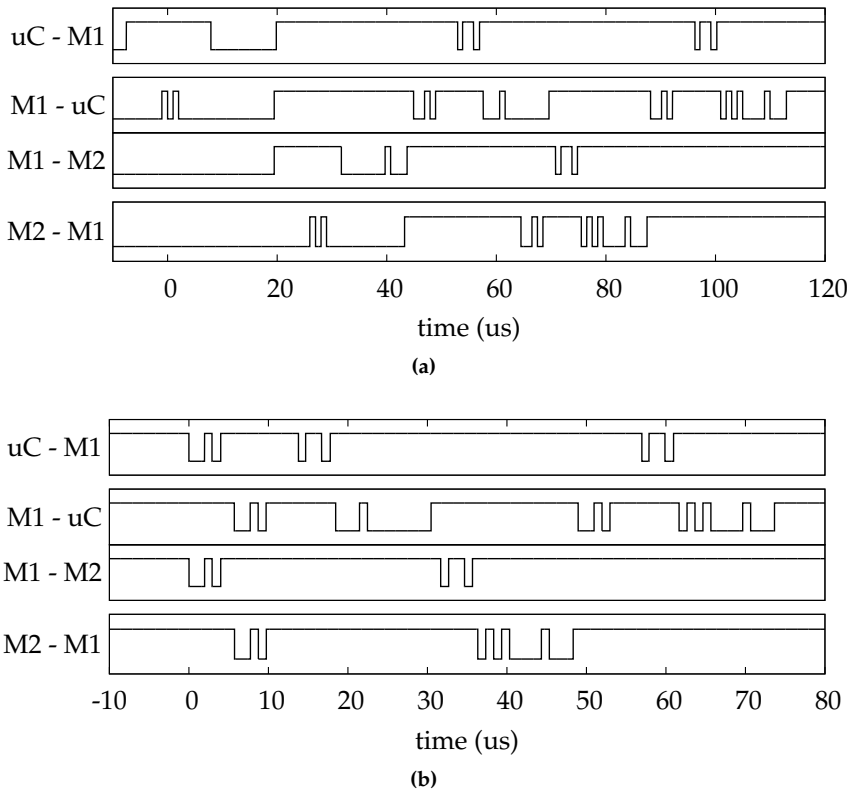


Figure 10.18 – Initialization signals (a) and polling signals (b) of a display of two modules using the free-form modular display driver.

layout related problem.

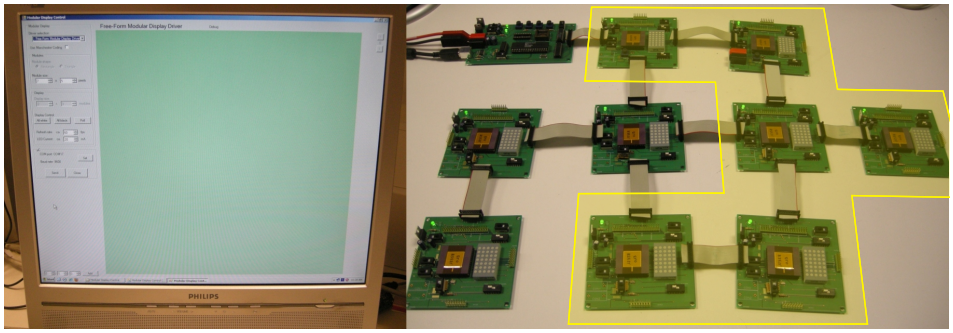
Initially, this problem can have a number of causes. There could be an error in the netlist-to-layout conversion itself. While this is highly unlikely (hopefully), it is still a possibility. But no errors are detected when LVS-check is performed (i.e. checking if the layout matches the designed circuit). Another possible cause is that one of the concerning logic gates encounters a too large fan-out. The logic gate is for example not strong enough to pull its output up in time. If this were the case, the problem should be solved when driving the chip at a lower frequency (giving the gates more time to pull up their output). When using oscillators of 10MHz (half of the designed frequency), the problem is still present. So unless there is a very large fan-out problem (frequency needs to be reduced even more), this is not the cause. Also the problem of crosstalk is a possibility. The free-form modular display driver is physically a lot larger than previous drivers, so it is possible that two tracks are close together for a long distance. Sharp transitions on one track can influence the data on the adjacent track. In any case, if there were to make a redesign, this is something to be looked at (See following section).

As said, the addresses are received and remembered correctly, which means that this driver still has the capability to detect the display shape. This process is illustrated in Figure 10.19. We start out with a blank canvas (Figure 10.19a). All modules are already connected, but they can be turned on and off individually. The yellow overlay shows which modules will participate in the initialization process. Figure 10.19b illustrates the result after the initialization is started. The active modules are recognized correctly. The red lines indicate how the tree structure is created, how the data will flow. The next modules are ready to be activated (yellow overlay). Figure 10.19c shows the result, modules added after the initialization is finished are recognized and added in the tree structure.

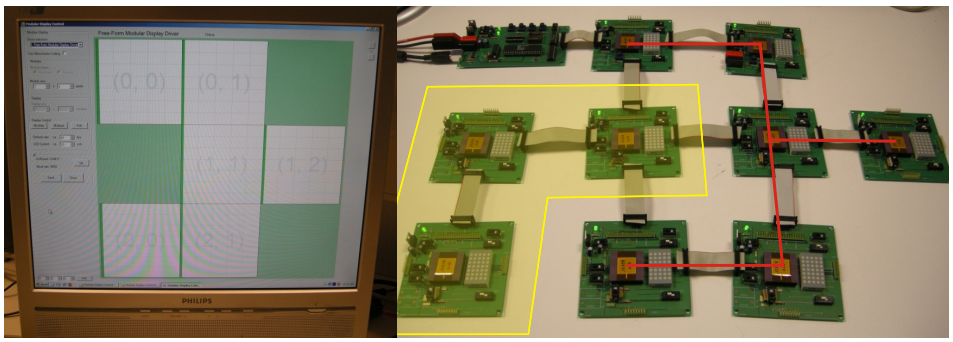
A last property that needs to be checked is how the display behaves when a module is removed. If we remove module (0,1) (top right), modules (1,1), (1,2), (2,1) and (2,0) will lose their path to the microcontroller. When the display was first turned on, they were dependent on (0,1) (Figure 10.19b), so they will need to find another path, which will be available through module (1,0). You can see how the display managed this in Figure 10.20. Since the used addresses have all reached the microcontroller successfully, the affected modules have found their new path and the tree structure has been adjusted.

10.3.8 Improved Free-Form Modular Display Driver

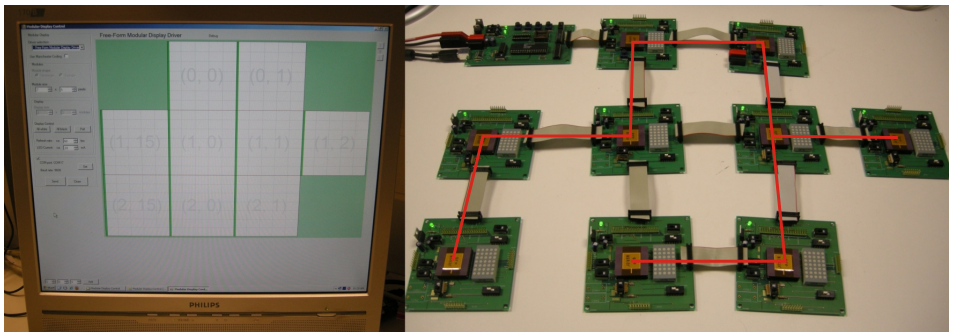
The last driver is the improved free-form modular display driver. During the initialization process, each module will now have to request an address directly from the microcontroller. This is done by setting up a connection between the module and the microcontroller by going through the send routine, discussed in Chapter 8. In order for the microcontroller to know how the modules are con-



(a)



(b)



(c)

Figure 10.19 – Initialization process using the free-form modular display driver. The yellow overlay shows which modules will be turned on next. The red lines represent the created tree structure.

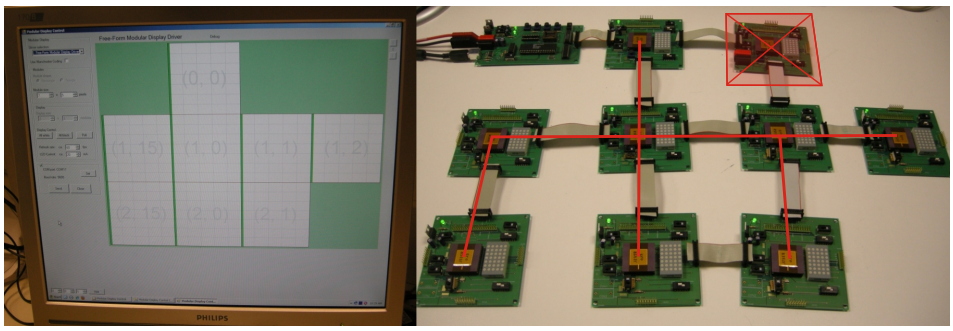


Figure 10.20 – After module (1,1) was removed, the affected modules have been rerouted and all the used addresses found their way to the microcontroller

nected, it needs to know (for each module) the address of the parent node, the child gate number of the parent node and the parent gate of the module. Modules can be added and removed, without interfering with the display operation.

Figure 10.21a shows the measured initialization process in a display of two modules (again connected as before). The microcontroller indicates that it is ready to send an address (high output) and the first module asks an address (preliminary address request, with parent gate number). After the receipt (address `0x00`) it is ready to provide a connection to the microcontroller (high output). The second module performs the preliminary address request, to which the first module will try to perform a complete address (with parent address, child gate number and parent gate number) request by going through the send routine. In this case, there are no more modules between the that module and the microcontroller, so the send routine only consists of one send request and one send permit. After the complete address request, the address is directly sent through to the second module. Initialization is ended by a 'send end' command of both modules.

In Figure 10.21b you can see the polling procedure for that same display. After the poll request of the microcontroller, both modules respond at the same time with a preliminary polling answer (including own address and parent gate number). After the microcontroller sent the 'ready' command to the first module, after processing its polling answer, the first module will go through the send routine in order to send the complete polling answer (with parent address, own address, parent gate number and child gate number) of the second module. Polling is ended with a 'send end' command of both modules.

This driver was also tested on a larger display. Figure 10.22 illustrates this. This driver makes it possible to connect the modules in any orientation. In Figure 10.22a you can see that some modules are upside down and others are sideways. In Chapter 8 I said that this driver could be used to create 3D shaped displays, and could also be used with triangular modules. This is still the case, of

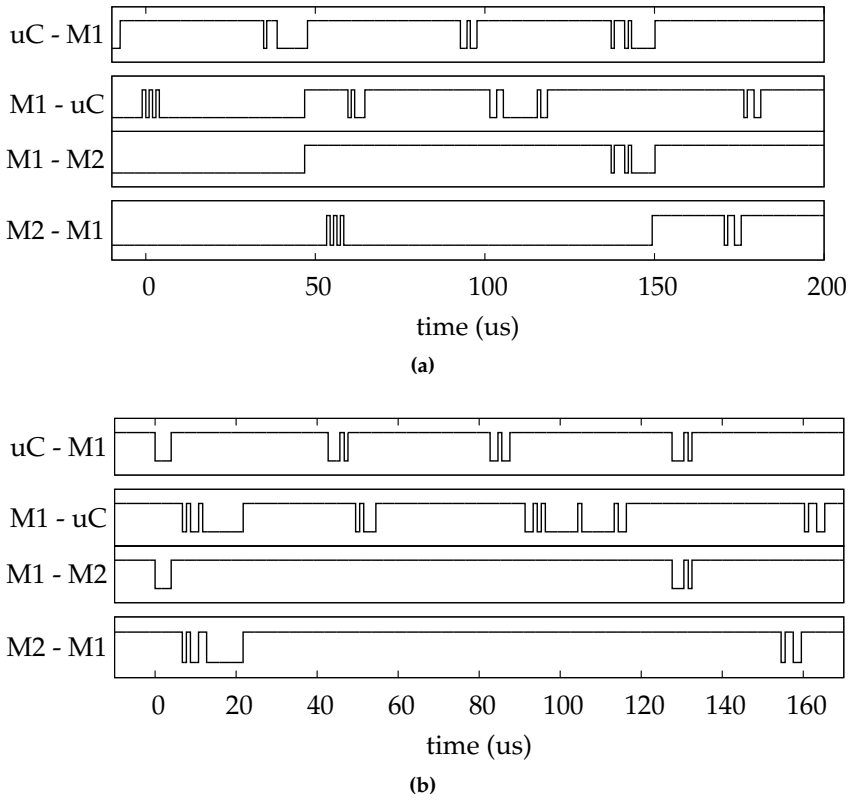


Figure 10.21 – Initialization signals (a) and polling signals (b) of a display of two modules, using the improved free-form modular display driver

course, but isn't tested explicitly, because of practical issues. Being that I didn't create triangular modules and that the modules I did create aren't built to be physically stacked. However, the protocols aren't dependent on this, so if this driver works correctly on the display in Figure 10.22a, it will also work on the other possible display structures.

We start the initialization procedure with only one module active. Figure 10.22b shows the result. The module is recognized and shown in the GUI. This module is orientated upright, which is indicated with the green bar at the left side of the module in the GUI. After a successful initialization, the microcontroller sends the default parameters and a default image. This time, this default image is again the address of the module. Since the microcontroller hands out addresses as they are requested, this will also be the order in which the modules are initialized. The blue arrow in the GUI indicates where the parent node is located, where the module gets its data from.

Figure 10.22c shows the result of turning the next two modules on. You can see that the orientation is also detected correctly. Module 1 is sideways (green bar at the bottom) and module 2 is upside down (green bar at the right). Figure 10.22d depicts the final result, when all modules are turned on. All the modules and their orientation have been detected correctly.

We can now use the generated display in the GUI to draw an image. Figure 10.23 shows an example and the result on the display. It shows that there has been taken account of the orientation of the modules.

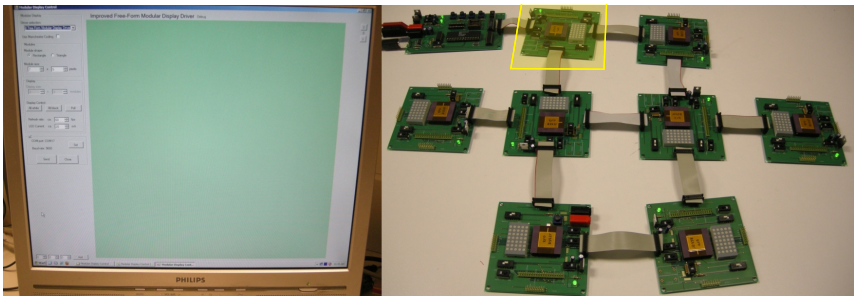
In Figure 10.24 you can see what happens if we remove module 2 (lower left corner). As you can see in Figure 10.22d, this module is the parent node of module 5 (lower right corner), so the latter will have to reroute in order to receive any data. Figure 10.24a shows the result after the removal. Module 2 has gone and module 5 now receives data from module 4. The blue arrow indicating its parent node is now pointed upwards. All active modules can still receive data, as proven by Figure 10.24b.

10.4 Future design considerations

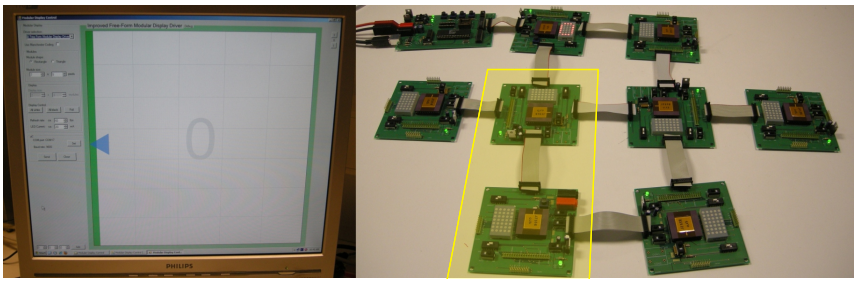
The FrIIDoM driver was tested and proven to be working for the greater part. However, there are some issues that need to be addressed if we would want to make another version of this driver. These changes might improve the performance of the driver and increase the overall robustness.

10.4.1 Issues in the free-form modular display driver

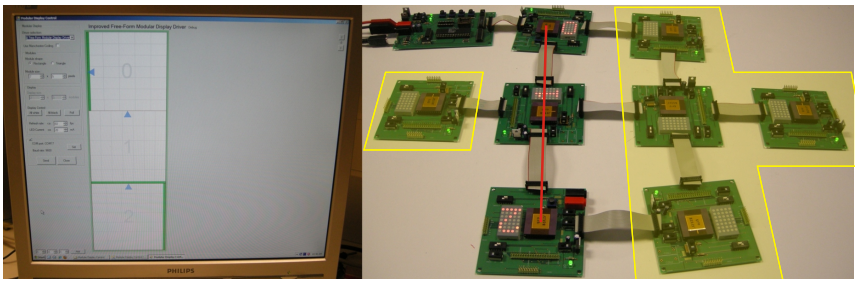
The biggest issue encountered was during the testing of the free-form modular display driver. While the addresses were received correctly, the modules did not



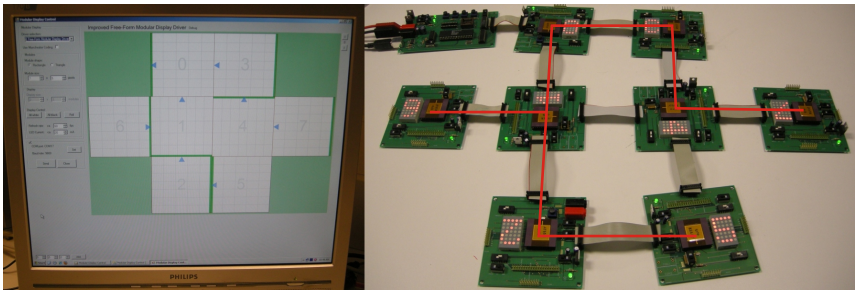
(a)



(b)



(c)



(d)

Figure 10.22 – Initialization process using the improved free-form display driver. The yellow overlay indicates which modules will be turned on next. The tree structure can be derived from the blue arrows in the GUI.

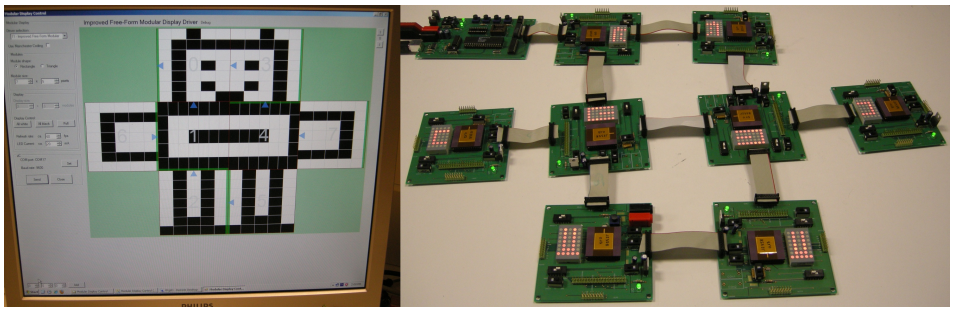
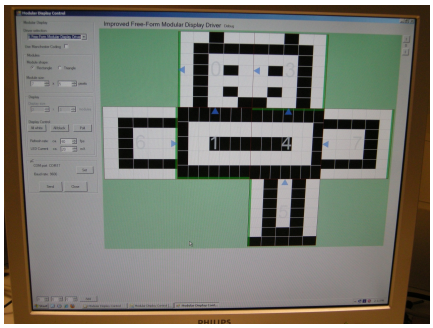
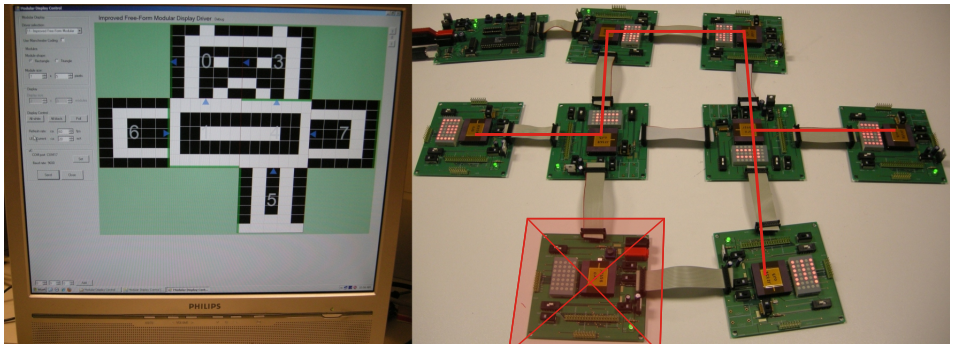


Figure 10.23 – After all modules have been initialized, the GUI is used to draw an image and display it.



(a)



(b)

Figure 10.24 – If a module is removed, it is detected by the system. The affected modules will reroute so they can still receive data.

succeed in recognizing their addresses in a data stream. As said, this is an issue that only occurs in the FrIIDoM driver (all drivers react the same way), but not in any previous simulations and tests. The problem must be layout related.

In Section 9.6 of previous chapter, I explained the concept of DFT, where scan chains are inserted in the digital logic in order to pinpoint structural errors in the layout. This method could be used to locate the differences between the gate-level netlist and the generated layout, and why these discrepancies have occurred (e.g. large fan-out, long parallel tracks). With this knowledge, the relevant part of the layout may be manually redesigned. Or, if it is a more general problem, the layout may be regenerated with adjusted settings.

10.4.2 Improvements on the Physical Layer

The FrIIDoM driver could also be improved on the Physical Layer of its protocol. I'm referring to the 7 layer OSI model, discussed in Chapter 4. In the Physical Layer, I include the communication itself, including possible Manchester (de)coding. The command use, address assignation, etc. belong to the Data Link Layer.

The problem in the Physical Layer is that in a real-life system, where modules are being connected and disconnected, there could be a lot of noise on the data line, because of drivers behaving unexpectedly when powered on or off. Switches, used to connect and disconnect signals can cause ground bounces. While the drivers try their best to filter out short glitches, longer irregularities (e.g. while connecting a module) are more difficult to eradicate. Glitches are recognized as commands or data sequences and problems occur.

A solution could be to provide the data sequences with longer fixed preambles instead of the lonely start bit. The chance that one of the glitches matches the preamble is quite small (depending of the size of the preamble of course). Same goes for the Manchester decoding. While it is shown to be working in previous section, there is still some miscommunication. A fixed preamble on each data package (e.g. `01010101`) can help the Manchester decoder estimate the clock frequency of the sending device (module or microcontroller) and adjust its counters (used for decoding) accordingly.

10.4.3 Clock adjustments

From previous section, we know that the created clock frequencies are slightly higher than anticipated. We can use this knowledge to adjust the design to better match the desired frequencies.

10.5 Significance and applications

10.5.1 Increasing the multiplexability in passive-matrix displays

As said in Section 5.1, the main goal of the first modular display driver was to increase the multiplexability (and thus the number of rows) of passive-matrix displays. We've shown that by dividing a passive-matrix display in modules with independent row electrodes, the number of rows can be increased because the limitation is now only dependent on the number of rows in a module, not on the display as a whole. But there might be some other perks.

10.5.2 Advantages for the ChLCD

We still know from Chapter 2 that ChLCD is a bistable display. Low voltages don't affect the state a pixel is in. Using the conventional minimal-swing driving scheme (See Section 5.5.1) pixels on the non-selected rows remain completely unaffected by the driving signals for the selected row. Because of this, multiplexability is not an issue for ChLCD, or any other display with these properties. It seems that a modular display driver would be completely useless here. There are some other benefits, though.

Driving a ChLCD is very slow. Every line takes about 10-20ms to drive. If we want to create a higher resolution display, the frame times can become very large. If we were to divide the display into areas that can update the display in parallel, the total frame time could be greatly reduced. As you probably already guessed, this is where the modular display driver can come in. The modules are independent, so they can drive their own part of the display all at the same time. The refresh rate can be increased to the maximum refresh rate of a single module.

But there's another advantage for bistable displays. Bistable displays do not require power to keep the image on their screen, which makes them highly wanted in low-power applications. With a modular display, each module can be updated separately. If you want a new image on the screen that is only different from the previous image in a couple of places, only the changed modules need an update. This can further reduce the power consumption in those displays.

10.5.3 Advantages for a LED display

A passive-matrix LED display (See Section 5.5.2) is not limited in multiplexability in the traditional way (i.e. Alt & Pleshko). LEDs on a non-selected row are not affected by the signals on the columns, because no current can flow through them. They are, however, limited in another way. When the current through a LED stops, the LED is immediately turned off. This means that a LED (or a row of LEDs) is only emitting light when this row is selected. The rest of the frame time,

these LEDs will be off. Due to our persistence of vision [1] this light will seam to be smeared out over the entire frame time, but the apparent light output will depend on the fraction of the time the LEDs are on (line time) in respect to the total frame time. The more rows in a display, the smaller this fraction is. So, if we impose a certain light output (i.e. a certain contrast), the number of rows is limited. This limitation can also be resolved by building a passive-matrix LED display from independent modules.

10.5.4 Creating a passive-matrix PDLC display

One of the liquid crystals that looked useful for creating a flexible e-paper is PDLC (See Chapter 2). In its reflective state, it produces a nice milky-white light. Incident light reflects of the many droplets and is diffusely scattered back. When the backplane of the display is a black light absorber, the transmissive state of the display produces a black pixel. You could also use a mirror as backplane. This would cause the white pixels to be even more white (more light is reflected back). Yet a pixel in the transmissive state would appear to be black, because of the shadows cast by the surrounding white pixels. Of course this will only work if there aren't any large black areas necessary. But if the e-paper simply needs to display text, it should work.

Another advantage is that a PDLC display is very easy to create (See below) and fairly easy to make flexible. Also, while the exact electro-optical characteristics are dependent on the thickness of the LC in the display, once the display is created, it can be pushed and bent without changing these characteristics too much. Compare this with a regular LC display (e.g. the LCD screen from your PC) where the image is deformed when you push your finger on the screen.

With these things in mind, we tried to create a PDLC display. The components that were used are PN393 as the pre-polymer and TL213 as the LC. Both are supplied by Merck [2]. We created a solution of 20%wt PN393 and 80%wt TL213. This (liquid) solution will, when cured with UV-light, become the PDLC. Since UV-light cures the PDLC, all actions need to take place in a yellow, UV-free room in the clean room.

A first test consisted of creating a PDLC display with glass carriers. The glass carriers were covered in a ITO (Indium Tin Oxide) coating. ITO is used to create a thin, flexible, transparent and electrically conductive film. Spacers (in our case spheres with 10μm diameter) were placed on one of the carriers. Next, a droplet of the LC solution is placed on the other carrier. Placing the first carrier (with the spacers at the bottom) on top of the second carrier, creates a display with a fixed width of 10μm. It is now ready to be cured. Curing parameters (curing intensity, curing time) can influence the electro-optical characteristics of the PDLC [2, 3]. A greater curing intensity will cause the LC droplets to be smaller. Smaller droplets will scatter the incident light more, but will also need a higher voltage to become

transparent. We used the Hg lamp as the UV source and cured the PDLC for 180s at an intensity of $12,5mW/cm^3$. The result is shown in Figure 10.25 (top). In the figure you see the display in its reflective state, when no voltage is applied (left), and in its transparent state, when 10V is applied (The difference between the on and off state are not very clear. The sample is not completely placed against the black background, letting light through from the back). The same procedure was performed, but now with flexible PET (PolyEthylene Terephthalate) carriers (Figure 10.25 (bottom)).

The next step was to create a display with more pixels. To do this the ITO on the PET carriers was lithographically etched and patterned to create row and column electrodes. See Figure 10.26 for the results. By applying voltages to the electrodes, we were able to turn a specific pixel on. From Chapter 2 we know that the multiplexablity is dependent on the maximal voltage were the pixel is considered off (V_{OFF}) and the minimal voltage were the pixel is considered on (V_{ON}). With our created display, $V_{OFF} = 2.2V$ and $V_{ON} = 7.0V$. Using Equation 2.2 we realize that this display isn't multiplexible at all. In literature, displays were created with a multiplexability of 3-4 lines [4]. This can be further improved to about 7 lines by using Multi-Line Adressing (MLA). So, technically it would be possible to create e-paper from PDLC using a modular approach. To create a somewhat higher resolution display, however, (which is kinda needed to create acceptable e-paper) too much modules would be required. We chose not to pursue this path and focus our energy to the intelligence of the modular display drivers itself.

10.5.5 Free-form displays

While the modular approach can increase the multiplexability in passive matrix systems, the final drivers are capable of a lot more. In Chapter 3 we briefly discussed some systems dealing with irregular, out of the (rectangular) box, displays. The systems described in this book might as well be added to that list.

The variety of display shapes possible with the first driver (Chapter 5) is quite limited, due to the fact that it has only one input and output gate. After all, creating a free-form display wasn't the intention for that driver. The second driver (Chapter 6) performs a lot better in that aspect. With its four input/output gates, a lot more different displays can be created. The shapes are still limited to a flat matrix of modules. The third driver (Chapter 7) won't allow any more display configurations than the second driver does since every module still has four input/output gates and the addresses are calculated the same way. However, the interface with the user is simplified (automatic configuration detection) and the display can be changed while it is running, which offers a different type of freedom. The fourth and last driver (Chapter 8) again provides an extra level in the variety of display shapes. Not only can the modules be connected in any way possible, the modules themselves can have a different shape.

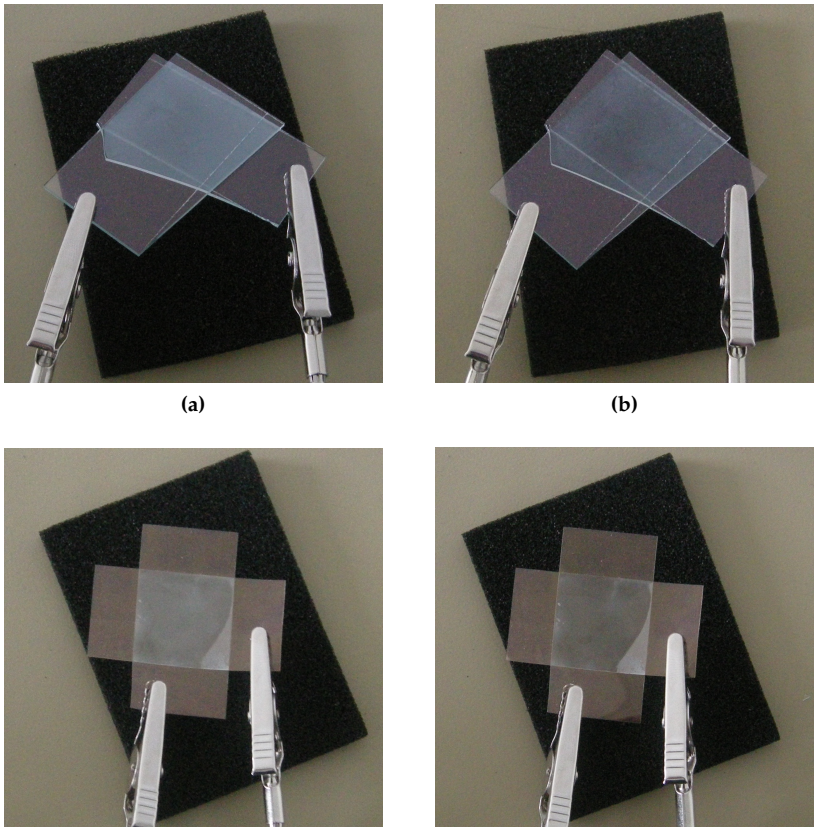
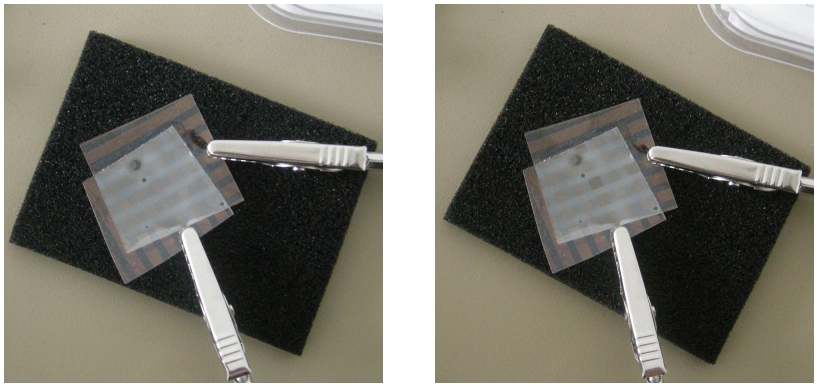


Figure 10.25 – A PDLC display with glass carriers (top) and PET carriers (bottom) in their reflective state (left) and its transparent state (right).



(a) (b)

Figure 10.26 – A passive-matrix PDLC display with PET carriers, with all pixels off ($V_{OFF} = 2.2V$) (left) and with one pixel on ($V_{ON} = 7.0V$) (right).

10.5.6 Using flexible modules

A whole new range of possibilities emerges when the modules are made flexible or even stretchable. This way, it becomes even more easy to create displays of even more irregular shapes. These modules could, for example, be used in clothing. The user takes a couple of modules and sews them together to create a display that fits the garment. To accomplish these flexible modules, it is of course necessary that the drivers themselves are flexible.

The Interuniversity MicroElectronics Center (IMEC), together with the Ghent University, has developed a new concept for packaging ultra-thin chips: the ultra-thin chip package (UTCP) [5]. The UTCP is based on the embedding of ultra-thin chips (normal sized chips can be polished to a thickness of 15 μm) in polyimide (PI). The total package is about 50-60 μm thick. This is so thin that the whole package is flexible. To give a bit of a feeling of such an UTCP, the process flow is shown in Figure 10.27. The base substrate is a uniform PI layer, applied on a rigid carrier. Next, a 30 μm photodefinable polyimide (PD PI) layer is spin coated. After exposing the PD PI with a mask, the unexposed regions are etched away by the developer, defining the cavities in the PD PI layer. After cure of the PD PI, adhesive is added and the ultra-thin chip is placed and fixed in the cavity. The chip and the adhesive are filling the cavity. Next, the top polyimide is applied, via openings are laser drilled and the metal layer (TiW/Cu) is sputtered. As vias with diameters down to 35 μm are realized using a tripled YAG laser, chips with contact pitches down to 100 μm can be used. After patterning of the metal layer the flat package can be released from the carrier.

The final result is depicted in Figure 10.28. The UTCP is bendable with a cur-

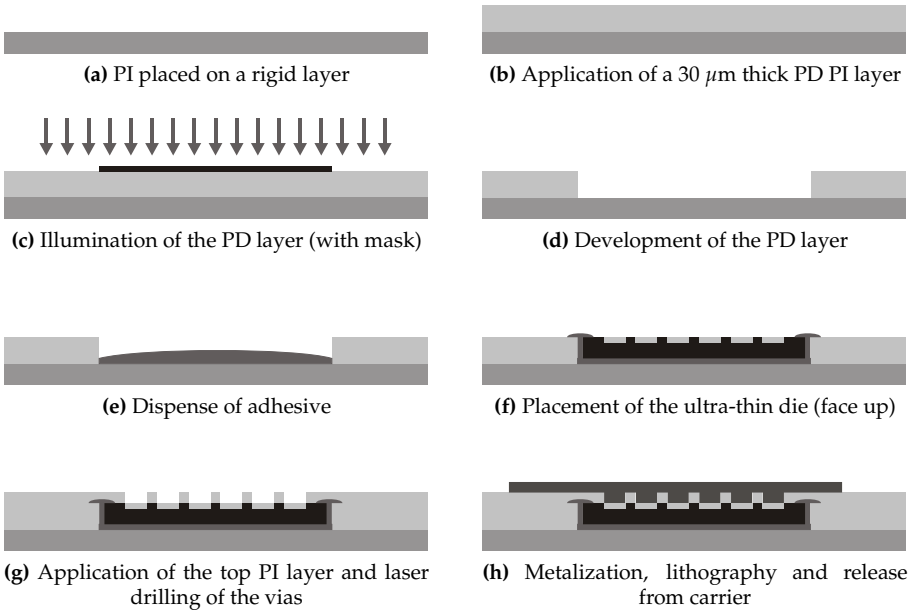
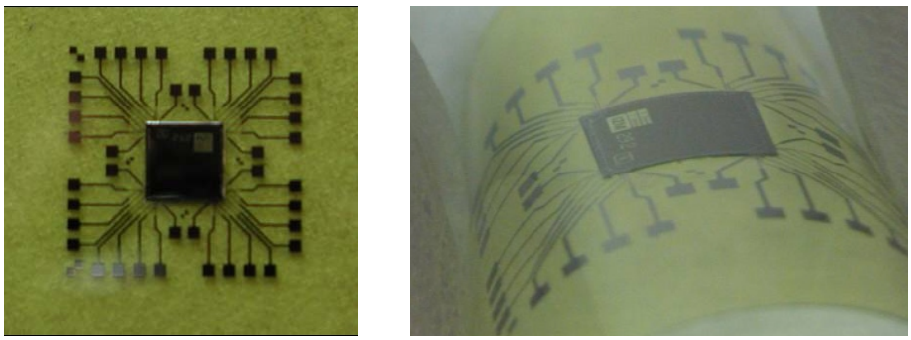


Figure 10.27 – The UTCP process flow



(a) (b)

Figure 10.28 – Final result of the UTPC process. The whole package is bendable

vature of about 5 mm, without damaging the silicon chip or the adhesive layer. This new concept of packaging ultra-thin chips gives us some great advantages. The most important advantage is that the whole package is very thin and flexible. When these chips are placed on the back of a (flexible) display, they will not interfere with the overall thickness or flexibility. Also, with the UTCP, you can embed several chips without having to deal with the Known Good Die problem. We can test the chips before embedding them, increasing the yield of the whole system. Since there is a fan-out created on the chip package, there is no need for a very fine pitch on the PCB or Flexible Printed Circuit (FCB).

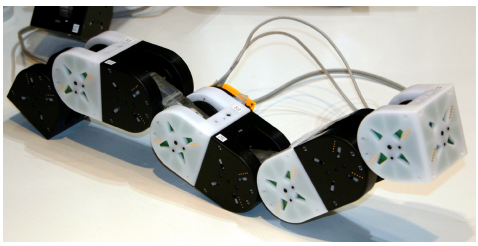
10.5.7 Outside the display world

While the drivers were primarily designed for controlling a display, the protocols are fairly general and could very well be used for applications outside the realm of displays. A system that consists of several independent modules may find its use here.

One particular system that comes to mind is the modular robot. A modular robot is built up by identical, movable modules that can be connected in several ways. Examples are the PolyBot from PARC [6] (Figure 10.29a), M-TRAN from AIST Japan [7] (Figure 10.29b), the CKBot developed at UPenn [8] (Figure 10.29c) and the SuperBot from the University of Southern California [9] (Figure 10.29d). The modules in these robots are independent from each other, but still have to work together to make the robot work as a whole. With these modules, the robot is (self-)reconfigurable, (self-)upgradeable, etc.

Most of these modular robots use a network protocol that is based on fixed addressing [8], some don't use addressing at all, like the SuperBot. The actions of a

(a)



(b)

(c)

(d)

Figure 10.29 – Examples of modular robots. The PolyBot (a), the M-TRAN (b), the CKBot (c) and the SuperBot (d).

certain module is dependent on the actions of the adjacent module [10]. However, it is clear that systems like that may find a benefit in one of the created protocols. Especially the last protocol, where the configuration of the modules can be detected, and the modules can be connected either way. With cube-like modules there would be six gates instead of the four used in the display modules, but the algorithms could stay the same.

References

[1] Wikipedia. Persistance of vision. [Online]. Available: http://en.wikipedia.org/wiki/Persistence_of_vision

[2] F. Bruyneel, "Introduction of Color in Reflective PDLC and PNLC Microdisplays," Ph.D. dissertation, Ghent University, 2002.

[3] P. Nolan, E. Jolliffe, and D. Coates, "Film-formation parameters affecting the electro-optic properties of PDLC films," in *Proceedings of the SPIE*, vol. 2408, 1995, pp. 2–13.

[4] G. Spruce and R. D. Pringle, "Polymer dispersed liquid crystal (PDLC) films," *Electronics & Communication Engineering Journal*, vol. 4, no. 2, pp. 91–100, April 1992.

[5] W. Christiaens, "Active and Passive Component Integration in Polyimide Interconnection Substrates," Ph.D. dissertation, Ghent University, 2009.

[6] M. Yim, D. Duff, and K. Roufas, "PolyBot: a Modular Reconfigurable Robot," in *Proceedings of the 2000 IEEE International Conference on Robotics and Automation*, 2000, pp. 514–520.

[7] A. Kamikura, S. Murata, E. Yoshida, H. Kurokawa, K. Tomita, and S. Kokaji, "Self-Reconfigurable Modular Robot," in *Proceedings of the 2001 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2001, pp. 606–612.

[8] M. Park, "Configuration Recognition, Communication Fault Tolerance and Self-reassembly for the CKBot," Ph.D. dissertation, University of Pennsylvania, 2009.

[9] B. Salemi, M. Moll, and W.-M. Shen, "SUPERBOT: A Deployable, Multi-Functional, and Modular Self-Reconfigurable Robotic System," in *Proceedings of the 2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2006, pp. 3636–3641.

[10] W.-M. Shen, B. Salemi, and P. Will, "Hormones for self-reconfigurable robots," in *Interbational Conference on Intelligent Autonomous Systems (IAS-6)*, 2000, pp. 918–925.

A witty saying proves nothing.
Voltaire (1694-1778)

11 Conclusion and Future Prospects

11.1 Main achievements

The main goal of this Ph.D. was to create a couple of intelligent drivers that could automate the configuration in modular displays. Modular displays can be used to increase the multiplexability (affecting contrast, resolution and brightness) in passive matrix displays or to create free-form displays. This can go from simple scalable displays to full 3D-shaped displays, as explained in Chapter 3.

In this work, four such drivers were presented. Chapter 5 explains the *first modular display driver*. This driver enables a modular display system, where the modules are connected in a bus network, to be configured automatically. All modules receive a unique address that can be used to send module-specific data.

The *improved modular display driver* is discussed in Chapter 6. This driver controls a modular system connected in a mesh network. This provides more freedom in the displays that can be created, but we must take care to prevent possible data loops.

In Chapter 7 we go a little bit further with the *free-form modular display driver*. While the modular systems that can be created with this driver remain the same, there is a lot of added functionality. The modular system is now clearly defined in a hierarchical, dynamical tree structure, created at start-up. The shape of the created display is detected and can be represented in a user interface. Modules can be added and removed when the display is running, without interfering with the

	Chapter 5	Chapter 6	Chapter 7	Chapter 8
Increase multiplexability	Yes	Yes	Yes	Yes
Free-Form Display	No	Yes	Yes	Yes
Configuration detection	No	No	Yes	Yes
Complex display shapes	No	No	No	Yes
Initialization speed	Fast	Fast	Fair	Slow

Table 11.1 – Overview of the properties of the created drivers.

operation of the display. The changes are detected, the GUI and the tree structure are updated.

The *improved free-from modular display driver* from Chapter 8 offers the same functionality, but there is again an increase in freedom of displays that can be created. Modules can be connected in any way possible. When the modular system is no longer fixed to a flat matrix of modules, the possibility emerges to create 3D-shaped displays. The driver also enables the use of triangular modules, further increasing the freedom of creation. Table 11.1 sums up the properties of the created drivers. The functionality of all four drivers have been implemented in VHDL and tested with modules equipped with an FPGA. After the successful tests, it was time to transform these drivers in one chip.

Chapter 9 elaborates on the creation of the *FrIIDoM driver*. This one driver includes the four drivers discussed above and is designed to drive an 8×8 passive matrix LED display. It has its own 8-bit adjustable current source (up to 50mA), row switches (up to 400mA), on-chip clock and power-on reset. The measurement results on this driver are presented in Chapter 10. Except for the address recognition problems of the free-form modular display driver, the four drivers seem to be working as they should. Also the current sources and switches proved to be functioning properly.

11.2 Future work

The first steps that should be taken if work was continued on these drivers is to eliminate the error in the free-form modular display driver, so that addresses in the data stream are correctly recognized. There is also room for improvement on the physical layer of the protocol, ensuring more robust communication. Maybe it could also be interesting to take a look at some other chip technologies. Since the LED drivers of the FrIIDoM chip have to source a fairly large current, a BiCMOS (Bipolar CMOS, where CMOS and bipolar junction transistor are both present) technology may result in a smaller and maybe cheaper chip.

Furthermore, there might be some other ideas to work out. We could take a look at implementing collision detection, further improving the communication qual-

ity. The modules and microcontroller could communicate through a wireless connection, which would need some adjustments to the used protocols. Including a gyroscope and magnetometer in the modules can make it possible to detect their physical orientation to create a real-life representation in the GUI.

As of now, the display has to consist of identical modules: all square, triangular, pentagonal, ... modules. It is possible to let the module detect its own shape (number of active gates) and send this information to the microcontroller, allowing the modular display to consist of modules of different shapes and even further increase the range of possible displays.

On the side of increasing the multiplexability, research could be done on the implementation of Multi-Line Addressing (Chapter 3).

Using this modular display system with bistable displays, the power efficiency can be increased. With some adjustments to the protocols, it can be even further increased when a specific low-power sleep state is introduced when the display nor the modules need to be active. When the image on the display needs to be changed, only the relevant modules need to wake up.

As said in Section 10.5.7, these protocols could find their way outside the display world as well. According to the application in mind, further improvements and extensions can be made. In the case of the modular robot, it could be interesting if every module can communicate with every other module (e.g. to separate 'legs' that have to be coordinated). For this, an extra protocol layer, the network layer, could be added, where every module acts as a router and is aware of its entire offspring.

As you can see, these protocols can form the basis of a lot more, and a lot more complex protocols, both inside and outside the display world.

A

VHDL implementation of *Main Control* (1)

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.all;
3  use IEEE.numeric_std.all;
4
5  entity main_ctrl_1_mdd is
6    -- Clock signals are not really clock signals,
7    --  but 'enables' for the registers.
8    port(
9      clk: in std_logic;
10     data_in: in std_logic;
11     rec_clk: in std_logic;
12     start_bit: in std_logic;
13
14     address: in std_logic_vector(7 downto 0);
15
16     data_rec: in std_logic;
17     Dp: in std_logic;
18
19       -- number of rows
20     param_in: in std_logic_vector(2 downto 0);
21
22     POR: in std_logic;
```

```
23      rst: in std_logic;
24
25        -- pulse, loads the address in Tx
26      load_out: out std_logic;
27      Tx_ena: out std_logic;
28
29        -- enables for reading address, param and data registers.
30      adr_ena: out std_logic;
31      param_ena: out std_logic;
32      data_ena: out std_logic;
33
34        -- Sequencer control
35      SC: out std_logic;
36
37      reset: out std_logic;
38      reset_hrd: out std_logic
39    );
40  end main_ctrl_1_mdd;
41
42  architecture main_ctrl_1_mdd of main_ctrl_1_mdd is
43    -- selects the clock that will be counted
44  signal clk_sel: std_logic;
45    -- 0 = rec_clk
46    -- 1 = bit ena
47    -- the clock that will be counted
48  signal count_clk: std_logic;
49  signal count: integer range 10 downto 0;
50    -- enables the bit clockfor sending
51  signal bit_clk_ena: std_logic;
52  signal bit_ena: std_logic;
53
54  signal main_state: std_logic_vector(1 downto 0);
55
56  signal load: std_logic;
57  signal load_2: std_logic;
58
59    -- bypass control
60  signal BC: std_logic;
61
62  signal adr_read: std_logic;
63  signal param_read: std_logic;
64  signal data_read: std_logic;
65  signal address_check: std_logic;
66  signal rec_addr: std_logic_vector (7 downto 0);
67
68  signal reset_P: std_logic;
69  signal reset_dp_int: std_logic;
```

```
70  signal reset_hrd_int: std_logic;
71
72  begin
73
74    reset <= reset_dp_int or reset_P or reset_hrd_int;
75    reset_hrd <= reset_hrd_int;
76    reset_hrd_int <= POR;
77
78    count_clk <= rec_clk when clk_sel='0' else bit_ena;
79    Tx_ena <= bit_ena;
80
81    adr_ena <= adr_read and rec_clk;
82
83      --generates the load-pulse for Tx
84    load_2 <= load when rising_edge(clk);
85    load_out <= load and not(load_2);
86
87  ------------ State Machine -->>>>>>>>>>>---
88    process(clk)
89    begin
90      if (rising_edge(clk)) then
91        if (rst = '1') then
92          main_state <= "00";
93
94          adr_read <= '0';
95          bit_clk_ena <= '0';
96
97          BC <= '0';
98          SC <= '0';
99
100         clk_sel <= '0';
101         load <= '0';
102         count <= 0;
103
104         reset_P <= '0';
105       else
106         case main_state is
107           when "00" => -- begin state -------- S0 ---
108             if (start_bit = '1') then
109               main_state <= "01";
110             end if;
111           when "01" => -- receive address ---- S1 ---
112             if (count = 8) then
113               count <= 0;
114               adr_read <= '0';
115               main_state <= "10";
116               reset_P <= '1';
```

```
117             else
118               clk_sel <= '0';
119               adr_read <= '1';
120               if (count_clk = '1') then
121                 count <= count + 1;
122               end if;
123             end if;
124           when "10" => -- send address ------- S2 ---
125             reset_P <= '0';
126             if (count = 10) then
127               count <= 0;
128               load <= '0';
129               bit_clk_ena <= '0';
130               main_state <= "11";
131             else
132               load <= '1';
133               bit_clk_ena <= '1';
134               clk_sel <= '1';
135               if (count_clk = '1') then
136                 count <= count + 1;
137               end if;
138             end if;
139           when "11" => -- normal operation --- S3 ---
140             -- Activate bypass and Sequencer
141             BC <= '1';
142             SC <= '1';
143           when others =>
144             main_state <= "11";
145         end case;
146       end if;
147     end if;
148   end process;
149 ---<<<<<<<<<< Stage Machine ---------------
150 ------------ Data/Param ---->>>>>>>>>>>>>---
151   -- If enabled (*_read), read param or image with the decoder
        clk ena
152   param_ena <= param_read and rec_clk;
153   data_ena <= data_read and rec_clk;
154
155   process(clk)
156   variable count: integer range 64 downto 0;
157   begin
158     if (rising_edge(clk)) then
159       if (rst = '1') then
160         count := 0;
161         reset_dp_int <= '0';
162         param_read <= '0';
```

```
163         data_read <= '0';
164         address_check <= '0';
165         rec_addr <= (others => '0');
166       elsif ((data_rec and BC) = '1') then
167         if (Dp = '1') then
168           -- Incoming image data
169           if (address_check = '1') then
170             -- Check address and if match (data_read = 1), read
                    data
171             --  Otherwise, just count data.
172             if (count = 8*(8-to_integer(unsigned(param_in))))
                    then
173               data_read <= '0';
174               reset_dp_int <= '1';
175             else
176               if (rec_addr = address) then
177                 data_read <= '1';
178               end if;
179               if (rec_clk = '1') then
180                 count := count + 1;
181               end if;
182             end if;
183           else
184             -- Receive address
185             if (count = 8) then
186               address_check <= '1';
187               count := 0;
188             else
189               if (rec_clk = '1') then
190                 rec_addr(7) <= data_in;
191                 for i in 0 to 6 loop
192                   rec_addr(i)<=rec_addr(i+1);
193                 end loop;
194                 count := count + 1;
195               end if;
196             end if;
197           end if;
198         elsif (Dp = '0') then
199           -- Incoming parameter data
200           if (count = 32) then
201             param_read <= '0';
202             reset_dp_int <= '1';
203           else
204             param_read <= '1';
205             if (rec_clk = '1') then
206               count := count + 1;
207             end if;
```

```
208           end if;
209         end if;
210       else
211         address_check <= '0';
212         reset_dp_int <= '0';
213         count := 0;
214       end if;
215     end if;
216   end process;
217 ---<<<<<<<<<< Data/Param ------------------
218 ------------ Bit_clk ------->>>>>>>>>>>>---
219   -- Used when sending, will eventually enable Tx
220   process(clk)
221   variable count: integer range 19 downto 0;
222   begin
223     if (rising_edge(clk)) then
224       if (rst = '1') then
225         bit_ena <= '0';
226         count := 0;
227       elsif (bit_clk_ena = '1') then
228         if (count = 19) then
229           bit_ena <= '1';
230           count := 0;
231         else
232           bit_ena <= '0';
233           count := count + 1;
234         end if;
235       else
236         bit_ena <= '0';
237         count := 0;
238       end if;
239     end if;
240   end process;
241 ---<<<<<<<<<< Bit_clk ---------------------
242 end main_ctrl_1_mdd;
```

B

VHDL implementation of *Main Control* (2)

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.all;
3  use IEEE.numeric_std.all;
4
5  entity main_ctrl_2_mdd is
6    port(
7      clk: in std_logic;
8      data_in: in std_logic;
9      rec_clk: in std_logic;
10     start_bit: in std_logic_vector(3 downto 0);
11
12     address: in std_logic_vector(7 downto 0);
13
14     data_rec: in std_logic;
15     Dp: in std_logic;
16
17     param_in: in std_logic_vector(2 downto 0);
18
19     POR: in std_logic;
20     rst: in std_logic;
21
22       -- Data from Parent node
```

```
23      data_int: out std_logic;
24        -- controls output gates
25      out_ctrl: out std_logic_vector(3 downto 0);
26        -- controls input gate
27      in_ctrl: out std_logic_vector(1 downto 0);
28
29        -- pulse, loads the address in Tx
30      load_out: out std_logic;
31      Tx_ena: out std_logic;
32
33        -- enables for reading address, param and data registers.
34      adr_ena: out std_logic;
35      param_ena: out std_logic;
36      data_ena: out std_logic;
37
38        -- Sequencer control
39      SC: out std_logic;
40
41      reset: out std_logic;
42      reset_hrd: out std_logic
43    );
44  end main_ctrl_2_mdd;
45
46  architecture main_ctrl_2_mdd of main_ctrl_2_mdd is
47    -- selects the clock that will be counted
48  signal clk_sel: std_logic;
49    -- 0 = rec_clk
50    -- 1 = bit ena
51  signal count_clk: std_logic;
52  signal count: integer range 10 downto 0;
53
54  signal bit_clk_ena: std_logic;
55  signal bit_ena: std_logic;
56
57  signal main_state: std_logic_vector(1 downto 0);
58  signal out_ctrl_int: std_logic_vector(3 downto 0);
59  signal load: std_logic ;
60  signal load_2: std_logic ;
61
62  signal BC: std_logic;
63
64  signal adr_read: std_logic;
65  signal param_read: std_logic;
66  signal data_read: std_logic;
67  signal address_check: std_logic;
68  signal rec_addr: std_logic_vector (7 downto 0);
69
```

```
70  signal reset_P: std_logic;
71  signal reset_dp_int: std_logic;
72  signal reset_hrd_int: std_logic;
73
74  begin
75
76    reset <= reset_dp_int or reset_P or reset_hrd_int;
77    reset_hrd <= reset_hrd_int;
78    reset_hrd_int <= POR ;
79
80    out_ctrl <= out_ctrl_int;
81    data_int <= data_in;
82
83    count_clk <= rec_clk when clk_sel='0' else bit_ena;
84    Tx_ena <= bit_ena;
85
86    adr_ena <= adr_read and rec_clk;
87
88    load_2 <= load when rising_edge(clk);
89    load_out <= load and not(load_2);
90
91  ------------ State Machine -->>>>>>>>>>>---
92    process(clk)
93    begin
94      if (rising_edge(clk)) then
95        if (rst='1') then
96          in_ctrl <= "00";
97          out_ctrl_int <= "1111";
98          main_state <= "00";
99
100         adr_read <= '0';
101         bit_clk_ena <= '0';
102
103         BC <= '0';
104         SC <= '0';
105
106         clk_sel <= '0';
107         load <= '0';
108         count <= 0;
109
110         reset_P <= '0';
111       else
112         case main_state is
113           when "00" => -- begin state -------- S0 ---
114             -- Choose Parent node
115             if (start_bit(0)='1') then
116               in_ctrl <= "00";
```

```
117             main_state <= "01";
118           elsif (start_bit(1)='1') then
119             in_ctrl <= "01";
120             main_state <= "01";
121           elsif (start_bit(2)='1') then
122             in_ctrl <= "10";
123             main_state <= "01";
124           elsif (start_bit(3)='1') then
125             in_ctrl <= "11";
126             main_state <= "01";
127           end if;
128         when "01" => -- receive address ---- S1 ---
129           -- adjust out_ctrl on incoming star tbits
130           out_ctrl_int <= out_ctrl_int and not(start_bit);
131           -- receive address
132           if (count = 8) then
133             count <= 0;
134             adr_read <= '0';
135             main_state <= "10";
136             reset_P <= '1';
137           else
138             clk_sel <= '0';
139             adr_read <= '1';
140             if (count_clk = '1') then
141               count <= count + 1;
142             end if;
143           end if;
144         when "10" => -- send address ------- S2 ---
145           reset_P <= '0';
146           if (count = 10) then
147             count <= 0;
148             load <= '0';
149             bit_clk_ena <= '0';
150             main_state <= "11";
151           else
152             load <= '1';
153             bit_clk_ena <= '1';
154             clk_sel <= '1';
155             if (count_clk = '1') then
156               count <= count + 1;
157             end if;
158           end if;
159         when "11" => -- normal operation --- S3 ---
160           -- Activate bypass and Sequencer
161           BC <= '1';
162           SC <= '1';
163         when others =>
```

```
164             main_state <= "11";
165         end case;
166       end if;
167     end if;
168   end process;
169 ---<<<<<<<<< Stage Machine ---------------
170 ------------ Data/Param ---->>>>>>>>>>>---
171   param_ena <= param_read and rec_clk;
172   data_ena <= data_read and rec_clk;
173
174   process(clk)
175   variable count: integer range 64 downto 0;
176   begin
177     -- See Appendix A
178   end process;
179 ---<<<<<<<<< Data/Param ------------------
180 ------------ Bit_clk ------->>>>>>>>>>>---
181   process(clk)
182   variable count: integer range 19 downto 0;--:= 0;
183   begin
184     -- See Appendix A
185   end process;
186 ---<<<<<<<<< Bit_clk ---------------------
187 end main_ctrl_2_mdd;
```

C

VHDL implementation of *Main Control* (3)

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.all;
3  use IEEE.numeric_std.all;
4
5  entity main_ctrl_3_mdd is
6    port(
7      clk: in std_logic;
8      data_in: in std_logic_vector(3 downto 0);
9      rec_clk: in std_logic_vector(3 downto 0);
10       --address available
11     adr_av: in std_logic_vector(3 downto 0);
12
13       -- own address
14     address: in std_logic_vector(7 downto 0);
15       -- address to be shouted
16     sht_address: in std_logic_vector(7 downto 0);
17
18       -- command received
19     cmd_rec: in std_logic_vector(3 downto 0);
20       -- when Rx detects missing module
21     gone: in std_logic_vector(3 downto 0);
22       -- address request received
```

```
23      adr_req: in std_logic_vector(3 downto 0);
24        -- address received
25      adr_ack: in std_logic_vector(3 downto 0);
26        -- shout request received
27      sht_req: in std_logic_vector(3 downto 0);
28        -- shout acknowledgement received
29      sht_ack: in std_logic_vector(3 downto 0);
30        -- shout received
31      sht: in std_logic_vector(3 downto 0);
32        -- shout end received
33      sht_end: in std_logic_vector(3 downto 0);
34      param: in std_logic_vector(3 downto 0);
35      data: in std_logic_vector(3 downto 0);
36
37      param_in: in std_logic_vector(2 downto 0);
38
39      POR: in std_logic;
40      rst: in std_logic;
41
42        -- data from parent node
43      data_int: out std_logic;
44        -- shout data
45      data_sht: out std_logic;
46        -- data for child nodes
47      data_ext: out std_logic;
48
49        -- enables for registers
50        --  (address, shout address, data)
51      adr_ena: out std_logic;
52      sht_adr_ena: out std_logic;
53      param_ena: out std_logic;
54      data_ena: out std_logic;
55
56      SC: out std_logic;
57
58        -- Output buffer (for Tx)
59      Tx_buf: out std_logic_vector(10 downto 0);
60        -- cmd: Tx(2 downto 0)
61        -- adres: Tx(10 downto 3)
62      load_out: out std_logic;
63      Tx_ena: out std_logic;
64        --
65      default_out: out std_logic;
66
67        -- control signals for the output multiplexers
68      mux_out_sel_0: out std_logic_vector(1 downto 0);
69      mux_out_sel_1: out std_logic_vector(1 downto 0);
```

```
70      mux_out_sel_2: out std_logic_vector(1 downto 0);
71      mux_out_sel_3: out std_logic_vector(1 downto 0);
72
73        -- reset Rx or Tx
74      reset_Rx: out std_logic_vector(3 downto 0);
75      reset_Tx: out std_logic;
76
77        -- reset after data
78      reset_dp: out std_logic;
79        -- reset after parent gone
80      reset_fll: out std_logic;
81      reset_hrd: out std_logic;
82
83      main_state_out: out std_logic_vector(3 downto 0)
84    );
85  end main_ctrl_3_mdd;
86
87  architecture main_ctrl_3_mdd of main_ctrl_3_mdd is
88
89  component wacht_mdd
90    generic (COUNT: integer := 1);
91    -- When 'start' is high, 'wacht_mdd' (re)starts counting.
92    --  When COUNT is reached, 'eind' becomes high and
93    --  remains high until reset or start = 0.
94    port(
95    --  ...
96      );
97  end component;
98  signal start_timeout: std_logic;
99  signal eind_timeout: std_logic;
100
101 -- signal declarations
102 -- ...
103 -- all signals can come from either the Parent,
104 --  or a Child:
105 -- <signal>_P, <signal>_C
106 -- ...
107 -- controls the output multiplexers
108 --  for children and parent
109 type out_sel is array(3 downto 0) of std_logic_vector(1 downto 0)
      ;
110 signal mux_P: std_logic_vector(1 downto 0);
111 signal mux_C: out_sel;
112 -- 00: default output
113 -- 01: cmd/Tx
114 -- 10: data_int
115 -- 11: data_ext
```

```
116 -- ...
117
118 begin
119
120 ------------ State Machine -->>>>>>>>>>>---
121 main_state_out <= main_state;
122 ----- signals -----
123   wacht_timeout: wacht_mdd
124   generic map(COUNT => 262144)
125   port map(
126 -- timeout is started with start_timeout. runs for 13ms
127   );
128   with clk_sel select
129   count_clk <= rec_clk(0) when "100",
130          rec_clk(1) when "101",
131          rec_clk(2) when "110",
132          rec_clk(3) when "111",
133          rec_clk_P when "000",
134          bit_ena when others;
135
136   load_2 <= load when rising_edge(clk);
137   load_out <= load and not(load_2);
138
139   Tx_ena <= bit_ena and snd_ena;
140   default_out <= adr_rec;
141   data_ext <= (not(out_ctrl(0)) or data_in(0)) and (not(out_ctrl
          (1)) or data_in(1)) and (not(out_ctrl(2)) or data_in(2)) and
           (not(out_ctrl(3)) or data_in(3));
142
143   with gate_no select
144   data_sht <= data_in(0) when "00",
145         data_in(1) when "01",
146         data_in(2) when "10",
147         data_in(3) when "11",
148         data_in(0) when others;
149
150     -- according to in_control, distinction
151     --  between child and parent signals are made
152   with in_ctrl select
153   -- rec_clk_P vs rec_clk_C
154   rec_clk_P <= rec_clk(0) when "00",
155         rec_clk(1) when "01",
156         rec_clk(2) when "10",
157         rec_clk(3) when "11",
158         rec_clk(0) when others;
159   rec_clk_C(0) <= rec_clk(0) when in_ctrl/="00" else '0';
160   rec_clk_C(1) <= rec_clk(1) when in_ctrl/="01" else '0';
```

```
161   rec_clk_C(2) <= rec_clk(2) when in_ctrl/="10" else '0';
162   rec_clk_C(3) <= rec_clk(3) when in_ctrl/="11" else '0';
163
164     -- analogous:
165   with in_ctrl select
166   -- data_int_int vs data_in_C
167   data_int <= data_int_int;
168   -- cmd_rec_P vs cmd_rec_C
169   -- gone_P vs gone_C
170   -- adr_req_P vs adr_req_C
171   -- adr_ack_P vs adr_ack_C
172   -- sht_req_P vs sht_req_C
173   -- sht_ack_P vs sht_ack_C
174   -- sht_P vs sht_C
175   -- sht_end_P vs sht_end_C
176   -- data_P vs data_C
177   -- param_P vs param_C
178
179   adr_ena <= adr_read and rec_clk_P;
180   param_ena <= param_read and rec_clk_P;
181   data_ena <= data_read and rec_clk_P;
182   sht_adr_ena <= sht_adr_read and count_clk;
183
184   mux_out_sel_0 <= mux_P when in_ctrl = "00" else mux_C(0);
185   mux_out_sel_1 <= mux_P when in_ctrl = "01" else mux_C(1);
186   mux_out_sel_2 <= mux_P when in_ctrl = "10" else mux_C(2);
187   mux_out_sel_3 <= mux_P when in_ctrl = "11" else mux_C(3);
188
189   reset_Rx(0) <= (reset_P or reset_dp_int or reset_fll_int or
          reset_hrd_int) when in_ctrl = "00" else (reset_C(0) or
          reset_fll_int or reset_hrd_int);
190   reset_Rx(1) <= ...
191   reset_Rx(2) <= ...
192   reset_Rx(3) <= ...
193
194   reset_Tx <= reset_Tx_int or reset_fll_int or reset_hrd_int;
195
196   reset_dp <= reset_dp_int or reset_fll_int or reset_hrd_int;
197   reset_fll <= reset_fll_int or reset_hrd_int;
198   reset_hrd <= reset_hrd_int;
199   reset_hrd_int <= POR ;
200
201   process(clk)
202   begin
203     if (rising_edge(clk)) then
204       if (rst = '1') then
205         -- reset every signal to 0.
```

```
206       else
207         case main_state is
208           when "0000" => -- begin state ------ S0 ---
209             if (reset_fll_int = '0') then
210               if (adr_av(0)='1') then
211                 in_ctrl <= "00";
212                 main_state <= "0001";
213               elsif (adr_av(1)='1') then
214                 in_ctrl <= "01";
215                 main_state <= "0001";
216               elsif (adr_av(2)='1') then
217                 in_ctrl <= "10";
218                 main_state <= "0001";
219               elsif (adr_av(3)='1') then
220                 in_ctrl <= "11";
221                 main_state <= "0001";
222               end if;
223             else
224               -- wait for the reset
225               reset_fll_int <= '0';
226             end if;
227           when "0001" => -- send adr_req ----- S1 ---
228             if (count >= 5) then
229               count <= 0;
230               load <= '0';
231               bit_clk_ena <= '0';
232               snd_ena <= '0';
233               mux_P <= "00";
234
235               reset_Tx_int <= '1';
236               start_timeout <= '1';
237
238               main_state <= "0010";
239             else
240               if (parent_gone = '1') then
241                 -- send adpt_req
242                 Tx_buf(2 downto 0) <= "111";
243                 Tx_buf(10 downto 3) <= (others => '0');
244               else
245                 -- send adr_req
246                 Tx_buf(2 downto 0) <= "001";
247                 Tx_buf(10 downto 3) <= (others => '0');
248               end if;
249               load <= '1';
250               bit_clk_ena <= '1';
251               clk_sel <= "001"; -- choose bit_clk
252               snd_ena <= '1';
```

```
253             mux_P <= "01";
254             if (count_clk='1') then
255               count <= count + 1;
256             end if;
257           end if;
258         when "0010" => -- receive address -- S2 ---
259           reset_Tx_int <= '0';
260           if (((cmd_rec_P and not(BC)) or cmd_rec_C(to_integer(
                  unsigned(gate_no)))) = '1') then
261             -- receive own address, or address from child node
262             start_timeout <= '0';
263             if (count >=8 or parent_gone = '1') then
264               count <= 0;
265               adr_read <= '0';
266               sht_adr_read <= '0';
267               bit_clk_ena <= '0';
268               reset_P <= '1' and (cmd_rec_P and not(BC));
269               reset_C(to_integer(unsigned(gate_no))) <= '1' and
                      cmd_rec_C(to_integer(unsigned(gate_no)));
270               if ( ( (adr_ack_P and not(adr_rec)) or ( (sht_C(
                      to_integer(unsigned(gate_no))) or  sht_end_C(
                      to_integer(unsigned(gate_no)))) and adr_rec) )
                      = '1') then
271                 adr_rec <= '1'; -- own address is received
272                 if (parent_gone = '1') then
273                   main_state <= "1111"; -- to final state
274                 elsif (sht_self = '0') then
275                   main_state <= "0011"; -- receive address
                        requests
276                 elsif (children = children_tmp) then -- passing
                        address
277                   main_state <= "1111"; -- to final state
278                 else  -- continue shout routine
279                   main_state <= "0101"; -- receive sht_req
280                 end if;
281               else -- wrong command
282                 if (adr_rec = '0') then -- no own address
283                   reset_fll_int <= '1';
284                   main_state <= "0000"; -- start over
285                 else
286                   main_state <= "1000"; -- receive sht_req
287                 end if;
288               end if;
289             else
290               if (adr_rec = '0' and adr_ack_P='1') then --
                      address from parent
291                 adr_read <= '1';
```

```
292                 clk_sel <= "000"; -- Parent clock
293                 start_shtreq <= not(BC);
294                   -- If the bypasses are already active (BC =
                          1)
295                   -- module cannot shout an address
296                   -- without checking the data line
297                   -- See state S5
298               elsif (adr_rec = '1' and (sht(to_integer(unsigned
                      (gate_no))) or sht_end(to_integer(unsigned(
                      gate_no))))='1') then -- address from child
299                 adr_end(to_integer(unsigned(gate_no))) <=
                        sht_end(to_integer(unsigned(gate_no)));
300                 sht_adr_read <= '1';
301                 clk_sel <= '1' & gate_no;
302                 start_shtreq <= not(BC);
303               else -- wrong command
304                 bit_clk_ena <= '1';
305                 clk_sel <= "001";
306               end if;
307               if (count_clk='1') then
308                 count <= count + 1;
309               end if;
310             end if;
311           elsif (eind_timeout = '1') then -- time out
312             start_timeout <= '0';
313             if (adr_rec = '0') then -- no own address
314               reset_fll_int <= '1';
315               main_state <= "0000"; -- start over
316             else
317               main_state <= "1000"; -- receive sht_req
318             end if;
319           end if;
320         when "0011" => -- receive adr_req/adpt_req - S3
321           reset_Tx_int <= '0';
322           reset_P <= '0';
323           if (count >= 20) then -- adr_req time out
324             count <= 0;
325             bit_clk_ena <= '0';
326             main_state <= "0101"; -- start sht_req
327           else
328             if (cmd_rec_C/="0000") then
329               count <= 0;
330               bit_clk_ena <= '0';
331               if ((adr_req_C  or adpt_req_C)/="0000") then
332                 main_state <= "0100"; -- send adr_ack
333                 if ((adr_req_C(0) or adpt_req_C(0))='1') then
334                   children_tmp(0)<='1';
```

```
335                   gate_no <= "00";
336                   reset_C(0) <= '1';
337                   adpt_req <= adpt_req_C(0);
338                 elsif ((adr_req_C(1) or adpt_req_C(1))='1')
                        then
339                   children_tmp(1)<='1';
340                   gate_no <= "01";
341                   reset_C(1) <= '1';
342                   adpt_req <= adpt_req_C(1);
343                 elsif ((adr_req_C(2) or adpt_req_C(2))='1')
                        then
344                   children_tmp(2)<='1';
345                   gate_no <= "10";
346                   reset_C(2) <= '1';
347                   adpt_req <= adpt_req_C(2);
348                 elsif ((adr_req_C(3) or adpt_req_C(3))='1')
                        then
349                   children_tmp(3)<='1';
350                   gate_no <= "11";
351                   reset_C(3) <= '1';
352                   adpt_req <= adpt_req_C(3);
353                 end if;
354               end if;
355             elsif (children_tmp = "0000") then
356               -- no children => start adr_req timer
357               reset_C <= "0000";
358               bit_clk_ena <= '1';
359               clk_sel <= "001";
360               if (count_clk = '1') then
361                 count <= count + 1;
362               end if;
363             else -- all adr_reqs are processed
364               reset_C <= "0000";
365               main_state <= "0101";
366             end if;
367           end if;
368         when "0100" => -- send adr_ack ----- S4 ---
369           reset_C <="0000";
370           if (count >=13 or (adpt_req='1' and count >= 7)) then
371             count <= 0;
372             load <= '0';
373             bit_clk_ena <= '0';
374             snd_ena <= '0';
375             mux_C(to_integer(unsigned(gate_no))) <= "00";
376
377             reset_Tx_int <= '1';
378
```

```
379               if (adpt_req = '1') then
380                 main_state <= "1111";
381                 adr_end(to_integer(unsigned(gate_no)))<='1';
382                 children <= children_tmp;
383                 adpt_req <='0';
384               elsif (sht_self = '0') then
385                 main_state <= "0011"; -- back to adr_reqs
386               else
387                 main_state <= "1000"; -- receive sht_req
388               end if;
389             else
390               Tx_buf(2 downto 0) <= "000";
391               if (adpt_req = '1') then
392                 Tx_buf(10 downto 3) <= (others => '1');
393               else -- calculate address
394                 case gate_no is
395                   when "00" =>
396                     Tx_buf(10 downto 3) <=std_logic_vector(
                          unsigned(address(7 downto 4))- 1) &
                          address(3 downto 0);
397                   when "01" =>
398                     Tx_buf(10 downto 3) <=address(7 downto 4) &
                          std_logic_vector(unsigned(address(3 downto
                           0))- 1);
399                   when "10" =>
400                     Tx_buf(10 downto 3) <=std_logic_vector(
                          unsigned(address(7 downto 4))+ 1) &
                          address(3 downto 0);
401                   when "11" =>
402                     Tx_buf(10 downto 3) <=address(7 downto 4) &
                          std_logic_vector(unsigned(address(3 downto
                           0))+ 1);
403                   when others =>
404                     Tx_buf(10 downto 3) <=std_logic_vector(
                          unsigned(address(7 downto 4))- 1) &
                          address(3 downto 0);
405                 end case;
406               end if;
407               load <= '1';
408               bit_clk_ena <= '1';
409               clk_sel <= "001";
410               snd_ena <= '1';
411               mux_C(to_integer(unsigned(gate_no))) <= "01";
412               if (count_clk='1') then
413                 count <= count + 1;
414               end if;
415             end if;
```

```
416           when "0101" => -- send sht_req ----- S5 ---
417             reset_P <= '0';
418             reset_C <= "0000";
419             if (start_shtreq = '1') then
420               if (count >= 5) then
421                 count <= 0;
422                 load <= '0';
423                 bit_clk_ena <= '0';
424                 snd_ena <= '0';
425                 mux_P <= "00";
426
427                 reset_Tx_int <= '1';
428                 start_timeout <= '1';
429
430                 main_state <= "0110";
431               else
432                 Tx_buf(2 downto 0) <= "010";
433                 Tx_buf(10 downto 3) <= (others => '1');
434                 load <= '1';
435                 bit_clk_ena <= '1';
436                 clk_sel <= "001";
437                 snd_ena <= '1';
438                 mux_P <= "01";
439                 if (count_clk='1') then
440                   count <= count + 1;
441                 end if;
442               end if;
443             else
444               if (count>=10) then
445                 count <= 0;
446                 bit_clk_ena <= '0';
447                 start_shtreq <= '1';
448               else
449                 if (cmd_rec_P = '0') then
450                   bit_clk_ena <= '1';
451                   clk_sel <= "001";
452                   if (count_clk = '1') then
453                     count <= count +1;
454                   end if;
455                 else
456                   count <= 0;
457                   bit_clk_ena <= '0';
458                 end if;
459               end if;
460             end if;
461           when "0110" => -- receive sht_ack -- S6 ---
462             reset_Tx_int <= '0';
```

```
463             if (cmd_rec_P='1') then
464               start_timeout <= '0';
465               reset_P <= '1';
466               if (sht_ack_P = '1') then
467                 main_state <= "0111"; -- shout
468                 children_tmp <= children_tmp or sht_req_C;
469                 -- adjust the child nodes during polling
470               else -- wrong command
471                 main_state <= "0101"; -- send sht_req
472               end if;
473             elsif (eind_timeout = '1') then -- timeout
474               start_timeout <= '0';
475               main_state <= "0101"; -- send sht_req
476             end if;
477           when "0111" => -- send sht/sht_end - S7 ---
478             reset_P <= '0';
479             if (count >= 13) then
480               count <= 0;
481               load <= '0';
482               bit_clk_ena <= '0';
483               snd_ena <= '0';
484               mux_P <= "00";
485               sht_self <= '1';
486
487               reset_Tx_int <= '1';
488               if ((children_tmp xor adr_end) = "0000") then
489                 -- all child nodes used sht_end
490                 main_state <= "1111";
491                 children <= children_tmp;
492               else -- back to receive sht_req
493                 main_state <= "1000";
494                 start_timeout <= '1';
495               end if;
496             else
497               if ((children_tmp xor adr_end) = "0000") then
498                 Tx_buf(2 downto 0) <= "101"; --sht_end
499               else
500                 Tx_buf(2 downto 0) <= "100"; --sht
501               end if;
502               if (sht_self = '1') then
503                 -- shout child address
504                 Tx_buf(10 downto 3) <= sht_address;
505               else
506                 -- shout own address
507                 Tx_buf(10 downto 3) <= address;
508               end if;
509               load <= '1';
```

```
510             bit_clk_ena <= '1';
511             clk_sel <= "001";
512             snd_ena <= '1';
513             mux_P <= "01";
514             if (count_clk='1') then
515               count <= count + 1;
516             end if;
517           end if;
518         when "1000" => -- receive sht_reqs - S8 ---
519           reset_Tx_int <= '0';
520           reset_P <= '0';
521           if (cmd_rec_C/="0000") then
522             start_timeout <= '0';
523             if (sht_req_C/="0000") then
524               if (children = children_tmp) then
525                 -- no own new child, let it pass
526                 main_state <= "1111";
527               else
528                 main_state <= "1001"; -- send sht_ack
529               end if;
530               if (sht_req_C(0)='1') then
531                 children_tmp(0)<='1';
532                 gate_no <= "00";
533                 reset_C(0) <= '1';
534               elsif (sht_req_C(1)='1') then
535                 children_tmp(1)<='1';
536                 gate_no <= "01";
537                 reset_C(1) <= '1';
538               elsif (sht_req_C(2)='1') then
539                 children_tmp(2)<='1';
540                 gate_no <= "10";
541                 reset_C(2) <= '1';
542               elsif (sht_req_C(3)='1') then
543                 children_tmp(3)<='1';
544                 gate_no <= "11";
545                 reset_C(3) <= '1';
546               end if;
547             else -- wrong command
548               reset_C <= cmd_rec_C;
549             end if;
550           elsif (eind_timeout = '1') then -- timeout
551             start_timeout <= '0';
552             -- reshout last address using sht_end
553             children_tmp <= adr_end;
554             start_shtreq <= '0';
555             main_state <= "0101";
556           else
```

```
557               reset_C <= "0000";
558               start_timeout <= '1';
559             end if;
560           when "1001" => -- send sht_ack ----- S9 ---
561             reset_C <= "0000";
562             if (count >= 5) then
563               count <= 0;
564               load <= '0';
565               bit_clk_ena <= '0';
566               snd_ena <= '0';
567               mux_C(to_integer(unsigned(gate_no))) <= "00";
568
569               reset_Tx_int <= '1';
570               main_state <= "0010";
571               start_timeout <= '1';
572             else
573               Tx_buf(2 downto 0) <= "011";
574               Tx_buf(10 downto 3) <= (others => '1');
575               load <= '1';
576               bit_clk_ena <= '1';
577               clk_sel <= "001";
578               snd_ena <= '1';
579               mux_C(to_integer(unsigned(gate_no))) <= "01";
580               if (count_clk='1') then
581                 count <= count + 1;
582               end if;
583             end if;
584           when "1111" => -- normal operation - SF ---
585             if (gone_P = '1') then
586               parent_gone <= '1';
587
588               start_shtreq <= '1';
589               out_ctrl <= "0000";
590               children <= "0000";
591               children_tmp <= "0000";
592               adr_end <= "0000";
593               BC <= '0';
594               SC <= '0';
595               adr_rec <= '0';
596               mux_P <= "00";
597               mux_C <= (others => "00");
598               bit_clk_ena <= '1';
599               clk_sel <= "001";
600               -- Wait a little bit befor resetting and
601               -- sending adpt_req. Surrounding modules
602               -- will see the dropping line as a command
603               if (count_clk = '1') then
```

```
604               if (count >= 10) then
605                 reset_fll_int <= '1'; -- full reset
606                 main_state <= "0000";
607                 bit_clk_ena <= '0';
608                 count <= 0;
609               else
610                 count <= count + 1;
611               end if;
612             end if;
613           elsif (gone_C /= "0000") then
614             -- A child or other offspring node is gone.
615             bit_clk_ena <= '1';
616             clk_sel <= "001";
617             if (count_clk = '1') then
618               -- don't close bypass to early
619               if (count = 0) then
620                 -- close bypass
621                 out_ctrl <= out_ctrl and not(gone_C);
622                 count <= count + 1;
623               elsif (count >= 4) then
624                 -- check data line after a couple of bits
625                 children <= children and (data_in or not(gone_C
                      ));
626                 out_ctrl <= children and (data_in or not(gone_C
                      ));
627                 adr_end <= children and (data_in or not(gone_C)
                      );
628                 reset_C <= gone_C;
629                 count <= 0;
630                 bit_clk_ena <= '0';
631               else
632                 count <= count + 1;
633               end if;
634             end if;
635           elsif (sht_req_P = '1') then -- polling
636             sht_self <= '0'; -- shout own address first
637             start_shtreq <= '1';
638             BC <= '0';
639             children_tmp <= "0000";
640             children <= "0000";
641             adr_end <= "0000";
642             out_ctrl <= "0000";
643             bit_clk_ena <= '1';
644             clk_sel <= "001";
645             if (count_clk = '1') then
646               mux_C <= (others => "00");
647               main_state <= "0101"; -- send sht_req
```

```
                reset_P <= '1';
                bit_clk_ena <= '0';
              end if;
            elsif ((adr_req_C or adpt_req_C)/="0000") then -- new
                /adopt child
              children_tmp <= children;
              main_state <= "0011";
            elsif ((sht_C or sht_end_C)/="0000" and reset_C = "
                0000") then
              -- some offspring node shouts address
              children_tmp <= children;
              main_state <= "0010"; -- pass address
            elsif (cmd_rec_P = '1' and (data_P or param_P)='0')
                then
              -- ignore all other commands from parent node
              reset_P <= '1';
            elsif (not(cmd_rec_C="0000")) then
              -- ignore all other commands from child node
              reset_C <= cmd_rec_C;
            else -- control signals during normal operation
              parent_gone <= '0';
              BC <= '1';
              SC <= '1';
              out_ctrl <= children;
              mux_P <= "11";  -- data_ext
              for i in 0 to 3 loop
                if (children(i)='1') then
                  mux_C(i) <= "10"; -- data_int
                else
                  mux_C(i) <= "00"; -- default
                end if;
              reset_Tx_int <= '0';
              reset_P <= '0';
              reset_C <= "0000";
              end loop;
            end if;
          when others =>
            main_state <= "1000";
        end case;
      end if;
    end if;
  end process;
---<<<<<<<<<< Stage Machine ---------------
------------ Data/Param ---->>>>>>>>>>>>---
-- See Appendix A
---<<<<<<<<<< Data/Param ------------------
------------ Bit_clk ------->>>>>>>>>>>>---
```

```
692 -- See Appendix A
693 ---<<<<<<<<< Bit_clk ----------------------
694 end main_ctrl_3_mdd;
```

D

VHDL implementation of *Main Control* (4)

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.all;
3  use IEEE.numeric_std.all;
4
5  entity main_ctrl_4_mdd is
6    port(
7      clk: in std_logic;
8      data_in: in std_logic_vector(3 downto 0);
9      rec_clk: in std_logic_vector(3 downto 0);
10     adr_av: in std_logic_vector(3 downto 0);
11
12     address: in std_logic_vector(7 downto 0);
13
14       -- command received
15     cmd_rec: in std_logic_vector(3 downto 0);
16       -- Rx detects missing modul
17     gone: in std_logic_vector(3 downto 0);
18       -- commands
19     send_req: in std_logic_vector(3 downto 0);
20     send_ack: in std_logic_vector(3 downto 0);
21     send_end: in std_logic_vector(3 downto 0);
22     ready: in std_logic_vector(3 downto 0);
```

```
23      adr_req1: in std_logic_vector(3 downto 0);
24      adr_req2: in std_logic_vector(3 downto 0);
25      adr_ack: in std_logic_vector(3 downto 0);
26      poll_req: in std_logic_vector(3 downto 0);
27      poll_ack1: in std_logic_vector(3 downto 0);
28      poll_ack2: in std_logic_vector(3 downto 0);
29      adpt_req: in std_logic_vector(3 downto 0);
30      adpt_ack: in std_logic_vector(3 downto 0);
31      param: in std_logic_vector(3 downto 0);
32      data: in std_logic_vector(3 downto 0);
33      reset: in std_logic_vector(3 downto 0);
34
35      param_in: in std_logic_vector(2 downto 0);
36        -- Rx receive buffers
37      rec_buf_0: in std_logic_vector(9 downto 0);
38      rec_buf_1: in std_logic_vector(9 downto 0);
39      rec_buf_2: in std_logic_vector(9 downto 0);
40      rec_buf_3: in std_logic_vector(9 downto 0);
41
42      POR: in std_logic;
43      rst: in std_logic;
44
45      data_int: out std_logic;
46      data_ext: out std_logic;
47
48      adr_ena: out std_logic;
49      param_ena: out std_logic;
50      data_ena: out std_logic;
51
52      SC: out std_logic;
53
54      Tx_buf: out std_logic_vector(23 downto 0);
55        -- cmd: Tx_buf(23 downto 20)
56        -- [parentgate: Tx_buf(19 downto 18)]
57        -- [address: Tx_buf(19 downto 12)]
58        -- Childgate: Tx_buf(11 downto 10)
59        -- receivebuffer: Tc_buf(9 downto 0)
60      load_out: out std_logic;
61      Tx_ena: out std_logic;
62      default_out: out std_logic;
63
64      mux_out_sel_0: out std_logic_vector(1 downto 0);
65      mux_out_sel_1: out std_logic_vector(1 downto 0);
66      mux_out_sel_2: out std_logic_vector(1 downto 0);
67      mux_out_sel_3: out std_logic_vector(1 downto 0);
68
69      reset_Rx: out std_logic_vector(3 downto 0);
```

```
70      reset_Tx: out std_logic;
71
72      reset_dp: out std_logic;
73      reset_fll: out std_logic;
74      reset_hrd: out std_logic;
75
76      main_state_out: out std_logic_vector (3 downto 0)
77    );
78  end main_ctrl_4_mdd;
79
80  architecture main_ctrl_4_mdd of main_ctrl_4_mdd is
81
82  component wacht_mdd
83    generic (COUNT: integer := 1);
84    -- When 'start' is high, 'wacht_mdd' (re)starts counting.
85    --  When COUNT is reached, 'eind' becomes high and
86    --  remains high until reset or start = 0.
87    port(
88    -- ...
89      );
90  end component;
91  signal start_timeout: std_logic;--:='0';
92  signal eind_timeout: std_logic;--:='0';
93
94  -- signal declarations
95  -- ...
96  -- all signals can come from either the Parent,
97  --  or a Child:
98  -- <signal>_P, <signal>_C
99  -- ...
100 -- controls the output multiplexers
101 --  for children and parent
102 type out_sel is array(3 downto 0) of std_logic_vector(1 downto 0)
      ;
103 signal mux_P: std_logic_vector(1 downto 0);
104 signal mux_C: out_sel;
105 -- 00: default output
106 -- 01: cmd/Tx
107 -- 10: data_int
108 -- 11: data_ext
109 -- ...
110
111 begin
112
113 main_state_out <= main_state;
114 ------------ State Machine -->>>>>>>>>>>---
115   --- signals ---
```

```
116   wacht_timeout: wacht_mdd
117   generic map(COUNT => 262144)  -- +- 13ms
118   port map(
119 -- timeout is started with start_timeout. runs for 13ms
120   );
121
122     -- according to in_control, distinction
123     --  between child and parent signals are made
124     -- ...
125
126   with gate_no select
127   rec_buf <= rec_buf_0 when "00",
128         rec_buf_1 when "01",
129               rec_buf_2 when "10",
130         rec_buf_3 when "11",
131         rec_buf_0 when others;
132
133     -- ...
134
135   process(clk)
136   begin
137     if (rising_edge(clk)) then
138       if (rst = '1') then
139         -- reset signals ...
140       else
141         case main_state is
142           when "0000" =>  -- begin state ----- S0 ---
143             if (reset_fll_int = '0') then
144               if (adr_av(0)='1') then
145                 in_ctrl <= "00";
146                 main_state <= "0001";
147               elsif (adr_av(1)='1') then
148                 in_ctrl <= "01";
149                 main_state <= "0001";
150               elsif (adr_av(2)='1') then
151                 in_ctrl <= "10";
152                 main_state <= "0001";
153               elsif (adr_av(3)='1') then
154                 in_ctrl <= "11";
155                 main_state <= "0001";
156               end if;
157             else
158               -- wait for the reset
159               reset_fll_int <= '0';
160             end if;
161           when "0001" =>  -- adr_req1/adpt_req S1 ---
162             reset_P <= '0';
```

```
163             if (count >= 8) then
164               count <= 0;
165               bit_clk_ena <= '0';
166               snd_ena <= '0';
167               mux_P <= "00";
168               reset_Tx_int <= '1';
169               load <= '0';
170
171               if (parent_gone = '1') then
172                 main_state <= "1100";
173               else
174                 main_state <= "0010";
175               end if;
176
177               start_timeout <= '1';
178             else
179               if (parent_gone = '1') then -- adpt_req
180                 Tx_buf(23 downto 20) <= "1101";
181                 Tx_buf(19 downto 0) <= (others => '0');
182               else            -- adr_req1
183                 Tx_buf(23 downto 20) <= "1010";
184                 Tx_buf(19 downto 18) <= (18=>in_ctrl(1), 19=>
                      in_ctrl(0));
185                 Tx_buf(17 downto 0) <= (others => '0');
186               end if;
187               load <= '1';
188               bit_clk_ena <= '1';
189               clk_sel <= "001";
190               snd_ena <= '1';
191               mux_P <= "01";
192               if (count_clk='1') then
193                 count <= count + 1;
194               end if;
195             end if;
196           when "0010" =>  -- receive address - S2 ---
197             reset_Tx_int <= '0';
198             reset_C <= "0000";
199             if (cmd_rec_P='1') then
200               start_timeout <= '0';
201               if (adr_ack_P='1') then
202                 if (adr_rec = '0') then
203                   if (count >= 8) then
204                     adr_read <= '0';
205                     reset_P <= '1';
206                     count <= 0;
207
208                     adr_rec <= '1';
```

```
209
210                   main_state <= "0011";
211                 else
212                   adr_read <= '1';
213                   clk_sel <= "000";
214                   if (count_clk='1') then
215                     count <= count + 1;
216                   end if;
217                 end if;
218               else -- if adr_rec = '1'
219                 if (count >= 9) then
220                   -- address must be bypassed competely
221                   reset_P <= '1';
222                   count <= 0;
223
224                     -- deactivate bypass
225                   mux_C(to_integer(unsigned(gate_no)))<= "00";
226                   main_state <= "0011";
227                 else
228                   clk_sel <= "000";
229                   if (count_clk ='1') then
230                     count <= count + 1;
231                   end if;
232                 end if;
233               end if;
234             else -- wrong command
235               if (count >= 9) then
236                 mux_C(to_integer(unsigned(gate_no))) <= "00";
237                 count <= 0;
238                 if (adr_rec = '0') then
239                   reset_fll_int <= '1';
240                   main_state <= "0000"; -- S0
241                 else
242                   main_state <= "0011"; -- S3
243                   reset_P <= '1';
244                 end if;
245               else
246                 clk_sel <= "000";
247                 if (count_clk='1') then
248                   count <= count + 1;
249                 end if;
250               end if;
251             end if;
252           elsif (eind_timeout = '1') then
253             start_timeout <= '0';
254             mux_C(to_integer(unsigned(gate_no))) <= "00";
255             if (adr_rec = '0') then
```

```
256               reset_fll_int <= '1';
257               main_state <= "0000"; -- S0
258             else
259               main_state <= "0011"; -- S3
260             end if;
261           end if;
262         when "0011" =>  -- receive cmd ----- S3 ---
263           reset_P <= '0';
264           if (count >= 20) then
265             -- adr_req1 timeout
266             count <= 0;
267             bit_clk_ena <= '0';
268             main_state <= "1001";
269           else
270             if (not(cmd_rec_C="0000")) then
271               -- command received
272               start_timeout <= '0';
273               count <= 0;
274               bit_clk_ena <='0';
275               reset_C <= "0000";
276               if (not((adr_req1_C or send_req_C or poll_ack1_C)
                      ="0000")) then
277                 start_sendreq <= not(BC);
278                 main_state <= "0100";
279                 if ((adr_req1_C(0) or send_req_C(0) or
                        poll_ack1_C(0)) = '1') then
280                   children_tmp <= children_tmp or "0001";
281                   gate_no <= "00";
282                 elsif ((adr_req1_C(1) or send_req_C(1) or
                        poll_ack1_C(1)) = '1') then
283                   children_tmp <= children_tmp or "0010";
284                   gate_no <= "01";
285                 elsif ((adr_req1_C(2) or send_req_C(2) or
                        poll_ack1_C(2)) = '1') then
286                   children_tmp <= children_tmp or "0100";
287                   gate_no <= "10";
288                 elsif ((adr_req1_C(3) or send_req_C(3) or
                        poll_ack1_C(3)) = '1') then
289                   children_tmp <= children_tmp or "1000";
290                   gate_no <= "11";
291                 end if;
292               elsif (not(send_end_C="0000")) then
293                 -- send_req is placed here so all adr_reqs are
                        processed first
294                 adr_end <= adr_end or send_end_C;
295                 reset_C <= send_end_C;
296               else -- wrong command
```

```
297                 reset_C <= cmd_rec_C;
298               end if;
299             elsif (children_tmp = "0000" and BC = '0') then
300               -- start adr_req timeout
301               bit_clk_ena <= '1';
302               clk_sel <= "001";
303               if (count_clk = '1') then
304                 count <= count + 1;
305               end if;
306             else
307               reset_C <= "0000";
308               if ((adr_end xor children_tmp)="0000") then
309                 start_timeout <= '0';
310                 if (children = children_tmp) then
311                   -- no own new child nodes
312                   main_state <= "1111"; -- SF
313                 else
314                   main_state <= "1001"; -- S9
315                 end if;
316               else -- send_end timeout
317                 start_timeout <= '1';
318                 if (eind_timeout = '1') then
319                   start_timeout <= '0';
320                   children_tmp <= adr_end;
321                 end if;
322               end if;
323             end if;
324           end if;
325         when "0100" =>  -- send send_req --- S4 ---
326           reset_P <= '0';
327           if (start_sendreq='1') then
328             if (count >= 6) then
329               count <= 0;
330               bit_clk_ena <= '0';
331               mux_P <= "00";
332               snd_ena <= '0';
333               reset_Tx_int <= '1';
334               load <= '0';
335               main_state <= "0101";
336
337               start_timeout <= '1';
338             else
339               Tx_buf(23 downto 20)<= "1000";
340               Tx_buf(19 downto 0)<= (others => '1');
341               load <= '1';
342               bit_clk_ena <= '1';
343               clk_sel <= "001";
```

```
344                 snd_ena <= '1';
345                 mux_P <= "01";
346                 if (count_clk ='1') then
347                   count <= count + 1;
348                 end if;
349               end if;
350             else
351               -- check data line
352               if (children = children_tmp) then
353                 -- no send_req if no own new child nodes
354                 main_state <= "0101"; -- S5
355                 start_timeout <= '1';
356               else
357                 if (count>=10) then
358                   count <= 0;
359                   bit_clk_ena <= '0';
360                   start_sendreq <= '1';
361                 else
362                   if (cmd_rec_P = '0') then
363                     bit_clk_ena <= '1';
364                     clk_sel <= "001";
365                     if (count_clk = '1') then
366                       count <= count +1;
367                     end if;
368                   else
369                     count <= 0;
370                     bit_clk_ena <= '0';
371                   end if;
372                 end if;
373               end if;
374             end if;
375           when "0101" =>  -- receive send_ack  S5 ---
376             reset_Tx_int <= '0';
377             if (cmd_rec_P='1') then
378               reset_P <= '1';
379               if (send_ack_P='1') then
380                 if (adr_req1(to_integer(unsigned(gate_no)))='1')
                      then
381                   main_state <= "0110"; -- send adr_req2
382                   reset_C(to_integer(unsigned(gate_no))) <= '1';
383                   start_timeout <= '0';
384                 elsif (poll_ack1(to_integer(unsigned(gate_no)))
                      ='1') then
385                   main_state <= "1011"; -- send poll_ack2
386                   reset_C(to_integer(unsigned(gate_no))) <= '1';
387                   start_timeout <= '0';
388                 elsif (send_req(to_integer(unsigned(gate_no)))
```

```
                =’1’) then
389               if (children = children_tmp) then
390                 -- no own new children
391                 main_state <= "1000";
392                 start_timeout <= ’1’;
393               else
394                 start_timeout <= ’0’;
395                 main_state <= "0111"; -- send send_ack
396               end if;
397               reset_C(to_integer(unsigned(gate_no))) <= ’1’;
398             end if;
399           else -- wrong command
400             start_timeout <= ’0’;
401             main_state <= "0011"; -- S3
402           end if;
403         elsif (eind_timeout = ’1’) then
404           if (reset_C /= "0000") then
405             start_timeout <= ’0’;
406             reset_C <= "0000";
407             main_state <= "0011";
408           else
409             reset_C <= "1111" xor send_end_C;
410           end if;
411         end if;
412       when "0110" =>  -- send adr_req2 --- S6 ---
413         reset_P <= ’0’;
414         reset_C <= "0000";
415         if (count >= 21) then
416           count <= 0;
417           bit_clk_ena <= ’0’;
418           snd_ena <= ’0’;
419           mux_P <= "00";
420           reset_Tx_int <= ’1’;
421           load <= ’0’;
422           main_state <= "0010";
423             -- activate bypass
424           mux_C(to_integer(unsigned(gate_no))) <= "10";
425
426           start_timeout <= ’1’;
427         else
428           Tx_buf(23 downto 20)<= "0110";
429           Tx_buf(19 downto 12)<= address;
430           Tx_buf(11 downto 10)<= (11=> gate_no(0), 10=>
                gate_no(1));
431           Tx_buf(9 downto 8) <= rec_buf(9 downto 8);
432           Tx_buf(7 downto 0)<= (others => ’1’);
433           load <= ’1’;
```

```
434             bit_clk_ena <= '1';
435             clk_sel <= "001";
436             if (count = 3) then
437               snd_ena <= '1';
438             end if;
439             mux_P <= "01";
440             if (count_clk='1') then
441               count <= count + 1;
442             end if;
443           end if;
444         when "0111" =>  -- send send_ack --- S7 ---
445           reset_P <= '0';
446           reset_C <= "0000";
447           if (count >= 6) then
448             count <= 0;
449             load <= '0';
450             bit_clk_ena <= '0';
451             snd_ena <= '0';
452               -- activate bypass
453             mux_C(to_integer(unsigned(gate_no))) <= "10";
454             out_ctrl(to_integer(unsigned(gate_no)))<='1';
455             mux_P <= "11";
456             reset_Tx_int <= '1';
457
458             main_state <= "1000";
459             start_timeout <= '1';
460           else
461             Tx_buf(23 downto 20)<= "0100";
462             TX_buf(19 downto 0)<= (others => '1');
463             load <= '1';
464             bit_clk_ena <= '1';
465             clk_sel <= "001";
466             snd_ena <= '1';
467             mux_C(to_integer(unsigned(gate_no))) <= "01";
468             if (count_clk='1') then
469               count <= count + 1;
470             end if;
471           end if;
472         when "1000" =>  --pass adr_req2/poll_ack2 S8
473           reset_P <= '0';
474           reset_Tx_int <= '0';
475           if (not(reset_C="0000")) then
476             reset_C <= "0000";
477           elsif (cmd_rec_C(to_integer(unsigned(gate_no)))='1')
                then
478             start_timeout <= '0';
479             if ((polling='0' and count>=13) or (polling='1' and
```

```
                count>=21)) then
480               count <= 0;
481               reset_C <= (adr_req2_C or poll_ack2_C);
482                 -- deactivate bypass
483               out_ctrl(to_integer(unsigned(gate_no)))<='0';
484               mux_P <= "00";
485               if (not((poll_ack2_C or adr_req2_C)="0000")) then
486                 if (polling = '1') then
487                   main_state <= "1110"; -- SE
488                 else
489                   main_state <= "0010"; -- S2
490                 end if;
491                 start_timeout <= '1';
492               else -- wrong command
493                   -- deactivate bypass
494                 mux_C(to_integer(unsigned(gate_no))) <= "00";
495                 main_state <= "0011";
496               end if;
497             else
498               clk_sel <= "1" & gate_no;
499               if (count_clk='1') then
500                 count <= count +1;
501               end if;
502             end if;
503           elsif (eind_timeout = '1') then
504             start_timeout <= '0';
505               -- activate bypass
506             out_ctrl(to_integer(unsigned(gate_no)))<='0';
507             mux_C(to_integer(unsigned(gate_no))) <= "00";
508             main_state <= "0011";
509           end if;
510         when "1001" =>  -- send send_end --- S9 ---
511           if (count >= 6) then
512             count <= 0;
513             bit_clk_ena <= '0';
514             snd_ena <= '0';
515             reset_Tx_int <= '1';
516             load <= '0';
517             main_state <= "1111";
518             polling <= '0';
519             children <= children_tmp;
520             BC <= '1';
521             SC <= '1';
522           else
523             Tx_buf(23 downto 20)<= "1100";
524             Tx_buf(19 downto 0)<= (others => '1');
525             load <= '1';
```

```
526             bit_clk_ena <= '1';
527             clk_sel <= "001";
528             snd_ena <= '1';
529             mux_P <= "01";
530             if (count_clk='1') then
531               count <= count + 1;
532             end if;
533           end if;
534         when "1010" =>  -- send poll_ack1 -- SA ---
535           reset_P <= '0';
536           if (count >= 16) then
537             count <= 0;
538             load <= '0';
539             bit_clk_ena <= '0';
540             snd_ena <= '0';
541             mux_P <= "00";
542             main_state <= "1110"; -- SE
543             reset_Tx_int <= '1';
544
545             start_timeout <= '1';
546           else
547             Tx_buf(23 downto 20) <= "1001";
548             Tx_buf(19 downto 12) <= address;
549             Tx_buf(11 downto 10) <= (11 => in_ctrl(0), 10=>
                  in_ctrl(1));
550             Tx_buf(9 downto 0) <= (others => '1');
551             load <= '1';
552             bit_clk_ena <= '1';
553             clk_sel <= "001";
554             snd_ena <= '1';
555             mux_P <= "01";
556             if (count_clk = '1') then
557               count <= count + 1;
558             end if;
559           end if;
560         when "1011" =>  -- send poll_ack2 -- SB ---
561           reset_P <= '0';
562           reset_C <= "0000";
563           if (count >= 29) then
564             count <= 0;
565             load <= '0';
566             bit_clk_ena <= '0';
567             snd_ena <= '0';
568             mux_P <= "00";
569             reset_Tx_int <= '1';
570               -- activate bypass
571             mux_C(to_integer(unsigned(gate_no))) <= "10";
```

```
572               start_timeout <= '1';
573               main_state <= "1110";
574             else
575               Tx_buf(23 downto 20) <= "0101";
576               Tx_buf(19 downto 12) <= address;
577               Tx_buf(11 downto 4) <= rec_buf(9 downto 2);
578               Tx_buf(3 downto 2) <= (3 => gate_no(0), 2 =>
                      gate_no(1));
579               TX_buf(1 downto 0) <= rec_buf(1 downto 0);
580               load <= '1';
581               bit_clk_ena <= '1';
582               clk_sel <= "001";
583               if (count = 3) then
584                 snd_ena <= '1';
585               end if;
586               mux_P <= "01"; --Tx
587               if (count_clk = '1') then
588                 count <= count + 1;
589               end if;
590             end if;
591           when "1100" =>  -- receive adpt_ack SC ---
592             reset_Tx_int <= '0';
593             if (cmd_rec_P = '1') then
594               start_timeout <= '0';
595               if (adpt_ack_P = '1') then
596                 parent_gone <= '0';
597                 BC <= '1';
598                 SC <= '1';
599                 adr_rec <= '1';
600
601                 reset_P <= '1';
602                 main_state <= "1111";
603               else    -- wrong command
604                 reset_fll_int <= '1';
605                 main_state <= "0000";
606               end if;
607             elsif (eind_timeout = '1') then
608               start_timeout <= '0';
609               reset_fll_int <= '1';
610               main_state <= "0000";
611             end if;
612           when "1101" =>  -- send adpt_ack --- SD ---
613             if (count >= 7) then
614               count <= 0;
615               load <= '0';
616               bit_clk_ena <= '0';
617               snd_ena <= '0';
```

```
618             reset_C <= "0000";
619             reset_Tx_int <= '1';
620             main_state <= "1111";
621           else
622             Tx_buf(23 downto 20) <= "0011";
623             Tx_buf(19 downto 0) <= (others => '1');
624             load <= '1';
625             bit_clk_ena <= '1';
626             clk_sel <= "001";
627             snd_ena <= '1';
628             for i in 0 to 3 loop
629               if (adpt_req_C(i)='1') then
630                 mux_C(i) <= "01"; -- Tx
631               end if;
632             end loop;
633             if (count_clk = '1') then
634               count <= count + 1;
635             end if;
636               -- adjust chldren vector
637             children <= children or adpt_req_C;
638             adr_end <= children or adpt_req_C;
639             reset_C <= adpt_req_C;
640           end if;
641         when "1110" =>  -- receive ready --- SE ---
642           reset_Tx_int <= '0';
643           reset_C <= "0000";
644           if (cmd_rec_P = '1') then
645             start_timeout <= '0';
646             clk_sel <= "001";
647             if (count_clk = '1') then
648               bit_clk_ena <= '0';
649                 -- deactivate bypass
650               mux_C(to_integer(unsigned(gate_no))) <= "00";
651               reset_P <= '1';
652               if (ready_P = '1') then
653                 ready_rec <= '1';
654                 main_state <= "0011";
655               else -- wrong command
656                   -- deactivate bypass
657                 mux_C(to_integer(unsigned(gate_no))) <= "00";
658                 if (ready_rec = '1') then
659                   main_state <= "0011";
660                 else
661                   main_state <= "1010";
662                 end if;
663               end if;
664             else
```

```
665                 bit_clk_ena <= '1';
666               end if;
667             elsif (eind_timeout = '1') then
668               start_timeout <= '0';
669                 -- deactivate bypass
670               mux_C(to_integer(unsigned(gate_no))) <= "00";
671               if (ready_rec = '1') then
672                 main_state <= "0011";
673               else
674                 main_state <= "1010";
675               end if;
676             end if;
677           when "1111" =>  -- normal operation SF ---
678             if (gone_P = '1') then
679               parent_gone <= '1';
680
681               out_ctrl <= "0000";
682               children <= "0000";
683               children_tmp <= "0000";
684               BC <= '0';
685               SC <= '0';
686               adr_rec <= '0';
687               mux_P <= "00";
688               mux_C <= (others => "00");
689               bit_clk_ena <= '1';
690               clk_sel <= "001";
691               if (count_clk = '1') then
692                 if (count >= 10) then
693                   reset_fll_int <= '1';
694                   main_state <= "0000";
695                   bit_clk_ena <= '0';
696                   count <= 0;
697                 else
698                   count <= count + 1;
699                 end if;
700               end if;
701             elsif (not(gone_C="0000")) then
702               bit_clk_ena <= '1';
703               clk_sel <= "001";
704               if (count_clk = '1') then
705                 if (count = 0) then
706                   out_ctrl <= out_ctrl and not(gone_C);
707                   count <= count + 1;
708                 elsif (count >= 4) then
709                   children <= children and (data_in or not(gone_C
                        ));
710                   out_ctrl <= children and (data_in or not(gone_C
```

```
                    ));
711                 adr_end <= children and (data_in or not(gone_C)
                    );
712                 reset_C <= gone_C;
713                 count <= 0;
714                 bit_clk_ena <= '0';
715               else
716                 count <= count + 1;
717               end if;
718             end if;
719           elsif (not(adr_req1_C="0000") or not(send_req_C="0000
                  ") or not(send_end_C="0000")) then
720             children_tmp <= children;
721             main_state <= "0011";
722           elsif (not(adpt_req_C="0000")) then
723             main_state <= "1101";
724           elsif(send_ack_P='1') then
725             reset_P <='1';
726           elsif (adr_ack_P ='1') then
727             children_tmp <= children;
728             main_state <= "0010";
729           elsif (poll_req_P = '1') then
730             polling <= '1';
731             ready_rec <= '0';
732             BC <= '0';
733             children <= "0000";
734             children_tmp <= "0000";
735             adr_end <= "0000";
736             out_ctrl <= "0000";
737             bit_clk_ena <= '1';
738             clk_sel <= "001";
739             if (count_clk = '1') then
740               mux_C <= (others => "00");
741               main_state <= "1010";
742               reset_P <= '1';
743               bit_clk_ena <= '0';
744             end if;
745           elsif (cmd_rec_P = '1' and (data_P or param_P)='0')
                  then
746             reset_P <= '1';
747           elsif (not(cmd_rec_C="0000")) then
748             reset_C <= cmd_rec_C;
749           else
750             out_ctrl <= children;
751               -- data_ext for parent
752             mux_P <= "11";
753             for i in 0 to 3 loop
```

```
754                 -- data_int for children
755                 if (children(i)='1') then
756                   mux_C(i) <= "10";
757                 else
758                   mux_C(i) <= "00";
759                 end if;
760               end loop;
761               bit_clk_ena <= '0';
762               reset_Tx_int <= '0';
763               reset_P <= '0';
764               reset_C <= "0000";
765             end if;
766           when others =>  -- tja
767         end case;
768       end if;
769     end if;
770   end process;
771 ---<<<<<<<<<< Stage Machine ---------------
772
773 ------------ Data/Param ---->>>>>>>>>>>>---
774   -- See Appendix A
775 ---<<<<<<<<<< Data/Param ------------------
776 ------------ Bit_clk ------->>>>>>>>>>>>---
777   -- See Appendix A
778 ---<<<<<<<<<< Bit_clk ---------------------
779 end main_ctrl_4_mdd;
```